MQL5 PROGRAMMING FOR TRADERS

Stanislav Korotky

MetaQuotes

Contenido

Programación en MQL5 para traders

El trading moderno depende en gran medida de la tecnología informática. La automatización traspasa ahora los límites de las bolsas y las oficinas de corretaje y se ha vuelto accesible para los usuarios de a pie a través de soluciones informáticas especializadas. Entre los pioneros en este campo destaca MetaTrader, que surgió a principios de la década de 2000. La última versión de la plataforma, MetaTrader 5, se mantiene a la vanguardia, evolucionando continuamente con características y funcionalidades innovadoras.

Un elemento clave que se perfecciona continuamente en MetaTrader 5 es su lenguaje de programación integrado MQL5, que permite a los operadores de trading ascender a un nuevo nivel de automatización de las operaciones, lo que se conoce de manera generalizada como trading algorítmico. Con MQL5, los operadores de trading pueden transformar sus estrategias en aplicaciones mediante la escritura de sus propios indicadores para el análisis, scripts para la ejecución de operaciones o Asesores Expertos para la completa automatización del trading. Al ser un sistema de trading automatizado, un Asesor Experto puede operar de forma autónoma, siguiendo los cambios de precios y alertando rápidamente a los operadores por correo electrónico o SMS.

El lenguaje de programación integrado permite a los operadores de trading aplicar casi cualquier concepto de trading, desde estrategias sencillas hasta complejos algoritmos basados en redes neuronales. MQL5 combina a la perfección las características tanto de lenguajes de programación específicos de un dominio como universales. A lo largo de los años, el lenguaje ha ido logrando valiosos avances, como la compatibilidad con gráficos 3D, la computación paralela mediante OpenCL, la integración con Python y la compatibilidad con bases de datos SQLite.

Si desea liberar todo el potencial de MetaTrader 5 debe profundizar en la programación. Este libro le ayudará a dominar MQL5 y a aprender a crear sus propias aplicaciones de trading.

Se da por sentado que el lector ya está familiarizado con MetaTrader 5. Otro requisito previo es comprender los principios fundamentales del funcionamiento de los terminales dentro de un sistema de información distribuido que facilite el comercio. El terminal Ayuda ofrece información detallada sobre todas las funciones disponibles.

Además, utilizando la API de MQL5, los operadores de trading pueden acceder a capacidades que van mucho más allá de la interfaz gráfica de usuario (GUI) de MetaTrader 5. Domine el lenguaje de programación para implementar escenarios complejos, automatizando diversos aspectos del funcionamiento del terminal y mejorando la eficacia de las estrategias de trading.

El libro está dividido en 7 partes, cada una de las cuales se centra en diferentes aspectos de la programación en MQL5.

- Parte 1: introduce los principios básicos de programación en MQL5 y MetaEditor, el marco estándar de MQL5. Los usuarios con experiencia en programación en otros lenguajes deberían tomar nota de las características del marco.
- Parte 2: explica los términos básicos, como tipos, instrucciones, operadores, expresiones, variables, bloques de código, estructuras de programa. Describe cómo se utilizan estos términos en el estilo de programación procedimental MQL5. Aquellos usuarios que conozcan bien MQL4 pueden saltarse esta parte y empezar a leer la Parte 3.
- Parte 3: trata de la programación orientada a objetos (POO) en MQL5. A pesar de su similitud con otros lenguajes que admiten el paradigma POO (especialmente con C++), MQL5 tiene ciertas características específicas. Para mi gusto, MQL5 es una especie de C±±.
- Parte 4: describe funciones integradas comunes aplicables a cualquier programa.

- Parte 5: aborda las características arquitectónicas de los programas MQL y su «especialización» en tipos para realizar diversas tareas de trading, como el análisis técnico mediante indicadores, la gestión de gráficos y el marcado de los mismos con la imposición de objetos gráficos en ellos, así como las respuestas a acciones y eventos interactivos en los que intervienen programas MQL.
- Parte 6: explica cómo analizar el entorno de trading y automatizar las operaciones mediante robots. Esta parte también presenta la interacción del programa con el probador en varios modos, incluida la optimización de estrategias.
- Parte 7: contiene información sobre el conjunto ampliado de API dedicadas que facilitan la integración de MQL5 con tecnologías adyacentes, como bases de datos, intercambio de datos de red, OpenCL, Python, etc.

A lo largo del libro, el material se presenta de forma equilibrada, combinando enfoques comunes, ejemplos y detalles técnicos. Se guía al lector pasando de un concepto a otro, como si se tratase de un problema del tipo del huevo y la gallina inherente al aprendizaje de programación. Para reforzar la comprensión, la mayoría de los programas MQL tratados en el libro están disponibles como códigos fuente para su exploración práctica en MetaEditor/MetaTrader 5.

Parte 1. Introducción a MQL5 y al entorno de desarrollo

Uno de los cambios más importantes de MQL5 en su reencarnación en MetaTrader 5 es que es compatible con el concepto de programación orientada a objetos (POO). En el momento de su aparición, el MQL4 (el lenguaje de MetaTrader 4) precedente se comparaba de forma convencional con el lenguaje de programación C, mientras que es más razonable buscar similitudes entre MQL5 y C++. Para ser justos, hay que señalar que hoy en día todas las herramientas de programación orientada a objetos que en origen sólo habían estado disponibles en MQL5 se han transferido a MQL4. Sin embargo, los usuarios que apenas saben programar siguen percibiendo la programación orientada a objetos como algo demasiado complicado.

En cierto sentido, este libro pretende hacer sencillas las cosas complejas. No sustituye, sino que se añade a la Referencia del Lenguaje MQL5 que se suministra con el terminal y que también está disponible en el sitio web mql5.com.

En este libro vamos a hablarle de forma coherente de todos los componentes y técnicas de programación en MQL5, paso a paso para que cada iteración sea clara y la tecnología POO vaya desplegando poco a poco su potencial, que es especialmente notable, como pasa con cualquier herramienta potente cuando se utiliza de forma adecuada y sensata. Como resultado, los desarrolladores de programas en MQL podrán elegir el estilo de programación que prefieran para una tarea concreta, es decir, no sólo el orientado a objetos, sino también el «antiguo» de procedimientos, así como utilizar diversas combinaciones de los mismos.

Los usuarios del terminal de trading pueden clasificarse cómodamente en «programadores» (aquellos que ya tienen cierta experiencia en programación en al menos un lenguaje) y «no programadores» (traders «puros» interesados en la capacidad de personalización del terminal mediante MQL5). Los primeros pueden, opcionalmente, saltarse la primera y la segunda parte de este libro, en las que se describen los conceptos básicos del lenguaje, y pasar directamente a descubrir las API específicas (interfaces de programación de aplicaciones, por sus siglas en inglés) integradas en MetaTrader 5. Para estos últimos, se recomienda una lectura progresiva.

Entre la categoría de «programadores», los que saben C++ parten con ventaja, ya que MQL5 y C++ son similares. Sin embargo, esta «medalla» tiene su reverso. La cuestión es que MQL5 no concuerda completamente con C++ (especialmente cuando se compara con los estándares recientes). Por tanto, los intentos de escribir una u otra estructura por hábito «como en los pluses» se verán interrumpidos con frecuencia por errores inesperados del compilador. Teniendo en cuenta elementos específicos del lenguaje, haremos todo lo posible por señalar estas diferencias.

Análisis técnico, ejecución de órdenes de trading o integración con fuentes de datos externas: todas estas funciones están a disposición de los usuarios del terminal tanto desde la interfaz de usuario como a través de las herramientas de software integradas en MQL5.

Dado que los programas en MQL5 deben realizar diferentes funciones, hay algunos tipos de programas especializados admitidos en MetaTrader 5. Se trata de una técnica habitual en muchos sistemas de software. Por ejemplo, en Windows, además de los programas habituales de ventanas, existen programas y servicios basados en línea de comandos.

Los siguientes tipos de programa están disponibles en MQL5:

- Indicadores: programas destinados a mostrar gráficamente arrays de datos calculados mediante una fórmula determinada, normalmente basada en la serie de cotizaciones;
- Asesores Expertos: programas para automatizar total o parcialmente las operaciones de trading;
- Scripts: programas destinados a realizar una acción cada vez;

- Servicios: programas para realizar acciones permanentes en segundo plano.

Más adelante hablaremos en detalle de las finalidades y particularidades de cada tipo. Es importante señalar ahora que todos ellos se crean en MQL5 y tienen mucho en común. Por lo tanto, empezaremos descubriendo las características comunes y poco a poco iremos conociendo la especificidad de cada tipo.

La característica técnica esencial de MetaTrader consiste en ejercer todo el control en el terminal del cliente, mientras que las órdenes iniciadas en él se envían al servidor. En otras palabras, las aplicaciones basadas en MQL sólo pueden funcionar dentro del terminal cliente, ya que la mayoría de ellas requieren una conexión «en vivo» con el servidor para funcionar correctamente. No hay aplicaciones instaladas en el servidor. El servidor se limita a procesar las órdenes recibidas del terminal de cliente y devuelve los cambios en el entorno de trading. Estos cambios también están disponibles para los programas en MQL5.

La mayoría de los tipos de programas en MQL5 se ejecutan en el contexto del gráfico, es decir, para lanzar un programa, hay que «lanzarlo» al gráfico deseado. La excepción es sólo un tipo especial, es decir, los servicios: estos están pensados para funcionar en segundo plano, sin fijarse al gráfico.

Recordamos que todos los programas en MQL5 están dentro de la carpeta de trabajo de MetaTrader 5, en la carpeta anidada que lleva por nombre */MQL5/<type>*, donde *<type>* es, respectivamente:

- *Indicators*
- *Experts*
- *Scripts*
- *Services*

Según la técnica de instalación de MetaTrader 5, la ruta a la carpeta de trabajo puede ser diferente (en concreto, con los derechos de usuario limitados en Windows, en un modo normal o portátil). Por ejemplo, puede ser:

```
C:/Program Files/MetaTrader 5/
```

o

```
C:/Users/<username>/AppData/Roaming/MetaQuotes/Terminal/<instance_id>/
```

El usuario puede saber dónde se encuentra exactamente esta carpeta ejecutando el comando *Archivo -> Abrir carpeta de datos* (disponible tanto en el terminal como en el editor). Además, cuando cree un nuevo programa, no necesita pensar en buscar la carpeta correcta gracias al uso del Asistente MQL integrado en el editor. Se invoca mediante el comando *Archivo -> Nuevo* y permite seleccionar el tipo de programa en MQL5 requerido. El archivo de texto correspondiente, que contiene una plantilla de código fuente, se creará automáticamente cuando sea necesario una vez completado el máster y, a continuación, se abrirá para su edición.

En la carpeta MQL5 hay otras carpetas anidadas, junto con las anteriores, y también directamente relacionadas con la programación en MQL5, pero nos referiremos a ellas más adelante.

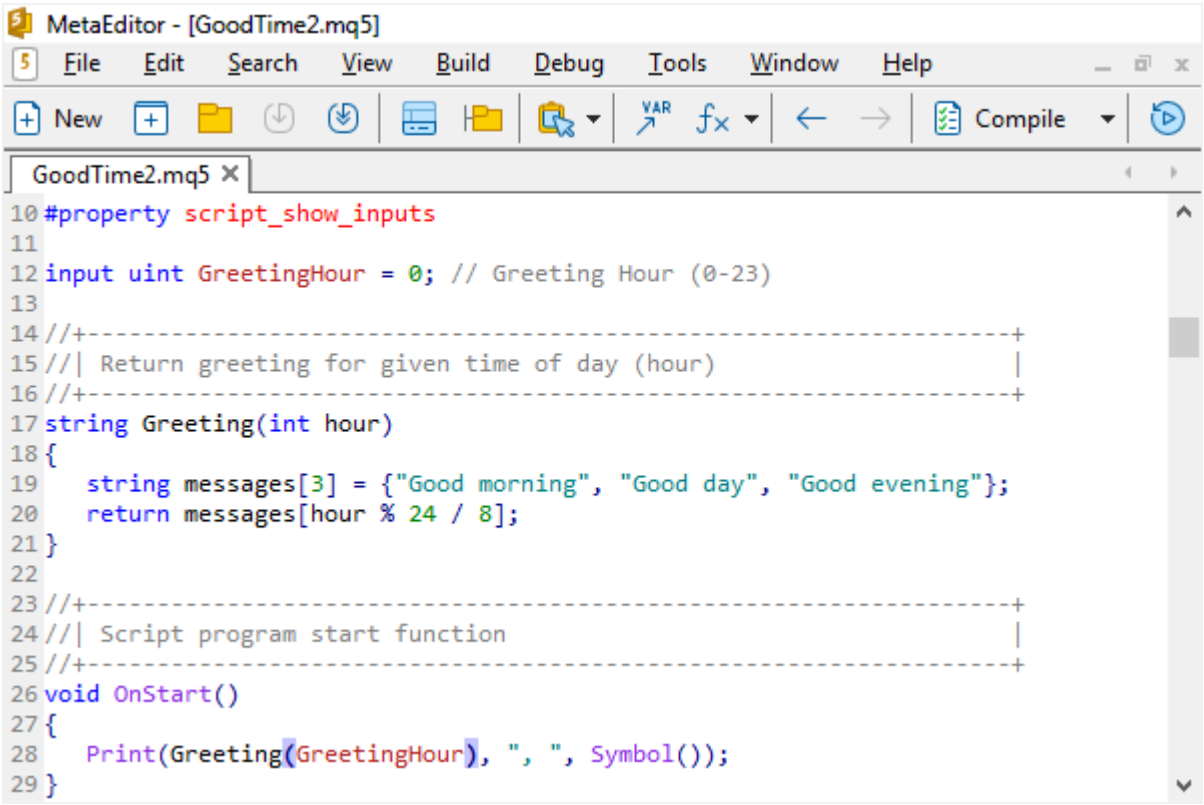
 Programación en MQL5 para Traders: códigos fuente del libro. Parte 1

 Los ejemplos del libro también están disponibles en proyecto público \MQL5\Shared Projects\MQL5Book

1.1 Edición, compilación y ejecución de programas

Todos los programas de MetaTrader 5 son compilables. Es decir, un código fuente escrito en MQL5 debe compilarse para obtener la representación binaria que será exactamente la que se ejecute en el terminal.

Los programas se editan y compilan con MetaEditor.



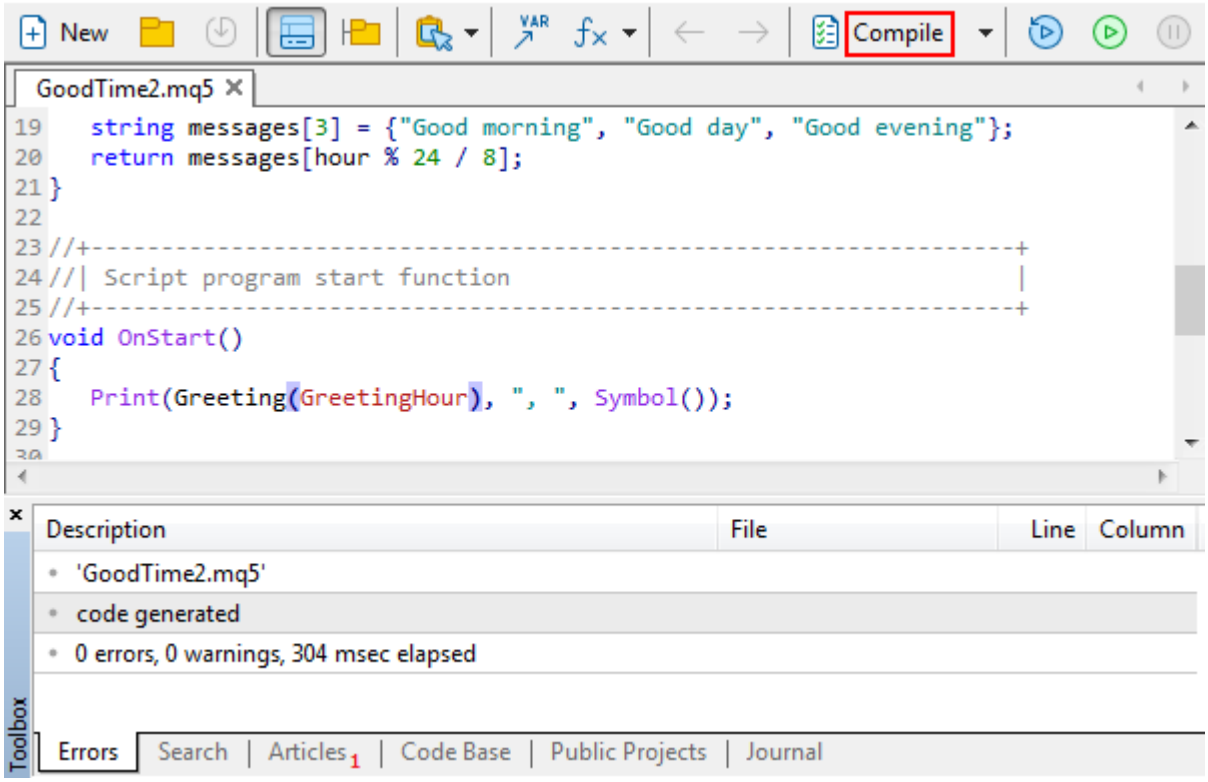
Edición de un programa MQL en MetaEditor

El código fuente es un texto escrito según las reglas de MQL5 y guardado como un archivo con la extensión *mq5*. El archivo que contiene un programa compilado tendrá el mismo nombre, mientras que su extensión será *ex5*.

En el caso más sencillo, un archivo ejecutable se corresponde con un archivo que contiene el código fuente; sin embargo, como veremos más adelante, la codificación de programas complejos requiere con frecuencia dividir el código fuente en varios archivos: el principal y algunos de apoyo que se habilitan desde el archivo principal de forma especial. En este caso, el archivo principal debe seguir teniendo la extensión mq5, mientras que los habilitados a partir de él deben tener la extensión mqh. Las sentencias procedentes de todos los archivos fuente entrarán entonces en el archivo ejecutable que se está generando. Así, varios archivos que contienen el código fuente pueden ser el punto de partida para crear un archivo ejecutable o programa. Todo esto que se menciona aquí para ofrecer una imagen completa, se va a presentar en la segunda parte del libro.

Utilizaremos el término «sintaxis MQL5» para denotar el conjunto de todas las reglas que permiten construir programas en MQL5. Sólo el estricto cumplimiento de la sintaxis permite codificar programas compatibles con el compilador. De hecho, enseñar a codificar consiste en introducir secuencialmente todas las reglas de un lenguaje concreto, que en nuestro caso es MQL5. Y este es el objetivo principal de este libro.

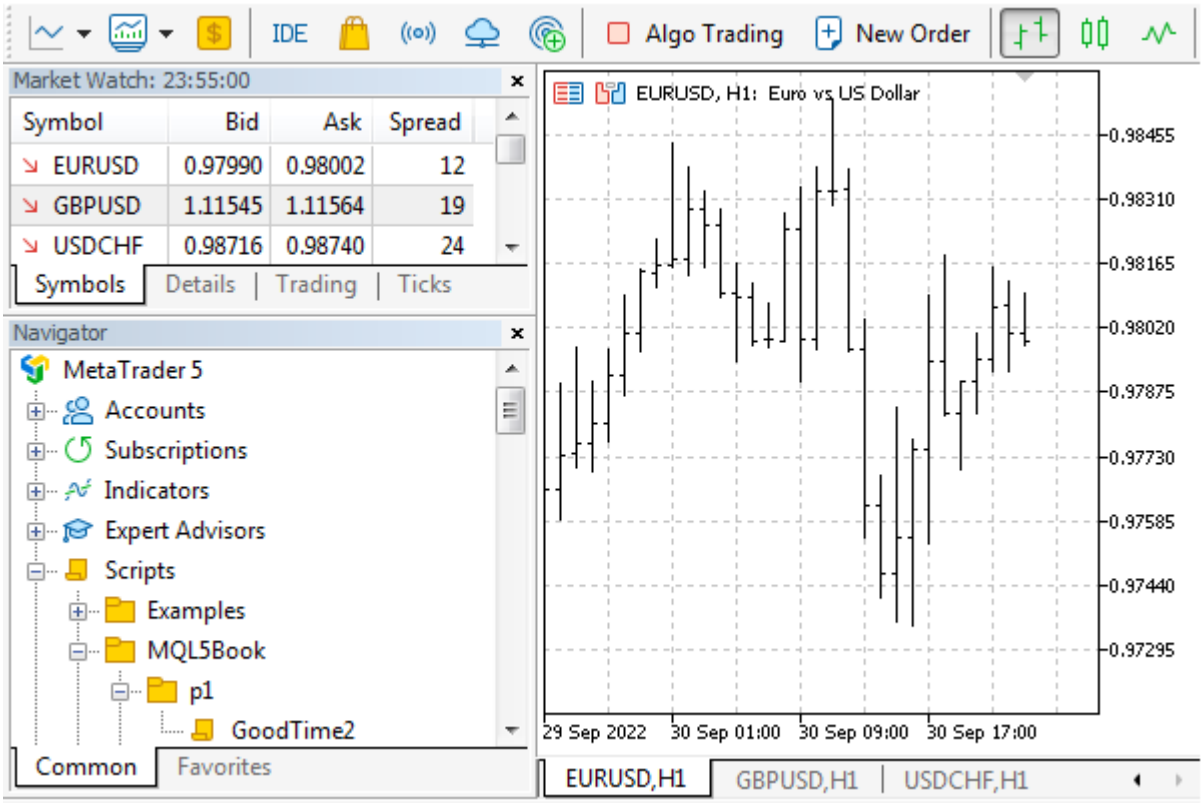
Para compilar un código fuente, podemos utilizar el comando MetaEditor *Archivo -> Compilar* o simplemente pulsar *F7*. Sin embargo, existen otros métodos especiales para compilar, de los que hablaremos más adelante. La compilación se acompaña de la visualización del estado cambiante en el registro del editor (donde un programa MQL5 se compone de varios archivos que contienen el código fuente, y la habilitación de cada archivo se marca en una sola línea de registro).



Compilación de un programa MQL5 en MetaEditor

Un indicio de que la compilación se ha realizado correctamente es la ausencia de errores («0 errores»). Las advertencias no afectan a los resultados de la compilación, sólo informan de posibles problemas. Por lo tanto, se recomienda solucionarlas del mismo modo que los errores (más adelante le explicaremos cómo hacerlo). Lo ideal sería que no hubiera ninguna advertencia («0 advertencias»).

Una vez compilado correctamente un archivo mq5, obtenemos un archivo del mismo nombre con la extensión *ex5*. MetaTrader 5 *Navegador* muestra en forma de árbol todos los archivos ejecutables ex5 situados en la carpeta MQL5 y sus subcarpetas, incluyendo el que se acaba de compilar.



Navegador MetaTrader 5 con un programa MQL5 compilado

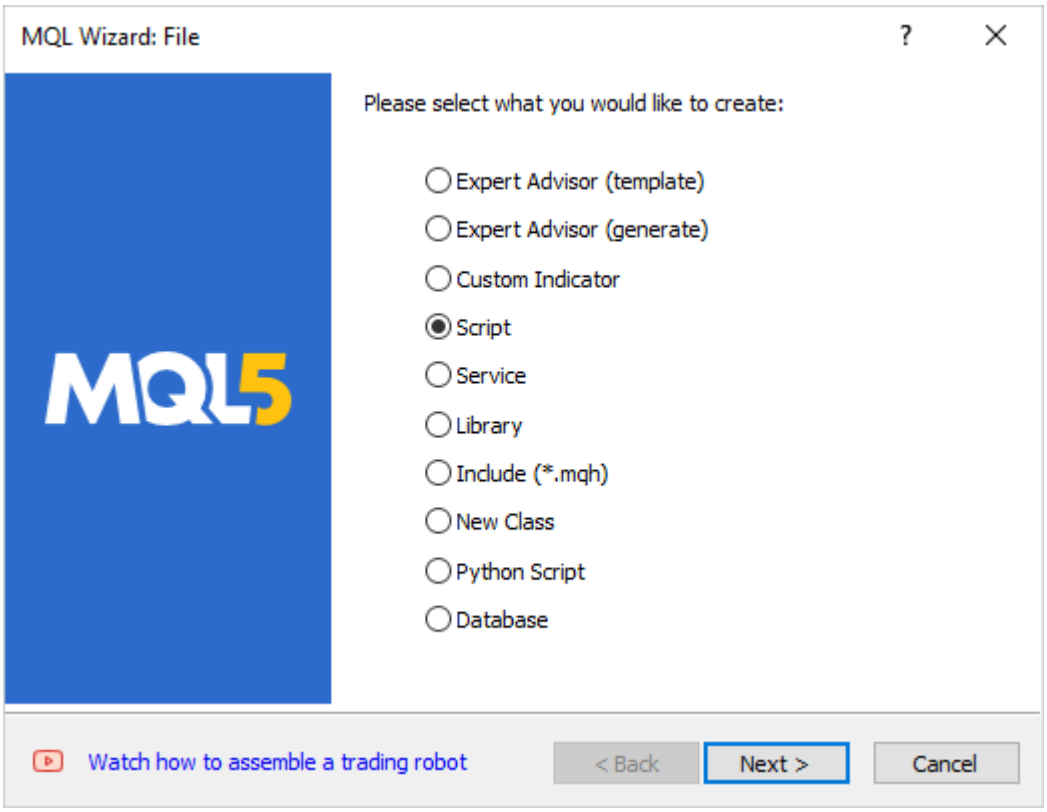
Los programas ya listos se lanzan en el terminal utilizando cualquier método familiar para el usuario. Por ejemplo, cualquier programa, que no sea un servicio, puede arrastrarse con el ratón desde *Navegador* al gráfico. Hablaremos de las características de los servicios por separado.

Además, los desarrolladores a menudo necesitan que el programa se ejecute en el modo de depuración para encontrar la causa de los errores. Existen varios comandos especiales para este fin, y nos referiremos a ellos en el apartado Corrección y depuración de errores.

1.2 Asistente MQL y borrador del programa

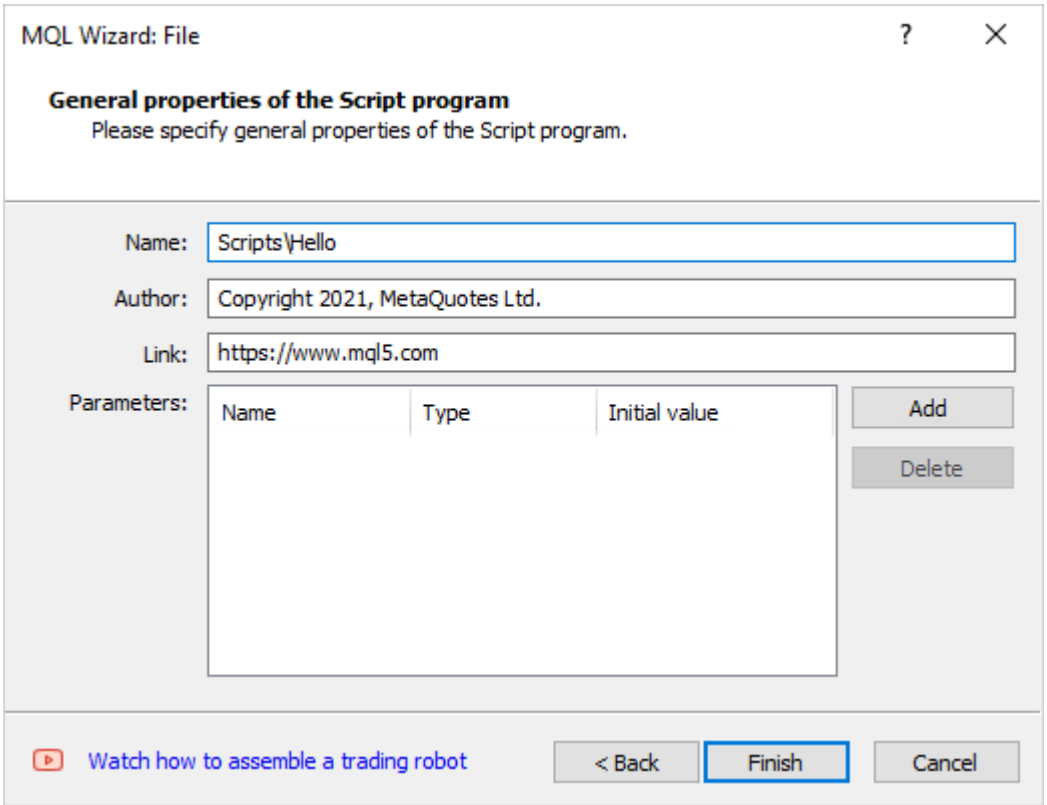
Aquí vamos a considerar el programa MQL más simple que en realidad no hace nada. Con ello se pretende introducir el proceso de escritura de un código fuente en el editor, compilarlo y lanzarlo en el terminal. Siguiendo de forma independiente los pasos que se indican a continuación, se asegurará de que la programación esté al alcance de los usuarios ocasionales y empezará a adaptarse al entorno de desarrollo integrado de los programas MQL5. Siempre será necesario para consolidar el material abordado.

Los programas MQL5 más sencillos son los scripts. Por lo tanto, lo que vamos a intentar crear es un script. Para ello, vamos a iniciar el Asistente de MQL5 (*Archivo -> Nuevo*). Como primer paso, seleccionamos *Script* en la lista de tipos y pulsamos *Siguiente*:



Creación de un script mediante el Asistente MQL. Paso 1

En el segundo paso, introduciremos el nombre del script en el campo Nombre, habiéndolo añadido tras la carpeta por defecto mencionada anteriormente y una barra invertida: «Scripts\». Por ejemplo, vamos a ponerle al script el nombre «Hello» (es decir, el campo Nombre contendrá la línea «Scripts\Hello») y, sin cambiar nada más, pulsamos *Finalizar*.



Creación de un script mediante el Asistente MQL. Paso 2

Como resultado, el Asistente creará un archivo llamado *Hello.mq5* y lo abrirá para su edición. El archivo se encuentra en la carpeta *MQL5/Scripts* (ubicación estándar para scripts) porque hemos utilizado la carpeta por defecto; sin embargo, podríamos añadir cualquier subcarpeta, o incluso una jerarquía de subcarpetas. Por ejemplo, si escribimos «Scripts\Exercise\Hello» en el campo *Nombre* en el paso 1 del Asistente, la subcarpeta *Exercise* se creará automáticamente en la carpeta *Scripts* y el archivo *Hello.mq5* se ubicará en esa subcarpeta.

Todos los ejemplos de este libro se encontrarán en las carpetas MQL5Book dentro de los catálogos asignados para los programas MQL de los tipos relevantes. Esto es necesario para facilitar la instalación de los ejemplos en su copia de trabajo del terminal y descartar cualquier conflicto de nombres con otros programas MQL que ya tenga instalados.

Por ejemplo, el archivo Hello.mq5 entregado como parte de este libro se encuentra en MQL5\Scripts\MQL5Book\p1\, donde p1 significa Parte 1 a la que este ejemplo se refiere.

La plantilla resultante del script *Hello.mq5* contiene el siguiente texto:

```
//+------------------------------------------------------------------+
//|                                                     Hello.mq5 |
//|                              Copyright 2021, MetaQuotes Ltd. |
//|                                          https://www.mql5.com |
//+------------------------------------------------------------------+

#property copyright "Copyright 2021, MetaQuotes Ltd."
#property link      "https://www.mql5.com"
#property version   "1.00"

//+------------------------------------------------------------------+
//| Script program start function                                    |
//+------------------------------------------------------------------+
void OnStart()
{
}

//+------------------------------------------------------------------+
```

Es este script el que se muestra en las capturas de pantalla anteriores de MetaEditor y MetaTrader 5.

Todas las cadenas que empiezan por «//» son los comentarios y no afectan a la intención del programa: no son procesados por el compilador ni ejecutados por el terminal. Sólo se utilizan para intercambiar información explicativa entre desarrolladores o para resaltar visualmente las partes del código con el fin de mejorar la legibilidad del texto. Por ejemplo, en esta plantilla, el archivo comienza con un bloque que contiene un comentario en el que se espera que se especifique el nombre del script y el copyright del autor. El segundo bloque de comentarios es el encabezamiento de la función principal del script, a la que nos referiremos con más detalle seguidamente. Por último, la última cadena de comentarios enfatiza visualmente el final del archivo.

Tres cadenas que comienzan con una directiva especial, *#property*, proporcionan al compilador algunos atributos que incorpora al programa de forma especial. En nuestro caso, por ahora no son importantes e incluso pueden suprimirse. Los directorios específicos están disponibles para cada tipo de programa MQL; los conoceremos en cuanto procedamos al aprendizaje de los tipos de programas concretos.

La parte principal del script, en la que vamos a describir la esencia de las acciones del programa, está representada por la función *OnStart*. Aquí tenemos que aprender los conceptos de 'bloque de código' y 'función'.

1.3 Sentencias, bloques de código y funciones

Así, en el script generado por el Asistente, la función *OnStart* aparece de la siguiente manera:

```
void OnStart()
{
}
```

Este es exactamente nuestro primer tema en el contexto de la programación en MQL5. También en este caso nos encontramos de inmediato con conceptos y secuencias de caracteres desconocidos. Para explicarlos, haremos una breve digresión.

Por lo general, un programa debe implementar las siguientes etapas típicas cuando se ejecuta:

- Definición de variables, es decir, celdas con nombre en la memoria del ordenador en las que almacenar datos;
- Organización de la entrada de datos de origen;
- Procesamiento de los datos (un algoritmo aplicado);
- Organización de la salida de los resultados.

Todas estas etapas no son necesarias en cuanto al mantenimiento de la corrección sintáctica del programa. Por ejemplo, si creamos un programa que calcule el producto de 2*2, obviamente no necesita ningún dato de entrada, ya que los números necesarios para la multiplicación están integrados en el texto del programa. Además, como 2 y 2 son valores constantes en esta expresión, no se necesitan celdas con nombre (variables) en el programa. Como de todos modos sabemos cuál es el doble dos, en realidad no necesitamos comunicar el número resultante. Por supuesto, un programa así carecería de toda función real. Sin embargo, sería absolutamente correcto desde un punto de vista técnico.

Y lo que es más interesante: el programa puede no contener ninguna sentencia sobre el procesamiento. Nuestra plantilla de script representa específicamente un programa 'null' de ejemplo. Pero, ¿qué es el fragmento de texto anterior?

En su día, Niklaus Wirth, uno de los grandes nombres de la programación, dio una sencilla definición generalizada de la programación como simbiosis de algoritmos y estructuras de datos.

Por «algoritmo» se entiende cualquier una secuencia de sentencias de un determinado lenguaje de programación. Una sentencia es un tipo de oración, es decir, un enunciado completo, articulado en un lenguaje de programación según las reglas sintácticas del mismo. El nombre mismo, «sentencia», sugiere que los ordenadores perciben esa oración como una guía para realizar operaciones. En otras palabras: las sentencias describen cuándo y cómo se procesarán las estructuras de datos aplicados requeridas. Ese es exactamente el motivo por el que la interpenetración de algoritmos y estructuras de datos permite poner en práctica las ideas del autor.

Por desgracia, en la mayoría de las tareas prácticas, el número de sentencias es tan grande que deben sistematizarse de alguna manera para que la persona reconozca y controle el comportamiento del programa.

También en este caso resulta útil el algoritmo «divide y vencerás», que se utiliza prácticamente en todos los ámbitos de la programación y bajo distintas formas. Las aprenderemos todas a medida que avancemos en este libro, ahora sólo estamos apuntando lo esencial.

Como es sabido, el algoritmo se reduce a dividir una tarea compleja de gran envergadura en otras más pequeñas y sencillas. Podemos comparar este proceso con la construcción de una casa o el ensamblaje de una nave espacial. Ambos «productos» se componen de múltiples módulos diferentes que, a su vez, constan de componentes, y estos últimos de piezas aún más pequeñas, y así sucesivamente.

Si ampliamos esta similitud a los algoritmos, podemos decir que las sentencias son pequeñas partes, mientras que el programa entero es la casa o la nave espacial. Por lo tanto, necesitamos bloques estructurales de tamaño intermedio.

Esa es la razón por la que es habitual, a la hora de implementar algoritmos, combinar sentencias relacionadas lógicamente en fragmentos más grandes con nombre: las funciones. En los lugares necesarios del programa, podemos dirigirnos a la función por su nombre (llamarla) y, al hacerlo, pedir al ordenador que ejecute todas las sentencias contenidas dentro de la función. El programa entero es, de hecho, el bloque externo más grande y, por lo tanto, también puede ser presentado por la función, desde la que se llama a funciones más pequeñas o se ejecutan sentencias de manera inmediata si no son muchas. Ahora nos acercamos a la función *OnStart*.

El nombre *OnStart* se reserva en los scripts para denotar la función última que el propio terminal invoca en respuesta a las acciones del usuario cuando éste lanza el script utilizando el comando del menú contextual o arrastrando el ratón sobre el gráfico. Así, el fragmento de código anterior define la función *OnStart* que predetermina el comportamiento de todo nuestro script.

Los que saben programar en otros lenguajes, como C, C++, Rust o Kotlin, pueden notar la similitud de esta función con la función *main*, el punto clave de acceder al programa.

Cualquier script debe contener la función *OnStart*. De lo contrario, la compilación puede finalizar con un error.

La función vacía *OnStart*, como la nuestra, comienza a ser ejecutada por el terminal (tan pronto como se lanza el script de la manera que sea) e inmediatamente termina su operación. En sentido estricto, todavía no se ha aplicado ningún algoritmo en nuestro script, pero ya existe una función stub para añadirlo.

En otros tipos de programas MQL existen también funciones especiales que el programador debe definir en su código. Hablaremos de las características específicas en las partes correspondientes del libro.

Analizaremos la sintaxis de definición de funciones en detalle en la segunda parte de este libro. A los efectos de realizar un análisis práctica de la misma, bastará con mencionar los siguientes elementos básicos para entender la descripción de *OnStart*.

Dado que las funciones suelen estar destinadas a obtener un resultado aplicable, las características del valor esperado se describen de manera especial en su definición: qué tipos de datos deben obtenerse y si estos son necesarios o no. Algunas funciones pueden realizar acciones que no requieren devolver ningún valor. Por ejemplo, una función puede estar destinada a cambiar la configuración del gráfico actual o a enviar notificaciones push cuando se alcanza el nivel de drawdown o porcentaje de caída predefinido en la cuenta. Todo esto puede programarse con las sentencias en la función, y no crea ningún dato nuevo (que tenga sentido devolver a cualquier otra parte del programa).

En nuestro caso, la situación es similar: como función principal del script, *OnStart* podría devolver su resultado sólo al entorno externo (directamente al terminal) una vez finalizado, pero esto no afectaría en modo alguno al funcionamiento del script en sí (porque ya ha finalizado).

Precisamente por eso, antes del nombre de la función *OnStart* aparece la palabra *void* que informa al compilador de que el resultado no nos importa (*void* significa vacío). *void* es una de las muchas palabras de procedimiento reservadas en MQL5. El compilador conoce el significado de todas las palabras reservadas y se guía por ellas a la hora de revisar el código fuente. En particular, un programador puede utilizar palabras reservadas para definir nuevos términos para el compilador, como la propia función *OnStart*.

Los paréntesis tras el nombre son parte integrante de la descripción de cualquier función: pueden incluir la lista de parámetros de la función. Por ejemplo, si escribiéramos una función que tomara el cuadrado de un número, tendríamos que proporcionarle un parámetro para ese número. A continuación podríamos llamar a esta función desde cualquier parte del programa, habiendo enviado un argumento sobre ella, es decir, el valor específico para el parámetro. Veremos más adelante cómo describir la lista de parámetros; ello no está en este ejemplo actual. Este requisito se plantea en la función *OnStart* ya que es invocada por el terminal en sí, y nunca envía nada a esta función como parámetros.

Por último, se utilizan llaves para marcar el principio y el final del bloque que contiene las sentencias. Inmediatamente después de la cadena con el nombre de la función, dicho bloque contendrá un conjunto de operaciones realizadas por esta función. Esto se denomina también cuerpo de la función. En este caso, no hay nada dentro de las llaves. Por lo tanto, la plantilla del script no hace nada.

La secuencia anterior de la palabra *void*, el nombre *OnStart*, una lista vacía de parámetros y un bloque de código vacío define la implementación mínima y vacía de la función *OnStart* para el compilador. Más adelante, añadiendo sentencias en el cuerpo de la función, ampliaremos la definición de la función *OnStart*.

Una vez ejecutado el comando *Compilar*, nos aseguraremos de que el script puede compilarse con éxito y que el programa terminado aparece en el *Navegador* del terminal en la carpeta *Scripts/MQL5Book/p1*. Esto se debe al hecho de que, en la carpeta correspondiente del disco, ahora está el archivo de *Hello.ex5*, lo que se puede comprobar fácilmente en cualquier administrador de archivos.

Podemos ejecutar el script en un gráfico, pero la única confirmación de su ejecución serán las entradas en el registro del terminal (pestaña *Log* de la ventana *Herramientas*; no confundir con la barra de herramientas):

```
Scripts       script Hello (EURUSD,H1) loaded successfully
Scripts       script Hello (EURUSD,H1) removed
```

Es decir: se ha cargado el script y se ha enviado el control a la función *OnStart*, pero se ha devuelto inmediatamente al terminal porque la función no hace nada, y después, el terminal se ha descargado el script del gráfico.

1.4 Primer programa

Intentemos añadir al script algo sencillo pero ilustrativo para demostrar su funcionamiento. Vamos a renombrar el script modificado como *HelloChart.mq5*.

En muchos libros de texto de programación, el ejemplo inicial imprime el solemne «Hola, mundo». En MQL5, un saludo similar podría tener el siguiente aspecto:

```
void OnStart()
{
  Print("Hello, world");
}
```

Pero vamos a hacerlo más informativo:

```
void OnStart()
{
  Print("Hello, ", Symbol());
}
```

Así, hemos añadido sólo una cadena con algunas estructuras lingüísticas.

Aquí, *Print* es el nombre de la función integrada en el terminal y destinada a mostrar mensajes en el registro *Asesores Expertos* (pestaña *Expert Advisors* de la ventana *Herramientas*; a pesar de llamarse *Expert Advisors*, la pestaña recoge mensajes de programas MQL de todo tipo). A diferencia de la función *OnStart* que definimos de forma independiente, la función *Print* ha sido definida para nosotros de antemano y para siempre. *Print* es una de las muchas funciones integradas que constituyen la API (interfaz de programación de aplicaciones) de MQL5.

La nueva línea en nuestro código denota la sentencia para llamar a la función *Print* enviándole la lista de argumentos (entre paréntesis) que se imprimirán en el registro. Los argumentos de la lista están separados por comas. En este caso, hay dos argumentos: la línea «Hola» y la llamada a otra función integrada, *Symbol*, que devuelve el nombre del instrumento activo del gráfico actual (el valor obtenido de ella pasará inmediatamente a la lista de argumentos de la función *Print*, en la ubicación desde la que se ha llamado a la función *Symbol*).

La función *Symbol* no tiene parámetros y, por lo tanto, no se le envía nada dentro de los paréntesis.

Por ejemplo, si el script se encuentra en el gráfico «EURUSD», la llamada a la función *Symbol()* devolverá «EURUSD» y, en cuanto al programa que se ejecuta, la sentencia relativa a la llamada a la función *Print* tendrá un nuevo aspecto: *Print("Hello, ", "EURUSD")*. Desde el punto de vista del usuario, por supuesto, todas estas llamadas a funciones y la sustitución dinámica de los resultados intermedios son fluidas e inmediatas. Sin embargo, para un programador es importante darse cuenta de cómo se ejecuta el programa paso a paso para evitar errores lógicos y lograr un cumplimiento estricto del plan concebido.

La línea «Hola» entre comillas dobles se denomina literal, es decir, una secuencia fija de caracteres que el ordenador percibe como un texto, tal y como está (tal como se introduce en el código fuente del programa).

Por lo tanto, la sentencia de impresión anterior debe imprimir los dos argumentos uno por uno en el registro, lo que debería dar como resultado de hecho la unión de las dos líneas y la obtención de «Hola, EURUSD».

Es importante destacar que la coma dentro de las comillas se imprimirá en el registro como parte de la línea y no se procesará de ninguna manera especial. A diferencia de ello, la coma que se coloca después de las comillas de cierre y antes de llamar a *Symbol()* es el carácter separador en la lista de argumentos, es decir, que afecta al comportamiento del programa. Si se omite la primera coma, el programa no perderá su corrección, aunque imprimirá la palabra «Hola» sin una coma después. Sin embargo, si se omite la segunda coma, el programa dejará de compilarse, ya que se romperá la sintaxis de la lista de argumentos de la función: todos los valores que contenga (en nuestro caso, son las dos líneas) deben estar separados por comas.

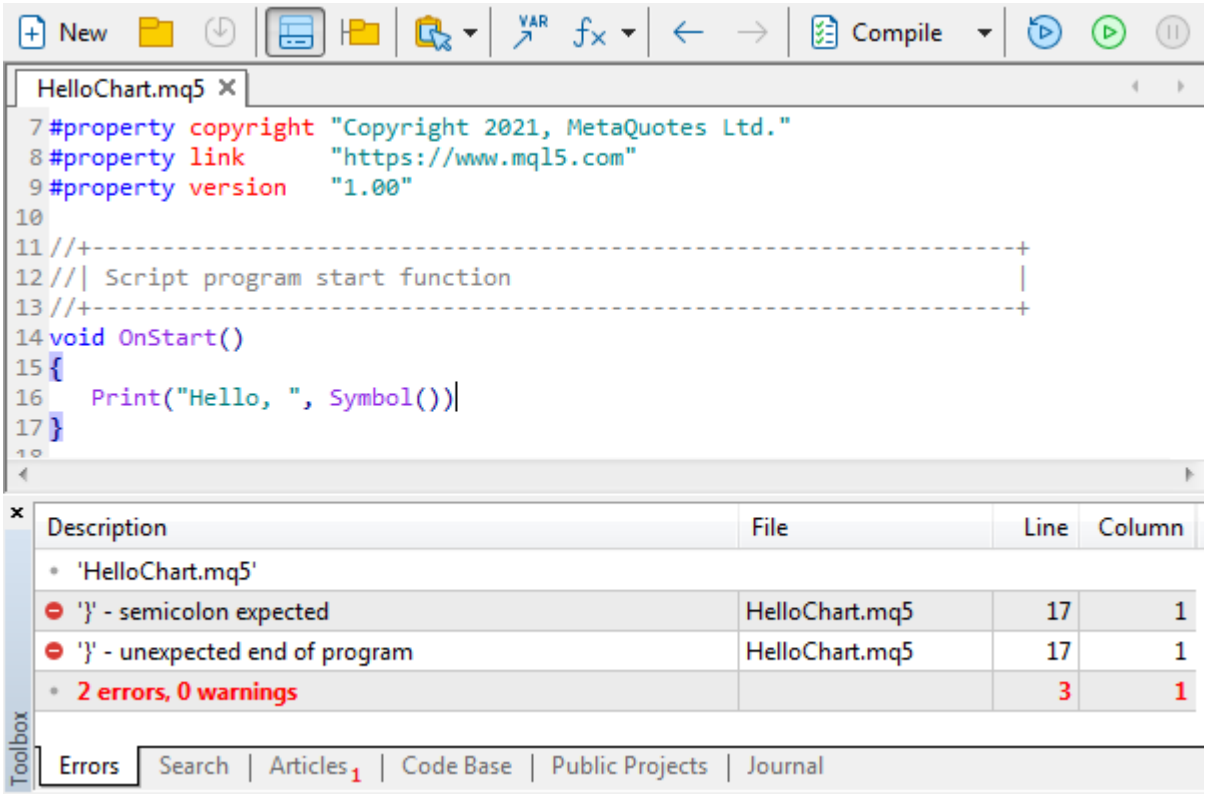
El error del compilador aparecerá de la siguiente manera:

```
'Symbol' - some operator expected HelloChart.mq5       16     19
```

El compilador se «queja» de la falta de algo antes de mencionar *Symbol*. Esto interrumpirá la compilación y el archivo ejecutable del programa no se creará. Por lo tanto, volveremos a poner la coma en su sitio.

Este ejemplo nos muestra lo importante que es seguir estrictamente la sintaxis del lenguaje. Los mismos caracteres pueden trabajar de forma diferente en distintas partes del programa. Así, incluso una pequeña omisión puede ser decisiva. Por ejemplo, observe el punto y coma al final de la línea que llama a *Print*. El punto y coma significa aquí el final de la sentencia. Si nos olvidamos de ponerlo, pueden producirse extraños errores de compilación.

Para comprobarlo, intentaremos eliminar este punto y coma y volver a compilar el script. Esto da lugar a la obtención de nuevos errores con la descripción del problema y su lugar en el código fuente.



Errores de compilación en el registro del MetaEditor

```
'}' - semicolon expected    HelloChart.mq5       17     1
'}' - unexpected end of program    HelloChart.mq5       17     1
```

El primer error especifica explícitamente la ausencia del punto y coma esperado por el compilador. Se propaga el segundo error: la llave de cierre que señalaba el final del programa se había detectado antes de que terminara la sentencia actual. En opinión del compilador, continúa, porque aún no se ha encontrado con el punto y coma. Es obvio cómo solucionar los errores: el punto y coma debe volver a colocarse en la posición correcta en la sentencia.

Vamos a compilar y lanzar el script arreglado. Aunque se ejecuta muy rápidamente y se elimina del gráfico prácticamente de inmediato, y aparece un registro que confirma la operación del script en el registro de *Experts*.

```
HelloChart (EURUSD,H1)      Hello, EURUSD
```

1.5 Tipos de datos y valores

Además de llamar a la función integrada *Symbol*, también podríamos utilizar nuestra propia función que hemos definido en el código fuente. Supongamos que queremos imprimir en el registro no sólo «Hola», sino diferentes saludos en función de la hora del día. Determinaremos la hora del día con precisión horaria: de 0 a 8 es por la mañana, de 8 a 16 por la tarde, y de 16 a 24 por la noche.

Es lógico sugerir que la estructura de definición de la nueva función debe ser similar a la de la función *OnStart* que ya conocemos. Sin embargo, su nombre debe ser único, es decir, no debe duplicar los nombres de otras funciones o palabras reservadas. Estudiaremos la lista de estas palabras más adelante en este libro de texto, aunque por ahora por suerte sugerimos que la palabra *Greeting* puede utilizarse como nombre.

Al igual que la función *Symbol*, esta función debe devolver una cadena; esta vez, sin embargo, la cadena debe ser una de las siguientes frases, dependiendo de la hora del día: «Buenos días», «Buenas tardes» o «Buenas noches».

Guiados por el sentido común, utilizamos aquí el concepto común de cadena. Al parecer, el compilador lo encuentra familiar, ya que vimos cómo había generado un programa imprimiendo el texto predefinido. Así, nos hemos acercado fácilmente al concepto de tipos en el lenguaje de programación, siendo uno de los tipos una cadena, es decir, una secuencia de caracteres.

En MQL5, este tipo se describe mediante la palabra clave *string*. Este es el segundo tipo que conocemos, el primero fue *void*. Ya hemos visto un valor de este tipo, sin saber que lo era: se trata del literal «Hola, ». Cuando nos limitamos a insertar una constante (en especial, algo así como un texto entrecomillado) en el código fuente, su descripción de tipo no es necesaria: define el tipo correcto automáticamente.

Utilizando la descripción de la función *OnStart* como muestra, podemos sugerir cómo debería aparecer la función *Greeting* en una primera aproximación.

```
string Greeting()
{
}
```

En este texto se indica nuestra intención de crear la función *Greeting*, que puede devolver un valor arbitrario del tipo *string*. Sin embargo, para que la función realmente devuelva algo, es necesario utilizar una sentencia especial con el operador *return* . Este es uno de los numerosos operadores MQL5; más adelante los exploraremos todos. Si la función tiene un tipo de valor de retorno distinto de *void*, debe contener el operador *return*.

En concreto, para devolver la antigua cadena de saludo «Hola, » de la función, debemos escribir:

```
string Greeting()
{
  return "Hello, ";
}
```

El operador *return* detiene la ejecución de la función y envía como resultado lo que está a su derecha. «Out» oculta el fragmento de código fuente desde el que se ha invocado a la función.

No hemos explorado todas las opciones para escribir expresiones que podrían formar una cadena arbitraria. Sin embargo, el caso más sencillo con el texto entrecomillado se traslada aquí sin cambios. Es importante que el tipo de valor de retorno coincida con el tipo de función, como en nuestro caso. Al final de la sentencia, ponemos un punto y coma.

No obstante, queríamos generar saludos distintos según la hora del día. Por tanto, la función debe tener un parámetro que defina la hora y que pueda tomar valores comprendidos entre 0 y 23. Obviamente, el número de la hora es un número entero, es decir, un número que no tiene parte fraccionaria. Está claro que el tiempo no se detiene en una hora, y que en ella se cuentan los minutos, siendo el número de minutos también un número entero. De nuevo, es inútil determinar la hora del día con precisión de un minuto. Por tanto, nos limitaremos a elegir el saludo únicamente por el número de la hora.

Para los valores enteros hay un tipo especial *int* en MQL5. Este valor debe enviarse a la función *Greeting* desde otro lugar del programa, desde el que esta función será invocada. Aquí nos hemos enfrentado por vez primera a la necesidad de describir una célula de memoria con nombre, es decir, una variable.

1.6 Variables e identificadores

Una variable es una celda de memoria que tiene un nombre único (para ser referenciada sin errores) y que puede almacenar los valores de un determinado tipo. Esta aptitud está garantizada por el hecho de que el compilador asigna a la variable exactamente la memoria necesaria para la misma en el formato interno especial: cada tipo tiene su tamaño y su formato de almacenamiento de memoria correspondiente. En la Parte 2 se ofrecen más detalles al respecto.

Básicamente, en el programa existe un término más estricto, el identificador, que se utiliza para los nombres de variables, funciones y muchas otras entidades que se descubrirán más adelante. El identificador cumple algunas reglas. En concreto, sólo puede contener caracteres latinos, números y guiones bajos, y no puede empezar por un número. Por eso, la palabra «Saludo» elegida para la función anterior cumple estos requisitos.

Los valores de una variable pueden ser diferentes y pueden modificarse mediante sentencias especiales durante la ejecución del programa.

Junto con su tipo y su nombre, una variable se caracteriza por el contexto, es decir, la zona del programa en la que se define y puede utilizarse sin errores de compilador. Nuestro ejemplo probablemente facilitará la comprensión de este concepto sin necesidad de un razonamiento técnico detallado al principio.

La cuestión es que una instancia concreta de una variable es el parámetro de la función. El parámetro sirve para enviar un determinado valor a la función. A este respecto, es obvio que el fragmento de código, en el que existe una variable de este tipo, debe limitarse al cuerpo de la función. En otras palabras: el parámetro puede utilizarse en todas las sentencias dentro del bloque de funciones, pero no fuera. Si el lenguaje de programación permitiera tales libertades, esto se convertiría en una fuente de

errores en abundancia debido a la posibilidad potencial de «estropear» la función en su interior a partir de un fragmento de programa aleatorio que no está relacionado con la función.

En todo caso, se trata de una definición ligeramente simplificada de una variable, lo cual es suficiente para esta sección introductoria. Más adelante estudiaremos algunos matices.

Por tanto, precisemos nuestros conocimientos sobre variables y parámetros: deben constar de tipo, nombre y contexto. Escribimos las dos primeras características en el código de forma explícita, mientras que la última resulta de la ubicación de la definición.

Veamos cómo podemos definir el parámetro del número de hora en la función *Greeting*. Ya conocemos el tipo deseado, se trata de *int*, y lógicamente podemos elegir el nombre: *hour*.

```
string Greeting(int hour)
{
  return "Hello, ";
}
```

Esta función seguirá devolviendo «Hola», sea cual sea la hora. Ahora deberíamos añadir algunas sentencias que seleccionaran diferentes cadenas como resultado, dependiendo del valor del parámetro *hour*. Recuerde que hay tres posibles opciones de respuesta de función: «Buenos días», «Buenas tardes» y «Buenas noches». Podríamos suponer que necesitamos 3 variables para describir estas cadenas. Sin embargo, en estos casos es mucho más cómodo utilizar un array, el cual garantiza un método unificado de codificación de algoritmos con acceso a elementos.

1.7 Asignación e inicialización, expresiones y arrays

Un array es un conjunto con nombre de celdas del mismo tipo que se encuentran en la memoria de forma contigua, y a cada uno de los cuales se puede acceder por su índice. En cierto sentido, se trata de una variable compuesta caracterizada por un identificador común, el tipo de valores almacenados y la cantidad de elementos numerados.

Por ejemplo, un array de 5 enteros puede describirse del siguiente modo:

```
int array[5];
```

El tamaño del array se especifica entre corchetes a continuación del nombre. Los elementos se numeran de 0 a N-1, siendo N el tamaño del array. Se accede a ellos, es decir, los valores se leen, utilizando una sintaxis similar. Por ejemplo, para imprimir el primer elemento del array anterior en el registro, podríamos escribir la siguiente sentencia:

```
Print(array[0]);
```

Tenga en cuenta que el índice 0 corresponde al primer elemento. Para imprimir el último elemento, la sentencia se sustituiría por lo siguiente:

```
Print(array[4]);
```

Se supone, por supuesto, que antes de imprimir un elemento del array, en él se ha escrito alguna vez un valor útil. Este registro se realiza mediante una sentencia especial, es decir, un operador de asignación. Una característica especial de este operador es el uso del símbolo '=', a cuya izquierda se especifica el elemento del array (o variable) en el que se realiza el registro, mientras que a su derecha se especifica el valor que se va a registrar o su «equivalente». Aquí, 'equivalente' oculta la capacidad del lenguaje para calcular expresiones aritméticas, lógicas y de otros tipos (las aprenderemos en la

Parte 2). La sintaxis de las expresiones es, en su mayor parte, similar a las reglas de escritura de las ecuaciones que aprendimos en aritmética y álgebra en la época escolar. Por ejemplo, en una expresión pueden utilizarse las operaciones de suma ('+'), resta ('-'), multiplicación ('*') y división ('/').

A continuación se muestran ejemplos de operadores para rellenar algunos elementos del array anterior.

```
array[0] = 10;                      // 10
array[1] = array[0] + 1;            // 11
array[2] = array[0] * array[1] + 1; // 111
```

Estas sentencias demuestran varios métodos de asignación y construcción de expresiones: En la primera línea, el literal 10 está escrito en el elemento *array[0]*, mientras que en la segunda y tercera líneas se utilizan las expresiones, cálculo que lleva a obtener los resultados especificados para mayor claridad visual en los comentarios.

Cuando en una expresión intervienen elementos de un array (o variables, en un caso general), el ordenador lee sus valores de la memoria durante la ejecución del programa y realiza con ellos las operaciones anteriores.

Es necesario distinguir el uso de variables y elementos de array a la izquierda y a la derecha del carácter '=' en la instrucción de asignación: a la izquierda hay un «receptor» de los datos procesados (siempre es único), mientras que a la derecha están las «fuentes» de datos iniciales para el cálculo (puede haber muchas «fuentes» en una expresión, como en la última cadena de este ejemplo, donde los valores de los elementos *array[0]* y *array[1]* se multiplican juntos).

En nuestros ejemplos se ha utilizado el carácter '=' para asignar los valores a los elementos de un array predefinido. Sin embargo, a veces es conveniente asignar valores iniciales a variables y arrays inmediatamente después de definirlos. Esto se denomina inicialización y para ello también se utiliza el carácter '='. Consideremos esta sintaxis en el contexto de nuestra tarea aplicada.

Vamos a describir el array de cadenas con las opciones de saludo dentro de la función *Greeting*:

```
string Greeting(int hour)
{
  string messages[3] = {"Good morning", "Good afternoon", "Good evening"};
  return "Hello, ";
}
```

En la sentencia añadida, no sólo se define el array *messages* con 3 elementos, sino también su inicialización, es decir, el llenado del mismo con los valores iniciales deseados. La inicialización resalta el carácter '=' sobre el nombre de la variable o array y la descripción del tipo. Para una variable, es necesario especificar un solo valor después de '=' (sin llaves), mientras que para un array, como podemos ver, podemos escribir varios valores separados por comas y encerrados entre llaves.

No confunda inicialización con asignación. El primero se especifica al definir una variable o array (y se hace una vez), mientras que el segundo se produce en sentencias específicas (se puede asignar la misma variable o elemento del array con diferentes valores una y otra vez). Los elementos de un array sólo pueden asignarse por separado: MQL5 no admite la asignación de todos los elementos a la vez, como es el caso de la inicialización.

El array *messages*, al estar definido dentro de la función, sólo está disponible dentro de ella, al igual que el parámetro *hour*. A continuación veremos cómo podemos describir las variables disponibles en todo el código del programa.

¿Cómo transformaremos el valor entrante de *hour* con el número de la hora en uno de los tres elementos?

Recordemos que, según nuestra idea, *hour* puede tener valores de 0 a 23. Si lo dividimos exactamente por 8, obtendremos los valores de 0 a 2. Por ejemplo, dividir 1 entre 8 nos dará 0, y 7 entre 8 dará 0 (en la división exacta, la parte fraccionaria se desprecia). Sin embargo, al dividir 8 entre 8 se obtiene 1, por lo que todos los números hasta 15 nos darán 1 al dividirlos entre 8. Los números del 16 al 23 se corresponderán con el resultado de la división de 2. Los enteros 0, 1 y 2 obtenidos se utilizarán como índices para leer el elemento del array *messages*.

En MQL5, la operación '/' permite calcular la división exacta para números enteros.

La expresión para obtener los resultados de la división es similar a las que hemos considerado recientemente para el *array*, sólo que hay que utilizar el parámetro *hour* y la operación '/'. Utilizaremos la siguiente sentencia como demostración de una posible implementación de la transformación *hour* en el índice de elementos:

```
int index = hour / 8;
```

Aquí, una nueva variable entera, *index*, se define e inicializa con el valor de la expresión anterior.

Sin embargo, podemos omitir guardar el valor intermedio en la variable *index* y transferir inmediatamente esta expresión (a la derecha de '=') dentro de corchetes, donde se especifica el número de elemento del array.

A continuación, en la sentencia con el operador *return*, podemos extraer el saludo pertinente de la siguiente manera:

```
string Greeting(int hour)
{
  string messages[3] = {"Good morning", "Good afternoon", "Good evening"};
  return messages[hour / 8];
}
```

La función está más o menos lista. No obstante, haremos algunas correcciones tras un par de secciones. De momento, guardemos el proyecto en un archivo con otro nombre, *GoodTime0.mq5*, e intentemos invocar nuestra función. Por esta razón, en *OnStart*, utilizaremos la llamada a *Greeting* dentro de la llamada a *Print*.

```
void OnStart()
{
  Print(Greeting(0), ", ", Symbol());
}
```

Hemos guardado la coma de separación (puesta dentro del literal «Hola, ») entre el saludo y el nombre del instrumento. Ahora hay tres argumentos en la llamada a la función *Print*: la primera y la última se calcularán sobre la marcha mediante llamadas, respectivamente, a las funciones *Greeting* y *Symbol*, mientras que la coma se enviará a imprimir tal cual.

Por ahora, estamos enviando la constante '0' a la función *Greeting*. Es su valor el que entrará en el parámetro *hour*. Una vez compilado y ejecutado el programa, podemos asegurarnos de que imprime el texto deseado en el registro.

```
GoodTime0 (EURUSD,H1)      Good morning, EURUSD
```

Sin embargo, en la práctica, los saludos deben seleccionarse dinámicamente, en función de la hora especificada por el usuario.

Así, hemos abordado la necesidad de organizar la introducción de datos.

1.8 Entrada de datos

La forma básica de transferir datos en un programa MQL es utilizar parámetros de entrada. Son similares a los de las funciones y las variables simples, en muchos aspectos, en concreto en términos de sintaxis de descripción y principios de su uso posterior en el código.

Además, la descripción de los parámetros de entrada presenta algunas diferencias esenciales:

- Se coloca en el texto fuera de todos los bloques (acabamos de descubrir los bloques que constituyen el cuerpo de las funciones, pero más adelante abordaremos los demás) o, en otras palabras, más allá de cualquier par de llaves;
- Comienza con la palabra clave *input*;
- Se inicializa con un valor predeterminado.

Normalmente se recomienda colocar los parámetros de entrada al principio del código fuente.

Por ejemplo, para definir un parámetro de entrada que permita introducir el número de la hora en nuestro script, deberá añadirse la siguiente cadena inmediatamente después del grupo de tres directivas *#property*:

```
input int GreetingHour = 0;
```

Este registro significa varias cosas.

- En primer lugar, en el script figura ahora la variable *GreetingHour*, que está disponible desde cualquier lugar del código fuente, incluso desde dentro de cualquier función. Esta definición se denomina definición de nivel global, lo que se debe a la ejecución del punto 1 de la lista anterior.
- En segundo lugar, el uso de la palabra clave *input* hace que dicha variable sea visible dentro del programa y en la interfaz de usuario, en el cuadro de diálogo de propiedades del programa MQL5, que se abre al iniciarse. Así, al iniciar el programa, un usuario establece el valor necesario de los parámetros (en nuestro caso, un parámetro*GreetingHour*), y estos se convierten en los valores de las variables correspondientes durante la ejecución del programa.

Fijémonos de nuevo en que el valor predeterminado que hemos especificado en el código se mostrará al usuario en el cuadro de diálogo. No obstante, el usuario podrá cambiarlo. En tal caso, es ese nuevo valor introducido manualmente el que se incluirá en el programa (no el valor de inicialización).

El valor inicial de los parámetros de entrada se ve afectado tanto por la inicialización en el código y la elección interactiva del usuario al lanzarlos, como por el tipo de programa MQL5 y la forma en que se lanza. La cuestión es que los diferentes tipos de programas MQL5 tienen diferentes ciclos de vida una vez que se lanzan en gráficos. Así, al colocarlos una sola vez en el gráfico, los indicadores y Asesores Expertos quedan «registrados» en él para siempre, hasta que el usuario los elimine explícitamente. Por lo tanto, el terminal recuerda los últimos ajustes seleccionados y los utiliza automáticamente; por ejemplo, al reiniciar el terminal. Sin embargo, los scripts no se guardan en gráficos entre las sesiones de terminal. Por tanto, al lanzar el script, sólo se nos mostrará el valor predeterminado.

Desafortunadamente, por alguna razón, la descripción de un parámetro de entrada no garantiza invocar el cuadro de diálogo de configuración al inicio del script (para scripts como tipo de programa MQL5 independiente). Para que esto suceda es necesario añadir al código una nueva directiva *#property* específica para el script:

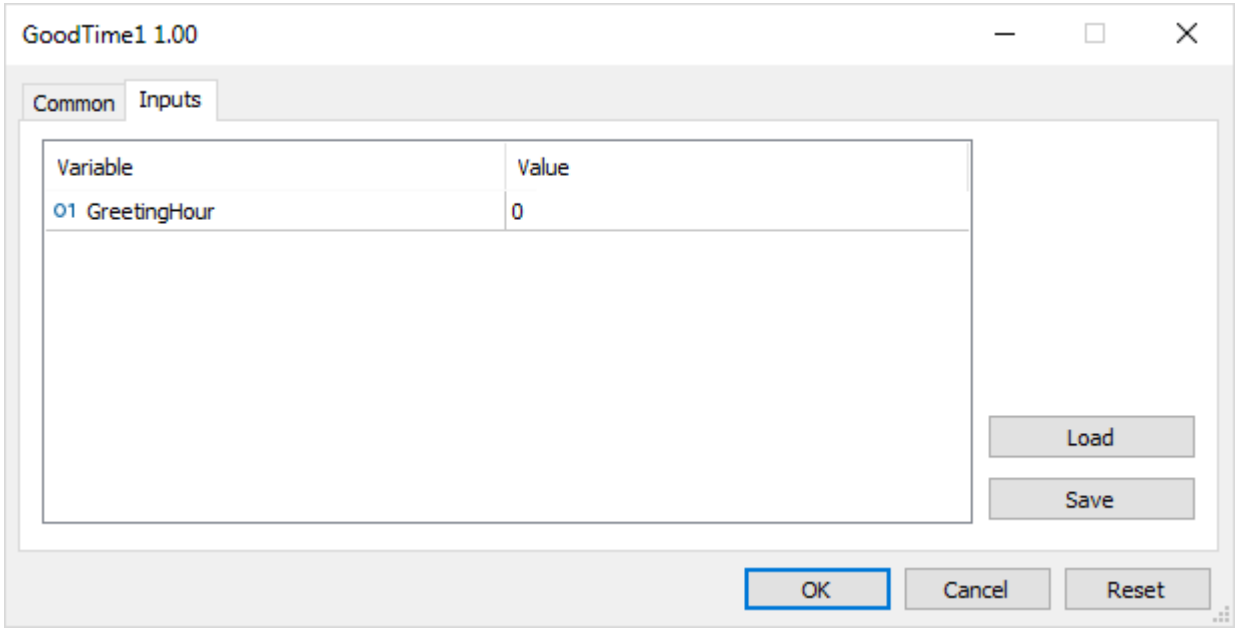
```
#property script_show_inputs
```

Como veremos más adelante, esta directiva no es necesaria para otros tipos de programas MQL5.

Necesitábamos *GreetingHour* para transferir su valor a la función *Greeting*. Para ello, basta con introducirlo en la llamada a la función *Greeting*, en lugar de 0:

```
void OnStart()
{
  Print(Greeting(GreetingHour), ", ", Symbol());
}
```

Teniendo en cuenta los cambios que hemos realizado para describir el parámetro de entrada, vamos a guardar la nueva versión del script en el archivo *GoodTime1.mq5*. Si lo compilamos e iniciamos, veremos el cuadro de diálogo de entrada de datos:



Cuadro de diálogo para introducir los parámetros del script GoodTime1.mq5

Por ejemplo, si editamos el valor *GreetingHour* a 10, el script mostrará el siguiente saludo:

```
GoodTime1 (EURUSD,H1)       Good afternoon, EURUSD
```

Se trata de un resultado correcto y esperado.

Sólo por diversión, vamos a ejecutar el script de nuevo e introducir 100. En lugar de una respuesta con sentido, obtendremos:

```
GoodTime1 (EURUSD,H1)       array out of range in 'GoodTime1.mq5' (19,18)
```

Acabamos de encontrarnos con un nuevo fenómeno: un error de ejecución. En este caso, el terminal notifica que en la posición 18 de la cadena 19, nuestro script ha intentado leer el valor de un elemento del array que tiene un índice inexistente (más allá del tamaño del array).

Puesto que los errores son compañeros permanentes y necesarios de un programador y tenemos que aprender a solucionarlos, vamos a hablar de ellos con más detalle.

1.9 Corrección y depuración de errores

El arte de programar se basa en la capacidad de dar instrucciones al programa sobre qué debe hacer y cómo debe hacerlo, y también de protegerlo frente a la posibilidad de que haga algo mal. Esta última labor es, por desgracia, mucho más difícil de ejecutar debido a múltiples factores no muy obvios que afectan al comportamiento del programa. Datos incorrectos, recursos insuficientes, errores de codificación propios y ajenos son sólo algunos de los problemas.

Nadie está a salvo de cometer errores al codificar programas. Los errores pueden producirse en distintas fases y se dividen acertadamente en:

- Errores de compilación devueltos por el compilador al identificar un código fuente que no cumple la sintaxis requerida (ya hemos visto tales errores más arriba); es más fácil solucionarlos porque el compilador se encarga de buscarlos;
- Errores de ejecución del programa devueltos por el terminal, si se da una condición incorrecta en el programa, como la división por cero, el cálculo de la raíz cuadrada de un negativo o el intento de hacer referencia a un elemento inexistente del array, como en nuestro caso; son más difíciles de detectar, ya que normalmente no se producen en cualquier valor de los parámetros de entrada, sino sólo en condiciones específicas;
- Errores de diseño del programa que conducen a su total apagado sin ningún dato por parte del terminal, como atascarse en un bucle infinito; estos errores pueden convertirse en los más complejos en términos de localización y reproducción, cuando la capacidad de reproducir un problema en el programa es una condición necesaria para solucionarlo a posteriori;
- Errores ocultos, en los que el programa parece funcionar sin problemas, pero el resultado proporcionado no es correcto; es fácil detectar si 2*2 no es 4, mientras que es mucho más difícil darse cuenta de las discrepancias.

Pero volvamos a la situación concreta con nuestro script. Según el mensaje de error que nos proporciona el entorno de ejecución del programa MQL, la siguiente sentencia es incorrecta:

```
return messages[hour / 8]
```

Al calcular el índice de un elemento del array, dependiendo del valor de la variable *hour*, se puede obtener un valor que supere el tamaño del array de tres.

El depurador incorporado en MetaEditor permite asegurarse de que esto realmente sucede así. Todos sus comandos están recogidos en el menú de depuración y ofrecen muchas funciones útiles. Aquí sólo nos vamos a fijar en dos: *Depurar -> Empezar con datos reales* (F5) y *Depurar -> Empezar con datos históricos* (Ctrl+F5). Puede obtener información sobre los otros en la Ayuda de MetaEditor.

Ambos comandos compilan el programa de una manera especial: con la información de depuración. Dicha versión del programa no está optimizada como en la compilación estándar (para obtener más detalles sobre la optimización, consulte Documentación), mientras que, al mismo tiempo, permite utilizar la información de depuración para «mirar dentro» del programa durante la ejecución: vea los estados de las variables y las pilas de llamadas a funciones.

La diferencia entre la depuración en datos reales y en datos históricos consiste en iniciar el programa en un gráfico en línea con la primera y en el gráfico de prueba en modo visual con la segunda. Para indicar al editor qué gráfico exactamente debe utilizar y con qué ajustes, es decir, símbolo, marco

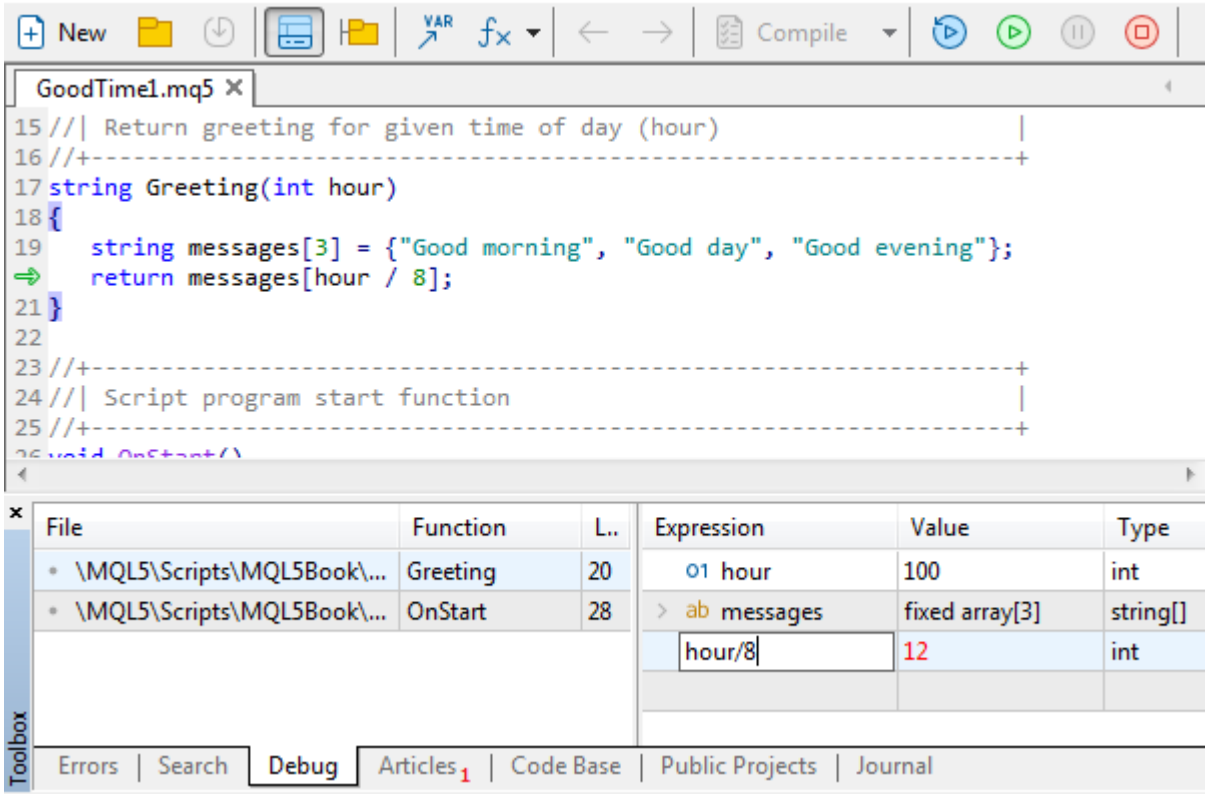
temporal, intervalo de fechas, etc., debe abrir para empezar el cuadro de diálogo *Ajustes -> Depurar* y rellenar en él los campos necesarios. La opción *Usar los ajustes especificados* debe estar activada. Si está desactivada, el primer símbolo de *Market Watch* y el marco temporal H1 se utilizarán en la depuración en línea, mientras que los ajustes del comprobador se utilizan al realizar la depuración en los datos históricos.

Tenga en cuenta que en el probador sólo se pueden depurar indicadores y Asesores Expertos. Sólo la depuración en línea está disponible para los scripts.

Vamos a ejecutar nuestro script utilizando F5 e introduciendo 100 en el parámetro *GreetingHour* para reproducir la situación problemática anterior. El script comenzará a ejecutarse y el terminal mostrará prácticamente de inmediato un mensaje de error, así como la solicitud para abrir el depurador.

```
Critical error while running script 'GoodTime1 (EURUSD,H1)'.
Array out of range.
Continue in debugger?
```

Una vez que respondemos afirmativamente, entramos en MetaEditor, donde aparece destacada la cadena actual del código fuente en la que se ha producido el error (fíjese en la flecha verde del campo de la izquierda).



MetaEditor en modo depuración en caso de error

La pila de llamadas actual se muestra en la parte inferior izquierda de la ventana: aquí se enumeran todas las funciones (en orden ascendente) que han sido invocadas antes de que la ejecución del código se detuviera en la cadena actual. En particular, en nuestro script, la función *OnStart* fue invocada (por el propio terminal), y la función *Greeting* fue invocada desde él (nosotros la invocamos desde nuestro código). En la parte inferior derecha de la ventana se muestra un panel de resumen. En él se pueden introducir los nombres de las variables, o las expresiones completas en la columna *Expression*, y ver sus valores en las columnas *Values* de la misma cadena.

Por ejemplo, podemos utilizar el comando *Añadir* del menú contextual o hacer doble clic con el ratón en la primera cadena libre para introducir la expresión «hora / 8» y asegurarnos de que es igual a 12.

Dado que la depuración se detuvo como consecuencia de un error, no tiene sentido continuar el programa; por tanto, podemos ejecutar el comando *Depurar -> Parar* (Mayús+F5).

En casos más complejos de una fuente de problemas no tan obvia, el depurador permite el seguimiento cadena a cadena de la secuencia de ejecución de las sentencias y del contenido de las variables.

Para resolver el problema es necesario asegurarse de que, en el código, el índice del elemento siempre cae dentro del rango de 0-2, es decir, se ajusta al tamaño del array. En sentido estricto, deberíamos haber escrito algunas sentencias adicionales comprobando que los datos introducidos eran correctos (en nuestro caso, *GreetingHour* sólo puede tomar un valor dentro del rango de 0-23), y luego mostrar un aviso o arreglarlo automáticamente en caso de violación de las condiciones.

En este proyecto introductorio no iremos más allá de una simple corrección: mejoraremos la expresión que calcula el índice del elemento para que su resultado siempre caiga dentro del rango requerido. Para ello, vamos a conocer un operador más: el operador módulo, que sólo funciona para números enteros. Para indicar esta operación se utiliza el símbolo «%». El resultado de la operación módulo es el resto de la división entera del dividendo por el divisor. Por ejemplo:

```
11 % 5 = 1
```

Aquí, con la división entera de 11 entre 5, obtendríamos 2, que se corresponde con el mayor factor de 5 dentro de 11, que es 10. El resto entre 11 y 10 es exactamente 1.

Para solucionar el error en la función *Greeting* basta con realizar previamente la división de módulo de *hour* entre 24, lo que garantizará que el número de la hora esté comprendido entre 0 y 23. La función *Greeting* tendrá el siguiente aspecto:

```
string Greeting(int hour)
{
  string messages[3] = {"Good morning", "Good afternoon", "Good evening"};
  return messages[hour % 24 / 8];
}
```

Aunque esta corrección seguramente funcionará bien (vamos a comprobarlo en un minuto), no afecta a otro problema que queda fuera de nuestro enfoque. La cuestión es que el parámetro *GreetingHour* es del tipo *int*, es decir, puede tomar valores tanto positivos como negativos. Si intentáramos introducir -8, por ejemplo, o un número «más negativo», obtendríamos el mismo error de ejecución, es decir, sobrepasar el array; sólo que, en este caso, el índice no sobrepasa el valor más alto (tamaño del array) sino que se hace más pequeño que el más bajo (en concreto, -8 lleva a referirse al elemento -1, curiosamente, los valores de -7 a -1 se muestran en el elemento 0 y no provocan ningún error).

Para solucionar este problema, sustituiremos el tipo del parámetro *GreetingHour* por el entero sin signo: utilizaremos *uint* en lugar de *int* (hablaremos de todos los tipos disponibles en la segunda parte, y aquí es *uint* lo que necesitamos). Guiado por el límite para la no negatividad de los valores, incorporado a nivel del compilador para *uint*, MQL5 garantizará de forma independiente que ni el usuario (en el cuadro de diálogo de propiedades) ni el programa (en su cálculo) «se vuelvan negativos».

Vamos a guardar la nueva versión del script como *GoodTime2*, compilarla y lanzarla. Introducimos el valor 100 para el parámetro *GreetingHour* y nos aseguramos de que, esta vez, el script se ejecute sin errores, mientras se imprime el saludo «Buenos días» en el registro del terminal. Este es el comportamiento esperado (correcto), ya que podemos utilizar una calculadora y comprobar que el

resto de la división de módulo de 100 entre 24 da 4, mientras que la división entera de 4 entre 8 es 0, lo que significa por la mañana, en nuestro caso. Desde el punto de vista del usuario, por supuesto, este comportamiento puede considerarse inesperado. Sin embargo, introducir 100 como número de hora también fue una acción inesperada del usuario. El usuario probablemente pensó que nuestro programa fallaría. Pero esto no sucedió, y eso es positivo. Por supuesto, con programas reales, los valores introducidos deben ser validados y el usuario debe ser informado de los errores.

Como medida adicional para evitar introducir un número incorrecto, también utilizaremos una función especial de MQL5 para dar un nombre más detallado y fácil de usar al parámetro de entrada. Para ello, utilizaremos un comentario después de la descripción del parámetro de entrada en la misma cadena. Por ejemplo, como este:

```
input uint GreetingHour = 0; // Greeting Hour (0-23)
```

Tenga en cuenta que hemos escrito las palabras del nombre de la variable por separado en el comentario (ya no es un identificador en el código, sino un consejo para el usuario). Además, hemos añadido entre paréntesis el intervalo de valores válidos. Al lanzar el script, el anterior *GreetingHour* aparecerá en el cuadro de diálogo para introducir los parámetros de la siguiente manera:

```
Greeting Hour (0-23)
```

Ahora podemos estar seguros de que, si se introduce 100 como hora, no es culpa nuestra.

Un lector atento puede preguntarse por qué hemos definido la función *Greeting* con el parámetro *hour* y le enviamos *GreetingHour* si podríamos utilizar en ella directamente el parámetro de entrada. La función, como fragmento lógico discreto de un código, sirve tanto para dividir el programa en partes visibles y fáciles de entender como para reutilizarlas posteriormente. Las funciones se suelen llamar desde varias partes del programa o forman parte de una biblioteca que está conectada a varios programas diferentes. Por tanto, una función escrita correctamente debe ser independiente del contexto externo y puede moverse de unos programas a otros.

Por ejemplo, si necesitamos transferir nuestra función *Greeting* a otro script, dejará de compilarse, ya que no contendrá el parámetro *GreetingHour*. No es del todo correcto exigir que se añada, porque el otro script puede calcular el tiempo de manera diferente. En otras palabras: al escribir una función, debemos hacer todo lo posible por evitar dependencias externas innecesarias. En lugar de ello, debemos declarar los parámetros de la función que se puedan rellenar con el código de llamada.

1.10 Salida de datos

En el caso de nuestro script, los datos se obtienen simplemente grabando el saludo en el registro mediante la función *Print*. En caso necesario, MQL5 permite guardar los resultados en archivos y bases de datos, enviarlos por Internet y mostrarlos como series gráficas (en indicadores) u objetos en gráficos.

La forma más sencilla de comunicar una sencilla información momentánea al usuario sin hacerle mirar el registro (que es una herramienta de servicio para monitorizar el funcionamiento de los programas y puede estar oculta en la pantalla) es la que proporciona la función *Comment* de la API de MQL5 . Se puede utilizar exactamente igual que la de Imprimir. Sin embargo, su ejecución hace que el texto no se muestre en el registro, sino en el gráfico actual, en su esquina superior izquierda.

Por ejemplo, una vez sustituido *Print* por *Comment* en el script del texto, obtendremos una función del tipo *Greeting*:

```
void OnStart()
{
  Comment(Greeting(GreetingHour), ", ", Symbol());
}
```

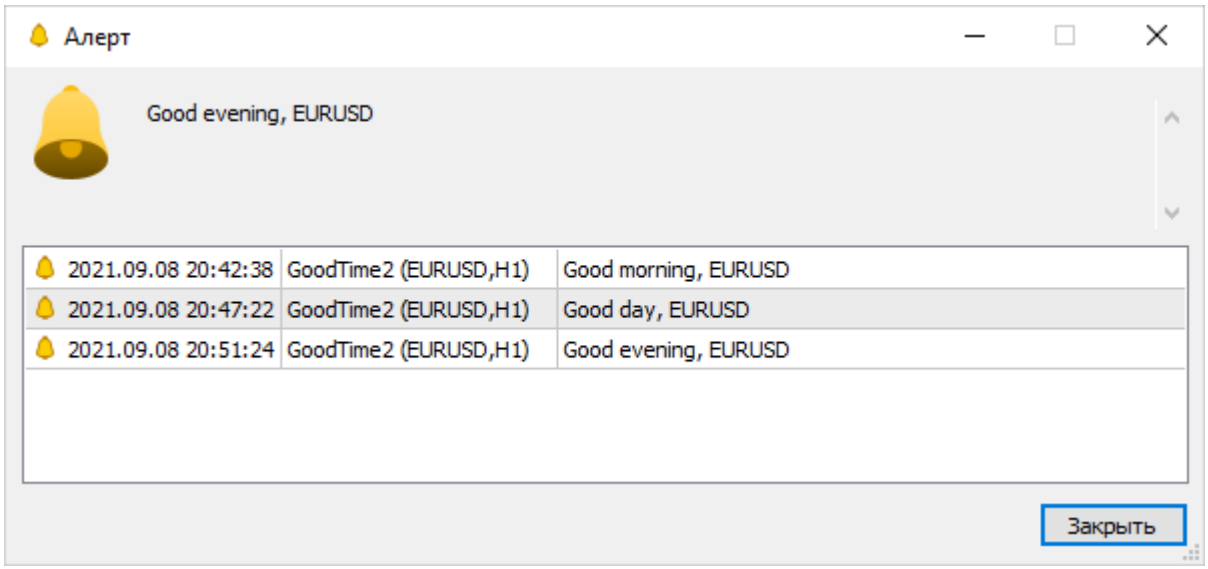
Una vez lanzado el script modificado en el terminal, veremos lo siguiente:

Visualización de información de texto en el gráfico mediante la función Comentario

Si necesitamos mostrar el texto al usuario y llamar al mismo tiempo su atención sobre un cambio en el entorno, relacionado con la nueva información, es mejor utilizar la función *Alert*. Ello envía una notificación a una ventana de terminal independiente que aparece sobre la ventana principal, acompañándola de una alerta sonora. Es útil, por ejemplo, en caso de una señal comercial o de eventos no rutinarios que requieran la intervención del usuario.

La sintaxis de *Alert* es idéntica a la de *Print* y *Comment*.

La imagen siguiente muestra el resultado de la operación de la función *Alert*.



Visualización de una notificación mediante la función Alert

Las versiones de los scripts con las funciones *Comment* y *Alert* no se adjuntan a este libro para que el lector pruebe y edite *GoodTime2.mq5* de forma independiente y reproduzca las capturas de pantalla que aquí se ofrecen.

1.11 Formato, sangría y espacios

MQL5 se encuentra entre los llamados lenguajes de forma libre, como los similares C y muchos otros. Esto significa que la colocación de símbolos de servicio, como corchetes u operadores, y palabras clave puede ser aleatoria, siempre que se respeten las reglas sintácticas. La sintaxis sólo limita la secuencia mutua de esos símbolos y palabras, mientras que el tamaño de la sangría al inicio de cada cadena o el número de espacios entre los elementos de la sentencia no tienen ningún significado para el compilador. En cualquier lugar del texto en el que sea necesario insertar un espacio para separar unos elementos de otros, como una palabra clave de tipo variable y un identificador de variable, puede utilizarse un mayor número de espacios. Además, en lugar de espacios, está permitido utilizar otros símbolos que denoten espacio vacío, como tabulaciones y saltos de línea.

Si hay un símbolo de separación (veremos más sobre ellos en la Parte 2) entre algunos elementos de la sentencia, como una coma ',' entre los parámetros de una función, entonces no es necesario utilizar ningún espacio.

Los cambios de formato del código fuente no modifican el código ejecutable.

Básicamente, hay muchos lenguajes que no son libres. En algunos de ellos, la formación de un bloque de código, que se realiza mediante la correspondencia de llaves en MQL5, se basa en sangrías iguales desde el borde izquierdo.

Gracias al formato libre, MQL5 permite a los programadores utilizar múltiples técnicas diferentes para dar forma al código fuente con el fin de mejorar su legibilidad y visibilidad y facilitar la navegación interna.

Veamos algunos ejemplos de cómo se puede grabar el texto fuente de la función *Greeting* desde nuestro script, sin cambiar su intención.

Aquí está la versión más "empaquetada" sin espacios ni saltos de línea excesivos (un salto de línea, denotado aquí con el símbolo '\', sólo se añade para cumplir con las restricciones sobre publicación de códigos fuente en este libro).

```
string Greeting(int hour){string messages[3]={"Good morning",\
"Good afternoon","Good evening"};return messages[hour%24/8];}
```

Esta es la versión, en la que se insertan espacios y saltos de línea excesivos.

```
string
Greeting ( int hour )
  {
    string messages [ 3 ]
            = {
                "Good morning" ,
                "Good afternoon" ,
                "Good evening"
              } ;

    return messages [ hour % 24 / 8 ] ;
  }
```

MetaEditor dispone de un estilizador de código integrado que permite formatear automáticamente el código fuente del archivo actual conforme a alguno de los estilos admitidos. Se puede seleccionar un estilo específico en el cuadro de diálogo *Tools -> Settings -> Styler*. El estilo se aplica mediante el comando *Tools -> Styler*.

Debe tener en cuenta que su libertad de espaciado es limitada. En particular, no puede insertar espacios en identificadores, palabras clave ni números. De lo contrario, el compilador no podrá reconocerlos. Por ejemplo, si insertamos sólo un espacio entre los dígitos 2 y 4 en el número 24, el compilador devolverá un montón de errores al intentar compilar el script.

He aquí una cadena modificada incorrectamente a sabiendas:

```
return messages[hour % 2 4 / 8];
```

Aquí está el registro de errores:

```
'GoodTime2.mq5'      GoodTime2.mq5 1      1
'4' - some operator expected      GoodTime2.mq5 19     28
'[' - unbalanced left parenthesis GoodTime2.mq5 19     18
'8' - some operator expected      GoodTime2.mq5 19     32
']' - semicolon expected    GoodTime2.mq5 19     33
']' - unexpected token      GoodTime2.mq5 19     33
5 errors, 0 warnings        6      1
```

Los mensajes del compilador no siempre son claros. Hay que tener en cuenta que, incluso en el primer error (en sucesión), existe una alta probabilidad de que la representación interna del programa (tal y como la percibió el compilador en «mitad de la frase») difiera considerablemente de lo que el programador ha sugerido. En concreto, en este caso, sólo el primer y el segundo error contienen la clave para entender el problema, mientras que todos los demás se propagan.

Según el primer error, el compilador esperaba encontrar el símbolo de una operación entre 2 y 4 (ya que percibe 2 y 4 como dos números diferentes y no como 24 separados por un espacio). La lógica alternativa consiste en que aquí se omite un corchete de cierre y el compilador muestra el segundo

error: «'[' - paréntesis izquierdo desequilibrado». Después de ejecutarse, la expresión queda completamente destruida, por lo que el número 8 y el corchete de cierre ']' aparecen como inapropiados para el compilador. Pero, de hecho, si suprimimos simplemente el espacio excesivo entre 2 y 4, la situación se normalizará.

Por supuesto, es mucho más fácil realizar un análisis de errores de este tipo cuando hemos añadido el problema de forma intencionada. No siempre entendemos en la práctica cómo solucionar esta o aquella situación. Incluso en el caso anterior, suponiendo que haya recibido este código roto de otro programador y los elementos del array no contengan información tan trivial, es fácil sospechar otra opción de corrección: debe quedar el 2 o el 4, ya que probablemente el autor haya querido sustituir un número por otro y no ha limpiado las «huellas».

1.12 Miniresumen

En la Parte 1 nos familiarizamos con el marco de MetaEditor, creamos una plantilla de script usando MQL Master y rellenamos el script poco a poco con código para resolver un problema sencillo. Para ello, utilizamos algunos principios básicos y estructuras sintácticas de MQL5. Después probamos el depurador en la práctica, solucionamos algunos problemas y llegamos a un funcionamiento estable del programa.

Nuestras muestras de script evolucionaron de la siguiente manera:

En las siguientes secciones de este libro empezaremos a explorar en detalle estas y otras muchas características de MQL5, los aspectos técnicos de la programación y sus aplicaciones para el trading.

Parte 2. Fundamentos de programación en MQL5

Como cualquier otro lenguaje de programación, MQL5 se basa en algunos conceptos fundamentales que sirven para crear estructuras más complejas y, con el tiempo, programas enteros. Aquí vamos a descubrir la mayoría de los conceptos, como tipos de datos, identificadores, variables, expresiones y operadores, así como las técnicas para combinar varias sentencias en el código para construir la lógica de funcionamiento de un programa.

El material les servirá de ayuda a nuestros lectores para progresar en la aplicación práctica independiente de la programación procedimental, una de las primeras tendencias en programación para la resolución de diversos problemas. De hecho, consiste en dar forma a un programa a partir de pequeños pasos (sentencias) que han de ejecutarse en la secuencia requerida para el tratamiento de datos. El script de texto que aparece en la Parte 1 de este libro es un ejemplo de este estilo.

En esta sección se cubre un amplio espectro de conceptos fundamentales y herramientas esenciales programar correctamente en MQL5, incluyendo las siguientes subsecciones:

Identificadores:

- Los identificadores constituyen la base del código de cualquier programa. En esta subsección se aborda el propósito y las reglas para nombrar identificadores en MQL5.

Tipos de datos integrados:

- MQL5 incluye una variedad de tipos de datos integrados, cada uno de ellos diseñado para almacenar y procesar tipos específicos de información. En esta sección se ofrece una visión global de los tipos de datos básicos.

Variables:

Las variables se utilizan para almacenar y gestionar datos en un programa. En la sección «Variables» se enseñan los fundamentos del trabajo con variables y se analiza cómo declararlas, inicializarlas y asignarles valores.

Arrays:

- Los arrays ofrecen una forma estructurada de almacenar datos. En esta sección se cubren los conceptos básicos de la creación y el uso de arrays en MQL5.

Expresiones:

- Las expresiones constituyen la base de los cálculos y la lógica de los programas. En esta subsección aprenderá a construir y evaluar expresiones en MQL5.

Conversión de tipos:

- La conversión de tipos de datos es una parte esencial de la programación. En la sección «Conversión de Tipos» se ofrece todo lo necesario para comprender el proceso relacionado con la conversión de datos entre diferentes tipos en MQL5.

Sentencias

- Las sentencias son órdenes que controlan la ejecución del programa. En esta sección veremos varios tipos de sentencias y sus aplicaciones.

Funciones:

- Las funciones permiten estructurar y reutilizar el código. En esta sección se profundiza en los fundamentos de la creación e invocación a funciones en MQL5.

Preprocesador:

- El preprocesador de MQL5 procesa el código fuente antes de la compilación. En la sección «Preprocesador» se describen los principios de uso de las directivas del preprocesador y su repercusión en el código.

Los principios de la programación procedimental servirán de base para el posterior aprendizaje de un paradigma más potente, a saber, la programación orientada a objetos (POO), a la que se hará referencia en la Parte 3.

Programación en MQL5 para Traders: códigos fuente del libro. Parte 2

Los ejemplos del libro también están disponibles en el proyecto público \MQL5\Shared Projects\MQL5Book

2.1 Identificadores

Como vamos a ver en breve, los programas se construyen a partir de múltiples elementos a los que hay que referirse con nombres únicos para evitar confusiones. Estos nombres son justamente lo que se denomina identificadores.

Un identificador es una palabra compuesta por ciertas reglas: en ella sólo pueden utilizarse caracteres latinos, guiones bajos ('_') y dígitos, y el primer carácter no puede ser un dígito. Pueden usarse letras minúsculas y mayúsculas.

La longitud máxima de un identificador es de 63 caracteres. El identificador no puede coincidir con ninguna palabra reservada de MQL5, como los nombres de tipo. Encontrará la lista completa de palabras de servicio en la Ayuda. El incumplimiento de cualquiera de las reglas de formación de identificadores provocará un error de compilación.

He aquí algunos identificadores correctos:

```
i              // single character
abc            // lower-case letters
ABC            // upper-case letters
Abc            // mixed-case letters
_abc           // underscore at the beginning
_a_b_c_        // underscore anywhere
step1          // digit
_1step         // underscore and digit
```

Ya hemos visto en el script *HelloChart* de qué forma se utilizan los identificadores como nombres de variables y funciones.

Se recomienda proporcionar identificadores con nombres significativos en los que sea evidente la finalidad o el contenido del elemento en cuestión. En algunos casos se utilizan identificadores de un solo carácter, de los que hablaremos en la sección dedicada a los bucles.

Existen algunas prácticas comunes para componer identificadores. Por ejemplo, si elegimos un nombre para una variable que almacena el valor del factor de beneficio, las siguientes serán buenas opciones:

```
ProfitFactor   // "camel" style, all words start with a capital letter
profitFactor   // "camel" style, all words but the first one start with a capital let
profit_factor  // "snake" style, the underscore is put between all words
```

En muchos lenguajes de programación se utilizan diferentes estilos para dar nombre a distintas entidades. Por ejemplo, se puede seguir una práctica en la que sólo los nombres de las variables empiecen por una letra minúscula, mientras que los nombres de las clases (véase la Parte 3) empiecen con letras mayúsculas. Esto ayuda al programador a analizar el código fuente cuando trabaja en equipo o si retoma su propio fragmento de código tras una larga pausa.

Además de los anteriores, existen otros estilos, algunos de los cuales se utilizan en casos especiales:

```
profitfactor   // "smooth" style, all letters are lower-case
PROFITFACTOR   // "smooth" style, all letters are upper-case
PROFIT_FACTOR  // "macro" style, all letters are upper-case with underscores between
```

A veces se utilizan sólo mayúsculas en los nombres de constantes.

El estilo «macro» se utiliza de forma convencional en los nombres de descripciones macro de preprocesador.

2.2 Tipos de datos integrados

El tipo de dato es un concepto fundamental que utilizamos cómodamente en nuestra vida cotidiana sin ni siquiera pensar en su existencia. Está implícito en función del significado de la información que intercambiamos y de los procedimientos de tratamiento admisibles para la misma. Por ejemplo, a la hora de realizar un control de nuestros activos domésticos, sumamos y restamos números que representan nuestros ingresos y gastos. Aquí, «número» describe un tipo, para el que nos damos cuenta por completo de los valores que puede tener y de operaciones aritméticas que se pueden hacer con él. En el contexto del trading existe un valor similar, el saldo de la cuenta corriente, en MetaTrader 5; por lo tanto, MQL5 proporciona un mecanismo para crear y manipular números.

A diferencia de los números, la información textual, como el nombre de un instrumento de trading, cumple otras normas. Aquí podemos construir una palabra a partir de letras o una frase a partir de palabras, pero es imposible calcular el total progresivo o la media aritmética de varias líneas. Así, 'línea' o 'cadena' es otro tipo de dato, no un dato numérico.

Además del propósito y un conjunto típico de operaciones significativas para cada tipo, hay otra cosa importante que diferencia a los tipos entre sí: su tamaño. Por ejemplo, el número de la semana no puede exceder de 52 dentro de un año, mientras que el número de segundos transcurridos desde el comienzo del año adopta una forma astronómica. Por lo tanto, para almacenar y procesar eficazmente valores tan diferentes en la memoria del ordenador se pueden seleccionar segmentos de distinto tamaño. Esto nos lleva a comprender que, de hecho, el concepto generalizado de 'número' puede esconder distintos tipos.

MQL5 permite el uso de algunos tipos de números que difieren tanto en los tamaños de las celdas de memoria asignadas a los mismos como en algunas características adicionales. En concreto, algunos números pueden adoptar valores negativos, como el beneficio flotante en pips, mientras que otros no, como los números de cuenta. Además, algunos valores no pueden tener parte fraccionaria y, por tanto, es más rentable representarlos con un tipo más estricto de 'enteros', frente a los de 'números con coma decimal' aleatorios. Por ejemplo, el saldo de una cuenta o el precio de un instrumento de trading suelen tener valores con una coma decimal. Al mismo tiempo, el número de órdenes en el historial o, de nuevo, el número de cuenta es siempre un número entero.

MQL5 admite un conjunto de tipos universales similares a los disponibles en la gran mayoría de lenguajes de programación. El conjunto incluye tipos enteros (de diferentes tamaños), dos tipos de números reales (con coma decimal) de diferente precisión, cadenas y caracteres simples, así como el tipo lógico, que sólo tiene dos valores posibles: *true* y *false*. Además, MQL5 proporciona sus propios tipos específicos que operan con el tiempo y el color.

Por completar la información, es de destacar que MQL5 permite ampliar el conjunto de tipos mediante la declaración de tipos aplicados en el código, es decir, estructuras, clases y otras entidades típicas de la programación orientada a objetos; pero ya los veremos más adelante.

Dado que el tamaño de la celda en la que se almacena el valor es un atributo importante del tipo, vamos a abordar la metodología de la memoria.

La unidad más pequeña de la memoria de un ordenador es el byte. En otras palabras: un byte es el tamaño más pequeño de una celda que un programa puede asignar a un valor independiente. Un byte consta de 8 'partículas' más pequeñas, los bits, cada una de las cuales puede encontrarse en dos estados: activado (1) o desactivado (0). Todos los ordenadores modernos utilizan este tipo de bits en el nivel inferior porque esta representación binaria de la información es práctico incorporarla al hardware (en la memoria de acceso aleatorio, en los procesadores o al transferir los datos por cables de red o vía WiFi).

El procesamiento de valores de distintos tipos está garantizado gracias a las diferentes interpretaciones de los estados de los bits en las celdas de memoria, de lo cual se encarga el compilador. Los programadores no suelen bajar al nivel de los bits; sin embargo, el lenguaje proporciona herramientas para ello (véase Operaciones a nivel de bits).

Hay palabras especiales reservadas en MQL5 para describir los tipos de datos. Ya conocemos algunos de estos tipos, como *void*, *int* y *string*, que vimos en la Parte 1. A continuación se ofrece una lista completa de los tipos, cada uno con una referencia rápida y su tamaño en bytes.

Por su finalidad, pueden dividirse condicionalmente en datos numéricos y datos codificados por caracteres (marcados en las columnas correspondientes), así como en otros tipos especializados, como cadenas, tipos lógicos (o booleanos), fecha/hora y color. El tipo void se sitúa aparte e indica que no existe ningún valor. Además de los tipos escalares, MQL5 proporciona tipos de objetos para operaciones con números complejos, matrices y vectores: complex, vector y matrix. Estos tipos se utilizan para resolver diversos problemas en álgebra lineal, modelización matemática, aprendizaje automático y otras áreas. Los estudiaremos en detalle en la Parte 4 del libro.

Tipo	Tamaño (bytes)	Número	Carácter	Nota
char	1	+	+	Carácter de un byte o un entero con signo
uchar	1	+	+	Carácter de un byte o un entero sin signo
short	2	+	+	Carácter de dos bytes o un entero con signo

Tipo	Tamaño (bytes)	Número	Carácter	Nota
ushort	2	+	+	Carácter de dos bytes o un entero sin signo
int	4	+		Entero con signo
uint	4	+		Entero sin signo
long	8	+		Entero con signo
ulong	8	+		Entero sin signo
float	4	+		Número de punto flotante con signo
double	8	+		Número de punto flotante con signo
enum	4	(int)		Enumeración
datetime	8	(ulong)		Fecha y hora
color	4	(uint)		Color
bool	1	(uchar)		Lógico
string	10+ variable			Cadena
void	0			Vacío
complex	16	+		Estructura con dos campos de tipo doble
vector	longitud del vector x tamaño del tipo	+		Array unidimensional de tipo real o complejo
matrix	filas x columnas x tamaño del tipo	+		Array bidimensional de tipo real o complejo

Dependiendo de su tamaño, en el tipo numérico pueden almacenarse diferentes rangos de valores. Además de lo anterior, el rango puede variar considerablemente para los números enteros y los números de punto flotante del mismo tamaño, ya que para ellos se utilizan diferentes representaciones internas. Todas estas telarañas se estudiarán en las secciones dedicadas a los tipos específicos.

Un programador es libre de elegir un tipo numérico en función de los valores previstos o las consideraciones de eficiencia o por razones de economía. En particular, el menor tamaño del tipo permite que quepan más valores de este tipo en la memoria, mientras que los números enteros se procesan más rápidamente que los de punto flotante.

Tenga en cuenta que los tipos numéricos y los codificados con caracteres se solapan parcialmente. Esto ocurre porque un carácter se almacena en memoria como un número entero, es decir, un código en la tabla de caracteres correspondiente: ANSI para caracteres de un byte o Unicode para los de dos bytes. ANSI es una norma que toma el nombre de un instituto (American National Standards Institute), mientras que Unicode, ya lo ha adivinado, significa (conjunto de caracteres) de Código Universal. Los caracteres Unicode se utilizan en MQL5 para hacer cadenas (tipo *string*). Los caracteres de un solo byte suelen ser necesarios a la hora de integrar los programas con fuentes de datos externas, como las de Internet.

Como ya se ha mencionado, los tipos numéricos pueden dividirse en números enteros y números de punto flotante. Vamos a verlos con más detalle.

2.2.1 Números enteros

Los tipos Números enteros están pensados para almacenar números sin decimales. Deben elegirse si el sentido práctico del valor excluye las fracciones. Por ejemplo, los números de las barras de un gráfico o de posiciones abiertas son siempre números enteros.

MQL5 permite elegir tipos enteros de 1 a 8 bytes de tamaño utilizando las palabras clave *char*, *short*, *int* y *long*, respectivamente. Todos ellos son de tipo con signo, es decir, pueden contener tanto valores positivos como negativos. Si es necesario, los tipos enteros que tengan los mismos tamaños pueden declararse sin signo (sus nombres empiezan por «u» de «unsigned»): *uchar*, *ushort*, *uint* y *ulong*.

En función del tamaño del tipo y de si lleva o no lleva signo, los rangos de valores potenciales son los que se muestran en la siguiente tabla:

Tipo	min	max
char	-128	127
uchar	0	255
short	-32768	32767
ushort	0	65535
int	-2147483648	2147483647
uint	0	4294967295
long	-9223372036854775808	9223372036854775807
ulong	0	18446744073709551615

No es necesario memorizar los valores límite anteriores para cada número entero. Hay muchas constantes con nombre predefinidas en MQL5 que se pueden utilizar en un código en lugar de números «mágicos», incluyendo los números enteros más bajos o más altos. Esta tecnología se aborda en una sección dedicada al preprocesador. Aquí nos limitamos a enumerar las constantes con nombre relevantes: CHAR_MIN, CHAR_MAX, UCHAR_MAX, SHORT_MIN, SHORT_MAX,USHORT_MAX, INT_MIN, INT_MAX, UINT_MAX, LONG_MIN, LONG_MAX y ULONG_MAX.

Vamos a explicar cómo se obtienen estos valores, para lo cual es necesario volver a bits y bytes.

El número de todas las combinaciones posibles de diferentes estados de 8 bits, activados y desactivados, dentro de un byte, es 256. Esto produce el rango de valores 0-255 que puede almacenarse en un byte. Sin embargo, su interpretación depende del tipo para el que se haya asignado este byte. El compilador garantiza diferentes interpretaciones en función de las sentencias del programador.

El bit de orden inferior (más a la derecha) de un byte significa 1, el segundo 2, el tercero 4, y así sucesivamente hasta el bit de orden superior, que significa 128. Es evidente que estos números son iguales a dos elevado a una potencia igual al número de bits (la numeración empieza por 0). Este es el efecto de utilizar el sistema binario.

Bits	**orden superior**							**orden inferior**
Número	7	6	5	4	3	2	1	0
Valor	128	64	32	16	8	4	2	1

Cuando todos los bits están activados, se obtiene la suma de todas las potencias de dos, es decir, 255 es el valor más alto para un byte. Si se reinician todos los bits, obtenemos cero. Si se activa un bit de orden inferior, el número es impar.

En la codificación de números con signo, el bit de orden superior se utiliza para marcar valores negativos. Por lo tanto, para un entero de un solo byte dentro del rango positivo, 127 se convierte en el valor más alto. Para los valores negativos hay 128 combinaciones posibles, es decir, el valor más bajo es -128. Cuando todos los bits de un byte están activados, ello se interpreta como -1. Si en un número de este tipo se reinicia el bit de orden inferior, obtendremos -2, etc. Si sólo se activa el bit de orden superior (signo) y todos los demás bits se reinician, obtendremos -128.

Esta codificación, que puede parecer irracional, se llama «adicional» y le permite unificar los cálculos de números con y sin signo a nivel del hardware. Además, le permite no perder un valor, lo que ocurriría si las regiones positivas y negativas se codificaran de forma idéntica: y es que así habríamos obtenido dos valores para el cero, es decir, un 0 positivo y un 0 negativo. Es más, esto provocaría ambigüedad.

Los números con más bytes, es decir, 2, 4 u 8, tienen una numeración consecutiva similar de bits y la progresión de sus respectivos valores. En todos los casos, un criterio para la negatividad del número es el bit de orden superior activado del byte de orden superior.

Así, podemos utilizar un byte para almacenar un entero sin signo (*uchar*, es decir, carácter sin signo abreviado) dentro del rango 0-255. También podemos escribir un entero con signo en el byte (para lo cual describiremos su tipo como *char*). En este caso, el compilador dividirá la cantidad disponible de combinaciones de 256 a partes iguales entre valores positivos y negativos, habiéndola desplegado sobre la región de -128 a 127 (el valor 256 es cero). Es evidente que los valores 0-127 se codificarán por igual a nivel de bits para bytes con y sin signo. Sin embargo, los valores absolutos grandes, a partir de 128, se convertirán en negativos (según el esquema descrito arriba). Esta «transformación» sólo tiene lugar en el momento de la lectura o de la realización de cualquier operación con el valor almacenado, con idéntica representación interna de los datos (estado de los bits).

Examinaremos esta cuestión con más detalle en la sección dedicada a la conversión de tipos.

Del mismo modo que con los enteros de un byte, es fácil calcular que el número de combinaciones de bits es de 65536 para 2 bytes. Por lo tanto, los rangos se forman para el entero de dos bytes con signo y sin signo, *short* y *ushort*. Otros tipos permiten almacenar valores aún mayores gracias al aumento de su tamaño en bytes.

Tenga en cuenta que el uso de un tipo sin signo con el mismo tamaño permite duplicar el valor positivo más alto. Esto puede ser necesario para almacenar cantidades potencialmente muy grandes para las que no pueden aparecer valores negativos. Por ejemplo, el número de orden en MetaTrader 5 es un valor del tipo *ulong*.

Ya hemos visto muestras de la descripción de números enteros en la Parte 1. En concreto, ahí se definió el parámetro de entrada *GreetingHour* del tipo *uint*:

```
input uint GreetingHour = 0;
```

A excepción de la palabra clave adicional, input, que hace que la variable sea visible en la lista de parámetros de un programa MQL, los demás componentes, es decir, el tipo, el nombre y la inicialización opcional tras el signo '=', son intrínsecos a todas las variables.

La sintaxis de la descripción de variables se estudiará en detalle en el apartado Variables. Por ahora, tenga en cuenta el método de registro de las constantes de tipo entero. Al describir una variable se pueden especificar constantes como valor por defecto (en el ejemplo anterior es 0). Además, las constantes pueden utilizarse en expresiones; por ejemplo, en un evento de fórmula.

Conviene recordar que las constantes de cualquier tipo, insertadas en el código fuente, se denominan literales (textualmente, «palabra por palabra»). Su nombre se debe a que se introducen en el programa «tal cual» y se utilizan inmediatamente en el punto de descripción. Los literales, a diferencia de muchos otros elementos del lenguaje, en particular las variables, no tienen nombre y no se puede hacer referencia a ellos desde otros puntos del programa.

Para los números negativos es obligatorio poner el signo menos '-' delante del número; sin embargo, el signo más '+' puede omitirse para los números positivos, es decir, las formas '+100' y '100' son idénticas.

Hay que tener en cuenta que los valores numéricos suelen registrarse en el código fuente con nuestra notación decimal habitual. No obstante, MQL5 permite utilizar la otra, es decir, la hexadecimal. Ello es conveniente para procesar información a nivel de bits (véase Operaciones a nivel de bits).

En las constantes decimales se permiten los números del 0 al 9 en números con cualquier orden de dígitos, mientras que para las hexadecimales, además de los dígitos, se utilizan también los símbolos latinos de *A* a *F* o de *a* a *f* (es decir, no afecta que sean mayúsculas o minúsculas). El «dígito hexadecimal» *A* se corresponde con el número 10 de la notación decimal, *B* con 11, *C*con 12, etc., y así hasta *F*, que es igual a 15.

Una característica distintiva de una constante hexadecimal es el hecho de que comienza con el prefijo 0x o 0X, seguido de los órdenes de dígitos significativos del número. Por ejemplo, el número 1 se registra como 0x1 en el sistema hexadecimal, mientras que el 16 como 0x10 (se requiere un dígito de orden superior adicional porque 16 es mayor que 15, es decir, 0xF). El 255 decimal se convierte en 0xFF.

Veamos algunos ejemplos más que ilustran diversas situaciones del uso de tipos enteros en la descripción de variables (adjuntos en el script *MQL5/Scripts/MQL5Book/p2/TypeInt.mq5*):

```
void OnStart()
{
  int x = -10;          // ok, signed integer x = -10
  uint y = -1;          // ok, but unsigned integer y = 4294967295
  int z = 1.23;         // warning: truncation of constant value, z = 1
  short h = 0x1000;     // ok, h = 4096 in decimal
  long p = 10000000000; // ok
  int w = 10000000000;  // warning, truncation..., w = 1410065408
}
```

La variable *x* se inicializa correctamente con el valor negativo permitido, -10.

La variable *y* no lleva signo. Por lo tanto, si se intenta registrar en él un valor negativo, se produce un efecto interesante. El número -1 tiene una representación en bits, que es interpretada por el programa de acuerdo con el tipo sin signo, *uint*. Por lo tanto, se obtiene el número 4294967295 (en realidad es igual a UINT_MAX).

La variable *z* se asigna con el número de punto flotante 1,23 (se analizarán en la siguiente sección) y el compilador advierte sobre el truncamiento de la parte fraccionaria. Como resultado, el entero 1 entra en la variable.

La variable *h* se inicializa correctamente mediante una constante en forma hexadecimal (0x1000 = 4096).

El valor grande 10000000000 se registra en las variables *p* y *w*, la primera de las cuales es de tipo entero largo (*long*) y se procesa correctamente, mientras que la segunda es de tipo normal (*int*) y, por tanto, llama la atención del compilador. Dado que la constante supera el valor máximo de *int*, el compilador trunca el exceso de dígitos de orden superior (bits) y, de hecho, 1410065408 se convierte en *w*.

Este comportamiento es una de las posibles evoluciones negativas de las conversiones de tipos que el programador puede o no suponer. En este último caso, ello conlleva un error potencial. Es evidente que, en este ejemplo concreto, se han seleccionado valores erróneos de forma intencionada para hacer una demostración de las advertencias. En un programa real no siempre es tan obvio qué valores está intentando guardar el programa en la variable entera. Por lo tanto, debe examinar detenidamente las advertencias del compilador y tratar de eliminarlas, una vez cambiado el tipo o especificado explícitamente la conversión de tipos requerida. Esta cuestión se abordará en la sección dedicada a la conversión de tipos.

Para los tipos enteros se definen operaciones aritméticas, a nivel de bits y de otros tipos (véase el capítulo Expresiones).

2.2.2 Números de punto flotante

En la vida cotidiana utilizamos números con coma (o punto) decimal, o números reales, con la misma frecuencia que los enteros. El propio nombre, «real», indica que, con dichos números, se puede expresar algo tangible del mundo real, como el peso, la longitud o la temperatura del cuerpo; es decir, todo aquello que se puede medir con una cantidad de unidades no entera, sino con «un poco más».

En trading también utilizamos a menudo los números reales. Por ejemplo, estos números se utilizan para expresar volúmenes o precios de símbolos en órdenes de trading (normalmente permiten las partes fraccionarias de un lote de tamaño completo).

En MQL5 se ofrecen 2 tipos reales: *float* para una precisión normal y *double* para una precisión doble.

En el código fuente, los valores constantes de los tipos *float* y *double* suelen registrarse como un número entero y una parte fraccionaria (cada uno de ellos es una secuencia de dígitos), separados por el carácter '.', como 1.23 o -789.01. Puede que no haya entero ni fracción (pero no ambos a la vez), pero el punto es obligatorio. Por ejemplo, .123 significa 0,123, mientras que 123. significa 123,0. 123 solo creará una constante de tipo entero.

No obstante, existe otra forma de registrar las constantes reales: la exponencial. En ella, la parte entera y la fraccionaria van seguidas de 'E' o 'e' (es indiferente que sea mayúscula o minúscula) y de un número entero que representa la potencia a la que hay 10 debe elevarse para obtener un factor adicional. Por ejemplo, las siguientes representaciones muestran el mismo número, 0.57, de forma exponencial:

```
  .0057e2
 0.057e1
 .057e1
 57e-2
```

Cuando se registran constantes reales, estas últimas se definen de forma predeterminada como tipo *double* (consumen 8 bytes). Para establecer el tipo *float* debe añadirse el sufijo 'F' (o 'f') a la constante de la derecha.

Los tipos *float* y *double* se diferencian por sus tamaños, rangos de valores y precisión de representación numérica. Todo ello se muestra en el cuadro siguiente.

Tipo	Tamaño (bytes)	Mínimo	Máximo	Precisión (órdene
float	4	$\pm 1.18 * 10^{-38}$	$\pm 3.4 * 10^{38}$	6-9, normalmente 7
double	8	$\pm 2.23 * 10^{-308}$	$\pm 1.80 * 10^{308}$	15-18, normalment

El rango de valores se muestra para ellos en términos absolutos: el mínimo y el máximo determinan la amplitud de los valores permitidos en las regiones positivas y negativas. De forma similar a los tipos enteros, existen constantes con nombre integradas para estos valores límite: FLT_MIN, FLT_MAX, DBL_MIN, DBL_MAX.

Tenga en cuenta que los números reales siempre tienen signo, es decir, no existen análogos sin signo para ellos.

Por precisión se entenderá la cantidad de dígitos significativos (dígitos decimales) que el número real del tipo correspondiente es capaz de almacenar sin distorsión.

De hecho, los números de los tipos reales no son tan precisos como los de los tipos enteros. Este es el precio que hay que pagar por su universalidad y una gama mucho más amplia de valores potenciales. Por ejemplo, si un entero de 4 bytes sin signo (*uint*) tiene el valor más alto de 4294967295, es decir, unos 4 millones, o $4.29*10^9$, entonces el real de 4 bytes (*float*) tiene $3.4 * 10^{38}$, que es 29 órdenes de magnitud superior. Para los tipos de 8 bytes, la diferencia es aún más perceptible: *ulong* puede guardar 18446744073709551615 ($18.44*10^{18}$, o ~18 quintillones), mientras que *double* puede guardar $1.80 * 10^{308}$, es decir, 289 órdenes de magnitud más. La inserción proporciona más detalles sobre la precisión.

Mantisa y exponente

La representación interna de los números reales en la memoria (en los bytes asignados a ellos) es bastante complicada. El bit de orden superior se utiliza como marcador del signo negativo (algo que también hemos visto en los tipos enteros). Los demás bits se dividen en dos grupos. El mayor contiene la mantisa del número, es decir, los dígitos significativos (nos referimos a dígitos binarios, es decir, bits). El menor guarda la potencia (exponente) a la que hay que elevar 10 para obtener el número almacenado al multiplicarlo por la mantisa. En concreto, para el tipo *float*, la mantisa tiene un tamaño de 24 bits (FLT_MANT_DIG), mientras que para *double* es de 53 (DBL_MANT_DIG). En términos de decimales convencionales (dígitos), obtendremos la misma precisión que se ha mostrado en la tabla anterior: 6 (FLT_DIG) es la menor cantidad de dígitos significativos para *float*, mientras que 15 (DBL_DIG) lo es para *double*. Sin embargo, dependiendo del número de que se trate, puede tener combinaciones «afortunadas» de bits, en correspondencia con una mayor cantidad de dígitos decimales. Los tamaños de los parámetros son de 8 y 11 bits para *float* y *double*, respectivamente.

Debido al exponente, los números reales tienen un rango de valores mucho mayor. Al mismo tiempo, con el aumento del exponente, también aumenta el «peso específico» del dígito de orden inferior de la mantisa. Esto significa que dos números reales vecinos que pueden representarse en la memoria del ordenador son sustancialmente diferentes. Por ejemplo, para el número 1.0, el «peso específico» del bit de orden inferior es 1.192092896e-07 (FLT_EPSILON) en el caso de *float*, y 2.2204460492503131e-016 (DBL_EPSILON) en el caso de *double*. En otras palabras: 1.0 es indistinguible de cualquier número cercano a él si dicho número es inferior a 1.192092896e-07. Esto puede parecer poco importante o que «no es para tanto», pero esta región de incertidumbre se agranda para los números más grandes. Si guarda en *float* un número de unos 1000 millones ($1*10^9$), los 2 últimos dígitos dejarán de almacenarse o restaurarse de forma segura desde la memoria (véase más abajo el ejemplo de código). No obstante, básicamente, el problema no es el valor absoluto de un número, sino la cantidad máxima de dígitos que contiene, que deben recuperarse sin pérdidas. Igualmente «bien» podemos intentar encajar un número representado como 1234.56789 (que estructuralmente se parece mucho al precio de un instrumento financiero) en *float*; y sus dos últimos dígitos «flotarán» debido a la falta de precisión de su representación interna.

En el caso de *double* se empezará a ver una situación similar para números mucho mayores (o para una cantidad mucho mayor de dígitos significativos), pero sigue siendo posible y ocurre a menudo en la práctica. Debe tener esto en cuenta cuando trabaje con números reales muy grandes o muy pequeños, y escribir sus programas con comprobaciones adicionales para evitar posibles pérdidas de precisión. En especial, debe comparar un número real con cero de una manera especial. Abordaremos esta cuestión en la sección sobre operadores de comparación.

A un lector atento le puede parecer que los tamaños de la mantisa y el exponente anteriores están mal especificados. Vamos a explicarlo con el ejemplo de *float*. Se almacena en la celda de memoria de 4 bytes de tamaño; es decir, consume 32 bits. Al mismo tiempo, los tamaños de la mantisa (24) y el exponente (8) suman ya 32. Entonces, ¿dónde está el bit con signo? La cuestión es que los profesionales informáticos han dispuesto almacenar la mantisa de forma «normalizada». Será más fácil entender en qué consiste eso si tenemos en cuenta en primer lugar la forma exponencial de registrar un número decimal normal. Digamos que el número 123.0 podría representarse como 1.23E2, 12.3E1, o 0.123E3. Se considera que una designación es la forma normalizada, en la que se coloca solo un dígito significativo (es decir, no cero) antes del punto. Para este número se trata de 1.23E2. Por definición, los dígitos del 1 al 9 se consideran dígitos significativos en notación decimal. Pasamos ahora sin dificultades a la notación binaria. Aquí sólo hay un dígito significativo, el 1. Resulta que la forma normalizada en notación binaria siempre empieza por 1, y se puede omitir (para no gastar memoria). De este modo, se puede ahorrar un bit en la mantisa. De hecho, contiene 23 bits (una unidad de orden superior más está implícita y se añade automáticamente al

reconstruir el número y recuperarlo de la memoria). Reducir la mantisa en 1 bit deja espacio para el bit con signo.

Predominantemente, allí donde se debe utilizar el tipo de punto flotante, elegimos *double* como más preciso. El tipo *float* sólo se utiliza para ahorrar memoria, como cuando se trabaja con arrays de datos muy grandes.

Algunos ejemplos de utilización de las constantes de tipos reales se muestran en el script *MQL5/Scripts/MQL5Book/p2/TypeFloat.mq5*.

```
void OnStart()
{
  double a0 = 123;       // ok, a0 = 123.0
  double a1 = 123.0;     // ok, a1 = 123.0
  double a2 = 0.123E3;   // ok, a2 = 123.0
  double a3 = 12300E-2;  // ok, a3 = 123.0
  double b = -.75;       // ok, b = -0.75
  double q = LONG_MAX;   // warning: truncation, q = 9.223372036854776e+18
                         //               LONG_MAX = 9223372036854775807
  double d = 9007199254740992; // ok, maximal stable long in double
  double z = 0.12345678901234567890123456789; // ok, but truncated
                           // to 16 digits: z = 0.1234567890123457
  double y1 = 1234.56789;  // ok, y1 = 1234.56789
  double y2 = 1234.56789f; // accuracy loss, y2 = 1234.56787109375
  float m = 1000000000.0;  // ok, stored as is
  float n =  999999975.0;  // warning: truncation, n = 1000000000.0
}
```

Las variables *a0*, *a1*, *a2* y *a3* contienen los mismos números (123.0) escritos con métodos diferentes.

En la constante de la variable *b*, el cero insignificante se omite delante del punto. Además, aquí está la demostración de cómo registrar un número negativo utilizando el signo menos, '-'.

Se intenta almacenar el mayor número entero en la variable *q*. En este lugar, el compilador emite una advertencia, porque *double* no puede representar LONG_MAX con precisión: en lugar de 9223372036854775807, habrá 9223372036854776000. Ello obviamente demuestra que, aunque los rangos de los valores de *double* superan ampliamente a los de los enteros, se logra debido a la pérdida de los dígitos de orden inferior.

A modo de comparación, el número entero máximo que el tipo *double* es capaz de guardar sin distorsiones se da como valor de la variable *d*. En la secuencia de enteros irá seguido de saltos esporádicos, si utilizamos *double* para ellos.

La variable *z* nos recuerda de nuevo la limitación en la cantidad máxima de dígitos significativos (16): una constante más larga se truncará.

Las variables *y1* y *y2*, en las que el mismo número se registra en formatos diferentes (*double* y *float*), permiten ver la pérdida de precisión debida a la transición a *float*.

De hecho, las variables *m* y *n* serán iguales, porque 999999975.0 se almacena de forma aproximada en la representación interna y se convierte en 1000000000.0.

Los tipos numéricos se suelen utilizar para calcular mediante fórmulas; para ellos se define un amplio conjunto de operaciones (véase Expresiones).

A veces, los cálculos pueden conducir a resultados incorrectos, es decir, que no pueden representarse como un número. Por ejemplo, no se puede definir la raíz de un número negativo o el logaritmo de cero. En estos casos, los tipos reales pueden guardar un valor especial denominado NaN (Not A Number, o No es un Número). De hecho, existen varios tipos de valores de este tipo que permiten, por ejemplo, diferenciar entre más infinito y menos infinito. MQL5 proporciona una función especial, *MathIsValidNumber*, que comprueba si el valor *double* es un número o uno de los valores NaN.

2.2.3 Tipos de caracteres

Los tipos de datos de caracteres están pensados para almacenar determinados caracteres (letras) con los que se forman cadenas (véase Cadenas). MQL5 tiene 4 tipos de caracteres: Dos de 1 byte (*char*, *uchar*) de tamaño y dos de 2 bytes (*short*, *ushort*) de tamaño. Los tipos prefijados con 'u' no llevan signo.

De hecho, los tipos de caracteres son enteros, ya que almacenan un código entero de un carácter de la tabla correspondiente: para *char* se trata de la tabla de caracteres ASCII (códigos 0-127); para *uchar*, de ASCII extendido (códigos 0-255); y para *short/ushort*, de la tabla Unicode (hasta 65535 caracteres en la versión sin signo). Por si es de su interés, ASCII es la abreviatura de American Standard Code for Information Interchange.

Para las cadenas MQL5 se utilizan caracteres *ushort* de 2 bytes. Los tipos *uchar* de 1 byte se utilizan normalmente para integrarse con programas externos al transferir los arrays de datos aleatorios que se empaquetan y desempaquetan en otros tipos según los protocolos aplicados, como por ejemplo para conectarse a una plataforma criptográfica.

Las constantes de caracteres se registran como letras encerradas entre comillas simples. No obstante, también puede utilizar la notación entera (véase Enteros) abordada anteriormente. Al mismo tiempo, el número entero debe estar dentro del rango de valores para el formato de 1 o 2 bytes.

Además, podemos utilizar la notación de secuencias de escape, que utilizan una barra invertida ('\') como primer carácter seguido de uno de los caracteres de control predefinidos y/o un código numérico. MQL5 admite las siguientes secuencias de escape:

- \n - nueva línea
- \r - retorno de carro
- \t - tabulación
- \\ - barra invertida
- \" - cita doble
- \' - comilla simple
- \X o \x - prefijo para especificar posteriormente un código numérico en formato hexadecimal
- \0 - prefijo para especificar posteriormente un código numérico en formato octal

Los métodos básicos de utilización de las constantes de los tipos de caracteres se indican en el script *MQL5/Scripts/MQL5Book/p2/TypeChar.mq5*.

```
void OnStart()
{
  char a1 = 'a';  // ok, a1 = 97, English letter 'a' code
  char a2 = 97;   // ok, a2 = 'a' as well
  char b = '£';   // warning: truncation of constant value, b = -93
  uchar c = '£';  // ok, c = 163, pound symbol code
  short d = '£';  // ok
  short z = '\0';    // ok, 0
  short t = '\t';    // ok, 9
  short s1 = '\x5c'; // ok, backslash code 92
  short s2 = '\\';   // ok, backslash as is, code 92 as well
  short s3 = '\0134';// ok, backslash code in octal form
}
```

Las variables *a1* y *a2* obtienen el valor del carácter 'a' (letra inglesa) de dos formas diferentes.

Se ha intentado registrar '£' en la variable *b*, pero su código, 163, está fuera del rango *char* (127); por lo tanto se «transforma» en -93 con signo (el compilador emite un aviso). Las variables de tipo *uchar* (c) y *short* (d) que le siguen perciben este código como normal.

Otras variables se inicializan utilizando secuencias de escape.

Los caracteres pueden procesarse con las mismas operaciones que los enteros (véase Expresiones).

2.2.4 Tipo string

El tipo string está pensado para almacenar información basada en texto y está marcado por la palabra clave *string*. Una cadena, o string, es una secuencia de caracteres *ushort* y es compatible con toda la gama Unicode, lo que incluye múltiples scripts nacionales. Por ejemplo, los nombres de los instrumentos financieros y los comentarios de las órdenes de trading son cadenas.

Por la naturaleza específica de las cadenas, su tamaño es un valor variable igual a la longitud duplicada del texto (cantidad de caracteres multiplicada por la «anchura» de un carácter, es decir, 2 bytes) más un carácter más. Este carácter adicional está pensado para el «cero final» (un carácter codificado como 0) que indica el final de la línea. Además, MQL5 utiliza algo de espacio para almacenar la información de servicio, es decir, una referencia al lugar de la memoria en el que comienza la cadena.

> A diferencia de C++, en MQL5 no se puede obtener la dirección de una cadena ni ninguna otra variable. El acceso directo a la memoria está prohibido en MQL5.

Una cadena literal se registra en el código fuente como una secuencia de caracteres entre comillas dobles. Por ejemplo, "EURUSD" o "$". Debemos distinguir entre cadenas formadas por un solo carácter, como "$", y los mismos caracteres por separado, como '$'. Se trata de tipos de datos diferentes.

Una cadena vacía aparece como "". Teniendo en cuenta el cero final implícito, consume 2 bytes, aparte de la información de servicio.

Si es necesario utilizar el carácter de comillas dobles dentro de la cadena, este debe ir precedido del carácter de barra invertida, transformándose en una secuencia de control, como "Pulse \"OK\"".

En el script *MQL5/Scripts/MQL5Book/p2/TypeString.mq5* se ofrecen ejemplos de inicialización de cadenas.

```
void OnStart()
{
   string h = "Hello";          // Hello
   string b = "Press \"OK\"";   // Press "OK"
   string z = "";               //
   string t = "New\nLine";      // New
                                // Line
   string n = "123";            // 123, text (not an integer value)
   string m = "very long message "
              "can be presented "
              "by parts";
   // equivalent:
   // string m = "very long message can be presented by parts";
}
```

La cadena "Hola" se coloca en la variable *h*.

El texto que contiene comillas dobles se escribe en la variable *b*.

La variable *z* se inicializa con una cadena vacía. Esto es básicamente equivalente a describir *z* sin inicialización, pero aquí hay que hilar más fino. Más adelante, según avancemos en el texto, en la sección de Inicialización de variables descubriremos que las cadenas no inicializadas obtienen un valor especial, NULL, a diferencia de "", para el que, como ya se ha dicho, la memoria se asigna para el cero de terminación. Esta diferencia afecta a la ejecución de la cadena operadores de comparación y algunas otras. A medida que avancemos iremos abordando todos estos aspectos.

La variable *t* obtendrá un texto que, cuando se imprima en el registro utilizando la función *Print* o se muestre por otros métodos, se dividirá en 2 cadenas.

La cadena "123" registrada en la variable *n* no es un número, aunque lo parezca. Existen algunas funciones en MQL5 para convertir texto en números y viceversa (véase la sección Transformación de datos). Además, existe un conjunto independiente de funciones para trabajar con cadenas.

Por comodidad, los literales largos pueden escribirse en varias cadenas, como en el caso de la variable *m*. La regla general es la siguiente: el compilador fusiona todos los literales hasta el punto y coma que marca el final de la descripción de la variable. En este tipo de aplicación de formato, la clave es no olvidar añadir un espacio intermedio dentro de cada fragmento de la cadena, si ello es necesario (por ejemplo, para separar las palabras del mensaje como en el ejemplo anterior).

Para las cadenas se define la operación de suma (concatenación), denotada por el carácter '+'. Hablaremos de ello en el capítulo dedicado a las expresiones (véase Operaciones aritméticas).

Los caracteres de la cadena pueden leerse por separado, refiriéndose a ellos como elementos de array (véase Utilización de arrays): si *s* es una cadena, entonces *s[i]* es el código del carácter i-ésimo de la misma, escriba *ushort*.

2.2.5 Tipo lógico (booleano)

El tipo lógico está pensado para almacenar características que sólo tienen 2 estados posibles: «activado» o «desactivado». Sus análogos de interfaz son opciones en diálogos de configuración de muchos programas, incluido MetaTrader 5: cada indicador puede estar activado o desactivado. La comprobación de los estados de tales características permite dividir la lógica de la ejecución del programa, de ahí el nombre de este tipo.

El tipo de lógica se define en MQL5 con la palabra clave *bool* y consume 1 byte de memoria. Para este tipo, se reservan dos constantes: *true* y *false*. Además, se admiten situaciones (y los programadores suelen aprovecharlas) en las que *bool* es el resultado de cálculos con números enteros y reales, interpretándose el valor 0 como *false* y cualquier otro como *true*.

También se admite la interpretación inversa del valor del tipo *bool* como un número: *true* se considera 1 y *false*, 0.

En el archivo *MQL5/Scripts/MQL5Book/p2/TypeBool.mq5* se ofrecen ejemplos de variables de tipo lógico.

```
void OnStart()
{
  bool t = true;          // true
  bool f = false;         // false
  bool x = 100;           // x = true
  bool y = 0;             // y = false
  int i = true;           // i = 1
  int j = false;          // j = 0
}
```

Para el tipo lógico se proporciona un conjunto de operaciones lógicas especiales (véase Operaciones lógicas (booleanas) y Operaciones de comparación).

2.2.6 Fecha y hora

MQL5 proporciona un tipo especial para almacenar datos de tiempo *datetime*. Como se deduce de su nombre, los valores de *datetime* incluyen tanto la fecha como la hora. No obstante, en caso necesario, pueden contener sólo la fecha o sólo la hora del día.

Los valores de este tipo se pueden utilizar en programas para supervisar eventos, como horas de trading, publicaciones de noticias o tiempos de espera para desactivar temporalmente la operación de trading del EA tras malas transacciones.

El tamaño de *datetime* en memoria es de 8 bytes. La representación interna de los datos es idéntica por completo a la del tipo *ulong*, ya que en su interior se almacena la cantidad de segundos que han transcurrido desde el 1 de enero de 1970. La fecha máxima admitida es 31 de diciembre de 3000.

Las constantes *datetime* se registran como una cadena literal encerrada entre comillas simples, precedida del carácter 'D'. Dentro de la cadena se asignan 6 campos, y los números de todos los elementos de fecha y hora tienen los siguientes formatos:

```
D'YYYY.MM.DD HH:mm:ss'
D'DD.MM.YYYY HH:mm:ss'
```

Aquí, YYYY indica el año; MM, el mes; DD, el día; HH, la hora; mm, los minutos, y ss, los segundos. Puede omitir la fecha o la hora. También es posible no especificar segundos o minutos con segundos.

Para el valor máximo permitido para la fecha se proporciona en MQL5 una constante especial, DATETIME_MAX, igual al valor entero 0x793406fff, que se corresponde con D"3000.12.31 23:59:59".

En el archivo *MQL5/Scripts/MQL5Book/p2/TypeDateTime.mq5* se muestran ejemplos de registro de los valores del tipo *datetime* .

```
void OnStart()
{
  // WARNINGS: invalid date
  datetime blank = D'';           // blank = day of compilation
  datetime noday = D'15:45:00';   // noday = day of compilation + 15:45
  datetime feb30 = D'2021.02.30'; // feb30 = 2021.03.02 00:00:00
  datetime mon22 = D'2021.22.01'; // mon22 = 2022.10.01 00:00:00
  // OK
  datetime dt0 = 0;                       // 1970.01.01 00:00:00
  datetime all = D'2021.01.01 10:10:30'; // 2021.01.01 10:10:30
  datetime day = D'2025.12.12 12';       // 2025.12.12 12:00:00
}
```

Las cuatro primeras variables llaman la atención del compilador sobre la fecha incorrecta. En el caso de *blank*, el literal está completamente vacío. En la variable *noday* no hay día. En ambos casos, el compilador sustituye la fecha de compilación en la constante. Las variables *feb30* y *mon22* contienen números incorrectos de día y mes. El compilador los corrige automáticamente, transfiriendo el exceso al campo de orden superior (el 30 de febrero se convierte en 2 de marzo, mientras que el mes 22 pasa a ser el mes 10 del año siguiente). Sin embargo, siempre es recomendable deshacerse de los avisos.

La variable *dt0* demuestra la inicialización del valor *datetime* con un número entero.

El tipo *datetime* admite el conjunto de operaciones inherentes a los números enteros (véase Expresiones). Esto permite, por ejemplo, añadir una cantidad predefinida de segundos a la hora (con lo que se obtiene un momento en el futuro) o calcular la diferencia entre fechas.

2.2.7 Color

MQL5 tiene un tipo especial para trabajar con el color, lo que permite colorear objetos gráficos.

Para indicar el tipo se utiliza la palabra clave *color*. Para el valor del tipo *color* se asignan 4 bytes de memoria. Su representación interna es un entero sin signo que contiene un color en el formato RGB (rojo, verde, azul, por sus siglas en inglés), es decir, con niveles de intensidad independientes para los colores rojo, verde y azul. La mezcla de estos tres componentes permite obtener cualquier tono de color visible. El verde y el rojo producirán amarillo, el rojo y el azul darán morado, etc.

Se asigna 1 byte a cada componente, es decir, los valores pueden ir del 0 al 255. Por ejemplo, tres ceros en todos los componentes producen el color negro, mientras que tres valores máximos de 255 se mezclan para formar blanco.

Si presentamos *color* como *uint* en notación hexadecimal, entonces los colores se distribuyen de la siguiente manera: 0x00BBGGRR, donde RR (rojo), GG (verde) y BB (azul) son enteros de un byte sin signo.

Para comodidad del usuario, MQL5 admite una forma especial de literales para registrar constantes de color. El literal representa un triplete de números separados por comas y encerrados entre comillas simples. El carácter 'C' se coloca antes del literal. Por ejemplo, C'0,128,255' significa un color con 0 para su componente rojo, 128 para el verde y 255 para el azul. También puede utilizarse la notación hexadecimal de los números: C'0x00,0x80,0xFF'.

Además, MQL5 incorpora una larga lista de tonos de color predefinidos, en la que todos empiezan por *clr*. Por ejemplo, clrMagenta, clrLightCyan y clrYellow. También se incluyen los primarios, por supuesto: clrRed, clrGreen, and clrBlue. La lista completa se encuentra en la Ayuda de MetaEditor.

A continuación se muestran algunos ejemplos de colores (también disponibles en el archivo *MQL5/Scripts/MQL5Book/p2/TypeColor.mq5*):

```
void OnStart()
{
  color y = clrYellow;          // clrYellow
  color m = C'255,0,255';       // clrFuchsia
  color x = C'0x88,0x55,0x01'; // x = 136,85,1 (no such predefined color)
  color n = 0x808080;           // clrGray
}
```

2.2.8 Enumeraciones

Las enumeraciones son un grupo de tipos integrados en MQL5, cada uno de los cuales contiene un conjunto de constantes con nombre que sirve para describir conceptos o propiedades relacionadas. Estas constantes también se denominan elementos de enumeración.

Por ejemplo, la enumeración ENUM_DAY_OF_WEEK contiene constantes para todos los días de la semana:

Identificador (ID)	Descripción	Valor
SUNDAY	Domingo	0
MONDAY	Lunes	1
TUESDAY	Martes	2
WEDNESDAY	Miércoles	3
THURSDAY	Jueves	4
FRIDAY	Viernes	5
SATURDAY	Sábado	6

La enumeración ENUM_ORDER_TYPE describe todos los tipos de órdenes admitidos en MetaTrader 5:

Identificador (ID)	Descripción	Valor
ORDER_TYPE_BUY	Orden de mercado para la compra	0
ORDER_TYPE_SELL	Orden de mercado para la venta	1
ORDER_TYPE_BUY_LIMIT	Orden pendiente Buy Limit	2
ORDER_TYPE_SELL_LIMIT	Orden pendiente Sell Limit	3
ORDER_TYPE_BUY_STOP	Orden pendiente Buy Stop	4
ORDER_TYPE_SELL_STOP	Orden pendiente Sell Stop	5
ORDER_TYPE_BUY_STOP_LIMIT	Al alcanzar el precio de la orden, la orden pendiente Buy Limit se coloca al precio StopLimit	6
ORDER_TYPE_SELL_STOP_LIMIT	Al alcanzar el precio de la orden, la orden pendiente Sell Limit se coloca al precio StopLimit	7
ORDER_TYPE_CLOSE_BY	Orden de cierre de una posición con la opuesta	8

Hay varias docenas de enumeraciones distintas, cuyos nombres llevan el prefijo «ENUM_». Las iremos descubriendo a medida que avancemos por las áreas correspondientes.

Cada enumeración es un tipo independiente. Sin embargo, su representación interna es idéntica, es decir, un número entero de cuatro bytes (*int*). Cada constante de enumeración se codifica con un número u otro, pero en la mayoría de los casos, el programador no necesita recordar estos números, ya que el la enumeración se utiliza precisamente para sustituir las representaciones internas por identificadores evidentes.

El compilador se asegura de que el valor de enumeración sea siempre una de las constantes redefinidas. De lo contrario, aparecerá un aviso o se producirá un error de compilación (contextualmente, véase el ejemplo).

Así es como aparece «debajo» la enumeración ENUM_DAY_OF_WEEK (script *MQL5/Scripts/MQL5Book/p2/TypeEnum.mq5*).

```
void OnStart()
{
  ENUM_DAY_OF_WEEK sun = SUNDAY;     // sun = 0
  ENUM_DAY_OF_WEEK mon = MONDAY;     // mon = 1
  ENUM_DAY_OF_WEEK tue = TUESDAY;    // tue = 2
  ENUM_DAY_OF_WEEK wed = WEDNESDAY;  // wed = 3
  ENUM_DAY_OF_WEEK thu = THURSDAY;   // thu = 4
  ENUM_DAY_OF_WEEK fri = FRIDAY;     // fri = 5
  ENUM_DAY_OF_WEEK sat = SATURDAY;   // sat = 6

  int i = 0;
  ENUM_DAY_OF_WEEK x = i; // warning: implicit enum conversion
  ENUM_DAY_OF_WEEK y = 1; // ok, equals to MONDAY
  ENUM_ORDER_TYPE buy = ORDER_TYPE_BUY;   // buy = 0
  ENUM_ORDER_TYPE sell = ORDER_TYPE_SELL; // sell = 1
  // ...

  // warning: implicit conversion
  //          from 'enum ENUM_DAY_OF_WEEK' to 'enum ENUM_ORDER_TYPE'
  //          'ENUM_ORDER_TYPE::ORDER_TYPE_SELL' will be used
  //          instead of 'ENUM_DAY_OF_WEEK::MONDAY'
  ENUM_ORDER_TYPE type = MONDAY;
  // compilation error: uncomment to reproduce
  // ENUM_DAY_OF_WEEK day = ORDER_TYPE_CLOSE_BY; // cannot convert enum
  // ENUM_DAY_OF_WEEK z = 10; // '10' - cannot convert enum
}
```

Todas las constantes de los días de la semana se codifican con números del 0 al 6, siendo el domingo el punto de partida. Básicamente, las constantes no tienen por qué constar de números consecutivos ni empezar por 0. Hay enumeraciones en las que esto no es así.

Tenga en cuenta que las mismas constantes pueden significar cosas distintas en tipos de enumeración diferentes. Por ejemplo, para las órdenes ORDER_TYPE_BUY y ORDER_TYPE_SELL de la enumeración ENUM_ORDER_TYPE se utilizan los mismos valores (0 y 1) que para los días de la semana SUNDAY y MONDAY de ENUM_DAY_OF_WEEK.

Cuando se copia el valor de una variable entera simple *i* en la variable de enumeración *x*, el compilador emite un aviso, ya que puede haber un valor distinto de las constantes permitidas en la variable *i* en la fase de ejecución del programa.

En la variable yregistramos el número 1, que significa MONDAY, y el compilador considera que esta es una operación correcta.

Un intento de escribir la constante de una enumeración en la variable de otra enumeración (como MONDAY para la variable *type* en el ejemplo anterior) puede dar lugar a un aviso sobre una conversión de tipo implícita. Esto ocurre si la constante que se está escribiendo tiene el mismo valor que uno de los elementos de la enumeración de destino. En otras palabras: cada una de las dos enumeraciones tiene su propio elemento con el valor correspondiente. Seguidamente, el compilador realiza de forma automática una conversión implícita en lugar del programador, pero utiliza un aviso para «pedir» al programador que compruebe si todo va según lo previsto: el hecho de que MONDAY vaya a sustituirse por ORDER_TYPE_SELL es extraño, sin duda; sin embargo, lo aquí lo hemos hecho a propósito con fines ilustrativos.

Si el elemento que se copia no coincide, por su valor, con ningún elemento de otra enumeración, se genera un error de compilación, ya que es imposible una conversión implícita, como cuando se escribe ORDER_TYPE_CLOSE_BY en la variable *day*.

La cadena comentada con la variable *z* también provoca un error de compilación, ya que el valor 10 no pertenece a ENUM_DAY_OF_WEEK. Si el programador está seguro de que, en un caso raro, sigue siendo necesario registrar un valor aleatorio en la variable de tipo enumeración, puede utilizar la conversión de tipos explícita.

La conversión de tipos explícita e implícita se abordará en la sección titulada Conversión de tipos.

MQL5 permite al programador declarar sus propias enumeraciones utilizando la palabra clave *enum*. Esta función se describe en la sección siguiente, Enumeraciones personalizadas (*enum*).

2.2.9 Enumeraciones personalizadas

Las enumeraciones personalizadas se basan estructuralmente en el tipo *int* y los principios de su uso coinciden completamente con lo que se ha comentado en la sección anterior en la que se abordaron las enumeraciones integradas. Por lo tanto, aquí vamos a describir las enumeraciones personalizadas, si bien, estrictamente hablando, no están integradas.

Ud. habrá de utilizar la palabra clave *enum* para describir su propia enumeración en el código MQL5. La forma de descripción más sencilla es la siguiente:

```
enum name
{
  element1,
  element2,
  element3
};
```

Esta descripción registra en el programa un tipo de enumeración denominado *name* con elementos encerrados entre llaves y separados por comas (su cantidad sólo está limitada por el valor más alto de *int*, lo que puede considerarse una ausencia de limitaciones en términos de tareas prácticas). Los identificadores *element1*, *element2* y *element3* pueden utilizarse en el programa dentro del contexto en el que se han definido: globalmente (es decir, fuera de todas las funciones) o dentro de una función (véase la sección Contexto, visibilidad y vida útil de las variables).

Fíjese en el punto y coma que sigue a la llave de cierre: es necesario porque la descripción de la enumeración es una sentencia separada, y después de cualquier sentencia de MQL5 debe escribirse punto y coma.

De manera predeterminada, los identificadores toman valores constantes, empezando por 0, y cada valor subsiguiente es 1 mayor que el anterior. Si es necesario, el programador puede definir un valor específico para cada elemento, después de '=' a la derecha del identificador. Por ejemplo, la entrada anterior es equivalente a esta otra:

```
enum name
{
  element1 = 0,
  element2 = 1,
  element3 = 2
};
```

Está permitido especificar como valor sólo constantes o expresiones que el compilador pueda calcular en la fase de compilación (para más detalles, consulte el ejemplo siguiente).

Si los valores no están definidos para todos los elementos, los valores omitidos se calculan automáticamente basándose en los más cercanos conocidos (precedentes) mediante la adición de 1. Por ejemplo:

```
enum name
{
  element1 = 1,
  element2,
  element3 = 10,
  element4,
  element5
};
```

Aquí, los dos primeros elementos toman los valores 1 y 2 (calculados), mientras que los que empiezan por el tercero toman 10 (especificado explícitamente), 11 y 12 (los dos últimos se calculan a partir de 10).

En el script *TypeUserEnum.mq5* hay algunos ejemplos de descripción de enumeraciones personalizadas.

```
const int zero = 0; // runtime value is not known at compile time

enum
{
  MILLION = 1000000
};

enum RISK
{
  // OFF      = zero, // error: constant expression required
  LOW      = -1,
  MODERATE = -2,
  HIGH     = -3,
};

enum INCOME
{
  LOW      = 1,
  MODERATE = 2,
  HIGH     = 3,
  ENORMOUS = MILLION,
};

void OnStart()
{
  enum INTERNAL
  {
    ON,
    OFF,
  };

  // int x = LOW; // ambiguous access, can be one of
  int x = RISK::LOW;
  int y = INCOME::LOW;
}
```

La enumeración INTERNAL muestra la posibilidad de describirla dentro de la función y, al hacerlo, limita la región de visibilidad o disponibilidad de este tipo, lo que resulta útil en términos de conflictos entre nombres.

La enumeración RISK muestra que pueden asignarse elementos con valores negativos. El elemento comentado OFF no puede describirse debido al intento de inicializarlo con una expresión no constante: en este caso se especifica la variable *zero*, cuyo valor no puede ser calculado por el compilador.

En la enumeración INCOME, el valor del elemento MILLION de la otra enumeración definida más arriba inicializa correctamente el elemento ENORMOUS. Las enumeraciones se crean en el momento de la compilación y, por lo tanto, están disponibles en las expresiones de inicialización.

La enumeración con MILLION no tiene nombre; tales enumeraciones se denominan anónimas y su aplicación básica es declarar constantes. Sin embargo, las enumeraciones con nombre se utilizan más a menudo para las constantes, ya que permiten agrupar los elementos por su significado.

Dado que en el ejemplo hay definidas 2 enumeraciones, ambas con elementos de idéntico nombre, especificar el identificador LOW al declarar la variable *x* da lugar al error de compilación «acceso ambiguo», ya que no queda claro a qué elemento de qué enumeración se refiere. Tenga en cuenta que los identificadores pueden tener (y tienen, en este caso) valores diferentes.

Para resolver este problema existe un operador de contexto especial: Dos dos puntos, '::'. Este operador ayuda a formar el identificador completo del elemento del lenguaje, es decir, el elemento de enumeración, en nuestro caso: en primer lugar se especifica el nombre de la enumeración; a continuación, el operador '::', y después, el identificador del elemento. Ejemplo: RISK::LOW e INCOME::LOW. Descubriremos todos los operadores en la sección correspondiente.

2.2.10 Tipo void

El tipo *void* es un tipo especial. Significa vacío (sin tipo) y no consume memoria. Sólo se utiliza para describir funciones que no devuelven ningún valor ni tienen ningún parámetro. Hemos visto un ejemplo de una función de este tipo: *OnStart* en el script *HelloChart* de la Parte 1. Esto se tratará con más detalle en la sección Funciones.

Es imposible utilizar el tipo *void* para describir variables; sin embargo, es el tipo básico para describir referencias a los objetos aleatorios de las clases. Esta posibilidad se describe en la Parte 3 relativa a la programación orientada a objetos.

2.3 Variables

En este capítulo conoceremos los principios básicos del trabajo con variables en MQL5, concretamente los relativos a los tipos de datos integrados. En particular, consideraremos la declaración y la definición de variables, las características especiales de la inicialización según lo requiera el contexto, la vida útil y los modificadores básicos que cambian las propiedades de las variables. Más adelante, basándonos en estos conocimientos, ampliaremos las capacidades de las variables con nuevos tipos personalizados (uniones, enumeraciones personalizadas y alias), clases, punteros y referencias.

Las variables en MQL5 proporcionan un mecanismo para almacenar datos de diversos tipos, desempeñando un papel importante en la organización de la lógica del programa y las operaciones con información de mercado. Esta sección incluye las siguientes subsecciones:

Declaración y definición de variables:

- La declaración de variables es el paso que consiste en crearlas en un programa. En esta sección veremos cómo declarar y definir variables, así como la forma de especificar sus tipos.

Contexto, ámbito y vida útil de las variables:

- Las variables pueden existir en diferentes contextos y ámbitos, lo que afecta a su disponibilidad y vida útil. En esta subsección se abordan estos aspectos, lo que le ayudará a entender cómo las variables interactúan con su código.

Inicialización:

- La inicialización de variables implica asignarles valores iniciales. Estudiamos métodos de inicialización que ayudan a evitar comportamientos indefinidos del programa.

Variables estáticas:

- Las variables estáticas conservan sus valores entre llamadas a funciones. En esta sección se explica cómo utilizar variables estáticas para almacenar información entre diferentes ejecuciones de código.

Variables constantes:

- Las variables constantes representan valores que no cambian durante la ejecución del programa. En esta sección se describe su uso y características.

Variables de entrada:

- Las variables de entrada se utilizan en los robots de trading para configurar los parámetros de la estrategia. Veremos cómo utilizarlos para crear sistemas de trading flexibles y personalizables.

Variables externas:

- Las variables externas permiten a los usuarios interactuar con el programa, ya que sus valores pueden cambiarse sin necesidad de modificar el código. En esta sección se explica cómo funcionan las variables externas.

2.3.1 Declaración y definición de variables

Una variable es una celda de memoria con nombre para almacenar los datos de un tipo específico. Para que el programa pueda manejar una variable, el programador debe declararla y/o definirla en el código fuente. En el caso general, los términos 'declaración' y 'definición' significan cosas distintas en relación con los elementos del programa, mientras que prácticamente siempre coinciden para las variables. Estos entresijos se abordarán cuando descubramos las funciones, las clases y las variables especiales (externas). Aquí vamos a utilizar ambos términos indistintamente, junto con el de 'descripción' como generalizador.

Podría darse por sentado que una declaración contiene una descripción de un elemento del programa con todos los atributos necesarios para su uso en el programa. La definición, sin embargo, contiene la implementación específica de este elemento, que se corresponde con la declaración.

Las declaraciones permiten al compilador interconectar todos los elementos del programa. A partir de las definiciones, el compilador genera un código ejecutable.

En el caso de las variables, su declaración actúa prácticamente siempre como su definición, ya que garantiza la asignación de memoria y la interpretación de su contenido de acuerdo con sus tipos (esto es exactamente una implementación de una variable). La única excepción es la declaración de variables con la palabra 'extern' (para obtener más detalles, véase la sección Variables externas).

Sólo a partir de la descripción de una variable puede utilizar sentencias especiales para introducir valores en ella, leerlos y hacer referencia al nombre de la variable para moverla de una parte del programa a otra.

En el caso más sencillo, una sentencia que describe una variable aparece de la siguiente manera:

```
type name;
```

En este caso, *name* debe cumplir los requisitos de construir identificadores. Como *type*, puede especificar cualquiera de los tipos incorporados que hemos visto en la sección anterior, o algunos otros tipos personalizados; un poco más adelante veremos cómo crearlos. Por ejemplo, la variable entera *i* se declara de la siguiente manera:

```
int i;
```

Si es necesario, puede describir simultáneamente varias variables del mismo tipo. En este caso, sus nombres se especifican en la sentencia, separados por comas.

```
int i, j, k;
```

Un factor importante es el lugar del programa en el que se encuentra la sentencia que contiene la descripción de la variable. Esto afecta al tiempo de vida de la variable y a su accesibilidad desde distintas partes del programa.

2.3.2 Contexto, ámbito y vida útil de las variables

MQL5 está entre los lenguajes de programación que utilizan llaves para agrupar las sentencias en bloques de código.

Recordemos que un programa se compone de bloques con sentencias, y un bloque debe existir de forma inequívoca. En los ejemplos de script de la Parte 1 vimos la función *OnStart*. El cuerpo de esta función (el texto entre llaves que sigue al nombre de la función) es exactamente ese bloque de código necesario.

Dentro de cada bloque se forma el contexto local, es decir, una región que limita la visibilidad y la vida útil de las variables descritas en su interior. Hasta ahora sólo hemos encontrado ejemplos en los que las llaves definen el cuerpo de las funciones; sin embargo, también pueden utilizarse para formar operadores compuestos, en la sintaxis de la descripción de clases y espacios de nombres. Todos estos métodos también definen regiones de visibilidad y se estudiarán en las secciones correspondientes. En esta fase sólo tenemos en cuenta un tipo de bloques locales, los que están dentro de funciones.

Además de regiones locales, cada programa tiene también un contexto global, es decir, una región con las definiciones de variables, funciones y otras entidades hechas más allá de otros bloques.

Por el lado del script sencillo, en el que el Asistente MQL ha creado la única función *OnStart* vacía, habrá entonces sólo 2 regiones: una global y otra local (dentro del cuerpo de la función *OnStart* , aunque esté vacía). El siguiente script lo ilustra con comentarios.

```
// GLOBAL SCOPE
void OnStart()
{
  // LOCAL SCOPE "OnStart"
}
// GLOBAL SCOPE
```

Tenga en cuenta que la región global se extiende por todas partes aparte de la función *OnStart* (tanto antes como después de ella). Básicamente, incluye todo más allá de cualquier función (si hubiera muchas), pero no hay nada en este script, aparte de *OnStart*.

Podemos describir variables, como *i, j, k*, en la parte superior del archivo, y se convertirán en globales.

```
// GLOBAL SCOPE
int i, j, k;
void OnStart()
{
  // LOCAL SCOPE "OnStart"
}
// GLOBAL SCOPE
```

Las variables globales se crean inmediatamente al iniciar un programa MQL en el terminal y existen durante todo el periodo de ejecución del programa.

El programador puede registrar y leer el contenido de las variables globales desde cualquier lugar del programa.

Básicamente se recomienda describir las variables globales sólo al principio, pero es necesario. Si movemos la declaración debajo de toda la función *OnStart*, no cambiará nada básicamente; sólo será difícil para otros programadores encontrarle el sentido al instante al código con variables, a cuyas definiciones primero hay que llegar.

Curiosamente, la propia función *OnStart* también se declara en el contexto global. Si añadimos otra función, ésta también se declarará en el contexto global. Recuerde cómo creamos la función *Greeting* en la Parte 1 y cómo la invocamos desde la función *OnStart*. Este es el efecto de que el nombre de la función y el método de hacer referencia a ella (cómo ejecutarla) se conozcan en todo el código fuente. Los espacios de nombres añaden algunas sutilezas, pero las veremos más adelante.

Una región local dentro de cada función pertenece sólo a dicha función: una región local está dentro de *OnStart*, y otra está dentro de *Greeting*, que es propia y difiere tanto de la región local de *OnStart* como de la global.

Las variables descritas en el cuerpo de la función se denominan locales y se crean según sus descripciones a partir de la llamada a la función correspondiente durante la ejecución del programa. Las variables locales sólo pueden utilizarse dentro del bloque que las contiene. No son visibles ni accesibles desde el exterior. Al salir de la función, las variables locales se destruyen.

Ejemplo de descripción de las variables *x, y, z* locales dentro de la función *OnStart*:

```
// GLOBAL SCOPE
int i, j, k;
void OnStart()
{
  // LOCAL SCOPE "OnStart"
  int x, y, z;
}
// GLOBAL SCOPE
```

Debe tenerse en cuenta que los pares de llaves pueden utilizarse tanto para describir la función y otras sentencias como para formar por sí mismas el bloque de código interno. El anidamiento de unidades es ilimitado.

Los bloques anidados suelen añadirse para minimizar el alcance de las variables utilizadas en una pequeña ubicación de código lógicamente aislada (si no es establecida por una función por un motivo u otro). Esto permite reducir la probabilidad de una falsa modificación de la variable donde no estaba prevista o algunos efectos secundarios no deseados debidos al intento de reutilizar la misma variable para diversas necesidades (lo que no es una buena práctica).

A continuación se muestra una función de ejemplo en la que el nivel de anidamiento de unidades es 2 (si consideramos que el bloque con el cuerpo de la función es el primer nivel) y se crean 2 bloques de este tipo que se ejecutarán consecutivamente.

```
void OnStart()
{
  // LOCAL SCOPE "OnStart"
  int x, y, z;

  {
    // LOCAL SUBSCOPE 1
    int p;
    // ... use p for task 1
  }

  {
    // LOCAL SUBSCOPE 2
    // y = p; // error: 'p' - undeclared identifier
    int p;    // from now 'p' is declared
    // ... use p for task 2
  }

  // p = x; // error: 'p' - undeclared identifier
}
```

Dentro de ambos bloques se describe la variable *p*, que en ellos se utiliza para diversos fines. De hecho, se trata de dos variables diferentes, aunque tengan el mismo nombre visible dentro de cada bloque.

Si la variable se sacara a la lista inicial de las variables locales de la función, podría contener algún valor restante al salir del primer bloque, rompiendo así el funcionamiento del segundo bloque. Además, el programador podría implicar ocasionalmente a *p* en algo más al principio de la función, y entonces los efectos secundarios podrían tener lugar en el primer bloque.

Más allá de cualquiera de los dos bloques anidados, la variable *p* es desconocida y, por tanto, cualquier intento de referirse a ella desde el bloque común de la función conduce a un error de compilación («identificador no declarado»).

También hay que tener en cuenta que una variable puede describirse no al principio del bloque, sino en su mitad o incluso más cerca del final. Entonces no se define en todo el bloque, sino sólo por debajo de su definición. Por lo tanto, se producirá el mismo error cuando se haga referencia a la variable por encima de su descripción.

Así, la región de ámbito de la variable puede diferir del contexto (todo el bloque).

Ambas versiones del problema se ilustran en un ejemplo: intente incluir cualquiera de las cadenas con las sentencias $p = x$ y $y = p$ y compile el código fuente.

Se asigna memoria a todas las variables locales de la función en cuanto se pasa el control dentro de la función. Sin embargo, este no es el final de su creación. A continuación, se inicializan (se establecen los valores iniciales), siendo la inicialización definida explícitamente por el programador o implícitamente por los valores por defecto del compilador. Al mismo tiempo, es esencial el contexto en el que se describen las variables.

2.3.3 Inicialización

Al describir variables, existe la posibilidad de establecer el valor inicial; este se especifica a continuación del nombre de la variable y del símbolo '=' y debe corresponderse con el tipo de variable o moldearse con arreglo al mismo (la conversión de tipos se puede encontrar en la correspondiente sección).

```
int i = 3, j, k = 10;
```

Aquí *i* y *k* se han inicializado explícitamente, mientras que *j*, no.

Como valor inicial se puede especificar tanto una constante (literal del tipo correspondiente) como una expresión (una especie de fórmula para realizar cálculos). Expondremos las expresiones por separado. Mientras tanto, veamos un ejemplo sencillo:

```
int i = 3, j = i, k = i + j;
```

Aquí, la variable *j* toma el mismo valor que la variable *i*, mientras que la variable *k* toma la suma de *i* y *j*. En sentido estricto, en los tres casos vemos expresiones. Sin embargo, la constante (3) es una opción de expresión degenerada especial. En el segundo caso, el único nombre de variable es una expresión, es decir, el resultado de la expresión será el valor de esta variable sin ninguna transformación. En el tercer caso, se accede en la expresión a dos variables, *i* y *j*, se ejecuta la operación de suma con sus valores y, después, el resultado pasa a la variable *k*.

Dado que la sentencia que contiene la descripción de varias variables se procesa de izquierda a derecha, el compilador ya conoce los nombres de las variables anteriores cuando analiza otra descripción.

Un programa suele contener muchas sentencias con descripciones de variables que son leídas por el compilador de forma natural de arriba a abajo. En inicializaciones posteriores se pueden utilizar nombres tomados de descripciones anteriores. Aquí están las mismas variables descritas por dos sentencias separadas.

```
int i = 3, j = i;
int k = i + j;
```

Las variables sin inicialización explícita también reciben algunos valores iniciales, pero dependen del lugar en el que se describió la variable, es decir, de su contexto.

Cuando no hay inicialización, las variables locales toman valores aleatorios en el momento de ser generadas: el compilador se limita a asignarles memoria en función del tamaño del tipo, mientras que se desconoce qué habrá en una dirección concreta (a menudo se reasignan varias zonas de memoria del ordenador para utilizarlas en programas diferentes después de que hayan dejado de ser necesarias para aquellos que se ejecutaron anteriormente).

Normalmente se sugiere que los valores de trabajo se introduzcan en variables locales sin inicializar en algún punto posterior del código del algoritmo, como por ejemplo utilizando las operaciones de asignación de las que hablaremos más adelante. Sintácticamente, esto es similar a la inicialización, ya que también utiliza el signo igual '=' para transferir el valor de la «estructura» situada a su derecha (puede ser una constante, variable, expresión o llamada a función) en la variable de la izquierda. Sólo una variable puede estar a la izquierda de '='.

El programador debe asegurarse de que la lectura de la variable no inicializada sólo tiene lugar cuando se le asigna un valor significativo. Si no es así, el compilador emite un aviso («posible uso de una variable no inicializada»).

Todo es diferente con las variables globales.

Un ejemplo de variables globales es el parámetro de entrada *GreetingHour* del script *GoodTime2* de la Parte 2. El hecho de que la variable haya sido descrita con la palabra clave *input* no afecta a sus otras propiedades como variable. Podríamos excluir su inicialización y describirla como sigue:

```
input uint GreetingHour;
```

Esto no cambiaría nada en el programa, ya que las variables globales son inicializadas implícitamente por el compilador usando cero si no hay inicialización explícita (mientras que antes también teníamos inicialización explícita con cero).

Cualquiera que sea el tipo de variable, la inicialización implícita es realizada siempre por un valor equivalente a cero. Por ejemplo, para una variable *bool* se establecerá *false*, mientras que para una variable *datetime* será D'1970,01.01 00:00:00'. Existe un valor especial, NULL, para las cadenas. Se trata, si se quiere, de una cadena aún más «vacía» que las comillas vacías «» porque aún queda memoria asignada a ellas, donde se coloca el único carácter nulo terminal.

Además de las variables locales y globales, existe otro tipo, esto es, las variables estáticas. El compilador las inicializa, también, implícitamente con cero, si el programador no ha escrito ningún valor explícitamente inicial. Se abordarán en la siguiente sección.

Vamos a crear un nuevo script, *VariableScopes.mq5*, con ejemplos de descripción de variables locales y globales (*MQL5/Scripts/MQL5Book/VariableScopes.mq5*).

```
// global variables
int i, j, k;    // all are 0s
int m = 1;      // m = 1                (place breakpoint on this line)
int n = i + m;  // n = 1
void OnStart()
{
  // local variables
  int x, y, z;
  int k = m; // warning: declaration of 'k' hides global variable
  int j = j; // warning: declaration of 'j' hides global variable
  // use variables in assignment statements
  x = n;     // ok, 1
  z = y;     // warning: possible use of uninitialized variable 'y'
  j = 10;    // change local j, global j is still 0
}
// compilation error
// int bad = x; // 'x' - undeclared identifier
```

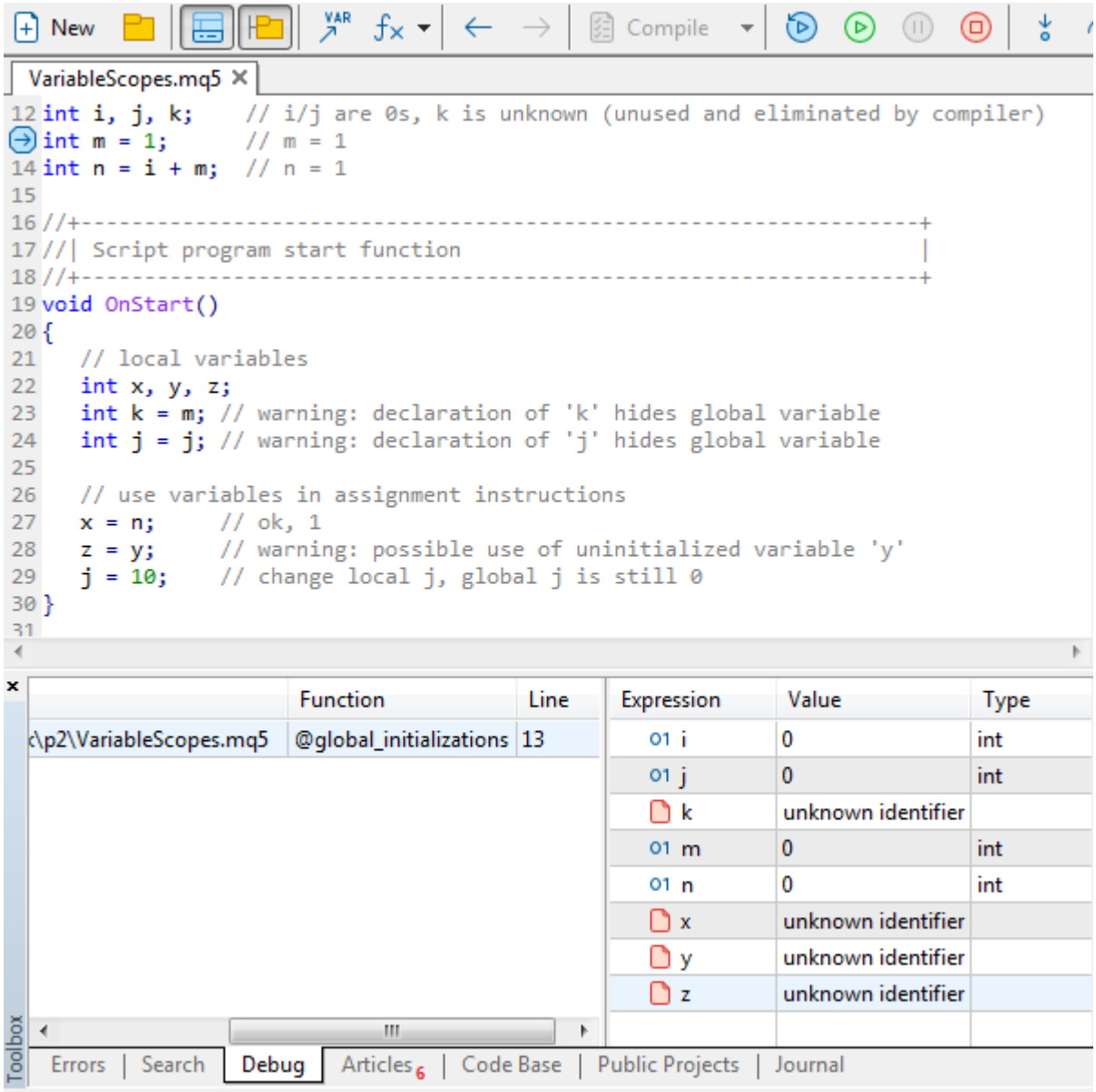
Hay que recordar que, al lanzar un programa de MQL, el terminal primero inicializa todas las variables globales y luego invoca una función que es el punto de partida para los programas de un tipo relevante. En este caso, se trata de *OnStart* para scripts.

Aquí, sólo las variables *i, j, k, m, n* son globales, ya que se describen fuera de la función (en nuestro caso, sólo tenemos una función, *OnStart*, que es necesaria para los scripts). *i, j, k* toman implícitamente el valor 0. *m* y *n* contienen 1.

Puede ejecutar el script en el modo de depuración paso a paso y asegurarse de que los valores de las variables cambian exactamente de esta manera. Para ello, debe establecer previamente un punto de interrupción en la cadena con la inicialización de una de las variables globales, como *m*. Coloque el

cursor de texto sobre esta cadena y ejecute *Depurar -> Cambiar punto de interrupción* (F9); y la cadena se resaltará con un signo azul en el campo de la izquierda, lo que indica que la ejecución del programa se detendrá aquí si comienza a funcionar en el depurador.

A continuación, debe ejecutar realmente el programa para depurarlo, para lo cual ejecute el comando *Depurar -> Empezar con datos reales* (F5). En este momento se abrirá un nuevo gráfico en el terminal, en el que comienza a ejecutarse este script (leyenda «VariableScopes (Debugging)» en la esquina superior derecha), pero se suspende inmediatamente y volvemos a MetaEditor. Deberíamos ver en él una imagen como la siguiente.



Depuración paso a paso y visualización de variables en MetaEditor

Una cadena que contiene un punto de interrupción se marca ahora con un signo de flecha: es la sentencia actual que el programa se dispone a ejecutar pero que aún no se ha ejecutado. Abajo a la izquierda se muestra la pila actual del programa, que por el momento consta de una sola entrada: @global_initializations. Puede introducir expresiones en la parte inferior derecha para controlar sus valores en tiempo real. Nos interesan los valores de las variables; por lo tanto, introduzcamos *i, j, k, m, n, x, y, z* de forma consecutiva (cada uno en una cadena separada).

Verá además que MetaEditor añade automáticamente variables del contexto actual para su visualización (por ejemplo, variables locales y entradas de la función, donde las sentencias se ejecutan dentro de la función). Pero ahora vamos a añadir *x*, *y* y *z* manualmente y por adelantado, sólo para mostrar que no están definidos fuera de la función.

Tenga en cuenta que, para las variables locales, se escribe «Identificador desconocido» en lugar de un valor, ya que todavía no se ha creado el bloque de funciones *OnStart*, donde se encuentran. Las variables globales *i* y *j* tendrán primero valores cero. La variable global *k* no se utiliza en ninguna parte y, por lo tanto, el compilador la excluye.

Si ejecutamos un paso de la ejecución del programa (ejecutar la sentencia en la línea de código actual) utilizando los comandos *Paso Into* (F11) o *Paso Over* (F10), veremos cómo la variable *m* toma el valor 1. Otro paso continuará la inicialización para la variable *n*, y también se convertirá en 1.

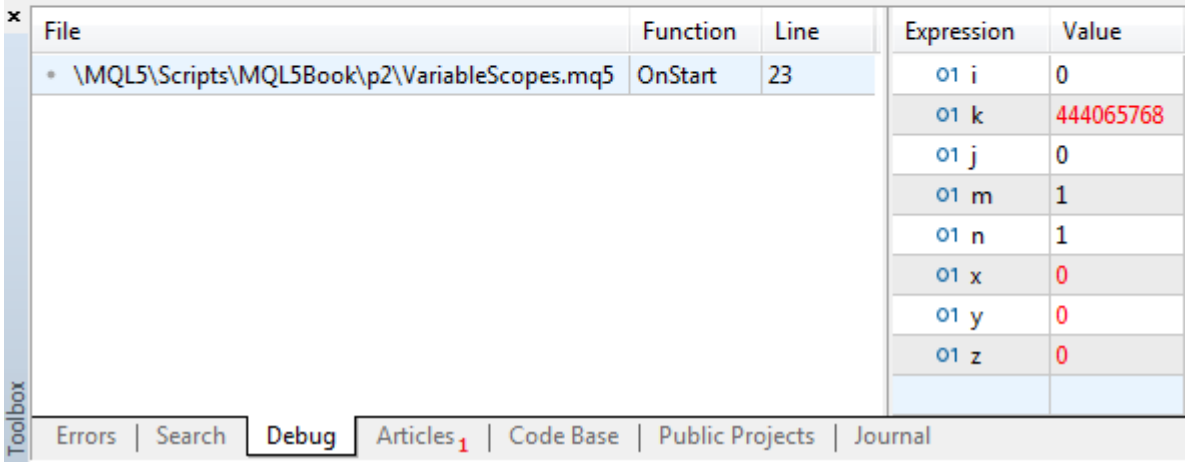
Aquí terminan las descripciones de las variables globales y, como sabemos, el terminal invoca a la función *OnStart* una vez finalizada la inicialización de las variables globales. En este caso, para entrar en la función *OnStart* en el modo paso a paso, pulse F11 una vez más (o bien, puede establecer otro punto de interrupción al principio de la función *OnStart*).

Las variables locales se inicializan cuando la ejecución de las sentencias del programa llega al bloque de código donde se han definido. Por lo tanto, las variables *x, y, z* sólo se crean al entrar en la función *OnStart*.

Cuando el depurador entre en la función *OnStart* podrá ver, con un poco de suerte, que realmente hay valores inicialmente aleatorios en *x*, *y* y *z*. En este caso, la «suerte» consiste en que estos valores aleatorios pueden ser nulos. Entonces será imposible diferenciarlas de la inicialización implícita con cero que realiza el compilador para las variables globales. Si el script se lanza repetidamente, la «basura» de las variables locales será probablemente diferente y más ilustrativa. No se inicializan explícitamente y, por tanto, su contenido puede ser de cualquier tipo.

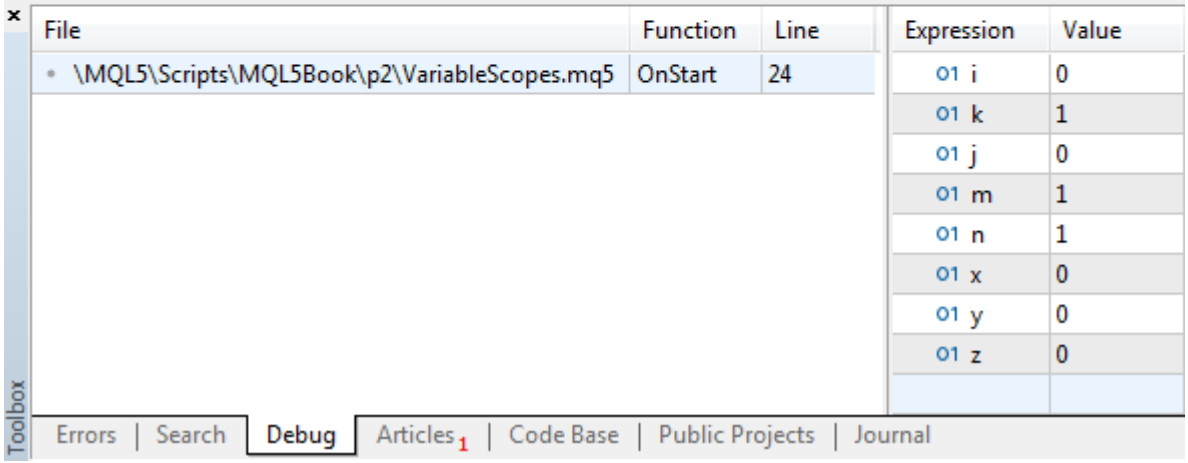
En la secuencia de imágenes siguiente puede ver la evolución de las variables utilizando el modo paso a paso del depurador. La cadena actual que va a ejecutarse (pero que aún no se ha ejecutado) se marca con una flecha verde en los campos con enumeración.

```
19 void OnStart()
20 {
21    // local variables
22    int x, y, z;
23    int k = m; // warning: declaration of 'k' hides global variable
24    int j = j; // warning: declaration of 'j' hides global variable
25
26    // use variables in assignment instructions
27    x = n;      // ok, 1
28    z = y;      // warning: possible use of uninitialized variable 'y'
29    j = 10;     // change local j, global j is still 0
30 }
```



Depuración paso a paso y visualización de variables en MetaEditor (cadena 23)

```
19 void OnStart()
20 {
21    // local variables
22    int x, y, z;
23    int k = m; // warning: declaration of 'k' hides global variable
24    int j = j; // warning: declaration of 'j' hides global variable
25
26    // use variables in assignment instructions
27    x = n;      // ok, 1
28    z = y;      // warning: possible use of uninitialized variable 'y'
29    j = 10;     // change local j, global j is still 0
30 }
```



Depuración paso a paso y visualización de variables en MetaEditor (cadena 24)

Más adelante en el código se demuestra cómo se pueden utilizar estas variables de la forma más sencilla en operadores de asignación. El valor de la variable global *n* se copia en la local *x* sin problemas ya que se ha inicializado *n*. Sin embargo, en la cadena en la que el contenido de la variable *y* se copia en la variable *z* aparece un aviso del compilador, ya que *y* es local y, en este momento, no se ha escrito nada en ella; es decir, no hay ninguna inicialización explícita, así como tampoco otros operadores que puedan establecer su valor.

Dentro de una función se permite describir variables con los mismos nombres que ya se utilizan para las variables globales. Una situación similar puede producirse en bloques locales anidados si se crea una variable en un bloque interno con el nombre existente en un bloque externo. Sin embargo, esta práctica no es recomendable, ya que puede dar lugar a errores lógicos. En tales casos, el compilador emite un aviso («la declaración oculta una variable global/local»).

Debido a esta redefinición, una variable local, como *k* en el ejemplo anterior, se superpone a la homónima global dentro de la función. Aunque tienen el mismo nombre, se trata de dos variables diferentes. La variable *k* local se conoce dentro de *OnStart*, mientras que la variable *k* global se conoce en todas partes excepto en *OnStart*. En otras palabras: cualquier operación dentro del bloque con la variable *k* sólo afectará a la variable local. Por lo tanto, al salir de la función *OnStart* (como si no fuera la única y principal función del script), descubriríamos que la variable global *k* sigue siendo igual a cero.

La variable local *j* no sólo se solapa con la variable global *j* sino que también es inicializada por el valor de esta última. En la cadena que contiene la descripción de *j* dentro de *OnStart*, la versión local de *j* todavía se está creando cuando el valor inicial para ella se lee de la versión global de *j*. Una vez definida con éxito la variable *j* local, este nombre se superpone a la versión global, y es la versión local a la que pertenecen los cambios posteriores en *j*.

Al final del código fuente hemos comentado el intento de declarar una variable global más, *bad*, en cuya inicialización se invoca el valor de la variable *x*. Esta cadena provoca un error de compilación, ya que la variable *x* es desconocida más allá de la función *OnStart* en la que ha sido definida.

2.3.4 Variables estáticas

A veces es necesario describir una variable dentro de una función, asegurando su existencia durante toda la ejecución del programa. Por ejemplo, queremos contar cuántas veces se ha llamado a esta función.

Una variable de este tipo no puede ser local, porque entonces perdería su «memoria larga», pues se creará cada vez que se llame a la función y se eliminará al salir de la misma. Técnicamente, podría describirse de forma global; sin embargo, si la variable sólo se utiliza en esta función, este enfoque es erróneo en términos de diseño del programa.

En primer lugar, una variable global puede modificarse accidentalmente desde cualquier lugar del programa.

En segundo lugar, imagine qué «zoo» de variables se crearía en la región global del programa si declaramos una variable global al menor pretexto. En lugar de ello se recomienda declarar las variables en el bloque más pequeño (si hay varios anidados) en el que se utilicen.

Por lo tanto, el contador de ejecuciones de la función debería describirse dentro de la función. Aquí es donde ayuda el nuevo atributo de las variables, su naturaleza estática.

Una palabra clave especial (modificador), *static*, colocada antes del tipo de variable en su declaración permite prolongar su vida útil a toda la duración de la ejecución del programa, es decir, la asemeja a las

globales. Por regla general, una variable estática sólo se define localmente, en una de las funciones. Por lo tanto, su visibilidad está limitada por el bloque de código correspondiente, como en una variable local normal.

Las variables estáticas también pueden describirse a nivel global, pero no difieren en nada de las globales normales (al menos, en el momento de escribir este libro). Varía con respecto a su comportamiento en C++: allí, su visibilidad está limitada por el archivo en el que se describen. En MQL5, un programa se ensambla sobre la base de un archivo principal mq5 y, tal vez, algunos archivos de cabecera (veáse la directiva*#include*); por lo tanto, las variables globales estáticas y también las normales están disponibles en todos los archivos fuente del programa.

Una variable estática local se crea sólo una vez: en el momento en que el programa entra por primera vez en la función donde se describe esta variable. Una variable de este tipo sólo se eliminará al descargar el programa. Si una función no se ha invocado nunca, las variables estáticas locales descritas en ella, si las hay, nunca se crearán.

Como ejemplo, modifiquemos la función *Greeting* de la Parte 1 para que emita saludos diferentes en cada llamada. Póngamosle al nuevo script el nombre *GoodTimes.mq5*.

Eliminaremos la entrada del script *GreetingHour* y el parámetro de la función *Greeting*. Dentro de la función *Greeting* describiremos una nueva variable estática, *counter*, de tipo entero, con el valor inicial de 0. Hay que recordar que se trata exactamente de una inicialización, y que se ejecutará una sola vez porque la variable es estática.

```
string Greeting()
{
  static int counter = 0;
  static string messages[3] =
  {
    "Good morning", "Good day", "Good evening"
  };
  return messages[counter++ % 3];
}
```

Puesto que ahora conocemos el modificador *static*, es razonable utilizarlo también para el array *messages*. La cuestión es que antes se declaraba como local, y se volvería a crear cada vez en múltiples llamadas de la función *Greeting* (y se eliminaría al salir). Esto no es eficiente.

Hay que recordar que un array es un conjunto con nombre de varios valores del mismo tipo, disponibles por índice especificado entre corchetes después del nombre. Gran parte de lo que se ha dicho sobre las variables se aplica directamente a los arrays. Otros matices del trabajo con arrays se abordarán en la sección Arrays.

Pero volvamos a nuestro problema actual. Se elige una opción del array basada en el valor de la variable *counter* en la sentencia *return*, y por el momento parece bastante cabalística:

```
  return messages[counter++ % 3];
```

Ya hemos mencionado casualmente la operación de módulo realizada mediante el carácter '%' en la Parte 1. Con ella garantizamos que el índice del elemento no podrá superar el tamaño del array: cualquiera que sea el contador, el módulo (o resto) de la división entre 3 será 0 o 1, o 2.

Lo mismo ocurre con la estructura *counter*++: significa sumar 1 al valor de la variable (incremento único).

Es importante tener en cuenta que, en esta notación, el incremento se producirá al haber calculado la expresión completa; en este caso, al dividir *counter % 3*. Esto significa que el recuento comenzará desde cero, es decir, el valor inicial. Existe la posibilidad de hacer un incremento antes de computar la expresión, escribiendo *++counter % 3*. En tal caso se empezaría a contar desde 1. Consideraremos las operaciones de este tipo en la sección Incremento y decremento.

Llamemos a la función *Greeting* desde *OnStart* 3 veces consecutivas.

```
void OnStart()
{
  Print(Greeting(), ", ", Symbol());
  Print(Greeting(), ", ", Symbol());
  Print(Greeting(), ", ", Symbol());
  // Print(counter); // error: 'counter' - undeclared identifier
}
```

Como resultado, veremos las tres cadenas anticipadas con todos los saludos uno tras otro en el registro.

```
GoodTimes (EURUSD,H1)       Good morning, EURUSD
GoodTimes (EURUSD,H1)       Good afternoon, EURUSD
GoodTimes (EURUSD,H1)       Good evening, EURUSD
```

Si seguimos llamando a la función, el contador aumentará y los mensajes irán rotando.

Un intento de hacer referencia a la variable *counter* al final de *OnStart* (comentado) no permitirá compilar el código, ya que la variable estática, aunque sigue existiendo, sólo está disponible dentro de la función *Greeting*.

Tenga en cuenta que las llaves se utilizan tanto para formar los bloques de código como para inicializar los arrays. Debe distinguir entre sus aplicaciones. Los arrays se tratarán en detalle en la sección correspondiente. Sin embargo, éstas no son todas las aplicaciones de las llaves: con su uso, aprenderemos más adelante a definir tipos, estructuras y clases personalizados. Las variables estáticas también pueden definirse dentro de estructuras y clases.

2.3.5 Variables constantes

Por paradójico que parezca, la mayoría de los lenguajes de programación admiten el concepto de variables constantes. En MQL5, se describen añadiendo el modificador *const*. Se coloca en la descripción de la variable, precediendo a su tipo, y significa que el valor de la variable no puede modificarse en modo alguno tras su inicialización por el valor inicial. Durante toda su vida, la variable tendrá el mismo valor, es decir, una constante.

El compilador simplemente evitará asignar la constante a un valor: el error «la constante no se puede modificar» aparecerá en la cadena correspondiente.

El modificador *const* tiene por objeto mostrar explícitamente la intención del programador de no modificar la variable en cuestión, si se trata de un valor fijo comúnmente conocido, como el índice EUR para calcular el índice USD, el número de semanas de un año, etc. Se recomienda utilizar siempre el modificador const si no se va a modificar la variable. Esto ayuda a evitar posibles errores más adelante, si el propio programador o alguien de entre sus colegas intenta escribir por accidente algo distinto en la constante.

Por ejemplo, podemos añadir el modificador *const* para el array *messages* en la función *Greeting*. Esto no parece muy útil para un programa tan pequeño. No obstante, dado que los programas tienden a crecer, cualquier cadena puede «encontrarse» tarde o temprano en un entorno de software mucho más complejo, como sentencias añadidas, modos de operación, etc. Por lo tanto, tiene sentido tener un plan B; sobre todo porque es muy sencillo.

```
string Greeting()
{
  static int counter = 0;
  static const string messages[3] =
  {
    "Good morning", "Good day", "Good evening"
  };
  // error demo: 'messages' - constant cannot be modified
  // messages[0] = "Good night";
  return messages[counter++ % 3];
}
```

En la cadena comentada, probamos a registrar la cadena «Buenas noches» en el primer elemento del array (recuerde que la numeración empieza por 0). En este caso, el sentido de esta acción es simplemente asegurarse de que el compilador impide hacer eso.

Como se ve fácilmente, los modificadores *static* y *const* pueden combinarse. El orden de registro no es importante.

Por cierto: en MQL5, las variables se convierten en constantes tanto al usar el modificador *const* como al declararlas con las variables de entrada del programa.

2.3.6 Variables de entrada

Cuando se inician, todos los programas en MQL5 pueden solicitar parámetros al usuario. La única excepción son las bibliotecas que no se ejecutan de forma independiente, sino como partes de otro programa (véase la sección correspondiente para saber más sobre Bibliotecas).

Los parámetros de entrada de los programas MQL son variables globales descritas en el código que tienen un modificador especial de *input* o *sinput*. Están disponibles en el cuadro de diálogo de propiedades del programa para que el usuario introduzca valores. Vimos una descripción de la variable de entrada *GreetingHour* en los scripts de la Parte 1.

Una característica especial de las variables de entrada es el hecho de que su valor no puede modificarse en el código del programa, es decir, se comporta como una constante.

Las variables de entrada sólo pueden ser de enumeraciones o tipos integrados simples. En el caso de las enumeraciones, los valores se introducen mediante una lista desplegable, mientras que en todos los demás casos se utilizan campos de entrada. No está permitido describir como *input*: Arrays, estructuras o uniones ni clases.

El desarrollador puede establecer el nombre del parámetro de entrada distinto del identificador de la variable. Este nombre se le mostrará al usuario en el cuadro de diálogo de propiedades del programa. Debe añadirse una descripción detallada como comentario en la definición del parámetro de entrada.

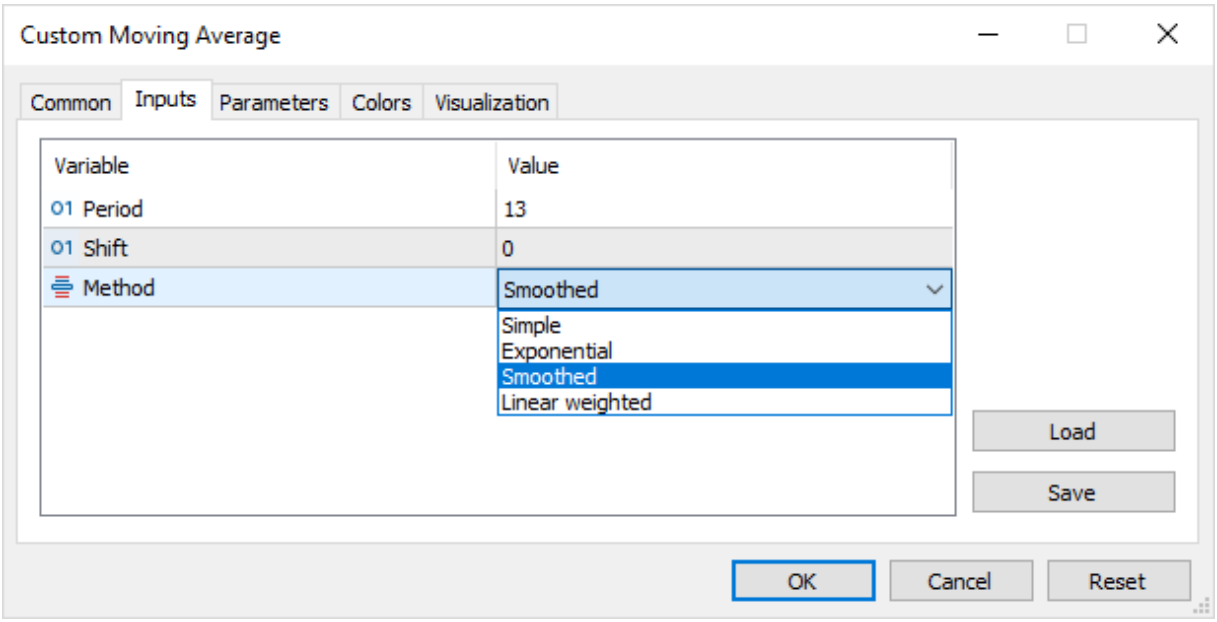
```
input int HourStart = 0; // Start of trading (hour, including):
input int HourStop = 0;  // End of trading (hour, excluding):
```

Esto permite hacer que la interfaz sea más fácil de usar, detallada y libre de las restricciones sintácticas que impone MQL5 a los identificadores. Además, los nombres (así como los comentarios) pueden estar en su lengua materna.

Por ejemplo, MetaTrader 5 viene con el código fuente del indicador *MQL5/Indicators/Examples/Custom Moving Average.mq5* con las variables de entrada:

```
input int            InpMAPeriod = 13;         // Period
input int            InpMAShift  = 0;          // Shift
input ENUM_MA_METHOD InpMAMethod = MODE_SMMA; // Method
```

Esta descripción genera el cuadro de diálogo de propiedades que se muestra a continuación.



Ejemplo de cuadro de diálogo de las propiedades del programa MQL

La longitud máxima de la representación textual de una variable de entrada como par identificador=valor, incluido el carácter "=", no puede superar los 255 caracteres (esta restricción viene impuesta por los protocolos internos de intercambio de datos del terminal y los agentes de pruebas). Este límite es especialmente importante para las variables de cadena, ya que los valores de otros tipos nunca lo superan. Como sabemos, la longitud de un identificador está limitada a 63 caracteres; por lo tanto, en función de la longitud del identificador, quedan entre 191 y 253 caracteres para el valor de la variable de la cadena de entrada. Todo el texto que supere el umbral combinado de 255 caracteres puede recortarse cuando se transfiere al probador. Si una cadena más larga tiene que ser introducida en su programa MQL, use múltiples campos de entrada (para continuar) o permita que el usuario especifique el nombre del archivo desde el que debe leerse el texto.

Para mayor comodidad en el funcionamiento de los programas MQL, las entradas se pueden combinar en bloques con nombre utilizando la palabra clave *group* (no es necesario el punto y coma al final de la cadena de grupo).

```
input group "group_name"
input type identifier = value;
...
```

Todas las variables con modificador *input* que siguen a la descripción del grupo (hasta la descripción de otro grupo o hasta el final del archivo) se muestran visualmente como una lista anidada bajo la cabecera del grupo en el cuadro de diálogo de propiedades del programa MQL. Además, se pueden desplegar o colapsar grupos de parámetros con un clic del ratón en el probador de estrategias aplicable tanto a indicadores como a Asesores Expertos (EA).

La palabra clave *sinput* es la abreviatura de *static input*, siendo ambas formas equivalentes.

Las variables descritas con los modificadores *sinput* y *static input* no pueden participar en la optimización. Sólo tiene sentido utilizarlas en Asesores Expertos siendo el único tipo de programa MQL que admite optimización. Para obtener más detalles, consulte la sección dedicada a Probar y optimizar Asesores Expertos.

2.3.7 Variables externas

> El material de esta sección es a la vez complejo y opcional; requiere el conocimiento de los conceptos que se basan en la analogía con C++ y los que se consideran a continuación. Al mismo tiempo, el efecto de la estructura lingüística descrita puede conseguirse de otra manera, mientras que su flexibilidad es una fuente potencial de errores.

MQL5 permite describir variables como externas. Esto se hace utilizando la palabra clave *extern* y sólo se permite en el contexto global.

Para una variable externa, la sintaxis básicamente repite una descripción normal pero adicionalmente tiene la palabra clave 'extern' mientras que la inicialización está prohibida:

```
extern type identifier;
```

Describir una variable como externa significa que su descripción se retrasa y debe producirse más adelante en el código fuente, normalmente en otro archivo (la conexión de archivos mediante *#include*directive se examinará en el capítulo dedicado al preprocesador). Varios archivos fuente diferentes pueden tener una descripción de la misma variable externa, es decir, que tengan tipos e identificadores idénticos. Todas estas descripciones se refieren a la misma variable.

Se supone que esta variable estará completamente descrita en uno de los archivos. Si la variable no está definida en ninguna parte del código sin la palabra clave *extern*, se devuelve el error de compilación «variable externa no resuelta» (similar a un error del enlazador en C++ en estos casos).

La descripción de una variable externa permite utilizarla eficazmente en el código fuente de un archivo concreto. En otras palabras, permite compilar un módulo determinado, aunque la variable no se cree en este módulo.

El uso de *extern* en MQL5 no es tan insistente como en C++ y en la mayoría de los casos, puede ser sustituido por la habilitación de un archivo de cabecera con descripciones generales de las variables que se van a declarar como *extern*. Basta con realizar estas definiciones de forma convencional. El compilador se asegura de añadir cada archivo adjunto al código fuente sólo una vez. Teniendo en cuenta que en MQL5 un programa siempre consta de una unidad compilable mq5, no hay aquí ningún problema de C ++, con el error potencial de las múltiples definiciones de la misma variable debido a la habilitación de la cabecera en diferentes unidades.

Aunque se adjunte un archivo mq5 adicional (no *mqh*) en la directiva *#include*, no compite en igualdad de condiciones con la unidad principal para la que se lanza la compilación, sino que se considera una de las cabeceras.

A diferencia de C++, MQL5 no permite especificar un valor inicial para una variable externa (la inicialización en C++ lleva a ignorar la palabra *extern*). Si intenta establecer un valor inicial obtendrá un error de compilación «no se permite la inicialización de variables externas».

En general, describir una variable como externa puede considerarse un tipo de descripción «imprecisa»: garantiza la aparición de la variable y excluye el error de anulación que se produciría si la variable se describiera en varios archivos sin el modificador *extern*.

Sin embargo, esto puede ser una fuente de errores. Si en diferentes archivos de cabecera, por coincidencia, se describen variables idénticas para diferentes propósitos, entonces ninguna palabra clave *extern* permite identificar un conflicto, mientras que con *extern* las variables se convertirán en una, y la lógica de funcionamiento del programa muy probablemente se romperá.

Como externos, tanto las variables como las funciones pueden describirse (se considerarán más abajo). Para las funciones, describirlas con el atributo como externas es un rudimento (es decir, se compila, pero no realiza ningún cambio). Las dos siguientes declaraciones de una función son equivalentes:

```
extern return_type name([parameters]);
       return_type name([parameters]);
```

En este sentido, la presencia o ausencia de *extern* sólo puede utilizarse para distinguir estilísticamente entre la descripción de una función desde la unidad actual (sin *extern*) o desde una externa (*extern* está presente).

Puede utilizar *extern* tanto en la unidad mq5 que se va a compilar como en los archivos de cabecera que se van a adjuntar.

Veamos algunas opciones para utilizar *extern*: se introducen en distintos archivos, a saber, el script principal *ExternMain.mq5* y 3 archivos adjuntos: *ExternHeader1.mqh*, *ExternHeader2.mqh* y *ExternCommon.mqh*.

En el archivo principal sólo se adjuntan *ExternHeader1.mqh* y *ExternHeader2.mqh*, mientras que necesitaremos *ExternCommon.mqh* un poco más adelante.

```
// source code from mqh files will be substituted implicitly
// in the main mq5 file, instead of these directives
#include "ExternHeader1.mqh"
#include "ExternHeader2.mqh"
```

En los archivos de cabecera se definen dos funciones de utilidad condicional: en el primero, la función *inc* para el incremento de la variable *x* , mientras que en el segundo, la función *dec* para el decremento de la variable *x*. Es la variable *x* la que se describe en ambos archivos como externa:

```
// ExternHeader1.mqh
extern int x;
void inc()
{
   x++;
}
// -----------------
// ExternHeader2.mqh
extern int x;
void dec()
{
   x--;
}
```

Debido a esta descripción, cada uno de los archivos mqh se compila de forma regular. Cuando se incluyen juntos en un archivo mq5 se compila también todo el programa.

Si la variable se definiera en cada archivo sin la palabra *extern*, el error de redefinición se produciría al compilar el programa en su conjunto. Si hubiéramos transferido la definición de *x* de los archivos de cabecera a la unidad principal, los archivos de cabecera habrían dejado de compilarse (puede que alguien no lo considere un problema, pero en programas más grandes, a los desarrolladores les gusta comprobar la capacidad de compilación de las correcciones inmediatas sin compilar el proyecto entero).

En el script principal definimos una variable (en este caso, con un valor inicial de 2, mientras que si no especificamos el valor, se utilizará el 0 de forma predeterminada) e invocamos las funciones de utilidad condicional, además de imprimir el valor *x*.

```
int x = 2;

void OnStart()
{
   inc();  // uses x
   dec();  // uses x
   Print(x); // 2
   ...
}
```

En el archivo *ExternHeader1.mqh* se encuentra la descripción de la variable *short z* (sin *extern*). Una descripción similar se comenta en el script principal. Si activamos esta cadena obtendremos el error antes mencionado («variable ya definida»). Esto se hace para ilustrar el problema potencial.

En *ExternHeader1.mqh* se describe también *extern long y*. Al mismo tiempo, en el archivo *ExternHeader2.mqh*, la variable externa homónima tiene otro tipo: *extern short y*. Si esta última descripción no se «moviera» a un comentario de forma preventiva, se produciría aquí el error de incompatibilidad de tipos («variable 'y' ya definida con un tipo diferente»). En resumen: o los tipos deben coincidir o las variables no deben ser externas. Si ambas opciones no son buenas, significa que hay un error tipográfico en el nombre de una de las variables.

Además, cabe señalar que la variable *y* no se inicializa explícitamente. Sin embargo, el script principal la invoca con éxito e imprime 0 en el registro:

```
long y;

void OnStart()
{
   ...
   Print(y); // 0
}
```

Por último, existe la posibilidad prevista en el script de probar una alternativa de las variables gemelas externas, ejemplificada por la variable ya conocida *x*. En lugar de describir *extern int x*, cada uno de los archivos *ExternHeader1.mqh* y *ExternHeader2.mqh* puede incluir otra cabecera común, *ExternCommon.mqh*, en la que figura la descripción de *int x* (sin *extern*). Se convierte en la única descripción de *x* en el proyecto.

Este modo alternativo de ensamblar el programa se habilita al activar la macro USE_INCLUDE_WORKAROUND: esto se encuentra en el comentario al principio del script:

```
#define USE_INCLUDE_WORKAROUND // this string was in the comment
#include "ExternHeader1.mqh"
#include "ExternHeader2.mqh"
```

En esta configuración, los archivos de inclusión particulares seguirán siendo compilables, como también lo es el proyecto entero. En un proyecto real, sin utilizar este método, el archivo mqh común se incluiría en *ExternHeader1.mqh* y *ExternHeader2.mqh* de forma incondicional (sin condiciones USE_INCLUDE_WORKAROUND). En este ejemplo, la conmutación entre los dos hilos de instrucciones se basa en USE_INCLUDE_WORKAROUND y sólo es necesaria para demostrar ambos modos. Por ejemplo, la versión simplificada de *ExternHeader2.mqh* debería aparecer de la siguiente manera:

```
// ExternHeader2.mqh
#include "ExternCommon.mqh" // int x; now here

void dec()
{
   x--;
}
```

Podemos comprobar en el registro de MetaEditor que el archivo *ExternCommon.mqh* sólo se ha cargado una vez, aunque se hace referencia a él tanto en *ExternHeader1.mqh* como en *ExternHeader2.mqh*.

```
'ExternMain.mq5'
'ExternHeader1.mqh'
'ExternCommon.mqh'
'ExternHeader2.mqh'
code generated
```

Si la variable *x* está «registrada» en *ExternCommon.mqh* no la redefiniremos (sin *extern*) en la unidad principal, ya que esto provocaría un error de compilación, pero podemos simplemente asignarle el valor deseado al principio del algoritmo.

2.4 Arrays

Un array es una herramienta para el almacenamiento basado en clúster y el procesamiento de los datos de tipos aleatorios. Son compatibles prácticamente con cualquier lenguaje de programación y

resultan especialmente importantes en MQL5 porque representan un método conveniente de organizar datos en serie relevantes para las tareas de trading. Las cotizaciones, las lecturas de indicadores, el historial de operaciones de trading de la cuenta con órdenes y transacciones, y las noticias son todos ellos ejemplos de datos en serie, es decir, secuencias de valores que varían en el tiempo.

El array puede considerarse una variable contenedora: puede contener una cantidad predefinida de valores del mismo tipo, que se identifican tanto por su nombre como por su índice (número de posición).

En esta sección vamos a ver la sintaxis habitual de la descripción de arrays y de las llamadas a los mismos, ejemplificada por tipos de datos integrados. Más adelante en este libro, con la adquisición de información sobre cómo ampliar el sistema de tipos gracias a la tecnología orientada a objetos, utilizaremos arrays junto con ello para acceder a nuevas oportunidades.

2.4.1 Características de los arrays

Antes de dar cuenta de las particularidades sintácticas de la declaración de arrays en MQL5 y de cómo trabajar con ellos, vamos a analizar algunos conceptos básicos de la construcción de arrays.

La característica principal de un array es el número de dimensiones. En un array unidimensional, sus elementos se colocan de uno en uno, como una fila de soldados, y un solo número (índice) basta para referirse a ellos. En un array de este tipo pueden guardarse los precios barra por barra de la apertura de un instrumento financiero para la profundidad dada del historial.

En un array bidimensional, sus elementos divergen en dos direcciones lógicamente perpendiculares, formando una especie de cuadrado (o rectángulo, en un caso general) en el que se necesitan dos índices para cada elemento, es decir, uno en cada dimensión. Un array de este tipo podría utilizarse para guardar quads de precios (Open, High, Low y Close) para cada barra del historial. Los números de barra se contarían con la primera dimensión, mientras que la segunda se utiliza para los números del 0 al 3, que indican uno de los tipos de precio.

Un array tridimensional es el equivalente de un cubo (o, más estrictamente en términos de geometría, un paralelepípedo rectángulo) con tres ejes. Siguiendo con el ejemplo del array de precios barra por barra, podríamos añadirle la tercera dimensión responsable de iterar los instrumentos financieros desde Observación de Mercado.

Para cada dimensión, el array tiene una longitud (tamaño) determinada que establece el rango de índices posibles. Si se supone que el historial se carga para 1000 barras y 10 instrumentos, obtendríamos un array de 1000 elementos de tamaño en la primera dimensión, 4 elementos en la segunda (OHLC) y 10 en la tercera.

El producto de los tamaños en todas las dimensiones proporciona el número total de elementos del array; en nuestro caso, es 40 000. En MQL5, no puede superar 2147483647 (máximo para int).

Ya de por sí es difícil imaginar una forma sólida para un array de 4 dimensiones porque vivimos en un mundo tridimensional. Sin embargo, MQL5 permite crear arrays de hasta cuatro dimensiones.

Cabe señalar que siempre se puede utilizar un array unidimensional en lugar de uno multidimensional con un número aleatorio de dimensiones, incluso más de 4. Se trata simplemente de organizar el recálculo de varios índices en uno continuo. Por ejemplo, si un array bidimensional tiene 10 columnas (dimensión 1, eje X) y 5 filas (dimensión 2, eje Y), puede transformarse en un array unidimensional con la misma cantidad de elementos, es decir, 50. En este caso, el índice del elemento se obtendrá mediante la siguiente fórmula:

```
index = Y * N + X
```

Aquí, N es el número de elementos en la primera dimensión, en nuestro caso, 10; se trata del tamaño de cada fila; Y es el número de fila (0..4), y X es el número de columna (0..9) en la fila.

Los tamaños entre dimensiones son otra característica que separa un array de una variable. Así, el número de dimensiones y el tamaño de cada dimensión deben especificarse de algún modo en la descripción, junto con el nombre del array y el tipo de datos (véase la sección siguiente).

Debe distinguir entre el tamaño de una variable (elemento del array) en bytes y el de un array como número de elementos que lo componen. En teoría, el tamaño total del array en términos de la memoria que consume debe ser el producto del tamaño de un elemento (dependiendo del tipo de datos) y el número de elementos. No obstante, esta fórmula no siempre funciona en la práctica. En particular, dado que las cadenas pueden tener distintas longitudes, resulta bastante difícil evaluar el volumen de memoria consumido por un array de cadenas.

Según el método de asignación de memoria, los arrays pueden ser dinámicos o de tamaño fijo.

En el código se describe un array de tamaño fijo con los tamaños exactos en todas las dimensiones. Es imposible cambiar su tamaño posteriormente. Sin embargo, a menudo se dan tareas prácticas en las que la cantidad de datos que se deben procesar es contingente y, por lo tanto, lo deseable sería redimensionar el array durante la operación del algoritmo. Para ello existen arrays dinámicos. Como veremos más adelante, estos arrays se describen sin especificar el tamaño de la primera dimensión y se pueden después «estirar» o «compactar» por medio de las funciones especiales de la API de MQL5.

En la documentación de MQL5 se utiliza terminología ambigua que denomina estáticos a los arrays de tamaño fijo. Este concepto también se utiliza para el modificador 'static' que se puede aplicar al array. Si un array de este tipo se declara dinámico, entonces es al mismo tiempo no estático en términos de asignación de memoria y estático en términos del modificador 'static'. Para evitar ambigüedades, el carácter estático en este libro sólo hará alusión al atributo de declaración.

Además de arrays dinámicos y de tamaño fijo, en MQL5 existen arrays especiales para almacenar cotizaciones y los búferes de los indicadores técnicos. Dichos arrays se denominan arrays de series temporales, ya que sus índices se corresponden con el tiempo. De hecho, estos arrays son unidimensionales y dinámicos. Sin embargo, a diferencia de otros arrays dinámicos, el propio terminal les asigna memoria. Los examinaremos en las secciones dedicadas a series temporales e indicadores.

2.4.2 Descripción de arrays

La descripción de arrays hereda algunas características de las descripciones de variables. Para empezar, debemos tener en cuenta que los arrays pueden ser globales y locales, según el lugar de su declaración. De forma similar a las variables, los modificadores *const* y *static* también pueden utilizarse para describir un array. Para un array unidimensional de tamaño fijo, la sintaxis de declaración es la siguiente:

```
type static1D[size];
```

Aquí, *type* y *static1D* denotan el nombre de tipo de los elementos y el identificador del array, respectivamente, mientras que *size* entre corchetes es una constante entera que define el tamaño.

Para los arrays multidimensionales deben especificarse varios tamaños, en función de la cantidad de dimensiones:

```
type static2D[size1][size2];
type static3D[size1][size2][size3];
type static4D[size1][size2][size3][size4];
```

Los arrays dinámicos se describen de forma similar, salvo que se realiza un salto en los primeros corchetes (antes de utilizar un array de este tipo, debe asignársele el volumen de memoria necesario mediante la función *ArrayResize*; véase la sección relativa a arrays dinámicos).

```
type dynamic1D[];
type dynamic2D[][size2];
type dynamic3D[][size2][size3];
type dynamic4D[][size2][size3][size4];
```

Para arrays de tamaño fijo se permite la inicialización: los valores iniciales se especifican para los elementos situados tras el signo igual, como una lista separada por comas, encerrándose toda la lista entre llaves. Por ejemplo:

```
int array1D[3] = {10, 20, 30};
```

Aquí, un array de enteros de 3 tamaños toma los valores 10, 20 y 30.

Con una lista de inicialización, no es necesario especificar entre corchetes el tamaño del array (para la primera dimensión). El compilador evaluará el tamaño automáticamente por la longitud de la lista. Por ejemplo:

```
int array1D[] = {10, 20, 30};
```

Los valores iniciales pueden ser tanto constantes como expresiones constantes, es decir, fórmulas que el compilador puede calcular durante la compilación. Por ejemplo, el siguiente array se rellena con el número de segundos de un minuto, una hora, un día y una semana (la representación como fórmulas es más ilustrativa que 86400 o 604800):

```
int seconds[] = {60, 60 * 60, 60 * 60 * 24, 60 * 60 * 24 * 7};
```

Estos valores suelen diseñarse como una macro del preprocesador al principio del código, y luego el nombre de esta macro se inserta en todos los lugares del texto en que sea necesario. Esta opción se describe en la sección relativa al Preprocesador.

El número de elementos de inicialización no puede exceder el tamaño del array. De lo contrario, el compilador dará el mensaje de error «demasiados inicializadores». Si la cantidad de valores es menor que el tamaño del array, los elementos en reposo se inicializan por cero. Por lo tanto, existe una breve notación para inicializar todo el array con ceros:

```
int array2D[2][3] = {0};
```

O simplemente llaves vacías:

```
int array2D[2][3] = {};
```

Ello funciona independientemente del número de dimensiones.

Para inicializar arrays multidimensionales, las listas deben estar anidadas. Por ejemplo:

```
int array2D[3][2] = {{1, 2}, {3, 4}, {5, 6}};
```

Aquí, el tamaño de la primera dimensión del array es 3; por lo tanto, dos comas enmarcan 3 elementos dentro de las llaves externas. Sin embargo, como el array es bidimensional, cada uno de sus elementos

es a su vez un array, siendo 2 el tamaño de cada uno de ellos. Por eso cada elemento representa una lista entre llaves, y cada lista contiene 2 valores.

Supongamos que necesitamos un array transpuesto (el primer tamaño es 2 y el segundo, 3); entonces, su inicialización cambiará:

```
int array2D[2][3] = {{1, 3, 5}, {2, 4, 6}};
```

Podemos omitir uno o más valores en la lista de inicialización, si es necesario, una vez marcados sus lugares con comas. Todos los elementos omitidos también se inicializarán con cero.

```
int array1D[3] = {, , 30};
```

Aquí, los primeros elementos serán iguales a 0.

La sintaxis del lenguaje permite colocar una coma después del último elemento:

```
string messages[] =
{
  "undefined",
  "success",
  "error",
};
```

Esto simplifica la adición de nuevos elementos, especialmente para las entradas de varias cadenas. En concreto, si nos olvidamos de introducir una coma delante del nuevo elemento añadido en un array de cadenas, resultará que las cadenas antigua y nueva se fusionan dentro de un mismo elemento (con el mismo índice), mientras que no aparecerá ningún elemento nuevo. Además, algunos arrays pueden ser generados automáticamente (por otro programa o por macros). Por lo tanto, la apariencia unificada de todos los elementos es natural.

«Montón» y «pila»

Con arrays que pueden ser potencialmente grandes es importante hacer distinguir entre ubicación global y local en la memoria.

La memoria para las variables globales y los arrays está distribuida dentro del «montón», es decir, la memoria libre de que dispone el programa. Esta memoria no está limitada prácticamente por nada, aparte de las características físicas del ordenador y el sistema operativo. El nombre de «montón» se explica por el hecho de que el programa siempre asigna o desasigna áreas de memoria de distinto tamaño, lo que hace que las áreas libres estén dispersas aleatoriamente por todo el volumen.

Las variables locales y los arrays se encuentran en la pila, es decir, una zona de memoria limitada asignada previamente al programa, especialmente para los elementos locales. El nombre de «pila» deriva del hecho de que, durante la ejecución del algoritmo, tienen lugar las llamadas anidadas a funciones,que acumulan sus datos internos según el principio de «apilamiento»: por ejemplo, el terminal llama a *OnStart*, una función de su código aplicado es invocada desde *OnStart*, después su otra función es invocada desde la anterior, etc. Al mismo tiempo, al entrar en cada función se crean sus variables locales, que siguen estando ahí cuando se llama a la función anidada. También crea variables locales, que entran en la pila algo por encima de las anteriores. Como resultado, una pila suele contener algunas capas de los datos locales de todas las funciones que se han activado en la ruta a la cadena de código actual. No es hasta que la función que se encuentra en la parte superior de la pila se haya completado que sus datos locales se eliminarán de ahí. En general, la pila es un almacenamiento que funciona según el principio FILO/LIFO (First In Last Out, Last In First Out).

Dado que el tamaño de la pila es limitado, se recomienda crear en ella sólo variables locales. No obstante, los arrays pueden ser lo bastante grandes como para agotar toda la pila muy pronto. Al mismo tiempo, la ejecución del programa finaliza con un error. Por lo tanto, debemos describir los arrays a nivel global como estáticos (*static*) o asignarles memoria dinámicamente (esto también se hace desde el montón).

2.4.3 Utilización de arrays

Los valores se escriben en, y se leen de, los elementos del array utilizando una sintaxis similar y especificando los índices necesarios entre corchetes. Para introducir un valor en un elemento, utilizaremos el método operación de asignación '='. Por ejemplo, para sustituir el valor del elemento 0 de un array unidimensional:

```
array1D[0] = 11;
```

La indexación empieza por 0. El índice del último elemento es igual a la cantidad de elementos menos 1. Por supuesto, podemos utilizar como índice tanto una constante como cualquier otra expresión que pueda reducirse al tipo entero (para más detalles sobre las expresiones, véase el capítulo siguiente), como una variable entera, una llamada a una función o un elemento de otro array con enteros (el direccionamiento indirecto).

```
int index;
// ...
// index = ... // assign an index somehow
// ...
array1D[index] = 11;
```

En el caso de los arrays multidimensionales, deben especificarse índices para todas las dimensiones.

```
array2D[index1][index2] = 12;
```

Los tipos enteros permitidos excluyen *long* y *ulong* para los índices. Si intentamos utilizar el valor de un «entero largo» como índice, este se convertirá implícitamente en *int*, por lo que el compilador emite el aviso «posible pérdida de datos debido a la conversión de tipos».

El acceso de lectura a los elementos del array se organiza según el mismo principio. Por ejemplo, así es como se puede imprimir un elemento del array en el registro:

```
Print(array2D[1][2]);
```

En el script *GoodTimes* hemos visto ya la descripción del array estático local *messages* con las cadenas de saludos (dentro de la función *Greeting*) y el uso de sus elementos en el operador *return*.

```
string Greeting()
{
  static int counter = 0;
  static const string messages[3] = // description
  {
    "Good morning", "Good day", "Good evening" // initialization
  };
  return messages[counter++ % 3];   // using
}
```

Al ejecutar *return* leemos el elemento que tiene el índice definido por la expresión *counter++ % 3*. El módulo de la división, 3, (denotado como '%') asegura que *counter*, que se incrementará cada vez que se incremente en 1, se forzará al rango de los valores correctos de los índices: 0, 1 o 2. Si no hubiera módulos de división, el índice del elemento solicitado superaría el tamaño del array, empezando por la 4ª llamada de esta función. En tales casos, se produce el error de tiempo de ejecución del programa («array fuera de rango») y se descarga del gráfico.

La API de MQL5 incluye funciones universales para muchas operaciones con arrays: la asignación de memoria (para arrays dinámicos), el rellenado, la copia, ordenación y la búsqueda en arrays se tratan en la sección Trabajar con arrays. Sin embargo, ahora vamos a presentar una de ellas: *ArrayPrint* permite la impresión de los elementos del array en el registro en un formato conveniente (considerando las dimensiones).

El script *Arrays.mq5* muestra algunos ejemplos de descripción de arrays y los resultados se imprimen en el registro. Más adelante analizaremos las manipulaciones con los elementos de los arrays, después de haber estudiado los bucles y las expresiones.

```
void OnStart()
{
  char array[100];      // without initialization
  int array2D[3][2] =
  {
    {1, 2},             // illustrative formatting
    {3, 4},
    {5, 6}
  };
  int array2Dt[2][3] =
  {
    {1, 3, 5},
    {2, 4, 6}
  };
  ENUM_APPLIED_PRICE prices[] =
  {
    PRICE_OPEN, PRICE_HIGH, PRICE_LOW, PRICE_CLOSE
  };
  // double d[5] = {1, 2, 3, 4, 5, 6}; // error: too many initializers
  ArrayPrint(array);    // printing random "garbage" values
  ArrayPrint(array2D);  // showing the 2D array in the log
  ArrayPrint(array2Dt); // a "transposed" appearance of the same data 2D
  ArrayPrint(prices);   // getting to know the values of the price enumeration elemen
}
```

A continuación se representa una de las opciones de entrada en el registro.

```
[ 0]   0   0   0   0   0   0   0   0   0   0   0   0 -87 105  82 119   0
       0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
[34]   0   0   0 -32  -3  -1  -1   7   0   0   2   0   0   0   0   0   0
       0   2   0   0   0   0   0   0   0 -96 104  82 119   0   0   0   0
[68]   0   0   3   0   0   0   0   0  -1  -1  -1  -1   0   0   0   0 100
      48   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    [,0][,1]
[0,]   1   2
[1,]   3   4
[2,]   5   6
    [,0][,1][,2]
[0,]   1   3   5
[1,]   2   4   6
2 3 4 1
```

El array que lleva por nombre *array* no tiene ninguna inicialización y, por lo tanto, la memoria asignada al mismo puede contener valores aleatorios. Los valores cambiarán en cada ejecución del script. Se recomienda inicializar siempre los arrays locales, por si acaso.

Los arrays *array2D* y *array2Dt* se imprimen en el registro de forma ilustrativa, como matrices. Esto no tiene nada que ver con el hecho de que hayamos formateado las listas de inicialización en el código fuente de la misma manera.

El array *prices* tiene el tipo de la enumeración integrada ENUM_APPLIED_PRICE. Básicamente, los arrays pueden ser de cualquier tipo, lo que incluye estructuras, punteros a funciones y otras cosas que vamos a ver. Dado que las enumeraciones se basan en el tipo *int*, los valores se muestran por dígitos, no por los nombres de los elementos (para obtener el nombre de un elemento específico de la enumeración existe la función EnumToString, pero su modo no es compatible con *ArrayPrint*).

La cadena con la descripción del array *d* contiene un error: la entidad de valores iniciales excede el tamaño del array.

2.5 Expresiones

Las expresiones son elementos esenciales de cualquier lenguaje de programación. Cualquiera que sea la idea práctica que subyace en un algoritmo, al final este se reduce al tratamiento de datos, es decir, a cálculos. La expresión describe el cálculo de un resultado a partir de uno o varios valores predefinidos. Los valores se denominan operandos, mientras que las acciones que se realizan con ellos se denotan mediante operaciones u operadores.

Como operadores que permiten manipular con operandos, en las expresiones se utilizan caracteres independientes o secuencias de los mismos, como '+' para sumar o '*' para multiplicar. Todos ellos forman varios grupos, como aritmética, a nivel de bits, comparación y lógica, además de algunos especializados.

Ya hemos utilizado expresiones en las secciones anteriores de este libro, por ejemplo para inicializar variables. En el caso más sencillo, la expresión es una constante (literal) que es el único operando, mientras que el resultado del cálculo es igual al valor del operando. No obstante, los operandos también pueden ser variables, elementos de arrays, resultados de llamadas a funciones (para las que la función se llama directamente desde la expresión), expresiones anidadas y otras entidades.

Todos los operadores sustituyen (devuelven) su resultado en la expresión padre, directamente en el lugar en el que había operandos, lo que permite combinarlos formando estructuras jerárquicas bastante complejas. Por ejemplo, en la siguiente expresión, el resultado de multiplicar las variables *b* por *c* se suma al valor de la variable *a*, y a continuación el valor obtenido se almacenará en la variable *v*:

```
v = a + b * c;
```

En esta sección analizaremos los principios generales de la construcción y el cálculo de varias expresiones, así como el conjunto estándar de operadores admitidos en MQL5 para los tipos integrados. Más adelante, en la parte dedicada a la programación orientada a objetos (POO), descubriremos cómo se pueden recargar (redefinir) los operadores para tipos personalizados, es decir, estructuras y clases, lo que nos permitirá utilizar objetos en expresiones y realizar con ellos acciones no estándar.

2.5.1 Conceptos básicos

Antes de pasar a los grupos específicos de operadores, debemos introducir algunos conceptos básicos que son inherentes a todos los operadores y afectan a su aplicabilidad y comportamiento en un contexto determinado.

En primer lugar, por la cantidad de operandos requeridos, los operadores pueden ser unarios y binarios. Como se deduce de los nombres, los unarios procesan un operando, mientras que los binarios procesan dos. En el caso de los binarios, el operador se coloca siempre entre los operandos. Entre los unarios, hay operadores que deben colocarse delante del operando y otros que deben colocarse detrás. Por ejemplo, el operador unario menos ('-') permite invertir el signo del valor:

```
int x = 10;
int y = -x;  // -10
```

Al mismo tiempo, existe un operador binario para la resta que utiliza el mismo carácter, '-'.

```
int z = x - y; // 10 - -10 -> 20
```

La elección de un operador (acción) correcto por parte del compilador en un contexto específico viene determinada por el contexto de uso en la expresión.

A cada operador se le asigna una prioridad. Esta determina el orden en que se calcularán los operadores en expresiones complejas en las que haya varios operadores. Los operadores de más alta prioridad se calculan primero, mientras que los de menor prioridad lo hacen en último lugar. Por ejemplo, en la expresión *1 + 2 * 3* hay dos operaciones (suma y multiplicación) y tres operandos. Dado que la multiplicación tiene una prioridad superior a la de la suma, primero se hallará el producto de *2 * 3* y luego se le sumará uno.

Más adelante facilitaremos la tabla completa de operaciones con prioridades.

Además, cada operador se caracteriza por la asociatividad, que puede ser izquierda o derecha y determina el orden en que se ejecutan los operadores sucesivos que tienen la misma prioridad. Por ejemplo, la expresión *10 - 7 - 1* puede calcularse en teoría de dos maneras:

- Restar 7 a 10 y luego restar 1 al 3 resultante, lo que da 2; o bien,
- Restar 1 a 7, lo que da 6, y luego restar 6 a 10, lo que da 4.

En el primer caso, los cálculos se realizaron de izquierda a derecha, lo que se corresponde con la asociatividad izquierda; dado que la operación de resta es, de hecho, asociativa izquierda, la primera respuesta es correcta.

La segunda opción de cómputo se corresponde con la asociatividad derecha y no se utilizará.

Veamos otro ejemplo en el que intervienen simultáneamente la prioridad y la asociatividad: *11 + 5 * 4 / 2 + 3*. Ambos tipos de operaciones, es decir, la suma y la multiplicación, se ejecutan de izquierda a derecha. Si las prioridades no fueran diferentes, obtendríamos 35, aunque 24 es la respuesta correcta. Cambiando a la asociatividad derecha nos daría 14.

Para redefinir explícitamente las prioridades en las expresiones, se pueden utilizar, por ejemplo, paréntesis: *(11 + 5) * 4 / (2 + 3)*. Lo que se encierra entre paréntesis se calcula antes, y el resultado intermedio se sustituye en la expresión que se utilizará en otras operaciones. Los grupos entre paréntesis pueden anidarse. Para obtener más detalles, consulte la sección Agrupación con paréntesis.

Un operador asociativo derecho puede ejemplificarse con el operador unario de negación lógica, '!'. En esencial, su tarea consiste en hacer *true* a partir de *false*, y viceversa. Al igual que con otros operadores unarios, la asociatividad significa en este contexto en qué lado del operador debe colocarse el operando. El símbolo '!' se coloca delante del operando, es decir, el operando está a la derecha.

```
int x = 10;
int on_off = !!x;  // 1
```

En este caso, la negación lógica se realiza dos veces: la primera vez con respecto a la variable *x* ('!' a la derecha) y la segunda vez con respecto al resultado de la negación anterior ('!' a la izquierda). Esta doble negación permite transformar cualquier valor distinto de cero en 1 al convertirlo en *bool* y viceversa.

La tabla final de operaciones también mostrará asociatividad.

Para terminar, el último punto, pero no por ello menos importante, en materia de procesamiento de expresiones es el orden de cálculo de los operandos. Debe distinguirse de la prioridad que pertenece a la operación, no a los operandos. El orden de cálculo de los operandos de las operaciones binarias no se define explícitamente, lo que da espacio al compilador para optimizar el código y mejorar su eficacia. El compilador sólo garantiza que los operandos se calcularán antes de ejecutar la operación.

Existe un conjunto limitado de operaciones para las que se define el orden de evaluación de los operandos. En particular, para la lógica AND ('&&') y OR ('||'), este es de izquierda a derecha, y la parte derecha puede omitirse si no afecta en nada debido al valor de la parte izquierda. Pero en que respecta al operador condicional ternario '?:', el orden es aún más intrincado, ya que se calculará una u otra rama al computar las primeras condiciones, dependiendo de su veracidad. Consulte las secciones siguientes para obtener más información.

El orden de evaluación de los operandos queda ilustrado por la situación en la que hay varias llamadas a una función en la expresión. Por ejemplo, utilicemos 4 funciones en la siguiente expresión:

```
a() + b() * c() - d()
```

Las reglas de prioridad y asociatividad sólo se utilizarán para los resultados intermedios de llamar a estas funciones, mientras que las llamadas en sí pueden ser generadas por el compilador en cualquier orden que «considere necesario» en función de las características del código fuente y la configuración del compilador. Por ejemplo, las funciones *b* y *c* que intervienen en la multiplicación pueden llamarse en el orden [*b(), c()*] o al revés, [*c(), b()*]. Si las funciones durante su ejecución pueden afectar a los mismos datos, su estado será ambiguo al calcular la expresión.

Un problema similar puede observarse al trabajar con arrays y operadores de incremento (véase Incremento y decremento).

```
int i = 0;
int a[5] = {0, 1, 2, 3, 4};
int w = a[++i] - a[++i];
```

Dependiendo de si se calcula en primer lugar el operando de diferencia izquierdo o derecho, podemos obtener -1 (*a[1] - a[2]*) o +1 (*a[2] - a[1]*). Dado que el compilador MQL5 no deja de mejorar, nada garantiza que el resultado actual (-1) se mantenga en el futuro.

Para evitar posibles problemas se recomienda no utilizar un operando repetidamente si ya ha sido modificado en la misma expresión.

Normalmente, en todas las expresiones puede haber operandos de distintos tipos. Esto lleva a la necesidad de convertirlos en un determinado tipo común antes de realizar cualquier acción con ellos. Si no hay conversión explícita de tipos, MQL5 realiza la conversión implícita cuando es necesario. Además, las reglas de conversión son diferentes para las distintas combinaciones de tipos. La conversión de tipos explícita e implícita se examina en la sección correspondiente.

2.5.2 Operación de asignación

Los resultados de los cálculos de las expresiones deben, normalmente, almacenarse en algún lugar. El operador de asignación denotado por '=' está pensado para este fin en el lenguaje. A la izquierda del mismo se coloca el nombre de una variable o de un elemento del array, en el que debe almacenarse el resultado, mientras que a la derecha se sitúa la expresión (de hecho, la fórmula de cálculo).

Ya hemos utilizado este operador para la inicialización de variables, que se ejecuta una sola vez, durante la creación de las mismas. No obstante, la asignación permite cambiar los valores de las variables en el transcurso del algoritmo un número arbitrario de veces. Por ejemplo:

```
int z;
int x = 1, y = 2;
z = x;
x = y;
y = z;
```

Las variables *x* e *y* se inicializaron con los valores 1 y 2, tras lo cual se utilizó la tercera variable auxiliar *z* y tres asignaciones para intercambiar los valores *x* e *y*.

El operador de asignación, como todos los operadores, devuelve su resultado en la expresión. Esto permite escribir las tareas en una secuencia.

```
int x, y, z;
x = y = z = 1;
```

Aquí, 1 se asignará primero a la variable *z*, luego a la variable *y* y, finalmente, a la variable *x*. Obviamente, este operador es asociativo a la derecha, porque el valor que se asigna se desplaza de derecha a izquierda en la expresión.

Podemos utilizar la asignación como parte de una expresión. Pero, como su prioridad es inferior a la de todos los demás operadores (salvo el de la «coma»; véase Prioridades de las operaciones), debe ir entre paréntesis (para obtener más detalles, consulte la sección sobre Agrupación con paréntesis). Este aspecto permite situaciones en las que los errores tipográficos, como '=' en lugar de '==', en las expresiones hacen que no se ejecuten las sentencias según lo previsto. Véase el ejemplo de este comportamiento en la sección dedicada a la sentencia*if*.

El operador de asignación impone ciertas limitaciones a lo que puede haber a la izquierda de '=' y a lo que puede haber a su derecha. En programación, estas entidades destinadas a simplificar el almacenamiento se denominan precisamente de la siguiente manera: LValue y RValue (basados en Left y Right).

LValue y RValue

LValue representa una entidad a la que se asigna memoria y, por lo tanto, se puede escribir en ella un valor. Los elementos variables y del array son los ejemplos conocidos de LValue. Una vez vista la POO, descubriremos otro representante de esta categoría, el objeto, en el que se puede recargar el operador de asignación. Un elemento obligatorio de LValue es la presencia de un identificador.

Hay que tener en cuenta que las variables y los arrays pueden describirse con la palabra clave const, y entonces no pueden actuar como LValue, ya que la modificación de constantes está prohibida.

RValue es un valor temporal que se utiliza en una expresión, como un literal o un valor devuelto debido a una llamada a una función o debido al cálculo de un fragmento de la expresión.

La categoría LValue es de carácter expansivo, es decir, permite colocar el objeto correspondiente a la izquierda de '=' pero no prohíbe utilizarlo, a la par de RValue, a la derecha de '='.

La categoría RValue, una vez más, tiene carácter limitativo, es decir, cualquier RValue sólo puede estar a la derecha de '='.

Como se utiliza un determinado elemento LValue a la derecha de '=', su identificador, de hecho, denota su contenido actual colocado en la fórmula de la expresión.

Sin embargo, si se utiliza un elemento de LValue a la izquierda de '=', su identificador indica una dirección de memoria (celda) en la que debe escribirse el nuevo valor (resultado del cálculo de la expresión).

Los distintos operadores tienen diferentes limitaciones en cuanto a si pueden utilizarse para los operandos de LValue o RValue. Por ejemplo, los operadores de incremento '++' y decremento '--' (véase Incremento y decremento) sólo puede utilizarse con LValue.

He aquí algunos ejemplos de lo que está y no está permitido hacer con los operadores de asignación (script *ExprAssign.mq5*):

```
// description of variables
const double cx = 123.0;
int x, y, a[5] = {1};
string s;
// assignment
a[2] = 21;       // ok
x = a[0] + a[1] + a[2]; // ok
s = Symbol();    // ok
cx = 0;          // const variable may not be changed
                 // error: 'cx' - constant cannot be modified
5 = y;           // 5 — this number (literal)
                 // error: '5' - l-value required
x + y = 3;       // to the left of RValue (expression computation result)
                 // error: l-value required
Symbol() = "GBPUSD"; // to the left of RValue with the function call result
                     // error: l-value required
```

El compilador devuelve un error de incumplimiento de las reglas de uso de operadores.

2.5.3 Operaciones aritméticas

Las operaciones aritméticas incluyen 5 binarias, es decir, suma, resta, multiplicación, división y módulo de división, y 2 unarias, esto es, más y menos. Los símbolos utilizados para cada una de esas operaciones figuran en la tabla siguiente.

En la columna de ejemplos, *e1* y *e2* son subexpresiones arbitrarias. La asociatividad se marca con 'L' (de izquierda a derecha) y 'R' (de derecha a izquierda). El número de la primera columna puede considerarse la prioridad de ejecución de las operaciones.

P	Símbolos	Descripción	Ejemplo	Orden de ejecución
2	+	Unario más	+e1	D-I
2	-	Unario menos	-e1	D-I
3	*	Multiplicación	e1 * e2	I-D
3	/	División	e1 / e2	I-D
3	%	Módulo de división	e1 % e2	I-D
4	+	Adición	e1 + e2	I-D
4	-	Resta	e1 - e2	I-D

El orden en la tabla se corresponde con la disminución de las prioridades: Unario más y menos se calculan antes de la multiplicación y la división, mientras que estas últimas, a su vez, antes de la suma y la resta.

```
double a = 3 + 4 * 5; // a = 23
```

De hecho, unario más no tiene ningún efecto en los cálculos, pero puede utilizarse para una mejor visualización de la expresión. Unario menos invierte el signo de su operando.

Las operaciones aritméticas se utilizan para los tipos numéricos o aquellos que pueden convertirse en ellos. El resultado del cálculo es un RValue. En computación, las posiciones de almacenamiento de los operandos enteros se extienden a menudo hasta el «mayor» de los enteros utilizados o hasta *int* (si todos los tipos de enteros fueran de menor tamaño), y también se convierten en un tipo común. Encontrará más información en la sección Conversión de tipos.

```
bool b1 = true;
bool b2 = -b1;
```

En este ejemplo, la variable *b1* se «expande» al tipo *int* con valor 1. La inversión del signo da -1, que en la conversión de tipos inversa a *bool* da *true* (porque -1 no es cero). El uso de tipos lógicos en cálculos aritméticos no es bienvenido.

Al dividir números enteros se obtiene un número entero, es decir, se omite la parte fraccionaria, si la hay. Ello puede comprobarse con el script *ExprArithmetic.mq5*.

```
int a = 24 / 7;      // ok: a = 3
int b = 24 / 8;      // ok: b = 3
double c = 24 / 7;   // ok: c = 3 (!)
```

Aunque la variable *c* se describe como *double*, hay enteros en la expresión para inicializarla; por lo tanto, la división realizada es un entero. Para realizar una división con una parte fraccionaria, al menos un operando debe ser de tipo real (el segundo también se convertirá a él).

```
double d = 24.0 / 7; // ok: d = 3.4285714285714284
```

El operador '%' calcula el resto de una división de enteros (sólo es aplicable a dos operandos de tipo entero).

```
int x = 11 % 5;   // ok: x = 1
int y = 11 % 5.0; // no real number can be used
                  // error: '%' - illegal operation use
```

Cuando los operandos tienen signos diferentes, los operadores '*' y '/' dan un número negativo. Las siguientes reglas se aplican al operador '%':

- Si el divisor del operador '%' es negativo, el signo "escapa";
- Si el dividendo del operador '%' es negativo, el resultado es negativo.

Esto es fácil de comprobar utilizando el cálculo alternativo del módulo de división: *m % n = m - m / n * n*. Debe tenerse en cuenta que la división *m / n* para números enteros se redondeará; por lo tanto, *m / n * n* no es igual a *m*, en el caso general.

En la sección Características de los arrays nos adentramos en la idea de que un array multidimensional podría representarse por medio de uno unidimensional debido al recálculo de los índices de sus elementos. También proporcionamos la fórmula para obtener un índice en un array unidimensional por medio de las coordenadas (número de columna X y número de fila Y a la longitud de cadena N) del array bidimensional.

```
index = Y * N + X
```

La operación '%' nos permite realizar un cálculo hacia atrás más conveniente, es decir, encontrar X e Y mediante el índice:

```
Y = index / N
X = index % N
```

Si se ha obtenido un resultado NaN (Not A Number, tal como infinito, raíz cuadrada de un número negativo, etc.) no presentable en algún momento del cálculo de la expresión, todas las operaciones posteriores con ella producirán también un NaN. Puede distinguirse de un número normal utilizando la función MathIsValidNumber (véase Funciones matemáticas).

```
double z = DBL_MAX / DBL_MIN - 1; // inf: Not A Number
```

Aquí, se resta de NaN (obtenido de la división) y vuelve a dar el NaN.

La operación de suma se define para cadenas y realiza la concatenación, es decir, las combina.

```
string s = "Hello, " + "world!"; // "Hello, World!"
```

Otras operaciones están prohibidas para las cadenas.

2.5.4 Incremento y decremento

Los operadores de incremento y decremento permiten escribir de forma simplificada el aumento o disminución en 1 de un operando. Estos aparecen la mayor parte de las veces dentro de bucles para modificar los índices al acceder a arrays u otros objetos que admitan la enumeración.

El incremento se indica mediante dos signos + consecutivos: '++'. La disminución se denota por dos signos menos consecutivos: '--'.

Existen dos tipos de operadores como estos: prefijo y postfijo.

Los operadores prefijos, como su nombre indica, se escriben delante del operando (*++x*, *--x*). Cambian el valor del operando, y este nuevo valor interviene en los cálculos posteriores de la expresión.

Los operadores postfijos se escriben detrás del operando (*x++*, *x--*). Sustituyen la copia del valor del operando actual en la expresión y luego cambian su valor (el nuevo valor no entra en la expresión). Encontrará ejemplos sencillos en el script *ExprIncDec.mq5*.

```
int i = 0, j;
j = ++i;       // j = 1, i = 1
j = i++;       // j = 1, i = 2
```

La forma postfija puede ser útil para una escritura más compacta de expresiones que combinan una referencia al valor precedente del operando y su modificación lateral (se necesitarían dos sentencias distintas para hacer un registro alternativo del mismo). En todos los demás casos se recomienda utilizar la forma de prefijo (no crea una copia temporal del valor «antiguo»).

En el siguiente ejemplo se invierte el signo en los elementos del array de forma consecutiva hasta que se encuentra el elemento de posición cero. El desplazamiento por los índices del array se garantiza mediante el post-incremento *k++* dentro del bucle *while*. Debido al postfijo, la expresión *a[k++] = -a[k]* actualiza primero el k-ésimo elemento y luego incrementa *k* en 1. A continuación, se comprueba que el resultado de la asignación no sea igual a cero (!= 0, véase la sección siguiente).

```
int k = 0;
int a[] = {1, 2, 3, 0, 5};
while((a[k++] = -a[k]) != 0){}
// a[] = {-1, -2, -3, 0, 5};
```

En la siguiente tabla se muestran los operadores de incremento y decremento por orden de prioridad:

P	Símbolos	Descripción	Ejemplo	Orden de ejecución
1	++	Post-incremento	e1++	I-D
1	--	Post-decremento	e1--	I-D
2	++	Incremento prefijo	++e1	D-I
2	--	Decremento prefijo	--e1	D-I

Todas las operaciones de incremento y decremento tienen una prioridad superior a las operaciones aritméticas. Los prefijos tienen menos prioridad que los postfijos. En el siguiente ejemplo, el valor «antiguo» de *x* se suma con el valor de *y*, a partir del cual se incrementa *x*. Si la prioridad del prefijo fuera mayor se realizaría el incremento de *y*, al que el nuevo valor, 6, se añadiría a *x*, y obtendríamos *z = 6, x = 0* (anterior).

```
int x = 0, y = 5;
int z = x+++y; // "x++ + y" : z = 5, x = 1
```

2.5.5 Operaciones de comparación

Como su nombre indica, estas operaciones están pensadas para comparar dos operandos y devolver una función lógica, *true* o *false*, dependiendo de la condición que se deba mantener en la comparación.

En la siguiente tabla se muestran todas las operaciones de comparación y sus propiedades, tales como símbolos utilizados, prioridades, ejemplos y asociatividad.

P	Símbolos	Descripción	Ejemplo	Orden de ejecución
6	<	Menos	e1 < e2	I-D
6	>	Mayor	e1 > e2	I-D
6	<=	Menor o igual que	e1 <= e2	I-D
6	>=	mayor o igual que	e1 >= e2	I-D
7	==	Igualdad	e1 == e2	I-D
7	!=	No es igual	e1 != e2	I-D

El principio de cada operación es comparar dos operandos utilizando el criterio de la columna que contiene su descripción. Por ejemplo, la entrada «*x* < *y*» significa comprobar si «*x* es menor que *y*». En consecuencia, el resultado de la comparación será *true* si *x* es realmente menor que *y*, y *false* en todos los demás casos.

Las comparaciones funcionan para operandos de cualquier tipo (para tipos diferentes se realiza una conversión de tipos).

Teniendo en cuenta la asociatividad izquierda y la devolución del resultado de tipo *bool*, construir una secuencia de comparaciones no funciona de forma tan obvia. Por ejemplo, una expresión hipotética para comprobar si el valor *y* se encuentra entre los valores de *x* y *z*, podría aparecer de la siguiente manera:

```
int x = 10, y = 5, z = 2;
bool range = x < y < z;   // true (!)
```

Sin embargo, dicha expresión se procesa de manera diferente. Hasta el compilador lo distingue con el aviso «Uso inseguro del tipo 'bool' en la operación».

Debido a la asociatividad izquierda, la condición izquierda *x* < *y* se comprueba primero, y su resultado se sustituye como valor temporal del tipo *bool* en la expresión que se desarrolla de la siguiente manera: *b* < *z*. A continuación, el valor de *z* se compara con *true* o *false* en la variable temporal *b*. Para comprobar si *y* se encuentra entre *x* y *z* debe utilizar dos operaciones de comparación combinadas con la operación lógica AND (se analizará en la sección siguiente).

```
int x = 10, y = 5, z = 2;
bool range = x < y && y < z;   // false
```

Al utilizar la comparación para igualdad/desigualdad se tendrán en cuenta las características de los tipos de operandos. Por ejemplo, los números de punto flotante contienen a menudo valores «aproximados» después de los cálculos (hemos tenido en cuenta la precisión de la representación de *double* y *float* en la sección Números reales). Por ejemplo, la suma de 0.6 y 0.3 no es 0.9 en sentido estricto:

```
double p = 0.3, q = 0.6;
bool eq = p + q == 0.9;         // false
double diff = p + q - 0.9;      // -0.00000000000000111
```

La diferencia hace $1*10^{-16}$, pero es suficiente para que la operación de comparación devuelva false.

Por lo tanto, los números reales deben compararse para ver su igualdad o desigualdad utilizando los operadores mayor-/menor-entonces para averiguar su diferencia y desviación aceptable que se determina manualmente, basándose en las características del cálculo, o se toma una universal. Recordemos que para *double* y *float* se definen las constantes de precisión integradas, DBL_EPSILON y FLT_EPSILON, válidas para el valor 1.0. Deben escalarse para comparar otros valores. En el guión *ExprRelational.mq5* se presenta una de las posibles realizaciones de la función *isEqual* para comparar números reales, que considera este aspecto.

```
bool isEqual(const double x, const double y)
{
   const double diff = MathAbs(x - y);
   const double eps = MathMax(MathAbs(x), MathAbs(y)) * DBL_EPSILON;
   return diff < eps;
}
```

Aquí utilizamos la función de obtener un valor absoluto sin signo (*MathAbs*) y el mayor de los dos valores (*MathMax*). Se describirán en la sección Funciones matemáticas de la Parte 4. La diferencia absoluta entre los parámetros de la función *isEqual* se compara con la tolerancia calibrada en la variable *eps* mediante la operación '<'.

En cualquier caso, esta función no puede utilizarse para comparar con el cero absoluto. Para ello, puede utilizar el siguiente planteamiento (probablemente requerirá alguna adaptación a sus necesidades específicas):

```
bool isZero(const double x)
{
   return MathAbs(x) < DBL_EPSILON;
}
```

Las cadenas se comparan lexicográficamente, es decir, letra por letra. El código de cada carácter se compara con el código del carácter situado en la misma posición de la segunda cadena. La comparación se realiza hasta que se encuentra una diferencia en los códigos o una de las cadenas termina. La proporción de la cadena será igual a la de los primeros caracteres diferentes, o se considerará que una cadena más larga es mayor que la más corta. Recuerde que las mayúsculas y las minúsculas tienen códigos diferentes y, por extraño que parezca, las mayúsculas tienen códigos más pequeños que las minúsculas.

Una cadena vacía "" (de hecho, almacena un terminal 0) no es igual al valor especial de NULL que significa que no hay cadena.

```
bool cmp1 = "abcdef" > "abs";     // false, [2]: 's' > 'c'
bool cmp2 = "abcdef" > "abc";     // true,  by length
bool cmp3 = "ABCdef" > "abcdef";  // false, by case
bool cmp4 = "" == NULL;           // false
```

Además, para comparar cadenas, MQL5 proporciona algunas funciones que se describirán en la sección Trabajar con cadenas.

Al comparar por igualdad/desigualdad, no se recomienda utilizar constantes *bool* : *true* o *false*. La cuestión es que, en expresiones como *v == true* o *v == false*, el operando *v* puede interpretarse intuitivamente como un tipo lógico, mientras que en realidad es un número. Como es sabido, el valor cero se considera *false* en los números, mientras que todos los demás se interpretan como *true* (a menudo queremos utilizarlo como indicación de que algún resultado está presente o ausente). Sin embargo, en este caso, la conversión de tipos va hacia atrás: *true* o *false* se «expanden» a un tipo numérico *v* y se hacen de hecho iguales a 1 y 0, respectivamente. Una comparación de este tipo tendrá un resultado distinto del esperado (por ejemplo, la comparación *100 == true* resultará falsa).

2.5.6 Operaciones lógicas

Las operaciones lógicas realizan cálculos sobre operandos lógicos y devuelven un resultado del mismo tipo.

P	Símbolos	Descripción	Ejemplo	Orden de ejecución
2	!	Negación lógica NOT	!e1	D-I
11	&&	AND lógico	e1 && e2	I-D
12	||	OR lógico	e1 || e2	I-D

La negación lógica NOT transforma *true* en *false* y *false* en *true*.

El operador lógico AND es igual a *true* si ambos operandos son iguales a *true*.

El operador lógico OR es igual a *true* si al menos un operando es igual a *true*.

Los operadores AND y OR siempre calculan los operandos de izquierda a derecha y, si es posible, utilizan el método abreviado de cálculo. Si el operando de la izquierda es igual a *false*, entonces el operador AND omite el segundo operando, ya que no afecta a nada: el resultado ya es *false*. Si el operando de la izquierda es igual a *true*, entonces el operador OR omite el segundo operando por la misma razón, ya que el resultado será, en cualquier caso, igual a *true*.

Esto se utiliza a menudo en los programas para evitar errores en el segundo operando (y siguientes). Por ejemplo, podemos protegernos contra el error de acceder a un elemento de array inexistente:

```
index < ArraySize(array) && array[index] != 0
```

Aquí utilizamos la función integrada *ArraySize* que devuelve la longitud de la matriz. Sólo si *index* es menor que la longitud se lee el elemento con este índice y se compara con cero.

También se utiliza la comprobación por contrarios, utilizando '||', por ejemplo:

```
ArraySize(array) == 0 || array[0] == 0
```

La condición es verdadera inmediatamente si el array es nulo. Y sólo si hay elementos continuará la comprobación adicional del contenido.

Si la expresión consta de varios operandos combinados mediante OR lógico, con el primer *true* (si existe) se obtendrá inmediatamente el resultado total de *true*. Sin embargo, si los operandos se combinan mediante AND lógico, entonces con el primer *false* se obtendrá inmediatamente el resultado total de *false*.

Por supuesto, puede combinar diferentes operaciones dentro de una misma expresión, teniendo en cuenta su diferente prioridad: Primero se ejecuta la negación, después las condiciones relacionadas con AND y al final las relacionadas con OR. Si se requiere otra secuencia, debe especificarse explícitamente mediante paréntesis.

Por ejemplo, la siguiente expresión sin paréntesis, A && B || C && D, es de hecho equivalente a (A && B) || (C && D). Para que el OR lógico se ejecute como el primero, debe escribirse entre paréntesis: A && (B || C) && D. Para obtener más detalles sobre el uso de paréntesis, consulte la sección Agrupación con paréntesis.

En el script *ExprLogical.mq5* se ofrecen ejemplos sencillos para comprobar en la práctica las operaciones lógicas.

```
int x = 3, y = 4, z = 5;
bool expr1 = x == y && z > 0;  // false, x != y, z does not matter
bool expr2 = x != y && z > 0;  // true,  both conditions are complied with
bool expr3 = x == y || z > 0;  // true,  it is sufficient that z > 0
bool expr4 = !x;               // false, x must be 0 to get true
bool expr5 = x > 0 && y > 0 && z > 0; // true, all 3 are complied with
bool expr6 = x < 0 || y > 0 && z > 0; // true, y and z are sufficient
bool expr7 = x < 0 || y < 0 || z > 0; // true, z is sufficient
```

En la cadena de cálculo *expr6*, el compilador emite el aviso: «Compruebe la precedencia de los operadores para detectar posibles errores; utilice paréntesis para aclarar la precedencia».

Las operaciones lógicas '&&' y '||' no deben mezclarse con las operaciones a nivel de bits '&' y '|' (se analizan en la sección siguiente).

2.5.7 Operaciones a nivel de bits

A veces es necesario procesar los números a nivel de bits. Para ello, existe un grupo de operaciones a nivel de bits aplicables a los tipos enteros.

Todos los símbolos y descripciones de los operadores a nivel de bits se proporcionan con su asociatividad y por orden de prioridad en la tabla siguiente.

P	Símbolos	Descripción	Ejemplo	Orden de ejecución
2	~	Complemento a nivel de bits (inversión)	~e1	D-I
5	<<	Desplazamiento a la izquierda	e1 << e2	I-D
5	>>	Desplazamiento a la derecha	e1 >> e2	I-D
8	&	AND a nivel de bits	e1 y e2	I-D
9	^	OR exclusivo a nivel de bits	e1 ^ e2	I-D
10	|	OR a nivel de bits	e1 | e2	I-D

De todo el grupo, sólo la operación '~' de complemento bit a bit es unaria, mientras que todas las demás son binarias.

En todos los casos, si el tamaño del operando es menor que *int/uint*, este se amplía de forma preliminar a *int/uint* añadiendo 0 bits en el orden superior. En función del tipo de operando con o sin signo, un bit de orden superior puede afectar al signo.

La aplicación estándar de Windows, Calculadora, puede ayudar a comprender la representación de los números a nivel de bits. Si selecciona el modo de funcionamiento Programador en el menú Ver aparecerán en el programa los grupos de botones de alternancia para seleccionar la representación del

número en forma hexadecimal (Hex), decimal (Dec), octal (Oct) o binaria (Bin). Es esta última la que muestra bits. Además, puede seleccionar el tamaño del número: 1, 2, 4 y 8 bytes. Los botones permiten ejecutar todas las operaciones consideradas: Not ('~'), And ('&'), Or ('|'), Xor ('^'), Lsh ('<<') y Rsh ('>>').

Dado que la calculadora utiliza números con signo, pueden aparecer valores negativos al cambiar al modo decimal (recuerde que el bit de orden superior se interpreta como un signo). Para facilitar el análisis es razonable excluir los menos que aparecen, para lo cual es necesario seleccionar el tamaño en bytes un grado superior. Por ejemplo, para comprobar los valores dentro del rango hasta 255 (uchar, entero de un byte sin signo), debe seleccionar 2 bytes (de lo contrario, sólo los valores decimales hasta 127 serán positivos, mientras que los demás se mostrarán en la región negativa).

El complemento a nivel de bits crea un valor en el que el bit 0 ocupa el lugar de todos los bits 1, mientras que el bit 1 ocupa el lugar de los bits 0. Por ejemplo, la negación de un byte con todos los bits cero da un byte con todos los bits 1. El número 50 aparece en el formato a nivel de bits como '00110010' (byte). Su inversión da '11001101'.

La unidad representada hexadecimalmente es 0x0001 (para *short*). La inversión de estos bits da 0xFFFE (véase el script *ExprBitwise.mq5*).

```
short v = ~1;  // 0xfffe = -2
ushort w = ~1; // 0xfffe = 65534
```

AND a nivel de bits comprueba cada bit en ambos operandos y, en las posiciones donde se encuentran dos bits establecidos (1), almacena el 1 bit en el resultado. En todos los demás casos (cuando sólo hay un bit establecido en un operando o se restablecen en ambos lugares), el bit 0 se escribe en el resultado.

OR a nivel de bits escribe bits 1 en el resultado si están en las posiciones donde hay un bit establecido en al menos uno de los dos operandos.

El OR exclusivo a nivel de bits escribe en el resultado los bits 1 en las posiciones donde hay un bit establecido en el primer o segundo operando, pero no en ambos a la vez. A continuación se muestra la representación binaria de dos números, X e Y, y los resultados de las operaciones a nivel de bits con ellos.

```
X       10011010   154
Y       00110111    55

X & Y   00010010    18
X | Y   10111111   191
X ^ Y   10101101   173
```

Cuando escriba expresiones complejas a partir de varios operadores diferentes, utilice la agrupación con paréntesis para no confundirse con las prioridades.

Las operaciones de desplazamiento desplazan bits a la izquierda ('<<') o a la derecha ('>>') en la cantidad de bits definida en el segundo operando que debe ser un entero no negativo. Como resultado, los bits de la izquierda (para '<<') o de la derecha (para '>>') se eliminan, ya que van más allá de los límites de la celda de memoria. Con el desplazamiento a la izquierda se añade a la derecha el número correspondiente de bits 0. Con el desplazamiento a la derecha, o bien se añaden 0 bits a la izquierda (si el operando es sin signo) o bien se reproduce el bit de signo (si el operando es con signo). En este último caso se añaden 0 bits a la izquierda para los números positivos y 1 bits para los negativos; es decir, se conserva el signo.

```
short q = v << 5;  // 0xffc0 = -64
ushort p = w << 5; // 0xffc0 = 65472
short r = q >> 5;  // 0xfffe = -2
ushort s = p >> 5; // 0x07fe = 2046
```

En el ejemplo anterior, el desplazamiento inicial a la izquierda «destruyó» los bits de orden superior de la variable *p*, mientras que el desplazamiento posterior a la derecha en la misma cantidad de bits los llenó de ceros, lo que llevó a disminuir el valor de 0xffc0 a 0x07fe.

El tamaño del desplazamiento (cantidad de bits) debe ser menor que el del tipo de operando (teniendo en cuenta su extensión potencial). De lo contrario, todos los bits iniciales se perderán.

El desplazamiento de 0 bits no modifica el número.

Las operaciones a nivel de bits '&' y '|' no deben mezclarse con las operaciones lógicas '&&' y '||' (se analizan en la sección anterior).

2.5.8 Operaciones de modificación

La modificación, que también se denomina asignación compuesta, permite combinar dentro de un operador operaciones aritméticas o a nivel de bits con operaciones de asignación normales.

P	Símbolos	Descripción	Ejemplo	Orden de ejecución
14	+=	Suma con asignación	e1 += e2	D-I
14	-=	Resta con asignación	e1 -= e2	D-I
14	*=	Multiplicación con asignación	e1 *= e2	D-I
14	/=	División con asignación	e1 /= e2	D-I
14	%=	Módulo de división con asignación	e1 %= e2	D-I
14	<<=	Desplazamiento a la izquierda con asignación	e1 <<= e2	D-I
14	>>=	Desplazamiento a la derecha con asignación	e1 >>= e2	D-I
14	&=	AND a nivel de bits con asignación	e1 &= e2	D-I
14	|=	OR a nivel de bits con asignación	e1 |= e2	D-I
14	^=	AND/OR a nivel de bits con asignación	e1 ^= e2	D-I

Estos operadores ejecutan la acción pertinente para los operandos *e1* y *e2*, tras lo cual el resultado se almacena en *e1*.

Una expresión como *e1 @= e2*, donde @ es cualquier operador de la tabla, es equivalente aproximadamente a *e1 = e1 @ e2*. La palabra «aproximadamente» subraya la presencia de algunos aspectos sutiles.

En primer lugar, si el lugar de *e2* está ocupado por una expresión con un operador de prioridad inferior a la de @, *e2* sigue calculándose antes que ella. Es decir, si la prioridad está marcada con paréntesis, obtendremos *e1 = e1 @ (e2)*.

En segundo lugar, si hay modificaciones laterales de variables en la expresión *e1*, sólo se realizan una vez. El siguiente ejemplo se ofrece la demostración.

```
int a[] = {1, 2, 3, 4, 5};
int b[] = {1, 2, 3, 4, 5};
int i = 0, j = 0;
a[++i] *= i + 1;           // a = {1, 4, 3, 4, 5}, i = 1
                           // not equivalent!
b[++j] = b[++j] * (j + 1); // b = {1, 2, 4, 4, 5}, j = 2
```

En este caso, los arrays *a* y *b* contienen elementos idénticos y se procesan mediante las variables de índice *i* y *j*. Al mismo tiempo, la expresión para el array *a* utiliza la operación '*=', mientras que la del array *b* utiliza la equivalente. Los resultados no son iguales: tanto las variables de índice como los arrays difieren.

Otros operadores serán útiles en problemas con manipulaciones a nivel de bits. Así, la siguiente expresión se puede utilizar para establecer un bit específico en 1:

```
ushort x = 0;
x |= 1 << 10;
```

Aquí, el desplazamiento 1 ('0000 0000 0000 0001') se realiza 10 bits a la izquierda, con lo que se obtiene un número con un 10º bit ajustado ('0000 0100 0000 0000'). La operación OR a nivel de bits copia este bit en la variable *x*.

Para reiniciar el mismo bit, escribiremos:

```
x &= ~(1 << 10);
```

Aquí, la operación de inversión se aplica a 1 desplazado 10 bits a la izquierda (que ya vimos en la expresión anterior), lo que hace que todos los bits cambien de valor: '1111 1011 1111 1111'. La operación AND a nivel de bit restablece los bits a cero (en este caso, uno) en la variable *x*, mientras que el resto de bits de *x* permanecen inalterados.

2.5.9 Operador ternario condicional

El operador ternario condicional permite describir en una sola expresión dos opciones de cálculo basadas en una condición determinada. La sintaxis del operador es la siguiente

```
condition ? expression_true : expression_false
```

La condición lógica debe especificarse en el primer operando 'condition' y puede ser una combinación arbitraria de operaciones de comparación y operaciones lógicas normales. Ambas ramas deben estar presentes.

Si la condición es verdadera, se calculará la expresión *expression_true*, mientras que si es falsa, se calculará *expression_false*.

Este operador garantiza que sólo se ejecute una de las expresiones *expression_true* y *expression_false*.

Los tipos de las dos expresiones deben ser idénticos; de lo contrario, se producirá un intento de conversión sus tipos de forma implícita.

Tenga en cuenta que el resultado de procesar expresiones en MQL5 siempre representa un RValue (en C++, si sólo hay LValues en las expresiones, entonces el resultado del operador también será LValue). Así, el siguiente código se compila bien en C++, pero da un error en MQL5:

```
int x1, y1; ++(x1 > y1 ? x1 : y1); // '++' - l-value required
```

Los operadores condicionales pueden anidarse, es decir, se permite utilizar otro operador condicional como condición o cualquiera de las ramas (*expression_true* o *expression_false*). Al mismo tiempo, no siempre puede quedar claro a qué se refieren las condiciones (si no se utilizan paréntesis para denotar explícitamente la agrupación). Veamos algunos ejemplos en *ExprConditional.mq5*.

```
int x = 1, y = 2, z = 3, p = 4, q = 5, f = 6, h = 7;
int r0 = x > y ? z : p != 0 && q != 0 ? f / (p + q) : h; // 0 = f / (p + q)
```

En este caso, la primera condición lógica representa la comparación *x > y*. Si es verdadero, se ejecuta la rama con la variable *z*. Si es falso, se comprueba la condición lógica adicional *p != 0 && q != 0*, también con dos opciones de expresión.

A continuación se muestran algunos operadores más en los que las condiciones lógicas se escriben en mayúsculas, mientras que las opciones de cálculo se escriben en minúsculas. Para simplificar, todas ellas se convierten en variables (del ejemplo anterior). En realidad, cada uno de los tres componentes puede ser una expresión más rica.

Para cada cadena puede hacer un seguimiento de cómo se obtiene el resultado, lo que se ha mostrado en el comentario.

```
bool A = false, B = false, C = true;
int r1 = A ? x : C ? p : q;                          // 4
int r2 = A ? B ? x : y : z;                          // 3
int r3 = A ? B ? C ? p : q : y : z;                  // 3
int r4 = A ? B ? x : y : C ? p : q;                  // 4
int r5 = A ? f : h ? B ? x : y : C ? p : q;          // 2
```

Dado que el operador es asociativo a la derecha, la expresión compuesta se analiza de derecha a izquierda, es decir, la estructura situada más a la derecha con tres operandos combinados por '?' y ':' se convierte en el operando de la condición externa escrita a la izquierda. A continuación, teniendo en cuenta esta sustitución, se vuelve a analizar la expresión de derecha a izquierda, y así sucesivamente, hasta obtener la estructura final completa de nivel superior '?:'.

Por lo tanto, las expresiones anteriores se agrupan de la siguiente manera (los paréntesis denotan la interpretación implícita del compilador, pero dichos paréntesis podrían añadirse a las expresiones para visualizar el código fuente, que es el enfoque que, de hecho, se recomienda):

```
int r0 = x > y ? z : ((p != 0 && q != 0) ? f / (p + q) : h);
int r1 = A ? x : (C ? p : q);
int r2 = A ? (B ? x : y) : z;
int r3 = A ? (B ? (C ? p : q) : y) : z;
int r4 = A ? (B ? x : y) : (C ? p : q);
int r5 = (A ? f : h) ? (B ? x : y) : (C ? p : q);
```

Para la variable *r5*, la primera condición *A ? f : h* calcula la condición lógica para la expresión subsiguiente y, por lo tanto, se transforma en *bool*. Como A es igual a *false*, el valor se toma de la variable *h*. No es igual a 0; por lo tanto, la primera condición se considera verdadera. El resultado es la rama actuadora *(B ? x : y)*, de la que se devuelve el valor de la variable *y*, ya que B es igual a *false*.

Deben estar los 3 componentes (una condición y 2 alternativas) en el operador. De lo contrario, el compilador generará el error «token inesperado»:

```
// ';' - unexpected token
// ';' - ':' colon sign expected
int r6 = A ? B ? x : y; // lack of alternative
```

En el lenguaje del compilador, un token es un fragmento indivisible del código fuente que tiene un significado o propósito independiente, como tipo, identificador, carácter de puntuación, etc. El compilador divide todo el código fuente en una secuencia de tokens. Los signos de los operadores considerados también son tokens. En el código anterior hay dos símbolos '?' y debe haber dos símbolos ':' que se correspondan con ellos, pero solo hay uno. Por lo tanto, el compilador «dice» que el símbolo de fin de sentencia ';' es prematuro y «pregunta» qué es exactamente lo que es deficiente: «signo de dos puntos esperado».

Dado que el operador condicional tiene una prioridad muy baja (13 en la tabla completa, véase Prioridades de las operaciones), se recomienda encerrarlo entre paréntesis. Esto facilita evitar situaciones en las que los operandos de un operador condicional podrían ser «atrapados» por operaciones vecinas que tengan prioridades más altas. Por ejemplo, si necesitamos calcular el valor de una determinada variable *w* mediante la suma de dos operadores ternarios, un planteamiento sencillo podría ser el siguiente:

```
int w = A ? f : h + B ? x : y;                    // 1
```

Esto funcionará de forma diferente a lo que habíamos pensado. Debido a su mayor prioridad, la suma *h + B* se considera una única expresión. Considerando su análisis sintáctico de derecha a izquierda, esta suma aparece como una condición y se convierte al tipo *bool*, lo que incluso es advertido por el compilador como «expresión no booleana». La interpretación del compilador puede visualizarse incluso mediante paréntesis:

```
int w = A ? f : ((h + B) ? x : y);                // 1
```

Para resolver el problema debemos colocar los paréntesis a nuestra manera.

```
int v = (A ? f : h) + (B ? x : y);                // 9
```

El anidamiento profundo de operadores condicionales afecta negativamente a la comprensibilidad del código. Deben evitarse los niveles de anidamiento superiores a dos o tres.

2.5.10 Coma

El operador coma se denota explícitamente como ',' y se coloca entre dos expresiones calculadas de forma independiente de izquierda a derecha. En otras palabras: este operador no realiza ninguna acción en sí, sino que permite simplemente especificar la secuencia de dos o más expresiones dentro de una sentencia.

Las expresiones situadas a la derecha en la secuencia pueden utilizar los resultados del cálculo de las expresiones de la izquierda, puesto que ya se han procesado.

El resultado del operador es el resultado de la expresión situada más a la derecha. El operador tiene la prioridad más baja.

Actualmente, el uso del operador en MQL5 está limitado por el encabezado de la sentencia for.

Ejemplo:

```
for(i=0,j=99; i<100; i++,j--)
   Print(array[i][j]);
```

Repitamos los aspectos clave del operador coma en MQL5:

Orden de evaluación:

- Las expresiones se procesan de izquierda a derecha. Así, las expresiones de la derecha pueden utilizar los resultados de las expresiones de la izquierda, puesto que ya se han procesado.

Resultado y prioridad:

- El resultado del operador coma es el valor de la expresión situada más a la derecha. Es importante tener en cuenta que el operador coma tiene la prioridad más baja, lo que significa que otros operadores de la expresión pueden tener prioridades más altas.

2.5.11 Operadores especiales sizeof y typename

sizeof

El operador *sizeof* devuelve el tamaño de su operando en bytes. Sintaxis del operador: *sizeof(x)*, donde *x* puede ser un tipo o una expresión. En este caso, la expresión no se computa, ya que el operador *sizeof* se ejecuta en la fase de compilación y, de hecho, se sustituye una constante en su lugar en la expresión.

Para arrays de tamaño fijo, el operador devuelve la cantidad total de memoria asignada, es decir, la multiplicación del número de elementos en todas las dimensiones por el tamaño del tipo en bytes. Para arrays dinámicos, devuelve el tamaño de una estructura interna que almacena las propiedades del array.

Veamos algunos ejemplos con explicaciones (*ExprSpecial.mq5*).

```
double array[2][2];
double dynamic1[][1];
double dynamic2[][2];
Print(sizeof(double));                          // 8
Print(sizeof(string));                          // 12
Print(sizeof("This string is 29 bytes long!")); // 12
Print(sizeof(array));                           // 32
Print(sizeof(array) / sizeof(double));          // 4 (quantity of elements)
Print(sizeof(dynamic1));                        // 52
Print(sizeof(dynamic2));                        // 52
```

El resultado que debe imprimirse en el registro se marca en los comentarios.

El tipo *double* ocupa 8 bytes. El tamaño del tipo *string* es 12. Estos 12 bytes almacenan la información de servicio que hemos mencionado en la sección relativa al tipo *string*. Esta memoria se asigna a cualquier cadena (incluso no inicializada). Tenga en cuenta que una cadena que contenga un texto de 29 caracteres también tiene un tamaño de 12. Esto se debe a que tanto una cadena vacía como una cadena con algún contenido tienen una estructura interna concebida para almacenar una referencia a la memoria. Para obtener la longitud del texto debemos utilizar la función *StringLen*.

El tamaño del array de tamaño fijo se calcula realmente como la multiplicación del número de elementos (2*2=4) por el tamaño del tipo *double* (8), un total de 32. En consecuencia, una expresión como *sizeof(array) / sizeof(double)* permite averiguar la entidad de los elementos que la componen.

Para los arrays dinámicos, el tamaño de la estructura interna es de 52 bytes. Las diferencias en las descripciones de los arrays *dynamic1* y *dynamic2* no afectan a este valor.

El operador *sizeof* es especialmente útil para obtener los tamaños de clases y estructuras.

typename

El operador *typename* devuelve una cadena con el nombre del parámetro que se le ha pasado, que puede ser un tipo o una expresión. Para los arrays, además de la palabra clave del tipo de datos, se imprime una etiqueta como un par de paréntesis (o varios, dependiendo de la dimensionalidad del array).

```
Print(typename(double));                    // double
Print(typename(array));                     // double [2][2]
Print(typename(dynamic1));                  // double [][1]
Print(typename(1 + 2));                     // int
```

Para los tipos personalizados, como clases, estructuras y otros (que veremos en la Parte 3), el nombre del tipo sigue a la categoría de la entidad, como «clase MyCustomType». Además, en el caso de las constantes, se añadirá el modificador «const» a la descripción de la cadena.

Por lo tanto, para conocer el nombre de tipo abreviado que consta de una palabra, utilice la macro TYPENAME del archivo adjunto *TypeName.mqh*.

Puede ser necesario aprender el nombre del tipo en las denominadas plantillas, que pueden generar a partir del código fuente realizaciones similares para diferentes tipos definidos en los parámetros de las plantillas.

2.5.12 Agrupación con paréntesis

En las secciones anteriores hemos visto ya más de una vez que algunas expresiones pueden provocar resultados inesperados debido a las prioridades de las operaciones. Para cambiar de forma explícita el orden de cálculo debemos utilizar paréntesis. Parte de la expresión encerrada en ellos obtiene una prioridad más alta en comparación con el entorno, sin tener en cuenta las prioridades por defecto. Se pueden anidar pares de paréntesis, pero no se recomienda hacer más de 3-4 niveles de anidamiento: es mejor dividir las expresiones demasiado complejas en varias más sencillas.

El script *ExprParentheses.mq5* muestra la evolución de la colocación de paréntesis dentro de una expresión. La intención inicial para ello es fijar el bit de la variable *flags* utilizando la operación de desplazamiento a la izquierda '<<'. El número de bit se toma de la variable *offset* si no es cero, o en caso contrario, como 1 (recuerde que la numeración empieza por cero). A continuación, el valor obtenido se multiplica por *coefficient*. No es necesario buscar ningún sentido práctico en este ejemplo. Sin embargo, también pueden darse estructuras más sofisticadas.

```
int offset = 8;
int coefficient = 10, flags = 0;
int result1 = coefficient * flags | 1 << offset > 0 ? offset : 1;     // 8
int result2 = coefficient * flags | 1 << (offset > 0 ? offset : 1);   // 256
int result3 = coefficient * (flags | 1 << (offset > 0 ? offset : 1)); // 2560
```

La primera versión, sin paréntesis, parece sospechosa incluso para el compilador. Emite un aviso que ya hemos visto: «expresión no booleana». La cuestión es que el operador condicional ternario tiene aquí la prioridad más baja de todos los operadores. Por esta razón, toda la parte izquierda situada delante de '?' se considera su condición. Dentro de la condición, los cálculos siguen el siguiente orden: multiplicación, desplazamiento de bits, comparación «más que» y OR a nivel de bits, que da como resultado un número entero. Por supuesto, se puede utilizar como *true* o *false*, pero se desea «comunicar» tales intenciones al compilador utilizando la conversión de tipos explícita. Si está ausente, el compilador considera que la expresión es sospechosa, y no en vano. El primer cálculo da como resultado 8. Esto es incorrecto.

Añadamos paréntesis en torno al operador ternario. El aviso del compilador desaparecerá. No obstante, la expresión sigue calculándose de forma errónea. Dado que la prioridad de la multiplicación es mayor que la de OR a nivel de bits, las variables *coefficient* y *flags* se multiplican antes de utilizar la máscara de bits, que se obtiene desplazando a la izquierda. El resultado es 256.

Finalmente, habiendo añadido otro par de paréntesis, obtendremos el resultado correcto: 2560.

2.5.13 Prioridades de las operaciones

Aquí está la tabla completa de todas las operaciones por orden de prioridad.

P	Símbolos	Descripción	Ejemplo	Orden de ejecución
0	::	Resolución de contexto	n1:: n2	I-D
1	()	Agrupación	(e1)	I-D
1	[]	Índice	[e1]	I-D
1	.	Desreferenciación	n1.n2	I-D
1	++	Post-incremento	e1++	I-D
1	--	Post-decremento	e1--	I-D
2	!	Negación lógica NOT	!e1	D-I
2	~	Complemento a nivel de bits (inversión)	~e1	D-I
2	+	Unario más	+e1	D-I
2	-	Unario menos	-e1	D-I

P	Símbolos	Descripción	Ejemplo	Orden de ejecución
2	++	Incremento prefijo	++e1	D-I
2	--	Decremento prefijo	--e1	D-I
2	(tipo)	Conversión de tipos	(n1)	D-I
2	&	Obtención de la dirección	&n1	D-I
3	*	Multiplicación	e1 * e2	I-D
3	/	División	e1 / e2	I-D
3	%	Módulo de división	e1 % e2	I-D
4	+	Adición	e1 + e2	I-D
4	-	Resta	e1 - e2	I-D
5	<<	Desplazamiento a la izquierda	e1 << e2	I-D
5	>>	Desplazamiento a la derecha	e1 >> e2	I-D
6	<	Menos	e1 < e2	I-D
6	>	Mayor	e1 > e2	I-D
6	<=	Menor o igual que	e1 <= e2	I-D
6	>=	Mayor o igual que	e1 >= e2	I-D
7	==	Igualdad	e1 == e2	I-D
7	!=	No es igual	e1 != e2	I-D
8	&	AND a nivel de bits	e1 & e2	I-D
9	^	OR exclusivo a nivel de bits	e1 ^ e2	I-D
10	|	OR a nivel de bits	e1 | e2	I-D
11	&&	AND lógico	e1 && e2	I-D
12	||	OR lógico	e1 || e2	I-D

P	Símbolos	Descripción	Ejemplo	Orden de ejecución
13	?:	Ternario condicional	c1 ? e1: e2	D-I
14	=	Asignación	e1 = e2	D-I
14	+=	Suma con asignación	e1 += e2	D-I
14	-=	Resta con asignación	e1 -= e2	D-I
14	*=	Multiplicación con asignación	e1 *= e2	D-I
14	/=	División con asignación	e1 /= e2	D-I
14	%=	Módulo de división con asignación	e1 %= e2	D-I
14	<<=	Desplazamiento a la izquierda con asignación	e1 <<= e2	D-I
14	>>=	Desplazamiento a la derecha con asignación	e1 >>= e2	D-I
14	&=	AND a nivel de bits con asignación	e1 &= e2	D-I
14	|=	OR a nivel de bits con asignación	e1 |= e2	D-I
14	^=	AND/OR a nivel de bits con asignación	e1 ^= e2	D-I
15	,	Coma	e1, e2	I-D

Como hemos visto, los corchetes se utilizan para especificar los índices de los elementos del array y, por lo tanto, tienen una de las prioridades más altas.

Además de los operadores que se han visto anteriormente, aquí hay algunos todavía desconocidos.

Descubriremos el operador resolución de contexto '::' dentro de la programación orientada a objetos (POO). También necesitaremos al mismo tiempo el operador de desreferenciación '.'. Los identificadores de tipos (clases) y sus propiedades, no las expresiones, actúan como sus operandos.

El operador de obtención de direcciones '&' sirve para pasar los parámetros de la función mediante referenciación y obtener las direcciones de objetos en la POO. En ambos casos, el operador se aplica a una variable (LValue).

Las operaciones explícitas de conversión de tipos se abordarán en el capítulo siguiente.

2.6 Conversión de tipos

En esta sección analizaremos el concepto de conversión de tipos, limitándonos por ahora a los tipos de datos integrados. Más adelante, tras estudiar la programación orientada a objetos (POO), la complementaremos con los matices inherentes a los tipos de objetos.

La conversión de tipo en MQL5 es el proceso de cambiar el tipo de datos de una variable o expresión. MQL5 admite tres tipos principales de conversión de tipos: implícita, aritmética y explícita.

Conversión implícita de tipos:

- Se produce automáticamente cuando una variable de un tipo se utiliza en un contexto que espera otro tipo. Por ejemplo, los valores enteros pueden convertirse implícitamente en valores reales.

Conversión aritmética de tipos:

- Surge durante operaciones aritméticas con operandos de diferentes tipos. El compilador intenta mantener la máxima precisión, pero avisa de la posible pérdida de datos. Por ejemplo, en la división de enteros, el resultado se convierte a un tipo real.

Conversión explícita de tipos:

- Permite al programador controlar la conversión de tipos. Ello se realiza de dos formas: estilo C ((destino)) y estilo «funcional» (destino()). Ello se utiliza cuando es necesario indicar explícitamente al compilador que realice una conversión entre tipos, por ejemplo al redondear números reales o cuando se requieren conversiones de tipo sucesivas.

Comprender las diferencias entre la conversión de tipos implícita, aritmética y explícita es crucial para garantizar la correcta ejecución de las operaciones y evitar la pérdida de datos. Este conocimiento ayuda a los programadores a utilizar eficazmente este mecanismo en el desarrollo de MQL5.

2.6.1. Conversión implícita de tipos

La conversión de tipos se produce automáticamente si se utiliza un tipo en algún punto del código fuente pero se espera otro, y existen reglas de conversión entre ellos. Dicha conversión se denomina conversión de tipo implícita y puede que no siempre se corresponda con la intención del programador. Además, algunas operaciones de conversión tienen efectos secundarios, y el compilador, al no saber si su uso es intencionado, resalta las líneas de código correspondientes con avisos. Para resolver estos problemas existe una sintaxis de conversión de tipos explícita (véase Reparto de tipos explícita).

Ya hemos visto varias reglas para la conversión implícita de tipos al estudiar los tipos y variables.

En concreto, si se asigna un valor de un tipo distinto a booleano a una variable *bool* , el valor 0 se considera *false* y todos los demás, *true*. En el caso más general, todas las expresiones que dan por sentada la presencia de condiciones lógicas se convierten al tipo *bool*. Por ejemplo, el primer operando de un operador condicional ternario siempre se convierte en *bool*.

Pero si un valor de tipo *bool* se asigna a un tipo numérico, entonces *true* se convierte en 1 y *false*, en 0.

Cuando se asigna un número real a una variable de tipo entero, la parte fraccionaria se descarta (el compilador emite un aviso). En cambio, cuando se asigna un número entero a una variable de tipo real,

se puede perder precisión (el compilador también emite un aviso). Ya hemos hablado de ello en las secciones sobre Números enteros y Números reales.

Si tenemos números enteros y de punto flotante, todo se convierte a números de punto flotante del tamaño máximo utilizado (normalmente *double*,, a menos que se especifique de forma explícita *float* o que el literal numérico tenga un sufijo 'f', como por ejemplo1234.56789f).

Para los enteros de diferentes tamaños también existen reglas de conversión: se expanden si es necesario, lo que significa que aumentan hasta el tamaño del tipo entero más grande utilizado en la expresión (véase Conversiones aritméticas de tipo).

Además de las expresiones, a menudo necesitamos convertir tipos de forma implícita durante la inicialización y la asignación, cuando los tipos a la derecha y a la izquierda del signo '=' no coinciden. Se aplican las mismas reglas de conversión cuando se pasan valores a través de parámetros de funciones y cuando se devuelven resultados de funciones (para obtener más detalles, consulte la sección Funciones).

Teniendo en cuenta lo anterior se puede realizar un gran número de conversiones en una sola línea de código. Si esto da lugar a avisos del compilador, es una buena idea asegurarse de que la conversión es intencionada y eliminar dichos avisos insertando una conversión de tipo explícita.

```
short s = 10;
long n = 10;
int p = s * n + 1.0;
```

En este ejemplo, al realizar una multiplicación, el tipo de la variable *s* se extiende al tipo del segundo operando *long* y se obtiene un resultado intermedio del tipo *long*. Como la constante 1.0 es del tipo *double*, el resultado del producto se convierte a *double* antes de la suma. El resultado global también es del tipo *double*; sin embargo, la variable *p* es del tipo *int* y, por lo tanto, se realiza una conversión implícita de *double* a *int*.

Los tipos especiales *datetime* y *color* se procesan según las reglas de los enteros con longitudes de 8 y 4 bytes, respectivamente. Pero para la fecha y la hora hay un límite más estricto en cuanto al valor máximo: se trata de 32535244799, que corresponde a D'3000,12.31 23:59:59'.

La mayoría de los tipos pueden convertirse de forma implícita a y desde cadenas, pero los resultados no siempre son adecuados, por lo que el compilador emite avisos «conversión implícita de 'número' a 'cadena'» y «conversión implícita de 'cadena' a 'número'» para que el programador pueda comprobarlas. Por ejemplo, convertir una cadena en un número entero permite que la cadena sólo contenga dígitos y caracteres '+'/'-' al principio. La conversión de una cadena a un número real permite, además de los números, la presencia de un punto '.' y la notación con «exponente» ('e' o 'E', por ejemplo +1,2345e-1). Si en la cadena se encuentra un carácter no admitido (por ejemplo, una letra), el resto de la cadena se descarta por completo.

Por ejemplo, la cadena de fecha y hora («2021,12.12 00:00») no puede asignarse sin pérdidas a una variable de tipo *datetime* porque *datetime* es un número entero (número de segundos). En este caso, la lectura del número de la cadena finalizará cuando se alcance el primer punto, es decir, el número obtendrá el valor 2021. Este número de segundos se corresponde con el minuto 34 del año 1970.

Existen funciones especiales para este tipo de conversiones (véase la sección Transformación de datos).

La única dirección de conversión de tipos implícita y explícita que está prohibida es de *string* a *bool*. En estos casos, el compilador muestra el mensaje de error «no se puede convertir de forma implícita el tipo 'cadena' a 'bool'».

Encontrará ejemplos de este capítulo en *TypeConversion.mq5*.

2.6.2. Conversiones aritméticas de tipo

En el cálculo aritmético y en las expresiones de comparación a menudo se utilizan como operandos valores de distintos tipos. Para procesarlos correctamente es necesario llevar los tipos a un cierto «denominador común». El compilador intenta hacerlo sin la intervención del programador, a menos que éste haya especificado reglas de conversión explícitas (véase Conversión explícita de tipos). En este caso, el compilador, siempre que sea posible, intenta conservar la máxima precisión cuando se trata de números. En concreto, ello produce un aumento de la capacidad de los números enteros y de la transición de números enteros a reales (si intervienen).

La expansión de números enteros implica la conversión de *bool*, *char*, *unsigned char*, *short*, *unsigned short* a *int* (o *unsigned int* si *int* no es lo suficientemente grande para almacenar números específicos). Los valores grandes pueden convertirse a *long* y *unsigned long*.

Si el tipo de la variable no es capaz de almacenar el resultado del tipo que se obtuvo al evaluar la expresión, el compilador emitirá un aviso.

```
double d = 1.0;
int x = 1.0 / 10; // truncation of constant value
int y = d / 10;   // possible loss of data due to type conversion
```

La expresión para inicializar las variables *x* y *y* contiene el número real 1.0, por lo que los otros operandos (la constante 10 en este caso) se convierten a *double*, y el resultado de la división también será del tipo *double*. Sin embargo, el tipo de las variables es *int*, y por lo tanto se produce una conversión implícita al mismo.

El cálculo 1.0 / 10 lo realiza el compilador durante la compilación y por lo tanto obtiene una constante de tipo *double* (0.1) Por supuesto, en la práctica, es poco probable que la constante de inicialización supere el tamaño de la variable receptora. Por lo tanto, la advertencia del compilador «truncamiento de valor constante» puede considerarse extraña. Sólo muestra el problema de la forma más simplificada.

Sin embargo, como resultado de los cálculos basados en variables, también pueden producirse pérdidas de datos similares. El segunda aviso del compilador que vemos aquí («posible pérdida de datos debido a la conversión de tipos») se produce con mucha más frecuencia. Además, la pérdida es posible no sólo al convertir de tipo real a entero, sino también al revés.

```
double f = LONG_MAX; // truncation of constant value
long m1 = 1000000000;
f = m1 * m1;         // possible loss of data due to type conversion
```

Como sabemos, el tipo *double* no puede representar con precisión números enteros grandes (aunque su rango de valores válidos es mucho mayor que *long*).

Otro aviso que podemos encontrar debido a la falta de coincidencia de tipo es «sobrante de constante de enteros».

```
long m1 = 1000000000;
long m2 = m1 * m1;                  // ok: m2 = 1000000000000000000
long m3 = 1000000000 * 1000000000; // integral constant overflow
                                    // m3 = -1486618624
```

Las constantes de enteros en MQL5 tienen el tipo *int*, por lo que la multiplicación de millón por millón se realiza teniendo en cuenta el rango de este tipo, que es igual a INT_MAX (2147483647). El valor 1000000000000000000 causa un sobrante, y *m3* obtiene el resto después de dividir este valor por el rango (encontrará más información al respecto en la barra lateral más abajo).

El hecho de que la variable receptora *m3* sea del tipo *long* no significa que los valores de la expresión deban convertirse previamente a ella. Esto sólo ocurre en el momento de la asignación. Para que la multiplicación se realice según las reglas de long es necesario especificar de algún modo el tipo *long* directamente en la propia expresión. Esto puede hacerse con una conversión explícita o mediante variables. En particular, la obtención del mismo producto utilizando una variable *m1* de tipo *long* (como *m1 * m1*) conduce al resultado correcto en *m2.*

Enteros con y sin signo

Los programas no siempre se escriben a la perfección, con protección frente a todos los fallos posibles. Por lo tanto, a veces ocurre que el número entero obtenido durante los cálculos no cabe en la variable del tipo entero seleccionado. Entonces, obtiene el resto de dividir este valor por el valor máximo (M) que puede escribirse en el número de bytes correspondiente (tamaño de tipo), más 1. Así, para tipos enteros con tamaños de 1 a 4 bytes, M + 1 es, respectivamente, 256, 65536, 4294967296 y 18446744073709551616.

Pero existe un matiz para los tipos con signo. Como sabemos, para los números con signo, el rango total de valores se divide aproximadamente a partes iguales entre áreas positivas y negativas. Por lo tanto, el nuevo valor «residual» puede superar en un 50 % de los casos el límite positivo o negativo. En este caso, el número se convierte en el «opuesto»: cambia de signo y acaba a una distancia M del original.

Es importante entender que esta transformación se produce sólo debido a una interpretación diferente del estado del bit en la representación interna, y el estado en sí es el mismo para los números con y sin signo.

Vamos a explicarlo con un ejemplo para los tipos enteros más pequeños: *char* y *uchar*.

Dado que *unsigned char* puede almacenar valores de 0 a 255, 256 mapea a 0, -1 mapea a 255, 300 mapea a 44, y así sucesivamente. Si intentamos escribir 300 en un *char* con signo normal, obtendremos también 44, porque 44 está en el rango de 0 a 127 (el rango positivo de *char*). Sin embargo, si establece las variables *char* y *uchar* en 3000, la imagen será diferente. El resto de 3000 dividido entre 256 es 184. Acaba en *uchar* sin cambios. Sin embargo, para *char*, la misma combinación de bits da como resultado -72. Es fácil comprobar que 184 y -72 difieren en 256.

En el siguiente ejemplo es fácil detectar el problema gracias al aviso del compilador.

```
char c = 3000;      // truncation of constant value
Print(c);           // -72
uchar uc = 3000;    // truncation of constant value
Print(uc);          // 184
```

Sin embargo, si obtiene un número extra grande durante el cálculo, no habrá ningún aviso.

```
char c55 = 55;
char sm = c55 * c55;  // ok!
Print(sm);            // 3025 -> -47
uchar um = c55 * c55; // ok!
Print(um);            // 3025 -> 209
```

Puede producirse un efecto similar cuando se utilizan números enteros con y sin signo del mismo tamaño en la misma expresión, ya que el operando con signo se convierte en operando sin signo. Por ejemplo:

```
uint u = 11;
int i = -49;
Print(i + i); // -98
Print(u + i); // 4294967258 = 4294967296 - 38
```

Cuando se suman dos números enteros negativos obtenemos el resultado esperado. La segunda expresión asigna la suma de -38 al número sin signo «opuesto» 4294967258.

> No se recomienda mezclar tipos con y sin signo en la misma expresión debido a estos posibles problemas.

Además, si restamos algo de un entero sin signo, tenemos que asegurarnos de que el resultado no sea negativo. De lo contrario, se convertirá en un número positivo y puede distorsionar la idea del algoritmo, en concreto la idea del *operador cíclico while* que comprueba en la variable la condición «mayor o igual que cero»: como los números sin signo son siempre no negativos, podemos obtener fácilmente un operador cíclico infinito, es decir, que el programa se cuelgue.

2.6.3. Conversión explícita de tipos

Para la conversión explícita de tipos, MQL5 admite dos formas de notación: el estilo C y el «funcional». El estilo C tiene la siguiente sintaxis:

```
target t = (target)s;
```

Aquí, *target* es el nombre del tipo de destino. Cualquier expresión puede ser una fuente de datos *s*. Si en ella se realiza alguna operación, debe encerrar la expresión entre paréntesis para que la conversión de tipo se aplique a toda la expresión.

Una sintaxis «funcional» alternativa es la siguiente:

```
target t = target(s);
```

Veamos un par de ejemplos:

```
double w = 100.0, v = 7.0;
int p = (int)(w / v);      // 14
```

Aquí, el resultado de dividir dos números reales se convierte explícitamente al tipo *int*. De este modo, el programador confirma su intención de descartar la parte fraccionaria, y el compilador no emitirá avisos. Cabe señalar que MQL5 dispone de un grupo de funciones para redondear los números reales de diversas maneras (véase Funciones matemáticas).

Si, por el contrario, desea realizar una operación sobre números enteros con un resultado real, deberá aplicar la conversión de tipo a los operandos (en la propia expresión):

```
int x = 100, y = 7;
double d = (double)x / y;  // 14.28571428571429
```

La conversión de uno de los operandos es suficiente para convertir automáticamente el resto al mismo tipo.

Si es necesario, puede realizar varias operaciones de conversión de tipos de forma secuencial. Dado que la operación de conversión es asociativa a la derecha, los tipos de destino se aplicarán en orden de derecha a izquierda. En el siguiente ejemplo, convertimos el cociente al tipo *float* (esta conversión permite una representación más compacta y con menos caracteres del valor) y, a continuación, a *string*. Sin una conversión explícita a *string* obtendríamos del compilador un aviso de «conversión implícita de número a cadena».

```
Print("Result:" + (string)(float)(w / v)); // Result:14.28571
```

No utilice la conversión explícita de tipos sólo para evitar un aviso del compilador. Si no tiene base práctica, está enmascarando un posible error en el programa.

2.7 Sentencias

Hasta ahora, hemos visto los tipos de datos, las declaraciones de variables y su uso en expresiones para cálculos. Sin embargo, estos son tan sólo pequeños ladrillos en el edificio al que se puede asemejar el programa. Incluso el programa más sencillo consta de bloques más grandes que permiten agrupar operaciones de tratamiento de datos relacionadas y controlar la secuencia de su ejecución. Estos bloques se denominan sentencias, y de hecho ya hemos utilizado algunas de ellas.

En concreto, la declaración de una variable (o de varias variables) es una sentencia. Asignar el resultado de la evaluación de la expresión a una variable también es una sentencia. En sentido estricto, la propia operación de asignación forma parte de la expresión, por lo que es más correcto denominar a una sentencia de este tipo sentencia de expresión. Por cierto: una expresión puede no contener un operador de asignación (por ejemplo, si simplemente llama a alguna función que no devuelve un valor, como *Print("Hello");*).

La ejecución del programa es la ejecución progresiva de sentencias: de arriba abajo y de izquierda a derecha (si hay varias sentencias en una línea). En el caso más sencillo, su secuencia se realiza linealmente, una tras otra. Para la mayoría de los programas, esto no es suficiente, por lo que existen varias sentencias de control que le permiten organizar bucles (repetición de cálculos) en los programas y la selección de opciones de funcionamiento del algoritmo en función de las condiciones.

Las sentencias son construcciones sintácticas especiales que representan el texto fuente escrito según las reglas. Las sentencias de un tipo determinado tienen sus propias reglas, pero hay algo en común: todos los tipos de sentencias terminan con un ';' excepto la sentencias sentencia compuesta. Esta puede prescindir del punto y coma porque su principio y su final están fijados por un par de llaves. Es importante señalar que gracias a la sentencia compuesta podemos incluir conjuntos de sentencias dentro de otras sentencias, construyendo estructuras jerárquicas arbitrarias de algoritmos.

En este capítulo nos familiarizaremos con todos los tipos de sentencias de control MQL5, además de consolidar las características de las sentencias de declaración y expresión.

2.7.1 Sentencias compuestas (bloques de código)

Una sentencia compuesta es un contenedor genérico para otras sentencias encerradas entre llaves '{' y '}'. Un bloque de código de este tipo puede utilizarse para definir el cuerpo de una función, tras el encabezado de otras sentencias de control si requieren más de una sentencia controlada, o simplemente como un bloque anidado por sí solo dentro del cuerpo de una función u otra sentencia. Esto permite crear un ámbito local y limitado para las variables. Ya hablamos de ello en la sección Contexto, ámbito y vida útil de las variables.

De forma generalizada, una declaración compuesta puede describirse del siguiente modo:

```
{
[statements]
}
```

En una descripción esquemática de este tipo, cualquier fragmento encerrado entre corchetes semicirculares y con el superíndice ^{opt} indica que es opcional. En este caso, no puede haber ninguna sentencia anidada dentro del bloque.

En las secciones siguientes veremos cómo se utilizan las sentencias compuestas en combinación con otros tipos de sentencias y qué pueden contener.

Hay un matiz que merece la pena destacar: después de la descripción de la sentencia compuesta no es necesario el punto y coma ';'. Esto la distingue de todas las demás sentencias.

2.7.2 Sentencias de declaración y definición

La declaración de una variable, array, función o cualquier otro elemento con nombre de un programa (incluidas las estructuras y clases, que se tratarán en la Parte 3) es una sentencia.

La declaración debe contener el tipo y el identificador del elemento (véase Declaración y definición de variables), así como un valor inicial opcional para inicialización. Además, al realizar una declaración, se pueden especificar modificadores adicionales que cambian ciertas características del elemento. En concreto, ya conocemos los modificadores *static* y *const* , y pronto se añadirán más. Los arrays requieren una especificación adicional de la dimensión y el número de elementos (véase Descripción de arrays), mientras que las funciones requieren una lista de parámetros (para obtener más detalles, véase Funciones).

La sentencia de declaración de variables puede resumirse de la siguiente manera:

```
[modifiers] identifier type
  [= initialization expressions] ;
```

Para un array, tiene este aspecto:

```
[modifiers] identifier type [ [size_1]opt ] [ [size_N] ]opt(3)
  [ = { initialization_list } ]opt ;
```

La principal diferencia es la presencia obligatoria de al menos un par de corchetes (el tamaño dentro de ellos puede indicarse o no; dependiendo de eso, obtendremos un array fijo o distribuido dinámicamente). En total, se permiten hasta 4 pares de corchetes (4 es el número máximo de medidas admitidas).

En muchos casos, una declaración puede actuar simultáneamente como definición, es decir, reserva memoria para el elemento, determina su comportamiento y permite utilizarlo en el programa. En concreto, la declaración de una variable o array es también una definición. Desde este punto de vista, una sentencia de declaración puede denominarse igualmente sentencia de definición, pero esto no se ha convertido en una práctica habitual.

Nuestro conocimiento básico de las funciones es suficiente para suponer de forma fiable cómo debe ser su definición:

```
type identifier ( [list_of_arguments] )
{
  [statements]
}
```

El tipo, el identificador y la lista de argumentos conforman el encabezado de la función.

Tenga en cuenta que se trata de una definición, ya que esta descripción contiene tanto los atributos externos de la función (interfaz) como las sentencias que definen su esencia interna (implementación). Esto último se hace con un bloque de código formado por un par de llaves e inmediatamente después del encabezado de la función. Como puede adivinar, este es un ejemplo de la sentencia compuesta que mencionamos en la sección anterior. En este caso se hace indispensable una tautología terminológica, pues está perfectamente justificada: la sentencia compuesta forma parte de la sentencia de definición de la función.

Un poco más adelante descubriremos por qué y cómo separar la descripción de la interfaz de la implementación y conseguir así la declaración de función sin definirla. También demostraremos la diferencia entre una declaración y una definición utilizando la clase a modo de ejemplo.

La sentencia de declaración hace que el nuevo elemento esté disponible por su nombre en el contexto del bloque de código (véase Contexto, ámbito y vida útil de las variables) en el que se encuentra la sentencia. Recordemos que los bloques forman el ámbito local de los objetos (variables, arrays). En la primera parte del libro hablamos de ello al describir la función de saludo.

Además de los ámbitos locales, existe siempre un ámbito global en el que también se pueden utilizar sentencias de declaración para crear elementos accesibles desde cualquier parte del programa.

Si no hay ningún modificador *static* en la sentencia de declaración y ésta se encuentra en algún bloque local, entonces el elemento correspondiente se crea e inicializa en el momento en que se ejecuta la sentencia (en sentido estricto, la memoria para todas las variables locales dentro de la función se asigna, en aras de la eficiencia, nada más entrar en la función, pero aún no están formadas en ese momento).

Por ejemplo, la siguiente declaración de la variable *i* al principio de la función *OnStart* garantiza que dicha variable se creará con el valor inicial especificado (0) en cuanto la función reciba el control (es decir, el terminal la invocará porque es la función principal del script).

```
void OnStart()
{
   int i = 0;
   Print(i);

   // error: 'j' - undeclared identifier
   // Print(j);
   int j = 1;
}
```

Gracias a la declaración de la primera sentencia, la variable *i* es conocida y está disponible en las líneas posteriores de la función; en concreto, en la segunda línea con la llamada a la función *Print*, que muestra el contenido de la variable en el registro.

La variable *j* descrita en la última línea de la función se creará justo antes del final de la función (esto, por supuesto, es insignificante, pero claro). Por lo tanto, esta variable no se conoce en todas las cadenas anteriores de esta función. Si se intenta enviar *j* al registro mediante una llamada a *Print* comentada se producirá un error de compilación por «identificador no declarado».

Los elementos declarados de esta forma (dentro de bloques de código y sin el modificador *static*) se denominan automáticos, ya que el programa en sí les asigna memoria al entrar en el bloque y los destruye al salir del bloque (en nuestro caso, después de salir de la función). Por ello, la zona de memoria en la que esto ocurre se denomina pila («último en entrar, primero en salir»).

Los elementos automáticos se crean en el orden en que se ejecutan las sentencias de declaración (primero *i*, luego *j*). La destrucción se realiza en orden inverso (primero *j*, luego *i*).

Si se declara una variable sin inicializarla y se empieza a utilizar en sentencias posteriores (por ejemplo, a la derecha del signo '=') sin escribir primero en ella un valor significativo, el compilador emite un aviso: «posible uso de variable no inicializada».

```
void OnStart()
{
   int i, p;
   i = p; // warning: possible use of uninitialized variable 'p'
}
```

Si una sentencia de declaración tiene el modificador *static*, el elemento correspondiente se crea una sola vez cuando la sentencia se ejecuta por primera vez y permanece en memoria, independientemente de la salida y de posibles entradas y salidas posteriores en el mismo bloque de código. Todos estos miembros estáticos se eliminan sólo cuando se descarga el programa.

A pesar del aumento de la vida útil, el alcance de dichas variables sigue estando limitado al contexto local en el que se definen, y sólo se puede acceder a ellas desde sentencias posteriores (situadas más abajo en el código).

Por el contrario, las sentencias de declaración en el contexto global crean sus elementos en el mismo orden en el que aparecen en el código fuente, inmediatamente después de que se cargue el programa (antes de que se llame a cualquier función de inicio estándar, como *OnStart* para scripts). Los objetos globales se eliminan en orden inverso cuando se descarga el programa.

Para demostrar lo anterior vamos a crear un ejemplo más «ingenioso» (*StmtDeclaration.mq5*). Recordando los conocimientos adquiridos en la primera parte, además de *OnStart* escribiremos una

función sencilla *Init* que se utilizará en expresiones de inicialización de variables y registrará una secuencia de llamadas.

```
int Init(const int v)
{
   Print("Init: ", v);
   return v;
}
```

La función *Init* acepta un único parámetro *v* de tipo entero *int*, cuyo valor se devuelve al código de llamada (*sentencia return*).

Esto permite utilizarlo como una envoltura para establecer el valor inicial de una variable; por ejemplo, para dos variables globales:

```
int k = Init(-1);
int m = Init(-2);
```

El valor del argumento pasado se introduce en las variables *k* y *m* al llamar a la función y volver de ella. Sin embargo, dentro de *Init* sacamos de forma adicional el valor con *Print*, y así podemos hacer un seguimiento de cómo se crean las variables.

Tenga en cuenta que no podemos utilizar la función *Init* en la inicialización de variables globales por encima de su definición. Si intentamos mover la declaración de la variable *k* por encima de la declaración *Init* obtendremos el error «'Init' es un identificador desconocido». Esta limitación sólo funciona para la inicialización de variables globales, ya que las funciones también se definen globalmente, y el compilador construye una lista de dichos identificadores de una sola vez. En todos los demás casos, el orden de definición de las funciones en el código no es importante, ya que el compilador primero las registra todas en la lista interna y, a continuación, vincula una a otra sus llamadas desde los bloques. En particular, puede mover toda la función *Init* y la declaración de las variables globales *k* y *m* debajo de la función *OnStart*: ello no romperá nada.

Dentro de la función *OnStart* describiremos varias variables más utilizando *Init*: *i* y *j* locales, así como *n* estática. Para simplificar, todas las variables reciben valores únicos a fin de poder distinguirlas.

```
void OnStart()
{
   Print(k);

   int i = Init(1);
   Print(i);
   // error: 'n' - undeclared identifier
   // Print(n);
   static int n = Init(0);
   // error: 'j' - undeclared identifier
   // Print(j);
   int j = Init(2);
   Print(j);
   Print(n);
}
```

Los comentarios muestran intentos erróneos de llamar a las variables relevantes antes de que estén definidas.

Ejecute el script y obtenga el siguiente registro:

```
Init: -1
Init: -2
-1
Init: 1
1
Init: 0
Init: 2
2
0
```

Como podemos ver, las variables globales se inicializaron antes de llamar a la función *OnStart* , y exactamente en el orden en que se encontraban en el código. Las variables internas se crearon en la misma secuencia en que se escribieron sus sentencias de declaración.

Si una variable está definida pero no se utiliza en ninguna parte, el compilador emitirá un aviso de «variable 'nombre' no utilizada». Esto es señal de un posible error del programador.

De cara al futuro, digamos que con la ayuda de las sentencias de declaración o definición se pueden introducir en el programa no sólo elementos de datos (variables, arrays) o funciones, sino también nuevos tipos definidos por el usuario (estructuras, clases, plantillas, espacios de nombres) que aún no conocemos. Estas sentencias sólo pueden hacerse a nivel global, es decir, fuera de todas las funciones.

Tampoco es posible definir una función dentro de otra función. El siguiente código no se compilará:

```
void OnStart()
{
   int Init(const int v)
   {
      Print("Init: ", v);
      return v;
   }
   int i = 0;
}
```

El compilador generará un error: «las declaraciones de función sólo se permiten en el ámbito global, de espacio de nombres o de clase».

2.7.3 Sentencias simples (expresiones)

Las sentencias simples contienen expresiones tales como asignar nuevos valores o resultados de cálculos a variables, así como llamadas a funciones.

Formalmente, la sintaxis es la siguiente:

```
expression ;
```

El punto y coma del final es importante. Dado que los códigos fuente de MQL5 admiten formato libre, el ';' es el único delimitador que le indica al compilador dónde terminó la declaración anterior y comenzó la siguiente. Por regla general, las declaraciones se escriben en líneas separadas, como por ejemplo así:

```
int i = 0, j = 1, k;   // declaration statement
++i;                   // simple statement
j += i;                // simple statement
k = (i + 1) * (j + 1); // simple statement
Print(i, " ", j);      // simple statement
```

Sin embargo, las normas no prohíben la escritura de código abreviado:

```
int i=0,j=1;++i;j+=i;k=(i+1)*(j+1);Print(i," ",j);
```

Si no fuera por los ';', las expresiones adyacentes podrían «pegarse» silenciosamente y dar lugar a resultados no deseados. Por ejemplo, la expresión *x = y - 10 * z* bien podría ser dos: *x = y;* y *-10 * z;* (-10 con un signo menos unario). ¿Cómo es posible?

El hecho es que es sintácticamente permisible escribir una sentencia que realmente funcione en vano, es decir, que no guarde el resultado. He aquí otro ejemplo:

```
i + j; // warning: expression has no effect
```

El compilador emite un aviso del tipo «la expresión no tiene efecto». La posibilidad de construir tales expresiones es necesaria porque los tipos de objeto, que veremos en la Parte 3, permiten la sobrecarga de operadores, es decir, podemos sustituir el significado habitual de los símbolos de los operadores por algunas acciones específicas. Entonces, si el tipo de *i* y *j* no es *int*, sino alguna clase con una operación de suma sobrescrita, tal notación tendrá efecto y el compilador no emitirá ningún aviso.

Las sentencias simples sólo pueden escribirse dentro de sentencias compuestas. Por ejemplo, llamar a la función *Print* fuera de una función no funcionará:

```
Print("Hello ", Symbol());
void OnStart()
{
}
```

Obtendremos una cascada de errores::

```
'Print' - unexpected token, probably type is missing?
'Hello, ' - declaration without type
'Hello, ' - comma expected
'Symbol' - declaration without type
'(' - comma expected
')' - semicolon expected
')' - expressions are not allowed on a global scope
```

La más relevante, en este caso, es la última: «las expresiones no están permitidas en el contexto global».

2.7.4 Visión general de las sentencias de control

Las sentencias de control están diseñadas para organizar la ejecución no lineal de otras sentencias, incluidas declaraciones, expresiones y sentencias de control anidadas. Pueden dividirse en 3 tipos:

- sentencias de repetición o bucles
- sentencias condicionales para elegir una de varias ramas de acciones alternativas

- sentencias de salto que cambian, si es necesario, el comportamiento estándar de los dos primeros tipos de sentencias

Las sentencias repeat y select constan de un encabezado (cada una con una sintaxis diferente) seguido de una sentencia controlada. Si una parte gestionada necesita especificar varias sentencias, utiliza una sentencia compuesta. Esta función no está disponible para las sentencias de salto. Sólo mueven el puntero interno en base al cual el programa determina qué sentencia se va a ejecutar en ese momento, de acuerdo con unas reglas especiales que abordaremos en las siguientes secciones.

En el caso más sencillo, sin sentencias de control, las sentencias se ejecutan secuencialmente, una tras otra, tal y como están escritas en el bloque de código (en particular, en el cuerpo de la función principal *OnStart* para scripts). Si en un bloque de código se encuentra una expresión con una llamada a otra función, el programa, según el mismo principio lineal, comienza a ejecutar sentencias dentro de la función llamada y, una vez ejecutadas todas, volverá al bloque de código de llamada y la ejecución continuará en la siguiente sentencia después de la llamada a la función. Las sentencias de control pueden cambiar significativamente esta lógica de trabajo.

Puede utilizar la selección dentro de bucles o viceversa, y el nivel de anidamiento es ilimitado. Sin embargo, demasiado anidamiento hace que el programa sea difícil de entender para el programador. Por lo tanto, se recomienda asignar (transferir) bloques de código a funciones (una o varias): dentro de cada función, tiene sentido mantener un nivel de anidamiento no superior a 2-3.

En MQL5 se admiten las siguientes sentencias de repetición:

- *for* bucle
- *while* bucle
- *do* bucle

Todos los bucles permiten ejecutar una o varias sentencias un número determinado de veces o hasta que se cumpla alguna condición booleana. Ejecutar el contenido de un bucle una vez se denomina iteración. Por regla general, los arrays se procesan en bucles o se realizan acciones periódicas repetitivas (normalmente en scripts o servicios).

Las sentencias condicionales incluyen:

- selección con *if*
- selección con *switch*

La primera permite especificar una o varias condiciones, en función de cuya verdad o falsedad se ejecutarán las opciones que tengan asignadas (una o varias sentencias). La última evalúa una expresión de tipo entero y selecciona una de varias alternativas en función de su valor.

Por último, las sentencias de salto son:

- *break*
- *continue*
- *return*

Más adelante abordaremos cada uno de ellas en detalle.

A diferencia de C++, MQL5 no tiene una declaración *go to*.

2.7.5 Operador cíclico For

Este bucle se implementa mediante una sentencia con la palabra clave *for*, de ahí su nombre. De forma generalizada, puede describirse como sigue:

```
for ( [initialization] ; [condition] ; [expression] )
  loop body
```

En el título, tras la palabra 'for', se indica lo siguiente entre paréntesis:

- Inicialización: sentencia para una inicialización de una sola vez antes del inicio del bucle;
- Condición: condición booleana que se comprueba al principio de cada iteración, y el bucle se ejecuta mientras sea verdadera;
- Expresión: fórmula de los cálculos realizados al final de cada iteración, cuando se han pasado todas las sentencias del cuerpo del bucle.

El cuerpo del bucle es una sentencia simple o compuesta.

Los tres componentes del encabezado son opcionales y pueden omitirse en cualquier combinación, incluida su ausencia.

La inicialización puede incluir la declaración de variables (junto con el establecimiento de valores iniciales) o la asignación de valores a variables ya existentes. Dichas variables se denominan variables de bucle. Si se declaran en el encabezado, su alcance y duración se limitan al bucle.

El bucle comienza a ejecutarse si, tras la inicialización, la condición es *true*, y continúa ejecutándose mientras sea cierta al comienzo de cada iteración posterior. Si durante la siguiente comprobación se incumple la condición, el bucle sale, es decir, el control se transfiere a la sentencia escrita después del bucle y su cuerpo. Si la condición es *false* antes del inicio del bucle (después de la inicialización), nunca se ejecutará.

La condición y la expresión suelen incluir variables de bucle.

Ejecutar un bucle significa ejecutar su cuerpo.

La forma más común del bucle *for* tiene una única variable de bucle que controla el número de iteraciones. En el siguiente ejemplo calculamos los cuadrados de los números del array *a*.

```
int a[] = {1, 2, 3, 4, 5, 6, 7};
const int n = ArraySize(a);
for(int i = 0; i < n; ++i)
   a[i] = a[i] * a[i];
ArrayPrint(a);    // 1  4  9 16 25 36 49
// Print(i);      // error: 'i' - undeclared identifier
```

Este bucle se ejecuta en los siguientes pasos:

1. Se crea una variable *i* con un valor inicial 0.
2. Se comprueba si la variable *i* es menor que el tamaño del bucle *n*. Mientras ello sea cierto, el bucle continúa. Si es falso, saltamos a la sentencia que llama a la función *ArrayPrint* .
3. Si la condición es verdadera, se ejecutan las sentencias del cuerpo del bucle. En este caso, el *i*-ésimo elemento del array obtiene el producto del valor inicial de este elemento por sí mismo, es decir, el valor de cada elemento se sustituye por su cuadrado.
4. La variable *i* se incrementa en 1.

A continuación se repite todo, empezando por el paso 2. Después de salir del bucle, su variable *i* se destruye, y cualquier intento de acceder a ella causará un error.

La expresión para el paso 4 puede ser de complejidad arbitraria, no sólo un incremento de la variable del bucle. Por ejemplo, para iterar sobre elementos pares o impares, se podría escribir *i += 2*.

Independientemente del número de sentencias que compongan el cuerpo del bucle, se recomienda escribirlo en una línea (o líneas) separada del encabezado. Esto facilita el proceso de depuración paso a paso.

La inicialización puede incluir múltiples declaraciones de variables, pero deben ser del mismo tipo porque son una sola declaración. Por ejemplo, para reordenar los elementos en orden inverso, puede escribir un bucle de este tipo (esto es sólo una demostración del bucle; existe una función *ArrayReverse* integrada que permite invertir el orden en un array, véase Copia y edición de arrays):

```
for(int i = 0, j = n - 1; i < n / 2; ++i, --j)
{
   int temp = a[i];
   a[i] = a[j];
   a[j] = temp;
}
ArrayPrint(a);    // 49 36 25 16  9  4  1
```

La variable auxiliar *temp* se crea y elimina en cada pasada del bucle, pero el compilador le asigna memoria una sola vez, como a todas las variables locales, al entrar en la función. Esta optimización funciona bien para los tipos integrados. Sin embargo, si un objeto de clase personalizado se describe en el bucle, entonces su constructor y su destructor se llamarán en cada iteración.

> Es aceptable cambiar la variable de bucle en el cuerpo del bucle, pero esta técnica sólo se utiliza en casos muy excepcionales. No se recomienda hacerlo, ya que puede dar lugar a errores (en particular, pueden saltarse elementos procesados o la ejecución puede entrar en un bucle infinito).

Para demostrar la capacidad de omitir componentes de encabezado, imaginemos el siguiente problema: necesitamos encontrar el número de elementos del mismo array cuya suma es menor que 100. Para ello, necesitamos una variable contador *k* definida antes del bucle porque debe seguir existiendo después de su finalización. También crearemos la variable *sum* para calcular la suma de forma acumulativa.

```
int k = 0, sum = 0;
for( ; sum < 100; )
{
  sum += a[k++];
}

Print(k - 1, " ", sum - a[k - 1]); // 2 85
```

Así, no hay necesidad de hacer la inicialización en el encabezado. Además, el contador *k* se incrementa utilizando un post-incremento directamente en la expresión que calcula la suma (al acceder a un elemento del array). Por lo tanto, no necesitamos una expresión en el título.

Al final del bucle imprimimos *k* y la suma menos el último elemento añadido, porque fue el que superó nuestro límite de 100.

Observe que estamos utilizando un bloque compuesto aunque sólo haya una sentencia en el cuerpo del bucle. Esto es útil porque, cuando el programa crece, ya está todo listo para añadir sentencias

adicionales dentro de los corchetes. Además, este enfoque garantiza un estilo uniforme para todos los bucles. La elección, en cualquier caso, corresponde no obstante al programador.

En la versión explícita y más abreviada posible, el encabezado del ciclo podría tener este aspecto:

```
for( ; ; )
{
   // ...       // periodic actions
   Sleep(1000); // pause the program for 1 second
}
```

Si no hay sentencias en el cuerpo de un bucle de este tipo que interrumpan el bucle debido a algunas condiciones, este se ejecutará indefinidamente. Aprenderemos a romper y probar condiciones en *Break* jump y *If* selection respectivamente.

Estos algoritmos de bucle suelen utilizarse en los servicios (están diseñados para un trabajo constante en segundo plano) para supervisar el estado del terminal o de los recursos de red externos. Suelen contener sentencias que pausan el programa en un intervalo especificado, por ejemplo mediante la función integrada *Sleep*. Sin esta precaución, un bucle infinito cargará el 100 % del núcleo de un procesador.

El script *StmtLoopsFor.mq5* contiene un bucle infinito al final, pero es sólo para fines de demostración.

```
for( ; ; )
{
   Comment(GetTickCount());
   Sleep(1000); // 1000 ms

   // the loop can be exited only by deleting the script at the user's command
   // after 3 seconds of waiting we will get the message 'Abnormal termination'
}
Comment("");  // this line will never be executed
```

En el bucle, una vez por segundo, el temporizador interno del ordenador (*GetTickCount*) se muestra mediante la función *Comment*: el valor aparece en la esquina superior izquierda del gráfico. Sólo el usuario puede interrumpir el bucle borrando todo el script del gráfico (botón «Eliminar» del cuadro de diálogo Expertos). Este código no comprueba si el usuario solicita detenerse dentro del bucle, aunque existe una función integrada *IsStopped* para este fin. Devuelve *true* si el usuario ha dado la orden de parar. En el programa, especialmente si hay bucles y cálculos a largo plazo, es conveniente prever la comprobación del valor de esta función y terminar voluntariamente el bucle y todo el programa al recibir *true*. De lo contrario, el terminal terminará forzosamente el script al cabo de 3 segundos de espera (con salida al registro «Terminación anormal»), lo que ocurrirá en este ejemplo.

Una versión mejor de este bucle debería ser:

```
for( ; !IsStopped(); ) // continue until user interrupt
{
   Comment(GetTickCount());
   Sleep(1000); // 1000 ms
}
Comment("");    // will clear the comment
```

Sin embargo, este bucle se implementaría mejor utilizando otra sentencia de repetición *while*. Por regla general, un bucle *for* sólo debe utilizarse cuando hay una variable de bucle obvia y/o un número predeterminado de iteraciones. En este caso no se cumplen estas condiciones.

Las variables de bucle suelen ser números enteros, aunque se permiten otros tipos, como *double*. Esto se debe a que la propia lógica del funcionamiento del bucle implica la numeración de las iteraciones. Además, siempre es posible calcular los números reales necesarios a partir de un índice entero, y con mayor precisión. Por ejemplo, el siguiente bucle itera sobre valores de 0.0 a 1.0 en incrementos de 0.01:

```
for(double x = 0.0; x < 1.0; x += 0.01) { ... }
```

Se puede sustituir por un bucle similar con una variable entera:

```
for(int i = 0; i < 100; ++i) { double x = i * 0.01; ... }
```

En el primer caso, al sumar *x += 0.01*, el error de los cálculos de punto flotante se acumula gradualmente. En el segundo caso, cada valor *x* se obtiene en una operación *i * 0.01*, con la máxima precisión disponible.

Es habitual dar a las variables de bucle los siguientes nombres de una sola letra, como por ejemplo, *i, j, k, m, p, q*. Se necesitan varios nombres cuando se anidan bucles o se calculan índices hacia delante (crecientes) y hacia atrás (decrecientes) dentro del mismo bucle.

Por cierto, aquí tienes un ejemplo de bucle anidado. El siguiente código calcula y almacena la tabla de multiplicar en un array bidimensional.

```
int table[10][10] = {0};
for(int i = 1; i <= 10; ++i)
{
   for(int j = 1; j <= 10; ++j)
   {
      table[i - 1][j - 1] = i * j;
   }
}
ArrayPrint(table);
```

2.7.6 Operador cíclico while

Este bucle se describe utilizando la palabra clave *while*. Repite la ejecución de las sentencias controladas en tanto la expresión lógica de su encabezado sea verdadera.

```
while ( condition )
  loop body
```

La condición es una expresión arbitraria de tipo booleano. La presencia de la condición es obligatoria. Si la condición es *false* antes del inicio del bucle, éste nunca se ejecutará.

A diferencia de C++, MQL5 no admite la definición de variables en el encabezado del bucle *while* .

Las variables incluidas en la condición deben definirse antes del bucle.

El cuerpo del bucle es una sentencia simple o compuesta.

El bucle *while* suele utilizarse cuando el número de iteraciones no está definido. Así, un ejemplo con el bucle que da un contador de tiempo de ordenador cada segundo se puede escribir utilizando un bucle *while* y comprobando la bandera de parada (llamando a la función *IsStopped*) de la siguiente manera (*StmtLoopsWhile.mq5):*

```
while(!IsStopped())
{
   Comment(GetTickCount());
   Sleep(1000);
}
Comment("");
```

Además, el bucle *while* es conveniente cuando la condición de finalización del bucle puede combinarse con la modificación de variables en una expresión. El siguiente bucle se ejecuta hasta que la variable *i* llega a cero (0 se trata como *false*).

```
int i = 5;
while(--i) // warning: expression not boolean
{
   Print(i);
}
```

Sin embargo, en este caso, la expresión del encabezado no es booleana (y se convierte implícitamente a *false* o *true*). El compilador genera la advertencia correspondiente. Es conveniente componer siempre las expresiones teniendo en cuenta las características esperadas (según las normas). A continuación se muestra la versión correcta del bucle:

```
int i = 5;
while(--i > 0)
{
   Print(i);
}
```

El bucle también puede utilizarse con una sentencia simple (sin bloque):

```
while(i < 10)
   Print(++i);
```

Tenga en cuenta que una sentencia simple termina con punto y coma. También demuestra que el cambio de la variable que se comprueba en el encabezado se realiza dentro del bucle.

Cuando trabaje con bucles, tenga cuidado al utilizar enteros sin signo. Por ejemplo, el bucle siguiente no terminará nunca, porque su condición siempre es verdadera (en teoría, el compilador podría emitir avisos en esos lugares, pero no lo hace). Después de cero, el contador se «convertirá» en un número positivo grande (UINT_MAX) y el bucle continuará.

```
uint i = 5;
while(--i >= 0)
{
   Print(i);
}
```

Desde el punto de vista del usuario, el programa MQL se congelará (dejará de responder a los comandos), aunque seguirá consumiendo recursos (procesador y memoria).

while puede anidarse como otros tipos de sentencias de repetición.

2.7.7 Operador cíclico Do

Este bucle es similar al bucle *while*, pero su condición se comprueba después del cuerpo del bucle. Por ello, las sentencias controladas deben ejecutarse al menos una vez.

Se utilizan dos palabras clave, *do* y *while*, para describir el bucle:

```
do
  loop body
while ( condition ) ;
```

Así, el encabezado del bucle está separado, y después de la condición lógica entre paréntesis debe haber un punto y coma. La condición no puede omitirse. Cuando se convierte en falsa, el bucle sale.

Las variables incluidas en la condición deben definirse antes del bucle.

El cuerpo del bucle es una sentencia simple o compuesta.

En el siguiente ejemplo se calcula una secuencia de números empezando por 1, en la que cada número siguiente se obtiene multiplicando el anterior por la raíz cuadrada de dos, la constante predefinida M_SQRT2 (*StmtLoopsDo.mq5*).

```
double d = 1.0;
do
{
   Print(d);
   d *= M_SQRT2;
}
while(d < 100.0);
```

El proceso termina cuando el número supera 100.

2.7.8 Selección if

La sentencia *if* tiene varias formas. En el caso más sencillo, ejecuta la sentencia dependiente si la condición especificada es verdadera:

```
if ( condition )
  statement
```

Si la condición es falsa, la sentencia se omite y la ejecución salta inmediatamente al resto del algoritmo (sentencias posteriores, si las hay).

La sentencia puede ser simple o compuesta. Una condición es una expresión de tipo booleano o transformable.

La segunda forma le permite especificar dos ramas de acciones: no sólo para la condición verdadera (enunciado_A), sino también para la falsa (enunciado_B):

```
if ( condition )
  statement_A
else
  statement_B
```

Cualquiera de las sentencias controladas que se ejecute, el algoritmo continuará después siguiendo las sentencias por debajo de la sentencia *if/else* .

Por ejemplo, un script puede seguir una estrategia diferente según el marco temporal del gráfico en el que se coloque. Para ello, basta con analizar el valor devuelto por la función integrada *Period*. El valor es del tipo ENUM_TIMEFRAMES. Si es inferior a PERIOD_D1, significa trading a corto plazo; en caso contrario, trading a largo plazo (*StmtSelectionIf.mq5*).

```
if(Period() < PERIOD_D1)
{
   Print("Intraday");
}
else
{
   Print("Interday");
}
```

Como sentencia de la rama *else* se permite especificar el siguiente operador *if* y, así, ordenarlos en una cadena de comprobaciones sucesivas. Por ejemplo, el siguiente fragmento cuenta el número de letras mayúsculas y símbolos de puntuación (más exactamente, letras no latinas) de una cadena.

```
string s = "Hello, " + Symbol();
int capital = 0, punctuation = 0;
for(int i = 0; i < StringLen(s); ++i)
{
   if(s[i] >= 'A' && s[i] <= 'Z')
      ++capital;
   else if(!(s[i] >= 'a' && s[i] <= 'z'))
      ++punctuation;

}
Print(capital, " ", punctuation);
```

El bucle se organiza a través de todos los caracteres de la cadena (la numeración empieza por 0) y la función *StringLen* devuelve la longitud de la cadena. El primer *if* comprueba cada carácter para ver si pertenece al rango de la 'A' a la 'Z' y, si tiene éxito, incrementa el contador de mayúsculas en 1. Si el carácter no pertenece a este rango se ejecuta el segundo *if* , en el que la condición de pertenencia al rango de letras minúsculas (*s[i] >= 'a' && s[i] <= 'z'*) se invierte con '!'. En otras palabras: la condición significa que el carácter no se encuentra en el intervalo dado. Dadas dos comprobaciones consecutivas, si el carácter no es una letra mayúscula (*else*) y tampoco una letra minúscula (segundo *if*), podemos concluir que el carácter no es una letra del alfabeto latino. En este caso, incrementamos el contador *punctuation* .

Las mismas comprobaciones podrían redactarse de forma más detallada, con bloques "{...}" para mayor claridad.

```
int capital = 0, small = 0, punctuation = 0;
for(int i = 0; i < StringLen(s); ++i)
{
   if(s[i] >= 'A' && s[i] <= 'Z')
   {
      ++capital;
   }
   else
   {
      if(s[i] >= 'a' && s[i] <= 'z')
      {
         ++small;
      }
      else
      {
         ++punctuation;
      }
   }
}
```

El uso de llaves ayuda a evitar errores lógicos asociados que pueden producirse cuando el programador sólo se guía por la sangría en el código. En concreto, el problema más común es el denominado *else* «colgante».

Cuando las sentencias *if* están anidadas, a veces hay menos ramas *else* que *if*. He aquí un ejemplo:

```
factor = 0.0;
if(mode > 10)
   if(mode > 20)
      factor = +1.0;
else
   factor = -1.0;
```

La sangría indica a qué tipo de lógica se refería el programador: *factor* debe convertirse en +1 cuando *mode* es mayor que 20, permanecer igual a 0 cuando *mode* está entre 10 y 20, y cambiar a -1 en caso contrario (*mode* <= *10*). Pero, ¿funcionará así el código?

En MQL5, se da por sentado que cada *else* hace referencia al *if* anterior más cercano (que no tiene un *else*). Como resultado, el compilador tratará las sentencias de la siguiente manera:

```
factor = 0.0;
if(mode > 10)
   if(mode > 20)
      factor = +1.0;
   else
      factor = -1.0;
```

Así, *factor* será -1 en el rango *mode* de 10 a 20, y 0 para *mode* <= *10*. Lo más interesante es que el programa no produce ningún error formal, ni durante la compilación ni durante la ejecución. Y, sin embargo, no funciona correctamente.

Para eliminar esos sutiles problemas lógicos se permite la colocación de llaves.

```
if(mode > 10)
{
   if(mode > 20)
      factor = +1.0;
}
else
   factor = -1.0;
```

Para mantener la coherencia del diseño es conveniente utilizar bloques en todas las ramas de la sentencia si ya se ha requerido al menos un bloque en ella.

Cuando utilice el bucle para comprobar la igualdad, tenga en cuenta la posibilidad de que se produzca un error tipográfico al escribir un '=' en lugar de dos caracteres '==='. Esto convierte la comparación en una asignación, y el valor asignado se analiza como una condición lógica. Por ejemplo:

```
// should have been x == y + 1, which would give false and skip the if
if(x = y + 1) // warning: expression not boolean
{
   // assigned x = 5 and treated x as true, so if is executed
}
```

El compilador producirá una advertencia «expresión no booleana».

2.7.9 Operador de selección switch

El operador *switch* permite elegir una de varias opciones de algoritmo. Por regla general, el número de opciones es significativamente superior a dos, ya que, de lo contrario, es más fácil utilizar la sentencia *if/else*. En teoría, la cadena de sentencias *if/else* permite tener un equivalente de *switch* en muchos casos (pero no en todos). Una característica importante de *switch* es que todas las opciones se seleccionan (identifican) en función del valor de la expresión entera, normalmente una variable.

En general, la sentencia *switch* tiene el siguiente aspecto:

```
switch ( expression )
{
   case constant-expression : statements [break; ]
   ...
   [ default : statements ]
}
```

El encabezado de la sentencia comienza con la palabra clave *switch*. Esta debe ir seguida de una expresión entre paréntesis y también se necesita el bloque con corchetes.

Los valores enteros que pueden obtenerse evaluando una expresión deben especificarse como constantes después de la palabra clave *case*. Una constante es un literal de cualquier tipo entero, como por ejemplo *int* (10, 123), *ushort* (caracteres 'A', 's', '*', etc.), o elementos enum. Aquí no se permiten números reales, variables ni expresiones.

Puede haber muchas de estas opciones *case*, o puede no haber ninguna, lo que se indica mediante corchetes semicirculares con el índice $^{opt(n)}$. Todas las variantes deben tener constantes únicas (sin repeticiones).

Para cada alternativa declarada con *case* debe escribirse una expresión después de los dos puntos, que se ejecutará si el valor de la expresión es igual a la constante correspondiente. De nuevo, una afirmación puede ser simple o compuesta. Además, es posible escribir varias sentencias simples sin encerrarlas entre llaves: seguirán ejecutándose como un grupo (sentencia compuesta).

Una o varias de estas sentencias pueden ir seguidas de la sentencia de salto *break*.

Si hay un *break*, después de ejecutar las sentencias anteriores de la rama *case*, la sentencia *switch* sale, es decir, el control se transfiere a las sentencias situadas por debajo de *switch*.

En ausencia de *break*, las sentencias de la rama siguiente o de varias ramas *case* continúan ejecutándose, es decir, hasta que se encuentra la primera *break* o el final del bloque *switch.* Esto se denomina «fall-through».

Así, la sentencia *switch* no sólo permite dividir el flujo de ejecución del algoritmo en varias alternativas, sino también combinarlas, algo que no está disponible para el operador *if*. Por otro lado, en la sentencia *switch*, a diferencia de *if*, no se puede seleccionar un rango de valores como condición para activar alternativas.

La palabra clave *default* permite establecer la variante del algoritmo por defecto, es decir, para cualesquiera otros valores de la expresión excepto para las constantes de todos los *cases*. La opción *default* puede no estar presente, o debe haber sólo una.

La secuencia en la que se enumeran las constantes *case* y *default* puede ser arbitraria.

Aunque todavía no exista algoritmo para la rama *default*, se recomienda hacerla explícitamente vacía, es decir, que contenga *break*. Un *default* vacío le recordará a usted y a otros programadores que existen otras opciones pero que no se consideran importantes, ya que, de lo contrario, la rama *default* tendría que señalar un error.

Varias variantes de *case* con diferentes constantes pueden enumerarse una debajo de otra (o de izquierda a derecha) sin sentencias, pero la última debe tener una sentencia. Estos *cases* combinados se indican en el diagrama con el índice [(i)].

He aquí el *switch* más fácil y más inútil:

```
switch(0)
{
}
```

Veamos un ejemplo más complejo con diferentes modos (*StmtSelectionSwitch.mq5*). En él, el operador *switch* se coloca dentro del bucle para mostrar cómo su trabajo depende de los valores de la variable de control *i*.

```
for(int i = 0; i < 7; i++)
{
   double factor = 1.0;

   switch(i)
   {
      case -1:
         Print("-1: Never hit");
         break;
      case 1:
         Print("Case 1");
         factor = 1.5;
         break;
      case 2: // fall-through, no break (!)
         Print("Case 2");
         factor *= 2;
      case 3: // same statements for 3 and 4
      case 4:
         Print("Case 3 & 4");
         {
            double local_var = i * i;
            factor *= local_var;
         }
         break;
      case 5:
         Print("Case 5");
         factor = 100;
         break;
      default:
         Print("Default: ", i);
   }

   Print(factor);
}
```

La opción -1 fallará porque el bucle cambia la variable *i* de 0 a 6 (ambos inclusive). Cuando *i* sea 0, se activará la rama *default*. También tomará el control cuando *i* sea igual a 6. Todos los demás valores *i* posibles se distribuyen según las directivas correspondientes de *case*. Al mismo tiempo, no hay ninguna sentencia *break* después del caso 2, por lo que el código de las opciones 3 y 4 se ejecutará además del 2 (en estos casos, siempre se recomienda dejar un comentario indicando que se ha hecho intencionadamente).

Los casos 3 y 4 tienen un bloque de sentencias común. No obstante, también es importante señalar aquí que si desea declarar una variable local dentro de una de las opciones de *case* , debe encerrar las sentencias en un bloque compuesto anidado ('{...}'). Aquí, se define de esa manera la variable *local_var*.

Conviene advertir que en el caso *default* no hay sentencia *break*. Esto es redundante porque *default* se escribe en último lugar en este caso. Sin embargo, muchos programadores aconsejan insertar *break* al final de cualquier opción, incluso de la última, porque puede dejar de ser la última en el proceso de modificaciones posteriores del código, y entonces es fácil olvidarse de añadir *break*, lo que probablemente conducirá a un error en la lógica del programa.

Si en *switch* no hay *default* y la expresión del encabezado no coincide con ninguna de las constantes de *case*, se omite *switch* por completo.

Como resultado de la ejecución del script obtendremos los siguientes mensajes en el log:

```
Default: 0
1.0
Case 1
1.5
Case 2
Case 3 & 4
8.0
Case 3 & 4
9.0
Case 3 & 4
16.0
Case 5
100.0
Default: 6
1.0
```

2.7.10 Operador break

El operador *break* está pensado para terminar de forma anticipada de los operadores cíclicos *for*, *while*, *do*, así como a la salida de la sentencia de selección *switch*. El operador sólo puede aplicarse dentro de las sentencias especificadas y sólo afecta a la que contiene inmediatamente a *break* si hay varias anidadas. Tras procesar la sentencia *break*, la ejecución del programa continúa hasta la sentencia siguiente al operador cíclico interrumpido o *switch*.

La sintaxis es muy sencilla: la palabra clave *break* y un punto y coma:

```
break ;
```

Cuando se utiliza dentro de bucles, *break* suele implementarse en una de las ramas del operador condicional *if/else*.

Considere un script que imprime el contador de tiempo actual del sistema una vez por segundo, pero no más de 100 veces. Contempla la gestión de la interrupción del proceso por parte del usuario; para ello, la función *IsStopped* se sondea en el operador condicional *if* y su sentencia dependiente contiene *break* (*StmtJumpBreak.mq5*).

```
int count = 0;
while(++count < 100)
{
   Comment(GetTickCount());
   Sleep(1000);
   if(IsStopped())
   {
      Print("Terminated by user");
      break;
   }
}
```

En el siguiente ejemplo se rellena una matriz diagonal con una tabla de multiplicar (la esquina superior derecha permanecerá rellena de ceros).

```
int a[10][10] = {0};
for(int i = 0; i < 10; ++i)
{
   for(int j = 0; j < 10; ++j)
   {
      if(j > i)
         break;
      a[i][j] = (i + 1) * (j + 1);
   }
}
ArrayPrint(a);
```

Cuando la variable del bucle interno *j* es mayor que la variable del bucle externo *i*, la sentencia *break* rompe el bucle interno. Por supuesto, esta no es la mejor manera de rellenar la matriz diagonalmente; sería más fácil hacer un bucle sobre *j* de 0 a *i* sin ningún *break*, pero aquí demuestra la presencia de construcciones equivalentes con *break* y sin *break*.

Aunque las cosas pueden no ser tan obvias en proyectos de producción, se recomienda evitar el operador *break* siempre que sea posible y sustituirlo por variables adicionales (por ejemplo, una variable booleana con un nombre *needAbreak* «revelador»), que deberían utilizarse en expresiones terminales en los encabezados de bucle para romperlas de la forma estándar.

Imagine que se utilizan dos bucles anidados para encontrar caracteres duplicados en una cadena. El primer bucle actualiza secuencialmente cada carácter de la cadena y el segundo recorre los caracteres restantes (hacia la derecha).

```
string s = "Hello, " + Symbol();
ushort d = 0;
const int n = StringLen(s);
for(int i = 0; i < n; ++i)
{
   for(int j = i + 1; j < n; ++j)
   {
      if(s[i] == s[j])
      {
         d = s[i];
         break;
      }
   }
}
```

Si los caracteres de las posiciones *i* y *j* coinciden, recuerde el carácter duplicado y salga del bucle mediante *break*.

Se podría dar por sentado que la variable *d* debería contener la letra 'l' tras la ejecución de este fragmento. Sin embargo, si coloca el script en el instrumento más popular «EURUSD», la respuesta será 'U'. El caso es que *break* sólo rompe el bucle interno, y tras encontrar el primer duplicado ('ll' en la palabra «Hola»), el bucle continúa en *i*. Por lo tanto, para salir de varios bucles anidados a la vez, deben tomarse medidas adicionales.

La forma más popular es incluir en la condición del bucle exterior (o de todos los bucles exteriores) una variable que se rellena en el bucle interior. En nuestro caso, ya existe una variable de este tipo: *d*.

```
for(int i = 0; i < n && d == 0; ++i)
{
   for(int j = i + 1; j < n; ++j)
   {
      if(s[i] == s[j])
      {
         d = s[i];
         break;
      }
   }
}
```

Al comprobar que *d* es igual a 0 se detendrá el bucle exterior después de encontrar el primer duplicado. Pero la misma comprobación puede añadirse al bucle interno, lo que elimina la necesidad de utilizar *break*.

```
for(int i = 0; i < n && d == 0; ++i)
{
   for(int j = i + 1; j < n && d == 0; ++j)
   {
      if(s[i] == s[j])
      {
         d = s[i];
      }
   }
}
```

2.7.11 Operador continue

La sentencia *continue* rompe la iteración actual del bucle más interno que contiene *continue* e inicia la siguiente iteración. La sentencia sólo puede utilizarse dentro de los bucles *for*, *while* y *do*. La ejecución de *continue* dentro de *for* da lugar al siguiente cálculo de la expresión en el encabezado del bucle (incremento/decremento de la variable del bucle), tras lo cual se comprueba la condición de continuación del bucle. Ejecutando *continue* dentro de *while* o *do* se comprueba inmediatamente la condición en el encabezado del bucle.

La sentencia consta de la palabra clave *continue* y un punto y coma:

```
continue ;
```

Suele situarse en una de las ramas de la sentencia condicional *if/else* o *switch* .

Por ejemplo, podemos generar una tabla de multiplicar con huecos: cuando el producto de dos índices sea impar, el elemento del array correspondiente seguirá siendo cero (*StmtJumpContinue.mq5*).

```
int a[10][10] = {0};
for(int i = 0; i < 10; ++i)
{
   for(int j = 0; j < 10; ++j)
   {
      if((j * i) % 2 == 1)
         continue;
      a[i][j] = (i + 1) * (j + 1);
   }
}
ArrayPrint(a);
```

Y así es como puede calcular la suma de los elementos positivos de un array:

```
int b[10] = {1, -2, 3, 4, -5, -6, 7, 8, -9, 10};
int sum = 0;
for(int i = 0; i < 10; ++i)
{
   if(b[i] < 0) continue;
   sum += b[i];
}
Print(sum); // 33
```

Observe que el mismo bucle puede reescribirse sin *continue* pero con un mayor anidamiento de bloques de código:

```
for(int i = 0; i < 10; ++i)
{
   if(b[i] >= 0)
   {
      sum += b[i];
   }
}
```

Así, el operador *continue* se utiliza a menudo para simplificar el formato del código (especialmente si hay que pasar varias condiciones). No obstante, la elección de uno u otro enfoque es una cuestión de preferencias personales.

2.7.12 Operador return

El operador *return* está diseñado para devolver el control de las funciones. Dado que todas las sentencias ejecutables están dentro de una función en concreto, se puede utilizar de forma indirecta para interrumpir conteniendo bucles *for*, *while*, y *do* de cualquier nivel de anidamiento. Hay que tener en cuenta que, a diferencia de *continue* y, sobre todo, de *break*, también se ignorarán todas las sentencias que sigan a bucles interrumpidos dentro de la función.

Sintaxis del operador *return* :

```
return ([expression]) ;
```

La necesidad de especificar una expresión viene determinada por la firma de la función (encontrará más información sobre este tema en la sección correspondiente). Para comprender de forma general cómo funciona *return* en el contexto de las sentencias de control, veamos un ejemplo con la función de script principal *OnStart*. Puesto que es del tipo *void*, es decir, no devuelve nada, el operador adopta la siguiente forma:

```
return ;
```

En la sección acerca de *break* implementamos un algoritmo para encontrar caracteres duplicados en una cadena. Para romper dos bucles anidados, no sólo utilizamos *break*, sino que también modificamos la condición del bucle exterior.

Con el operador *return*, esto puede hacerse de una forma más sencilla (*StmtJumpReturn.mq5*).

```
void OnStart()
{
   string s = "Hello, " + Symbol();
   const int n = StringLen(s);
   for(int i = 0; i < n; ++i)
   {
      for(int j = i + 1; j < n; ++j)
      {
         if(s[i] == s[j])
         {
            PrintFormat("Duplicate: %c", s[i]);
            return;
         }
      }
   }

   Print("No duplicates");
}
```

Si se encuentra la igualdad en el operador *if*, mostramos el símbolo y salimos de la función. Si este algoritmo estuviera en una función personalizada distinta de *OnStart* podríamos definir un tipo de retorno para ella (por ejemplo, *ushort* en lugar de *void*) y pasar el carácter encontrado utilizando la forma completa *return* al código de llamada.

Como se sabe que la letra doble 'l' existe en la cadena de prueba, la sentencia que sigue a los bucles (*Print*) no se ejecutará.

2.7.13 Sentencia vacía

La sentencia vacía es la más sencillo del lenguaje. Consta de un solo carácter: el punto y coma ';'

Una sentencia vacía se utiliza en el programa en aquellos lugares en los que la sintaxis requiere la presencia de una sentencia, pero la lógica del algoritmo ordena no hacer nada.

Por ejemplo, el siguiente bucle *while* se utiliza para encontrar un espacio en una cadena. La esencia entera del algoritmo se realiza directamente en el encabezado del bucle, por lo que su cuerpo debe estar vacío. Podríamos escribir un bloque vacío de llaves, pero aquí también funcionaría una sentencia vacía. (*StmtNull.mq5*).

```
int i = 0;
ushort c;
string s = "Hello, " + Symbol();
while((c = s[i++]) != ' ' && c != 0); // intentional ';' (!)
if(c == ' ')
{
   Print("Space found at: ", i);
}
```

Tenga en cuenta que si se omite el punto y coma al final del encabezado *while* (quizás por accidente), entonces la sentencia *if* será tratada como el cuerpo del bucle. Como resultado, la función *Print* no enviará ningún mensaje al registro. De hecho, el programa no funcionará correctamente, aunque sin errores perceptibles.

La situación contraria también es posible: un punto y coma extra después del encabezado del bucle (donde no debería haber estado) «separará» el cuerpo del bucle de la cabecera, es decir, sólo se ejecutará una sentencia vacía en el bucle.

En este sentido, se debe comprobar la presencia de puntos y comas opcionales en el código, y siempre que se coloquen intencionadamente, dejar un comentario con explicaciones.

Por cierto: desde un punto de vista formal, la sentencia vacía también se utiliza en la sentencia *for* cuando omitimos la expresión de inicialización. De hecho, siempre hay inicialización:

```
for ( [initialization] ; [end loop condition]; [post-expression] )
  loop body
```

El primer carácter ';' forma parte de una sentencia de inicialización, que puede ser una expresión o una sentencia vacía; ambas contienen el carácter ';' al final, y la segunda no contiene nada más que ';'. Así se alcanza la opcionalidad (vaciedad).

2.8 Funciones

Una función es un bloque con nombre y sentencias. Casi todo el algoritmo de aplicación del programa está contenido en funciones. Fuera de las funciones sólo se realizan operaciones auxiliares, como crear y eliminar variables globales.

La ejecución de sentencias dentro de una función se produce cuando invocamos esa función. Algunas funciones, las principales, son invocadas automáticamente por el terminal cuando se producen diversos eventos. También se conocen como los puntos de entrada del programa MQL o controladores de eventos. En concreto, ya sabemos que cuando ejecutamos un script en un gráfico, el terminal llama a su función principal *OnStart*. En otros tipos de programas existen otras funciones invocadas por el terminal, de las que hablaremos en detalle en los capítulos quinto y sexto que tratan de la arquitectura de trading de la API de MQL5.

En este capítulo descubriremos cómo definir y declarar una función, describir y pasarle parámetros, y devolver el resultado de su trabajo desde la función.

También hablaremos de la sobrecarga de funciones, es decir, la posibilidad de proporcionar varias funciones con el mismo nombre, y de lo útil que esto puede resultar.

Por último, nos familiarizaremos con un nuevo tipo: un puntero a una función.

2.8.1 Definición de funciones

Una definición de función consta del tipo de valor que devuelve, un identificador, una lista de parámetros entre paréntesis y un cuerpo, es decir, un bloque de código con sentencias. Los argumentos de la lista están separados por comas. A cada parámetro se le asigna un tipo, un nombre y, opcionalmente, un valor por defecto.

```
result_type function_identifier ( [parameter_type parameter_identifier
                                        = value_by_default] ,... )
{
  [statement]
   ...
}
```

Se permite crear funciones sin parámetros: en ese caso no hay lista, y se colocan corchetes vacíos después del nombre de la función (no se pueden omitir). Opcionalmente, puede escribir la palabra clave *void* entre los corchetes para enfatizar que no hay parámetros. Por ejemplo, así:

```
void OnStart(void)
{
}
```

La combinación de tipo de retorno, número y tipos de parámetros de la lista se denomina prototipo o firma de función. Diferentes funciones pueden tener el mismo prototipo.

En secciones anteriores hemos visto ya definiciones de funciones como *OnStart* y *Greeting*. Ahora vamos a intentar implementar el cálculo de los números de Fibonacci como función de prueba. Estas cifras se calculan mediante la siguiente fórmula:

```
f[0] = 1
f[1] = 1
f[i] = f[i - 1] + f[i - 2], i > 1
```

Los dos primeros números son 1, y todos los números siguientes son la suma de los dos anteriores. Damos el comienzo de la serie: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...

Puede calcular el número en un índice dado utilizando la siguiente función (*FuncFibo.mq5*).

```
int Fibo(const int n)
{
   int prev = 0;
   int result = 1;
   for(int i = 0; i < n; ++i)
   {
      int temp = result;
      result = result + prev;
      prev = temp;
   }
   return result;
}
```

Toma un parámetro *n* de tipo *int* y devuelve un resultado de tipo *int*. El parámetro *n* tiene el modificador *const* porque no vamos a cambiar *n* dentro de la función (una declaración tan explícita de las restricciones sobre los «derechos» de las variables es bienvenida porque ayuda a evitar errores aleatorios).

Las variables locales *prev* y *result* almacenarán los valores actuales de los dos últimos números de la serie. En el bucle sobre *i* calculamos su suma, obteniendo el siguiente número de la secuencia. Previamente, el valor antiguo *result* se escribe en la variable *temp*, a fin de que después de la suma se transfiera a *prev*.

Tras ejecutar el bucle un número determinado de veces, la variable *result* contiene el número deseado. Lo devolvemos desde la función utilizando la sentencia *result*.

El parámetro de entrada de una función es también una variable local que se inicializará con el valor real durante la llamada a la función. Este valor se pasa «fuera» de la sentencia con la llamada a la función.

Los nombres de los parámetros deben ser únicos y no deben coincidir con los nombres de las variables locales.

El cuerpo de una función es un bloque de código que define el ámbito y la vida útil de las variables locales. Su definición y principios de funcionamiento se han abordado en las secciones Sentencias de declaración y definición e Inicialización.

2.8.2 Llamada a una función

Se llama a una función cuando se menciona su nombre en una expresión. Tras el nombre, debe haber un par de paréntesis, en los que se indican los argumentos correspondientes a los parámetros de la función (si existe una lista de parámetros en su definición), separados por comas.

Un poco más adelante examinaremos el tipo puntero de función, que permite crear variables que apunten a una función con características específicas, y luego llamarla no por su nombre, sino a través de esta variable.

Siguiendo con el ejemplo de la función *Fibo*, vamos a llamarla desde la función *OnStart*. Para ello, vamos a crear una variable *f* en la que almacenar el número resultante, y en su expresión de inicialización indicamos el nombre de la función *Fibo* y un número entero (por ejemplo, 10) como argumento, entre paréntesis.

```
void OnStart()
{
   int f = Fibo(10);
   Print(f); // 89
}
```

No es necesario crear una variable para recibir un valor de una función. En lugar de ello, puede llamar a la función directamente desde una expresión, como «2*Fibo(10)» o «Print(Fibo(10))». A continuación, su valor se sustituirá en la expresión en el lugar de la llamada. Aquí, la variable auxiliar *f* se introduce para implementar la llamada y devolución de un valor en una sentencia separada.

El proceso de llamada incluye los siguientes pasos:

- Se suspende la ejecución de la secuencia de sentencias de la función de llamada (*OnStart*);
- El valor del argumento se introduce en el parámetro de entrada *n* de la función de llamada (*Fibo*);
- Comienza la ejecución de sus sentencias;
- Cuando está totalmente terminada, devuelve el resultado (recuerde la sentencia *return* que hay dentro);
- El resultado se escribe en la variable *f*;
- A continuación, la ejecución de la función *OnStart* continúa, es decir, el número se imprime en el registro (*Print*).

Para cada llamada a función, el compilador genera código binario auxiliar (el programador no tiene que preocuparse de ello). La idea de este código es que antes de llamar a la función, coloca la posición actual del programa en la pila, y una vez completada la llamada, la recupera y la utiliza para volver a las sentencias siguientes a la llamada a la función. Cuando una función llama a otra, ésta llama a una función más, la segunda llama a una tercera, y así sucesivamente, las direcciones de retorno de las transiciones a lo largo de la jerarquía de funciones llamadas se acumulan en la pila (de ahí el nombre de pila). A medida que se procesan las llamadas a funciones anidadas, la pila se borrará en orden inverso. Tenga en cuenta que la pila también asigna memoria para las variables locales de cada función.

2.8.3 Parámetros y argumentos

Los argumentos pasados a la función con su llamada son los valores iniciales de los parámetros correspondientes de la función. El número, el orden y los tipos de argumentos deben coincidir con el prototipo de la función. Sin embargo, el orden en que se calculan los argumentos no está definido (véase la sección Conceptos básicos). Dependiendo de las particularidades del código fuente y de las consideraciones en materia de optimización, el compilador puede elegir una opción que le resulte conveniente. Por ejemplo, dada una lista de dos argumentos, el compilador podría evaluar primero el segundo argumento y después, el primero. Sólo se garantiza que ambos argumentos se evalúen antes de la llamada.

Cada argumento se asigna al parámetro correspondiente de la misma manera que las variables se inicializan, con conversiones implícitas si es necesario. Antes de que se inicie la función se garantiza que todos sus parámetros tengan los valores especificados. Por ejemplo, dependiendo de los argumentos pasados, las llamadas a la función *Fibo* pueden provocar los siguientes efectos (descritos en los comentarios):

```
// warnings
double d = 5.5;
Fibo(d);          // possible loss of data due to type conversion
Fibo(5.5);        // truncation of constant value
Fibo("10");       // implicit conversion from 'string' to 'number'
// errors
Fibo();           // wrong parameters count
Fibo(0, 10);      // wrong parameters count
```

Todas las advertencias se refieren a conversiones implícitas que el compilador realiza porque los tipos de los valores no coinciden con los tipos de los parámetros. Deben considerarse errores potenciales y eliminarse. El error «recuento incorrecto de parámetros» se produce cuando demasiados argumentos o demasiado pocos.

En teoría, un parámetro de función no tiene por qué tener un nombre, es decir, el tipo por sí solo es suficiente para describir el parámetro.argumentos o demasiado pocos. Esto suena bastante extraño porque no podremos acceder a un parámetro sin nombre dentro de la función. Sin embargo, al crear programas basados en algunas interfaces estándar, a veces hay que escribir funciones que deben corresponderse a prototipos dados. En este caso, algunos parámetros dentro de la función pueden ser innecesarios. Entonces, para indicar explícitamente este hecho, el programador puede omitir sus nombres. Por ejemplo, la API de MQL5 requiere la implementación de la función de manejo de eventos *OnDeinit* con el siguiente prototipo:

```
void OnDeinit(const int reason);
```

Si no necesitamos el parámetro *reason* en el código de la función, podemos omitirlo en la descripción:

```
void OnDeinit(const int);
```

La función de gestión de eventos del terminal suele ser invocada por el propio terminal, pero si necesitáramos llamar a una función similar (con un parámetro anónimo) desde nuestro código, entonces necesitaríamos pasar todos los argumentos, independientemente de si los parámetros tienen nombre o no.

2.8.4 Parámetros de valor y parámetros de referencia

Los argumentos pueden pasarse a una función de dos formas: por valor y por referencia.

Todos los casos que hemos visto hasta ahora pasan por valor. Esta opción significa que el valor del argumento preparado por el fragmento de código de llamada se copia en una nueva variable, la variable de entrada correspondiente de la función. En caso contrario, el argumento y la variable de entrada no están relacionados. Todas las manipulaciones posteriores con la variable dentro de la función no afectan en modo alguno al argumento.

Para describir un parámetro de referencia, añada un signo ampersand '&' a la derecha del tipo. Muchos programadores prefieren añadir un «ampersand» al nombre de un parámetro, enfatizando así que el parámetro es una referencia al tipo dado. Por ejemplo, las siguientes entradas son equivalentes:

```
void func(int &parameter);
void func(int & parameter);
void func(int& parameter);
```

Cuando se llama a una función no se crea una variable local correspondiente para un parámetro de referencia. En lugar de ello, el argumento especificado para este parámetro pasa a estar disponible dentro de la función con el nombre (alias) del parámetro de entrada. Así, el valor no se copia, sino que se utiliza en la misma dirección de la memoria. Por lo tanto, las modificaciones de un parámetro dentro de una función se reflejan en el estado de su argumento asociado. De ello se desprende una característica importante.

Sólo se puede especificar una variable (LValue, véase Operador de asignación) como argumento para un parámetro de referencia. De lo contrario, obtendremos el error «parámetro pasado como referencia, variable esperada».

El paso por referencia se utiliza en varios casos:

- Para mejorar la eficacia del programa eliminando la copia del valor;
- Para pasar datos modificados de una función al código de llamada cuando devolver un único valor con *return* no es suficiente.

El primer punto es especialmente relevante para variables potencialmente grandes, como cadenas o arrays.

Para distinguir entre la primera y la segunda finalidad de un parámetro de referencia, se recomienda a los autores de la función que añadan el modificador const cuando no se espere que cambie el parámetro dentro de la función. Esto le recordará a Ud. y les dejará claro a otros desarrolladores que pasar una variable dentro de una función no provocará efectos secundarios.

Si no se aplica el modificador *const* a los parámetros de referencia allí donde sea posible, pueden surgir problemas en toda la jerarquía de llamadas a funciones. El hecho es que llamar a tales funciones requerirá argumentos no constantes. De lo contrario, se producirá el error «la variable constante no puede pasarse como referencia». Como resultado, puede resultar gradualmente que todos los

parámetros de todas las funciones deban despojarse del modificador *const* por el bien de la capacidad de compilación del código. De hecho, esto amplía el ámbito de posibles errores con la corrupción involuntaria de variables. La situación debería corregirse a la inversa: poner *const* allí donde no se requiera la devolución y modificación de valores.

Para comparar las formas de pasar parámetros en el script *FuncDeclaration.mq5* se implementan varias funciones: *FuncByValue* — paso por valor, *FuncByReference* — paso por referencia, *FuncByConstReference* — paso por referencia constante.

```
void FuncByValue(int v)
{
   ++v;
   // we are doing something else with v
}

void FuncByReference(int &v)
{
   ++v;
}

void FuncByConstReference(const int &v)
{
   // error
   // ++v; // 'v' - constant cannot be modified
   Print(v);
}
```

En la función *OnStart* invocamos todas estas funciones y observamos su efecto en la variable *i* utilizada como argumento. Tenga en cuenta que pasar un parámetro por referencia no cambia la sintaxis de llamada a la función.

```
void OnStart()
{
   int i = 0;
   FuncByValue(i);          // i cannot change
   Print(i);                // 0
   FuncByReference(i);      // i is changing
   Print(i);                // 1
   FuncByConstReference(i); // i cannot change, 1
   const int j = 1;
   // error
   // 'j' - constant variable cannot be passed as a reference
   // FuncByReference(j);

   FuncByValue(10);         // ok
   // error: '10' - parameter passed as reference, variable expected
   // FuncByReference(10);
}
```

El literal sólo puede pasarse a la función *FuncByValue*, ya que otras funciones requieren una referencia, es decir, una variable, como argumento.

La función *FuncByReference* no puede invocarse con la variable *j*, ya que esta última se declara como una constante, y esta función declara la capacidad (o intención) de cambiar su parámetro ya que no está equipada con el modificador *const*. Esto genera el error «la variable constante no puede pasarse como referencia».

El script también describe la función *Transpose*: transpone una matriz de 2x2 pasada como array bidimensional por referencia.

```
void Transpose(double &m[][2])
{
   double temp = m[1][0];
   m[1][0] = m[0][1];
   m[0][1] = temp;
}
```

Su llamada desde *OnStart* demuestra el cambio esperado en el contenido del array local **a**.

```
double a[2][2] = {{-1, 2}, {3, 0}};
Transpose(a);
ArrayPrint(a);
```

En MQL5, los parámetros del array se pasan siempre como una estructura interna de un array dinámico (véase la sección Características de los arrays). En consecuencia, la descripción de dicho parámetro debe tener necesariamente un tamaño abierto en la primera dimensión, es decir, está vacía dentro del primer par de corchetes.

Esto no impide, si es necesario, pasar a la función el argumento real, que es un array con un tamaño fijo (como en nuestro ejemplo). Sin embargo, funciones como *ArrayResize* no podrán redimensionar o reorganizar de otro modo un array fijo enmascarado de este tipo.

Los tamaños del array en todas las dimensiones excepto la primera deben coincidir tanto para el parámetro como para el argumento. De lo contrario, obtendremos un error de «conversión de parámetros no permitida». En concreto, la función *TransposeVector* se define en el siguiente ejemplo:

```
void TransposeVector(double &v[])
{
}
```

Un intento de invocarla en un array bidimensional ***a*** se comenta en *OnStart* porque genera el error anterior: las dimensiones del array no coinciden.

Además de pasar parámetros por valor o por referencia, existe otra opción: pasar un puntero. A diferencia de C++, MQL5 sólo admite punteros para tipos de objeto (classes). Examinaremos esta característica en la tercera parte.

2.8.5 Parámetros opcionales

MQL5 ofrece la posibilidad de especificar valores por defecto para los parámetros a la hora de describir una función. Para ello se utiliza la sintaxis de inicialización, es decir, un literal del tipo correspondiente a la derecha del parámetro, después del signo '='. Por ejemplo:

```
void function(int value = 0);
```

Al invocar una función pueden omitirse los argumentos de dichos parámetros. A continuación, sus valores se establecerán en sus valores por defecto. Estos parámetros se denominan opcionales (optional).

Los parámetros opcionales deben aparecer al final de la lista de parámetros. En otras palabras: si el *i*-ésimo parámetro se declara con inicialización, entonces todos los parámetros subsiguientes también deben tenerla. De lo contrario, se muestra un error de compilación «falta el valor por defecto para el parámetro». A continuación se describe una función con un problema de este tipo.

```
double Largest(const double v1, const double v2 = -DBL_MAX,
               const double v3);
```

Hay dos soluciones: o bien el parámetro *v3* debe tener también un valor por defecto, o bien el parámetro *v2* debe convertirse en obligatorio.

Sólo se pueden omitir argumentos opcionales cuando se llama a una función de derecha a izquierda. Es decir: si la función tiene dos parámetros y ambos son opcionales, al llamarla no se puede omitir el primero pero especificar el segundo. El único valor pasado se comparará con el primer parámetro, y el segundo se considerará omitido. Si faltan ambos argumentos, los paréntesis vacíos siguen siendo necesarios.

Considere la función de encontrar el número máximo de tres. El primer parámetro es obligatorio, los dos últimos son opcionales e iguales por defecto al número mínimo posible de tipo *double*. Así, cada uno de ellos, en ausencia de un valor explícitamente pasado, será sin duda menor que (o, en casos extremos, igual a) todos los demás parámetros.

```
double Largest(const double v1, const double v2 = -DBL_MAX,
               const double v3 = -DBL_MAX)
{
   return v1 > v2 ? (v1 > v3 ? v1 : v3) : (v2 > v3 ? v2 : v3);
}
```

Así es como puede invocarla:

```
Print(Largest(1));       // ok: 1
Print(Largest(0, -2));   // ok: 0
Print(Largest(1, 2, 3)); // ok: 3
```

Con la ayuda de parámetros opcionales, MQL5 implementa el concepto de funciones con un número variable de parámetros en funciones personalizadas.

MQL5 no admite la sintaxis de elipsis para definir funciones con un número variable de parámetros, como hace C++. Al mismo tiempo, existen funciones integradas en la API de MQL5 que se describen mediante elipsis y aceptan un número variable de parámetros arbitrarios. Por ejemplo, se trata de la función Print. Su prototipo tiene el siguiente aspecto: void Print(argument, ...). Por lo tanto, podemos invocarlo con hasta 64 argumentos separados por comas (excluyendo arrays) y los mostrará en el registro.

2.8.6 Devolver valores

Las funciones pueden devolver los valores de tipos integrados, estructuras con campos de tipos integrados, así como punteros a funciones y punteros a objetos class. El nombre del tipo se escribe en

la definición de la función antes del nombre. Si la función no devuelve nada, se le debe asignar el tipo *void* .

Para devolver una función de array debe utilizar parámetros pasados por referencia (véase Parámetros de valor y parámetros de referencia).

Se devuelve un valor utilizando la sentencia *return* en la que se especifica una expresión después de la palabra clave *return*. Puede utilizarse cualquiera de las dos formas:

```
return expression ;
```

o

```
return ( expression ) ;
```

Si la función es del tipo *void*, la sentencia *return* se simplifica:

```
return ;
```

La sentencia *return* no puede contener ninguna expresión dentro de la función *void*: el compilador generará un error «'return' - función 'void' devuelve un valor».

Para este tipo de funciones, en teoría, no es necesario utilizar *return* al final del bloque con el cuerpo de la función. Lo hemos visto en el ejemplo de la función *OnStart* .

Si la función tiene un tipo distinto de *void*, entonces la sentencia *return* debe ser obligatoria. Si no está presente, se producirá el error de compilación «no todas las rutas de control devuelven un valor».

```
int func(void)
{
   if(IsStopped()) return; // error: function must return a value
                           // error: not all control paths return a value
}
```

Es importante tener en cuenta que el cuerpo de una función puede tener varias sentencias return. En concreto, en caso de salidas anticipadas por condición. Cualquier sentencia *return* interrumpe la ejecución de la función en el lugar donde se encuentra.

Si una función debe devolver un valor (porque no sea del tipo *void*) y no se especifica en el operador *return*, el compilador generará el error «la función debe devolver un valor». A continuación se ofrece la versión correcta para el compilador de la función *func* (*FuncReturn.mq5*).

```
int func(void)
{
   if(IsStopped()) return 0;
   return 1;
}
```

Si el valor devuelto difiere del tipo de función especificado, el compilador intentará una conversión implícita. En caso de que los tipos requieran una conversión explícita se generará un error.

Para devolver un valor se crea de forma implícita una variable temporal y se pone a disposición del código de llamada.

Después de que veamos los tipos de objetos (véase el capítulo sobre Clases) y la capacidad de devolver punteros a objetos desde las funciones, volveremos a analizar cómo pasarlos de forma segura. A diferencia de C ++, las funciones en MQL5 no son capaces de devolver referencias. Si se intenta

declarar una función con un ampersand en el tipo de resultado se produce el error «'&' - no se puede utilizar la referencia».

2.8.7 Declaración de funciones

La declaración de una función describe un prototipo sin especificar el cuerpo de la función. En lugar de un bloque con cuerpo, se pone punto y coma.

La declaración es necesaria para que el compilador pueda comprobar en fragmentos de código posteriores cómo se llama correctamente a la función por su nombre, pasándole argumentos y obteniendo el resultado.

Toda la definición de la función (incluido el cuerpo) es también una declaración, por lo que no es necesario declarar una función además de la definición.

Por ejemplo, la declaración de la función *Fibo* anterior podría tener este aspecto:

```
int Fibo(const int n);
```

Se utilizan declaraciones y definiciones de funciones independientes cuando se construye un programa a partir de varios archivos con texto fuente: entonces, la declaración se realiza en el archivo de encabezado con la extensión *mqh* (véase la sección sobre la directiva #include del preprocesador), que está incluida en los archivos en los que se utiliza la función, y la definición de la función se implementa sólo en uno de los archivos. La coincidencia de la firma de la función en la declaración y la definición proporciona protección contra errores. En otras palabras: una única declaración garantiza la coherencia de los cambios realizados en todo el código fuente

Si declaramos una función y la llamamos en algún lugar del código, pero no proporcionamos una definición totalmente apropiada para la misma, el compilador arrojará el error «La función 'Nombre' debe tener un cuerpo». Esto suele ocurrir cuando hay erratas o imprecisiones en la declaración o en la definición, así como en el proceso de cambio de los códigos fuente, cuando parte de las correcciones ya se han hecho y la otra parte probablemente se ha olvidado.

Si la función se declara y no se utiliza en ninguna parte, el compilador tampoco requiere su definición; dicho elemento simplemente se «recorta» del programa binario.

En la sección Sentencias de declaración y definición vimos un ejemplo de la función *Init* (script *StmtDeclaration.mq5*), que se utilizó para inicializar variables. Allí, en concreto, se demostró el problema de que la variable global *k* no puede definirse antes de la función *Init*, ya que el valor inicial *k* se obtiene llamando a *Init*. El compilador arroja el error «'Init' es un identificador desconocido».

Ahora sabemos que ese problema puede resolverse con una declaración. En el script *FuncDeclaration.mq5* hemos añadido la siguiente declaración de la función *Init* antes de la variable *k*, y dejado la definición *Init* después de *k*.

```
// preliminary declaration
int Init(const int v);
// before adding preliminary declaration above
// here was an error: 'Init' is an unknown identifier
int k = Init(-1);
int Init(const int v)
{
   Print("Init: ", v);
   return v;
}
```

Ahora el script se compila normalmente. Técnicamente, en este caso podríamos simplemente mover la función por encima de la variable sin una declaración preliminar. Lo hicimos para explicar el concepto. Sin embargo, hay casos de dependencia mutua de los elementos del lenguaje entre sí (por ejemplo, las clases), cuando es imposible prescindir de una declaración previa dentro del mismo archivo.

2.8.8 Recursión

Está permitido llamar a la misma función desde sentencias dentro de una función. Este tipo de llamadas se denominan recursivas.

Volvamos al ejemplo del cálculo de los números de Fibonacci. Siguiendo la fórmula para calcular cada número como la suma de los dos anteriores (excepto los dos primeros, que son iguales a 1), es fácil escribir una función recursiva para calcular los números de Fibonacci.

```
int Fibo(const int n)
{
   if(n <= 1) return 1;

   return Fibo(n - 1) + Fibo(n - 2);
}
```

Una función recursiva debe ser capaz de devolver el control sin recursión, como en nuestro caso dentro de la sentencia condicional *if* para los índices 0 y 1. De lo contrario, la secuencia de llamadas a funciones podría continuar indefinidamente. En la práctica, dado que las llamadas a funciones no finalizadas se acumulan en una zona limitada de la memoria denominada pila (véase la sección Sentencias de declaración y definición y la barra lateral «Montón» y «Pila» en la sección Descripción de arrays), tarde o temprano la función terminará con el error en tiempo de ejecución «Stack overflow» (desbordamiento de pila). Este problema se muestra en la función *FiboEndless*.

```
int FiboEndless(const int n)
{
   return FiboEndless(n - 1) + FiboEndless(n - 2);
}
```

Tenga en cuenta que este no es un error de compilación. En tal caso, el compilador ni siquiera generará un aviso (aunque, técnicamente, podría hacerlo). El error se produce durante la ejecución del script. Se imprimirá en el diario *Experts* del terminal.

La recursión puede producirse no sólo cuando se llama a una función desde la propia función. Por ejemplo, si la función *F* llama a la función *G* que, a su vez, llama a la función *F*, este caso es una recursión indirecta. Así, la recursión puede producirse como resultado de llamadas cíclicas de cualquier profundidad.

2.8.9 Sobrecarga de funciones

MQL5 permite definir funciones con el mismo nombre pero con distinto número o tipo de parámetros en el mismo código fuente. Este enfoque se denomina sobrecarga de funciones y suele aplicarse cuando una misma acción puede ser desencadenada por diferentes entradas. Las diferencias en las firmas permiten al compilador determinar automáticamente a qué función llamar en función de los argumentos pasados. Pero hay algunas particularidades.

Las funciones no pueden diferir únicamente en cuanto a tipo de devolución. En este caso, el mecanismo de sobrecarga no se activa y se devuelve el error «la función ya se ha definido y tiene un tipo diferente».

Si funciones del mismo nombre tienen distinto número de parámetros y los parámetros «extra» se declaran opcionales, el compilador no podrá determinar a cuál de ellas llamar. Esto generará el error «llamada ambigua a función sobrecargada con los mismos parámetros».

Cuando se llama a una función sobrecargada, el compilador hace coincidir los argumentos y parámetros en las sobrecargas disponibles. Si no se encuentra una coincidencia exacta, el compilador intenta añadir o eliminar el modificador *const* y realizar la expansión de tipos numéricos y la conversión aritmética. En el caso de los punteros a objetos se utilizan reglas de herencia de clases.

Con un número diferente de parámetros o tipos de parámetros no relacionados en la misma posición (como un número y una cadena), la elección suele estar clara. Sin embargo, si los tipos de parámetros deben convertirse implícitamente de uno a otro, puede surgir la ambigüedad.

Por ejemplo, tenemos dos funciones de suma:

```
double sum(double v1, double v2)
{
   return v1 + v2;
}

int sum(int v1, int v2)
{
   return v1 + v2;
}
```

Entonces, la siguiente llamada dará lugar a un error:

```
sum(1, 3.14); // overloaded function call is ambiguous
```

Aquí, el compilador se siente igual de incómodo con cada una de las sobrecargas: para la función *double sum(double v1, double v2)* es necesario convertir implícitamente el primer argumento en *double*, y para *int sum(int v1, int v2)* es necesario convertir el segundo argumento en *int*.

El término 'sobrecarga' debe interpretarse en el sentido de que un nombre reutilizado se «carga» con «deberes» varias veces más pesados que un nombre normal utilizado sólo para una función.

Intentemos sobrecargar la función para la transposición de matrices. Ya teníamos un ejemplo para un array de 2x2 (véase Parámetros de valor y parámetros de referencia). Vamos a realizar la misma operación para un array de 3x3. El tamaño de un parámetro de array multidimensional en dimensiones superiores (distintas de cero) cambia el tipo, es decir, doble [][2] es diferente de doble [][3]. Por lo tanto, sobrecargaremos la versión antigua de la función:

```
void Transpose(double &m[][2]);
```

Añadiendo uno nuevo (*FuncOverload.mq5*):

```
void Transpose(double &m[][3]);
```

En la implementación de la nueva versión es conveniente utilizar la función de ayuda *Swap* para intercambiar dos elementos de matriz en los índices dados.

```
void Transpose(double &m[][3])
{
   Swap(m, 0, 1);
   Swap(m, 0, 2);
   Swap(m, 1, 2);
}

void Swap(double &m[][3], const int i, const int j)
{
   static double temp;

   temp = m[i][j];
   m[i][j] = m[j][i];
   m[j][i] = temp;
}
```

Ahora podemos llamar a ambas funciones desde *OnStart* utilizando la misma notación para arrays de diferentes tamaños. El propio compilador generará una llamada a las versiones correctas.

```
double a[2][2] = {{1, 2}, {3, 4}};
Transpose(a);
...
double b[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
Transpose(b);
```

Es importante señalar que el modificador *const* del parámetro, aunque cambia el prototipo de la función, no siempre es una diferencia suficiente para la sobrecarga. Dos funciones del mismo nombre, que sólo difieren en la presencia y ausencia de *const* para algún parámetro, pueden considerarse iguales. Esto provocará un error del tipo «la función ya está definida y tiene cuerpo». Este comportamiento se produce porque, para los parámetros de valor, el modificador *const* se descarta cuando se asigna el argumento (porque un parámetro de valor, por definición, no puede cambiar el argumento en el código de llamada), y esto no permite seleccionar una de las varias funciones solapadas basándose en él.

Para demostrarlo, basta con añadir una función en el script:

```
void Swap(double &m[][3], int i, int j);
```

Se trata de una sobrecarga infructuosa de la ya existente:

```
void Swap(double &m[][3], const int i, const int j);
```

La única diferencia entre las dos funciones son los modificadores *const* para los parámetros *i* y *j*. Por lo tanto, ambos son adecuados para llamar con argumentos del tipo *int* y pasar por valor.

Cuando los parámetros se pasan por referencia, la sobrecarga con una diferencia de sólo atributos *const*/non-*const* tiene éxito porque, para las referencias, el modificador *const* es importante (cambia el

tipo y elimina la posibilidad de conversión implícita). Esto se demuestra en el script con un par de funciones:

```
void SwapByReference(double &m[][3], int &i, int &j)
{
   Print(__FUNCSIG__);
}

void SwapByReference(double &m[][3], const int &i, const int &j)
{
   Print(__FUNCSIG__);
}

void OnStart()
{
   // ...
   {
      int i = 0, j = 1;
      SwapByReference(b, i, j);
   }
   {
      const int i = 0, j = 1;
      SwapByReference(b, i, j);
   }
}
```

Se dejan como stubs casi vacíos, en los que se imprime la firma de cada función mediante la llamada *Print(__FUNCSI__)*. Esto permite garantizar que se llame a la versión adecuada de la función en función del atributo *const* de los argumentos.

2.8.10 Punteros de función (typedef)

MQL5 tiene la palabra clave *typedef* , que le permite describir un tipo especial de puntero de función.

> A diferencia de C ++, donde *typedef* tiene una aplicación mucho más amplia, en MQL5 *typedef* se utiliza sólo para los punteros de función.

La sintaxis para una nueva declaración de tipo es:

```
typedef function_result_type ( *function_type )( [list_of_input_parameters] ) ;
```

El identificador *function_type* define un nombre de tipo que se convierte en sinónimo (alias) de un puntero a cualquier función que devuelva un valor del tipo dado *function_result_type* y acepte una lista de parámetros de entrada (*list_of_input_parameters*).

Por ejemplo, podemos tener 2 funciones con los mismos prototipos (dos parámetros de entrada de tipo *double* y el tipo de resultado también es *double*) que realizan operaciones aritméticas diferentes: suma y resta (*FuncTypedef.mq5*).

```
double plus(double v1, double v2)
{
   return v1 + v2;
}

double minus(double v1, double v2)
{
   return v1 - v2;
}
```

Su prototipo común es fácil de describir para utilizarlo como puntero:

```
typedef double (*Calc)(double, double);
```

Esta entrada introduce en el programa el tipo *Calc*, con el que se puede definir una variable o parámetro para guardar o pasar una referencia a cualquier función con dicho prototipo, incluidas las funciones *plus* y *minus*. Este tipo es un puntero porque en la descripción se utiliza el carácter '*' (**Calc*). Descubriremos más sobre las características del asterisco aplicadas a los punteros cuando estudiemos la programación orientada a objetos.

Es conveniente utilizar esta clase de punteros para crear algoritmos personalizados que puedan llamar «sobre la marcha» a diferentes funciones correspondientes al alias, dependiendo de los datos de entrada.

En concreto, podemos introducir una función calculadora generalizada:

```
double calculator(Calc ptr, double v1, double v2)
{
   if(ptr == NULL) return 0;
   return ptr(v1, v2);
}
```

Su primer parámetro se declara con el tipo *Calc* . Gracias a ello, podemos pasarle una función arbitraria con un prototipo adecuado y, como resultado, realizar alguna operación, cuya esencia desconoce la propia función *calculator*. Para ello, delega la llamada en un puntero: *ptr(v1, v2)*. Dado que *ptr* es un puntero a una función, esta sintaxis no sólo se asemeja a una llamada a una función, sino que en realidad llama a la función que contiene el puntero.

Tenga en cuenta que comprobamos previamente el parámetro *ptr* con el valor especial NULL (NULL es el equivalente a cero para los punteros). El hecho es que el puntero puede no apuntar a ninguna parte, es decir, puede no estar inicializado. Así, en el script, tenemos una variable global descrita:

```
Calc calc;
```

No tiene punteros. Si no fuera por la «protección» contra NULL, llamar a *calculator* con un puntero «vacío» *calc* provocaría un error en tiempo de ejecución «Llamada a puntero de función no válida».

Las llamadas a la función *calculator* con diferentes punteros en el primer parámetro darán los siguientes resultados (mostrados en los comentarios):

```
void OnStart()
{
   Print(calculator(plus, 1, 2));   //  3
   Print(calculator(minus, 1, 2));  // -1
   Print(calculator(calc, 1, 2));   //  0
}
```

Tenga en cuenta que si no hay inicialización explícita, todos los punteros de función se rellenan con valores cero. Esto se aplica tanto a las variables globales como a las locales de un tipo determinado.

Un tipo de puntero definido con *typedef* puede devolverse desde funciones, por ejemplo:

```
Calc generator(ushort type)
{
   switch(type)
   {
      case '+': return plus;
      case '-': return minus;
   }
   return NULL;
}
```

Además, el tipo de puntero de función se utiliza a menudo para funciones callback (*callback*, véase *FuncCallback.mq5*). Supongamos que tenemos una función *DoMath* que realiza cálculos largos (probablemente, está implementada en una biblioteca). En cuanto a comodidad y facilidad de uso de la interfaz, sería estupendo mostrar al usuario una indicación de progreso. Para ello, puede definir un tipo especial de puntero de función para notificaciones sobre el porcentaje de trabajo completado (*ProgressCallback*), y añadir un parámetro de este tipo a la función *DoMath*. En el código *DoMath*, debería llamar periódicamente a la función pasada:

```
typedef void (*ProgressCallback)(const float percent);

void DoMath(double &bigdata[], ProgressCallback callback)
{
   const int N = 1000000;
   for(int i = 0; i < N; ++i)
   {
      if(i % 10000 == 0 && callback != NULL)
      {
         callback(i * 100.0f / N);
      }

      // long calculations
   }
}
```

A continuación, el código de llamada puede definir la función callback necesaria, pasarle un puntero a *DoMath* y recibir actualizaciones a medida que avanza el cálculo.

```
void MyCallback(const float percent)
{
   Print(percent);
}

void OnStart()
{
   double data[] = {0};
   DoMath(data, MyCallback);
}
```

Los punteros de función sólo funcionan con funciones personalizadas definidas en MQL5. No pueden señalar a funciones integradas de la API de MQL5.

2.8.11 Inlining

Para mejorar la eficiencia del código, los compiladores modernos suelen utilizar el siguiente truco: al generar código ejecutable, algunas llamadas a funciones se sustituyen directamente por el cuerpo de la función (sus sentencias). Esta técnica se denomina inlining. Esto acelera la operación al evitar la sobrecarga asociada a la organización de la llamada y el retorno de la función. Desde el punto de vista del programador, inlining no cambia nada.

MQL5 admite inlining por defecto. En caso necesario, puede desactivarse, pero sólo en el modo de perfilado de código. La palabra clave *inline* está reservada en MQL5 por compatibilidad con los códigos fuente C++. Su presencia o ausencia antes de la definición de la función no afecta al programa generado.

2.9 Preprocesador

Hasta este momento hemos estado estudiando la programación MQL5, asumiendo que los códigos fuente son procesados por el compilador, que convierte su representación textual en binaria (ejecutable por el terminal). Sin embargo, la primera herramienta que lee y, si es necesario, convierte los códigos fuente es el preprocesador. Esta utilidad integrada en MetaEditor se controla mediante directivas especiales insertadas directamente en el código fuente y puede resolver una serie de problemas a los que se enfrentan los programadores a la hora de preparar códigos fuente.

Al igual que el preprocesador C++, MQL5 admite la definición de sustituciones de macros (*#define*), la compilación condicional (*#ifdef*) y la inclusión de otros archivos fuente (*#include*). En este capítulo exploraremos estas posibilidades. Algunas de ellas tienen limitaciones en comparación con C++.

Además de las directivas estándar, el preprocesador de MQL5 tiene sus propias directivas específicas; en concreto, un conjunto de propiedades de programas MQL (#property) e importación de funciones desde EX5 y DLL independientes (*#import*). Las abordaremos en las partes quinta, sexta y séptima cuando estudiemos diversos tipos de programas MQL.

Todas las directivas de preprocesador comienzan con el signo de almohadilla '#' seguido de una palabra clave y parámetros adicionales, cuya sintaxis depende del tipo de directiva.

Se recomienda iniciar una directiva de preprocesador desde el principio mismo de la línea, o al menos tras una sangría con espacio en blanco (si las directivas están anidadas). Insertar una directiva dentro de las sentencias del código fuente se considera un mal estilo de programación (a diferencia de MQL5, el preprocesador de C++ no lo permite en absoluto).

Las directivas de preprocesador no son sentencias del lenguaje y no deben terminar con un ';'. Las directivas suelen continuar hasta el final de la línea actual. En algunos casos, pueden ampliarse de forma especial para las líneas siguientes, que se abordarán por separado.

Las directivas se ejecutan secuencialmente, en el mismo orden en que aparecen en el texto y teniendo en cuenta el procesamiento de las directivas anteriores. Por ejemplo, si otro archivo está conectado a un archivo mediante la directiva *#include* y se define una regla de sustitución en el archivo incluido mediante *#define*, entonces esta regla empieza a funcionar para todas las líneas de código posteriores, incluidos los archivos de encabezado que se incluyen más tarde.

El preprocesador no procesa los comentarios.

2.9.1 Inclusión de archivos fuente (#include)

La directiva *#include* se utiliza para incluir el contenido de otro archivo en el código fuente. Esta directiva produce la misma acción que si el programador copiara el texto del archivo de inclusión en el portapapeles y lo pegara en el archivo actual en el lugar donde se utiliza la directiva.

Dividir el código fuente en varios archivos es una práctica habitual cuando se escriben programas complejos. Estos programas se construyen de forma modular, de modo que cada módulo o archivo contiene código relacionado lógicamente que resuelve una o varias tareas relacionadas.

Los archivos de inclusión también se utilizan para distribuir bibliotecas (conjuntos de algoritmos ya creados). La misma biblioteca puede incluirse en distintos programas. En este caso, la actualización de la biblioteca (la actualización de su archivo de encabezado) se aplicará automáticamente en todos los programas durante su próxima compilación.

Si los archivos principales de los programas MQL deben tener la extensión mq5, los archivos de inclusión suelen tener la extensión *mqh* (la 'h' al final de la palabra significa «header», es decir, «encabezado» en inglés). Al mismo tiempo, está permitido utilizar la directiva *#include* para otros tipos de archivos de texto, como por ejemplo, **.txt* (véase más adelante). En cualquier caso, cuando se incluye un archivo, el programa final combinado del archivo principal mq5 y todos los encabezados deben seguir siendo sintácticamente correctos. Por ejemplo, incluir un archivo con información binaria (como una imagen png) romperá la compilación.

Existen dos tipos de declaraciones *#include* :

```
#include <file_name>
#include "file_name"
```

En la primera, el nombre del archivo va entre corchetes angulares. El compilador busca estos archivos en el directorio de datos del terminal en la subcarpeta MQL5/Include/.

Para la segunda, con el nombre entre comillas, la búsqueda se realiza en el mismo directorio que contiene el archivo actual que utiliza la sentencia #include.

En ambos casos, el archivo puede ubicarse en subcarpetas dentro del directorio de búsqueda. En este caso, debe especificar toda la jerarquía relativa de carpetas antes del nombre de archivo en la directiva. Por ejemplo, además de MetaTrader 5, hay muchos archivos de arranque de uso común, entre los que se encuentra *DateTime.mqh* con un conjunto de métodos para trabajar con la fecha y la hora (están diseñados como estructuras, que son las construcciones del lenguaje que abordaremos en la Parte 3 dedicada a la POO). El archivo *DateTime.mqh* se encuentra en la carpeta *Tools* . Para incluirlo en su código fuente, debe utilizar la siguiente directiva:

```
#include <Tools/DateTime.mqh>
```

Para demostrar cómo incluir un archivo de encabezado de la misma carpeta que el archivo fuente con la directiva vamos a analizar el archivo *Preprocessor.mq5*. Contiene la siguiente directiva:

```
#include "Preprocessor.mqh"
```

Hace referencia al archivo *Preprocessor.mqh*, que en realidad se encuentra junto a *Preprocessor.mq5*.

Un archivo de inclusión puede, a su vez, incluir otros archivos. En concreto, dentro de *Preprocessor.mqh* está el siguiente código:

```
double array[] =
{
   #include "Preprocessor.txt"
};
```

Significa que el contenido del array se inicializa a partir del archivo de texto dado. Si miramos dentro de *Preprocessor.txt* veremos el texto que cumple con las reglas de sintaxis de inicialización de arrays:

```
1, 2, 3, 4, 5
```

Así, es posible recopilar el código fuente de los componentes personalizados, incluso generarlo utilizando otros programas.

Tenga en cuenta que si el archivo especificado en la directiva no se encuentra, la compilación fallará.

El orden en el que se incluyen varios archivos determina el orden en el que se procesan las directivas del preprocesador que contienen.

2.9.2 Visión general de las directivas de sustitución de macros

Las directivas de sustitución de macros incluyen dos formas de la directiva #define:

- simple, normalmente para definir una constante;
- definición de una macro como una pseudofunción con parámetros.

Además, existe una directiva *#undef* para deshacer cualquiera de las definiciones anteriores de *#define*. Si no se utiliza *#undef*, cada macro definida será válida hasta el final de la compilación del código fuente.

Las macros se registran y luego se utilizan en el código por su nombre, siguiendo las reglas de los identificadores. Por convención, los nombres de las macros se escriben en mayúsculas. Los nombres de las macros pueden superponerse a los nombres de variables, funciones y otros elementos del código fuente. El uso intencionado de este hecho otorga la flexibilidad de cambiar y generar código fuente sobre la marcha. Sin embargo, la coincidencia involuntaria de un nombre de macro con un elemento del programa dará lugar a errores.

El principio de funcionamiento de ambas formas de sustitución de macros es el mismo. Mediante la directiva *#define* se introduce un identificador que se asocia a un determinado fragmento de texto, o definición. Si el preprocesador encuentra un identificador dado más adelante en el código fuente, lo sustituye por el texto asociado al mismo. Hacemos hincapié en que el nombre de la macro sólo se puede utilizar en el código compilado después del registro (esto es similar a los principios de declaración de variables, pero sólo en la etapa de compilación).

La sustitución del nombre de una macro por su definición se denomina expansión. El análisis del código fuente se produce progresivamente y por una línea en una pasada, pero la expansión en cada línea puede realizarse un número arbitrario de veces, como en un bucle, siempre que se encuentren nombres de macros en el resultado. No puede incluir el mismo nombre en una definición de macro: al sustituirla, una macro de este tipo dará lugar a un error de «identificador desconocido».

En la Parte 3 del libro hablaremos de las plantillas, que también le permiten generar (o, de hecho, replicar) código fuente, pero con reglas diferentes. Si hay tanto directivas de sustitución de macros como plantillas en el código fuente, primero se expanden las macros y luego se genera el código a partir de las plantillas.

Los nombres de las macros aparecen resaltados en rojo en el MetaEditor.

2.9.3 Forma simple de #define

La forma simple de la directiva #define registra un identificador y la secuencia de caracteres por la que el identificador debe ser reemplazado en todas partes en los códigos fuente después de la directiva, hasta el final del programa, o antes de la directiva #undef con el mismo identificador.

Su sintaxis es:

```
#define macro_identifier [text]
```

El texto comienza después del identificador y continúa hasta el final de la línea actual. El identificador y el texto deben ir separados por un número arbitrario de espacios o tabuladores. Si la secuencia de caracteres requerida es demasiado larga, para facilitar la lectura puede dividirla en varias líneas colocando un carácter de barra invertida '\' al final de la línea.

```
#define macro_identifier text_beginning \
                         text_continued \
                         text_ending
```

El texto puede consistir en cualquier construcción de lenguaje: constantes, operadores, identificadores y signos de puntuación. Si sustituye *macro_identifier* en lugar de las construcciones encontradas en el código fuente, todas ellas se incluirán en la compilación.

La forma simple se utiliza tradicionalmente para varios fines:

1. Declaraciones de banderas, que luego se utilizan para comprobaciones de compilación condicional ;
2. Declaraciones de constantes con nombre;
3. Notación abreviada de sentencias comunes.

El primer punto se caracteriza por el hecho de que no es necesario especificar nada después del identificador: la presencia de una directiva con un nombre ya es suficiente para que se registre el identificador correspondiente y pueda utilizarse en directivas condicionales *#ifdef/ifndef*. Para ellas, sólo es importante si el identificador existe o no, es decir, funciona en el modo bandera: declarado / no declarado. Por ejemplo, la siguiente directiva define la bandera DEMO:

```
#define DEMO
```

Puede utilizarse, por ejemplo, para crear una versión de demostración del programa de la que se excluyan determinadas funciones (véase el ejemplo de la sección de compilación condicional).

La segunda forma de utilizar una directiva simple permite sustituir los «números mágicos» del código fuente por nombres fáciles de usar. Los «números mágicos» son constantes insertadas en el texto fuente cuyo significado no siempre está claro (porque un número es sólo un número: conviene al menos explicarlo en un comentario). Además, el mismo valor puede estar disperso por distintas partes del código, y si el programador decide cambiarlo por otro, tendrá que hacerlo en todos los sitios (y esperar que no se le haya escapado nada).

Con una macro con nombre, estos dos problemas se resuelven fácilmente. Por ejemplo, un script puede preparar un array con números de Fibonacci hasta una determinada profundidad máxima. Tiene sentido definir una macro con un tamaño de array predefinido y utilizarla en la descripción del propio array (*Preprocessor.mq5*).

```
#define MAX_FIBO 10

int fibo[MAX_FIBO]; // 10

void FillFibo()
{
   int prev = 0;
   int result = 1;

   for(int i = 0; i < MAX_FIBO; ++i) // i < 10
   {
      int temp = result;
      result = result + prev;
      fibo[i] = result;
      prev = temp;
   }
}
```

Si posteriormente el programador decide que es necesario aumentar el tamaño del array, basta con que lo haga en un lugar: en la directiva *#define*. Así, la directiva define de hecho un determinado parámetro del algoritmo que está «programado» en el código fuente y no está disponible para la configuración del usuario. Esta necesidad se plantea con bastante frecuencia.

Cabe preguntarse en qué se diferencia la definición a través de *#define* de una variable constante en el contexto global. De hecho, podríamos declarar una variable con el mismo nombre y finalidad, e incluso conservar las mayúsculas:

```
const int MAX_FIBO = 10;
```

Sin embargo, en este caso, MQL5 no permitirá definir un array con el tamaño especificado, ya que sólo se permiten constantes entre corchetes, es decir, literales (y una variable constante, a pesar de su nombre similar, no es una constante). Para resolver este problema podríamos definir un array como dinámico (sin especificar antes un tamaño) y luego asignarle memoria mediante la función *ArrayResize*: pasar una variable como tamaño no es difícil en este caso.

Los enums ofrecen una forma alternativa de definir una constante con nombre, pero se limita únicamente a valores enteros. Por ejemplo:

```
enum
{
   MAX_FIBO = 10
};
```

Pero la macro puede contener un valor de cualquier tipo.

```
#define TIME_LIMIT     D'2023.01.01'
#define MIN_GRID_STEP  0.005
```

La búsqueda de nombres de macros en los textos fuente para su sustitución se realiza teniendo en cuenta la sintaxis del lenguaje, es decir, los elementos indivisibles, como identificadores de variables o literales de cadena, permanecerán inalterados, aunque incluyan una subcadena que coincida con una de las macros. Por ejemplo, dada la macro XYZ siguiente, la variable XYZAXES se mantendrá tal cual, y el nombre XYZ (dado que es exactamente el mismo que la macro) se cambiará a ABC.

```
#define XYZ ABC
int XYZAXES = 3; // int XYZAXES = 3
int XYZ = 0;     // int ABC = 0
```

Las sustituciones de macros le permiten incrustar su código en el código fuente de otros programas. Esta técnica la utilizan normalmente las bibliotecas que se distribuyen como archivos de encabezado mqh y se conectan a programas que utilizan las directivas *#include* .

En concreto, para los scripts podemos definir nuestra propia implementación de biblioteca de la función *OnStart*, que debe realizar algunas acciones adicionales sin afectar a la funcionalidad original del programa.

```
void OnStart()
{
   Print("OnStart wrapper started");
   // ... additional actions
   _OnStart();
   // ... additional actions
   Print("OnStart wrapper stopped");
}

#define OnStart _OnStart
```

Supongamos que esta parte se encuentra en el archivo de encabezado incluido (*Preprocessor.mqh*).

El preprocesador renombrará en el código fuente la función original *OnStart* (en *Preprocessor.mq5*) por *_OnStart* (se entiende que este identificador no se utiliza en ningún otro lugar para algún otro propósito). Y la nueva versión de *OnStart* del encabezado llama a *_OnStart*, «envolviéndola» en sentencias adicionales.

La tercera forma habitual de utilizar la sencilla *#define* es acortar la notación de las construcciones del lenguaje. Por ejemplo, el título de un bucle infinito puede denotarse con una palabra LOOP:

```
#define LOOP for( ; !IsStopped() ; )
```

Y aplicado en código:

```
LOOP
{
   // ...
   Sleep(1000);
}
```

Este método es también la técnica principal para utilizar la directiva *#define* con parámetros (véase más adelante).

2.9.4 Forma #define como pseudofunción

La sintaxis de la forma paramétrica #define es similar a la de una función.

```
#define macro_identifier(parameter,...) text_with_parameters
```

Una macro de este tipo tiene uno o varios parámetros entre paréntesis. Los parámetros se separan con comas. Cada parámetro es un identificador simple (a menudo, una sola letra). Además, todos los parámetros de una macro deben tener identificadores diferentes.

Es importante que no haya ningún espacio entre el identificador y el paréntesis de apertura; de lo contrario, la macro se tratará como una forma simple en la que el texto de sustitución comienza con un paréntesis de apertura.

Una vez registrada esta directiva, el preprocesador buscará en los códigos fuente las líneas de forma:

```
macro_identifier(expression,...)
```

Se pueden especificar expresiones arbitrarias en lugar de parámetros. El número de argumentos debe coincidir con el número de parámetros de la macro. Todos los casos que se encuentren se sustituirán por *text_with_parameters*, en los que, a su vez, los parámetros se sustituirán por las expresiones pasadas. Cada parámetro puede aparecer varias veces, en cualquier orden.

Por ejemplo, la siguiente macro encuentra el máximo de dos valores:

```
#define MAX(A,B) ((A) > (B) ? (A) : (B))
```

Si el código contiene la sentencia:

```
int z = MAX(x, y);
```

el preprocesador la «expandirá» a:

```
int z = ((x) > (y) ? (x) : (y));
```

La sustitución de macros funcionará para cualquier tipo de datos (para los que sean válidas las operaciones aplicadas dentro de la macro).

Sin embargo, la sustitución también puede tener efectos secundarios. Por ejemplo, si el parámetro real es una llamada a una función o una sentencia que modifica la variable (digamos, *++x*), entonces la acción correspondiente puede realizarse varias veces (en lugar de la única vez prevista). En el caso de MAX, esto ocurrirá dos veces: durante la comparación y al obtener valores en una de las ramas del operador '?:'. A este respecto, tiene sentido convertir estas macros en funciones siempre que sea posible (sobre todo teniendo en cuenta que en MQL5 las funciones se incorporan automáticamente).

Hay paréntesis alrededor de los parámetros y alrededor de toda la definición de la macro. Se utilizan para garantizar que la sustitución de expresiones como parámetros o de la propia macro dentro de

otras expresiones no distorsione el orden de cálculo debido a prioridades diferentes. Supongamos que la macro define la multiplicación de dos parámetros (aún no encerrados entre paréntesis):

```
#define MUL(A,B) A * B
```

Entonces, el uso de la macro con las siguientes expresiones producirá resultados inesperados:

```
int x = MUL(1 + 2, 3 + 4); // 1 + 2 * 3 + 4
```

En lugar de la multiplicación *(1 + 2) * (3 + 4)*, que da 21, tenemos *1 + 2 * 3 + 4*, es decir, 11. La definición de macro apropiada debería ser así:

```
#define MUL(A,B) ((A) * (B))
```

Puede especificar otra macro como parámetro de macro. Además, también puede insertar otras macros en una definición de macro. Todas estas macros se sustituirán secuencialmente. Por ejemplo:

```
#define SQ3(X) (X * X * X)
#define ABS(X) MathAbs(SQ3(X))
#define INC(Y) (++(Y))
```

El siguiente código imprimirá 504 (*MathAbs* es una función integrada que devuelve el módulo de un número, es decir, sin signo):

```
int x = -10;
Print(ABS(INC(x)));
// -> ABS(++(Y))
// -> MathAbs(SQ3(++(Y)))
// -> MathAbs((++(Y))*(++(Y))*(++(Y)))
// -> MathAbs(-9*-8*-7)
// -> 504
```

En la variable *x*, el valor -7 permanecerá (debido al triple incremento).

Una definición de macro puede contener paréntesis no coincidentes. Esta técnica se utiliza, por regla general, en un par de macros, uno de los cuales debe abrir un fragmento de código determinado y la otra debe cerrarlo. En este caso, los paréntesis no emparejados de cada uno de ellos se convertirán en emparejados. En concreto, en los archivos de la biblioteca estándar disponibles en el paquete de distribución de MetaTrader 5, en *Controls/Defines.mqh*, se definen las macros EVENT_MAP_BEGIN y EVENT_MAP_END, que se utilizan para formar la función de procesamiento de eventos en objetos gráficos.

El preprocesador lee línea por línea todo el texto fuente del programa, empezando por el archivo principal mq5 e insertando en su lugar los textos de los archivos de encabezado encontrados. En el momento en que se lee cualquier línea de código se forma un cierto conjunto de macros que ya están definidas. No importa en qué orden se definieron las macros: es muy posible que una macro haga referencia en su definición a otra, que se describió tanto arriba como abajo en el texto. Sólo es importante que en la línea de código fuente donde se utiliza el nombre de la macro se conozcan las definiciones de todas las macros referenciadas.

Veamos un ejemplo:

```
#define NEG(x) (-SQN(x))*TEN
#define SQN(x) ((x)*(x))
#define TEN 10
...
Print(NEG(2)); // -40
```

En este caso, la macro NEG utiliza las macros SQN y TEN, que se describen a continuación, y esto no nos impide utilizarlo con éxito en el código después de los tres *#define*-s.

No obstante, si cambiamos la posición relativa de las filas por la siguiente:

```
#define NEG(x) (-SQN(x))*TEN
#define SQN(x) ((x)*(x))
...
Print(NEG(2)); // error: 'TEN' - undeclared identifier
...
#define TEN 10
```

obtenemos un error de compilación de «identificador no declarado».

2.9.5 Operadores especiales '#' y '##' dentro de definiciones #define

Dentro de las definiciones de macros se pueden utilizar dos operadores especiales:

- Un único símbolo de almohadilla '#' delante del nombre de un parámetro de macro convierte el contenido de ese parámetro en una cadena; sólo se permite en macros de función;
- Un doble símbolo hash '##' entre dos palabras (tokens) las combina, y si el token es un parámetro de macro, se sustituye su valor, pero si el token es un nombre de macro, se sustituye tal cual, sin expandir la macro; si como resultado del «pegado» se obtiene otro nombre de macro, se expande.

En los ejemplos de este libro, a menudo utilizamos la siguiente macro:

```
#define PRT(A) Print(#A, "=", (A))
```

Llama a la función *Print* , en la que la expresión pasada se muestra como una cadena gracias a #A, y tras el signo «igual» se imprime el valor real de A.

Para demostrar '##', analicemos otra macro:

```
#define COMBINE(A,B,X) A##B(X)
```

Con ella podemos generar de hecho una llamada a la macro SQN definida arriba:

```
Print(COMBINE(SQ,N,2)); // 4
```

Los literales SQ y N se concatenan, tras lo cual la macro SQN se expande a ((2)*(2)) y produce el resultado 4.

La siguiente macro le permite crear una definición de variable en el código generando su nombre dados los parámetros de la macro:

```
#define VAR(TYPE,N) TYPE var##N = N
```

A continuación, la línea de código

```
VAR(int, 3);
```

equivale a lo siguiente:

```
int var3 = 3;
```

La concatenación de tokens permite la implementación de un bucle abreviado sobre los elementos del array utilizando una macro.

```
#define for_each(I, A) for(int I = 0, max_##I = ArraySize(A); I < max_##I; ++I)

// describe and somehow fill in the array x
double x[];
// ...
// implement loop through the array
for_each(i, x)
{
   x[i] = i * i;
}
```

2.9.6 Anulación de la sustitución de macros (#undef)

Las sustituciones registradas con *#define* pueden deshacerse si ya no son necesarias tras un determinado fragmento de código. Para ello se utiliza la directiva *#undef*.

```
#undef macro_identifier
```

Ello es útil en especial si necesita definir la misma macro de diferentes maneras en distintas partes del código. Si el identificador especificado en *#define* ya ha sido registrado en alguna línea de código anterior (por otra directiva *#define*), entonces la antigua definición se sustituye por la nueva, y el preprocesador genera la advertencia «redefinición de macro». El uso de *#undef* evita el aviso al tiempo que indica explícitamente la intención del programador de no utilizar una determinada macro más adelante en el código.

#undef no puede anular la definición de macros predefinidas.

2.9.7 Constantes predefinidas del preprocesador

MQL5 tiene varias constantes predefinidas que equivalen a macros simples, pero son definidas por el propio compilador. En la siguiente tabla se enumeran algunos de sus nombres y significados.

Nombre	Descripción
__COUNTER__	Contador (cada mención en el texto durante la ampliación de la macro da lugar a un aumento de 1)
__DATE__	Fecha de compilación (día)
__DATETIME__	Fecha y hora de compilación
__FILE__	Nombre del archivo compilado
__FUNCSIG__	Firma de la función actual

Nombre	Descripción
__FUNCTION__	Nombre de la función actual
__LINE__	Número de línea en el archivo compilado
__MQLBUILD__, __MQL5BUILD__	Versión del compilador
__RANDOM__	Número aleatorio de tipo ulong
__PATH__	Ruta al archivo compilado
_DEBUG	Definido al compilar en modo depuración
_RELEASE	Definido al compilar en modo normal

2.9.8 Compilación condicional (#ifdef/#ifndef/#else/#endif)

Las directivas de compilación condicional permiten incluir y excluir fragmentos de código del proceso de compilación. Las directivas *#ifdef* y *#ifndef* marcan el inicio del fragmento de código que controlan. El fragmento termina con la directiva *#endif* . En el caso más sencillo, la sintaxis de *#ifdef* es la siguiente:

```
#ifdef macro_identifier
  statements
#endif
```

Si una macro con el identificador especificado se define arriba en el código utilizando *#define*, entonces este fragmento de código participará en la compilación. En caso contrario, se excluye. Además de las macros definidas en el código de la aplicación, el entorno proporciona un conjunto de constantes predefinidas; en concreto, los indicadores _RELEASE y _DEBUG (véase la sección Constantes predefinidas), cuyos nombres también pueden comprobarse en directivas de compilación condicional.

La forma extendida *#ifdef* permite especificar dos fragmentos de código: el primero se incluirá si el identificador de macro está definido, y el segundo si no lo está. Para ello, se inserta un separador de fragmentos *#else* entre *#ifdef* y *#endif*.

```
#ifdef macro_identifier
  statesments_true
#else
  statements_false
#endif
```

La directiva *#ifndef* funciona de forma similar, pero los fragmentos se incluyen y excluyen según la lógica inversa: si la macro especificada en el encabezado no está definida, se compila el primer fragmento, y si está definida, se compila el segundo fragmento.

Por ejemplo, dependiendo de la presencia de la sustitución de macro DEMO, podremos llamar o no a la función de cálculo de los números de Fibonacci.

```
#ifdef DEMO
   Print("Fibo is disabled in the demo");
#else
   FillFibo();
#endif
```

En este caso, si el modo DEMO está activado, en lugar de llamar a la función se mostraría un mensaje en el registro, pero como en el script *Preprocessor.mq5* y en todos los archivos incluidos no hay ninguna definición DEMO *#define*, la compilación procede de acuerdo con la rama *#else*, es decir, la llamada a la función *FillFibo* entra en el archivo ejecutable ex5.

Las directivas pueden anidarse.

```
#ifdef _DEBUG
   Print("Debugging");
#else
   #ifdef _RELEASE
      Print("Normal run");
   #else
      Print("Undefined mode!");
   #endif
#endif
```

2.9.9 Propiedades generales del programa (#property)

Mediante la directiva *#property*, un programador puede establecer algunas propiedades de un programa MQL. Algunas de estas propiedades son generales, es decir, aplicables a cualquier programa, y vamos a verlas aquí. Las propiedades restantes son típicas para tipos específicos de programas MQL5 y se discutirán en las secciones pertinentes de la Parte 5 cuando se describa la API de MQL5.

La directiva *#property* tiene el siguiente formato:

```
#property key value
```

La clave es una de las propiedades enumeradas en la siguiente tabla, en la primera columna. En la segunda columna se especifica cómo se interpretará el valor.

Propiedad	Valor
copyright	Cadena con información sobre el titular de los derechos de autor
link	Cadena con un enlace al sitio del desarrollador
version	Cadena con el número de versión del programa (para MQL5 Market, debe estar en formato «X.Y», donde X e Y son números enteros correspondientes a los números de compilación mayor y menor).
description	Línea con la descripción del programa (se permiten varias directivas *#description* y sus contenidos se combinan)
icon	Cadena, ruta al archivo con el logotipo del programa en formato ICO
stacksize	Entero que especifica el tamaño de la pila en bytes (el valor por defecto es de 4 a 16 MB, dependiendo del tipo de programa y entorno, 1 MB =

Propiedad	Valor
	1024*1024 bytes); si es necesario, el tamaño aumenta hasta 64 MB (máximo).

Todas las propiedades de cadena mencionadas anteriormente son la fuente de información del cuadro de diálogo de propiedades del programa, que se abre al iniciarse. Sin embargo, para los scripts, este cuadro de diálogo no se muestra de manera predeterminada. Para cambiar este comportamiento debe especificar adicionalmente la directiva *#property script_show_inputs*. Además, la información sobre los derechos se muestra en una descripción emergente al pasar el cursor del ratón por encima del programa en el *Navegador* de MetaTrader 5.

Las propiedades *copyright*, *link* y *version* se han visto ya en todos los ejemplos anteriores de este libro.

El tamaño de pila *stacksize* es una recomendación: si el compilador encuentra variables locales (normalmente arrays) en el código fuente que superan el valor especificado, la pila se incrementará automáticamente durante la compilación, pero hasta un máximo de 64 MB. Si se supera el límite, el programa ni siquiera podrá iniciarse: en el registro (pestaña *Log*, y no *Experts*) aparecerá el error «Se ha superado el tamaño de pila de 64MB. Reduzca la memoria ocupada por variables locales».

Tenga en cuenta que estos análisis y prevenciones sólo tienen en cuenta una instantánea fija del programa en el momento del lanzamiento. En el caso de las llamadas a funciones recursivas, el consumo de memoria de la pila puede aumentar considerablemente y provocar un error de desbordamiento de pila, pero ya en la fase de ejecución del programa. Para obtener más información sobre la pila, véase la nota en Descripción de arrays.

Las directivas *#property* sólo funcionan en el archivo mq5 compilado, y se ignoran en todos los incluidos con *#include*.

Parte 3. Programación orientada a objetos en MQL5

En algún momento del proceso de desarrollo de software se pone de manifiesto el problema de que los tipos integrados y el conjunto de funciones no son suficientes para la aplicación efectiva de los requisitos. La complejidad de gestionar muchas de las pequeñas entidades que componen el programa crece como una bola de nieve y exige utilizar algún tipo de tecnología capaz de mejorar la comodidad, la productividad y la calidad del trabajo del programador.

Una de estas tecnologías, implementada a nivel de muchos lenguajes de programación, se denomina Orientada a Objetos, y el estilo de programación basado en ella, Programación Orientada a Objetos (POO), respectivamente. El lenguaje de programación MQL5 también lo admite y, por tanto, pertenece a la familia de los lenguajes orientados a objetos, como C++.

Del nombre de la tecnología se deduce que está organizada en torno a objetos. En esencia, un objeto es una variable de un tipo definido por el usuario, es decir, un tipo definido por un programador utilizando herramientas de MQL5. La posibilidad de crear tipos que modelen el área temática hace que los programas sean más comprensibles y simplifica su redacción y mantenimiento.

En MQL5 existen varios métodos para definir un nuevo tipo, y cada método se caracteriza por algunas características que describiremos en las secciones pertinentes. Según el método de descripción, los tipos definidos por el usuario se dividen en clases, estructuras y asociaciones. Cada uno de ellos puede combinar datos y algoritmos, es decir, describir el estado y el comportamiento de objetos aplicados.

En la primera parte del libro mencionamos la cita de uno de los padres de la programación, Nicklaus Wirth, que decía que los programas son una simbiosis de algoritmos y estructuras de datos. Así, los objetos son esencialmente miniprogramas: cada uno es responsable de resolver su propia tarea, aunque pequeña, pero completa desde el punto de vista lógico. Componiendo objetos en un único sistema se puede construir un servicio o producto de complejidad arbitraria. Así, con la POO obtenemos una nueva interpretación del principio de «divide y vencerás».

La programación orientada a objetos debe considerarse una alternativa más potente y flexible al estilo de programación procedimental que exploramos en la segunda parte. Al mismo tiempo, ambos enfoques no deben contraponerse: si es necesario, pueden combinarse, y en las tareas más sencillas, la programación orientada a objetos puede dejarse de lado.

Así, en esta tercera parte del libro estudiaremos los fundamentos de la POO y las posibilidades de su implementación práctica en MQL5. Además, hablaremos de plantillas, interfaces y espacios de nombres.

Programación en MQL5 para Traders: códigos fuente del libro. Parte 3

Los ejemplos del libro también están disponibles en el proyecto público \MQL5\Shared Projects\MQL5Book

3.1 Estructuras y uniones

Una estructura es el tipo de objeto más fácil de entender, así que empezaremos con ella nuestra introducción a la programación orientada a objetos. Las estructuras tienen mucho en común con las clases, que son los principales bloques de construcción en la programación orientada a objetos, por lo que el conocimiento de las estructuras nos ayudará en el futuro cuando pasemos a las clases. Al mismo tiempo, las estructuras presentan ciertas diferencias, algunas de las cuales pueden considerarse

limitaciones y otras, ventajas. En particular, las estructuras no pueden tener funciones virtuales, pero pueden utilizarse para la integración con DLL de terceros.

La elección entre estructuras y clases en la implementación del algoritmo se basa tradicionalmente en los requisitos de acceso a los elementos del objeto y la presencia de lógica empresarial interna. Si se necesita un contenedor sencillo con datos estructurados y no es necesario comprobar si su estado es correcto (en programación esto se denomina «invariante»), entonces una estructura funcionará perfectamente. Si desea restringir el acceso y admitir la escritura y la lectura de acuerdo con algunas reglas (que se formalizan en forma de funciones asignadas al objeto, de las que hablaremos más adelante), entonces es mejor utilizar clases.

MQL5 tiene tipos integrados de estructuras que describen entidades demandadas para trading, en particular tasas (*MqlRates*), ticks (*MqlTick*), fecha y hora (*MqlDateTime*), solicitudes de operaciones (*MqlTradeRequest*), resultados de solicitudes (*MqlTradeResult*) y muchas otras. Hablaremos de ellos en la Parte 6 de este libro.

3.1.1 Definición de estructuras

Una estructura consta de variables; estas pueden ser integradas o de otros tipos definidos por el usuario. El propósito de la estructura es combinar datos relacionados lógicamente en un único contenedor. Supongamos que tenemos una función que realiza un cálculo determinado y acepta un conjunto de parámetros: número de barras que muestran un historial de cotizaciones para su análisis, fecha en que se inició el análisis, tipo de precio y número de señales asignadas (por ejemplo, armónicas).

```
double calculate(datetime start, int barNumber,
                 ENUM_APPLIED_PRICE price, int components);
```

En realidad, puede haber más parámetros y no será fácil pasarlos a la función como una lista. Además, basándose en los resultados de varios cálculos, tiene sentido guardar algunos de los mejores ajustes en algún tipo de array. Por lo tanto, es conveniente representar un conjunto de parámetros como un único objeto.

La descripción de la estructura con las mismas variables tiene el siguiente aspecto:

```
struct Settings
{
   datetime start;
   int barNumber;
   ENUM_APPLIED_PRICE price;
   int components;
};
```

La descripción comienza con la palabra clave *struct* seguida del identificador que elijamos. A continuación aparece un bloque de código entre llaves, dentro del cual se describen las variables incluidas en la estructura que se denominan también campos o miembros de una estructura. Hay un punto y coma después de las llaves, ya que toda la notación es una sentencia que define un nuevo tipo y al final de las mismas es necesario escribir ';'.

Una vez definido el tipo, podemos aplicarlo del mismo modo que los tipos integrados. En concreto, el nuevo tipo permite describir la variable *Settings* en el programa de la forma habitual.

```
Settings s;
```

Es importante señalar que una única descripción de estructura permite crear un número arbitrario de variables de estructura e incluso arrays de este tipo. Cada instancia de estructura tendrá su propio conjunto de elementos, y estos contendrán valores independientes.

Para acceder a los miembros de una estructura se proporciona un operador de desreferenciación especial: el carácter de punto '.'. A la izquierda debe haber una variable de tipo estructura, y a la derecha, un identificador de uno de los campos disponibles en la misma. A continuación se explica cómo asignar un valor a un elemento de la estructura:

```
void OnStart()
{
   Settings s;
   s.start = D'2021.01.01';
   s.barNumber = 1000;
   s.price = PRICE_CLOSE;
   s.components = 8;
}
```

Hay una forma más cómoda de rellenar la estructura: la inicialización agregada. En este caso, el signo '=' se escribe a la derecha de la variable de estructura, seguido de una lista, separada por comas, con los valores iniciales de todos los campos entre llaves.

```
Settings s = {D'2021.01.01', 1000, PRICE_CLOSE, 8};
```

Los tipos del valor deben coincidir con los tipos del elemento correspondiente. Se permite especificar menos valores que el número de campos: los campos restantes recibirán entonces valores cero.

Tenga en cuenta que este método sólo funciona cuando la variable está inicializada, en el momento de su definición. Es imposible asignar de esta forma el contenido de una estructura ya existente: obtendremos un error de compilación.

```
Settings s;
// error: '{' - parameter conversion not allowed
s = {D'2021.01.01', 1000, PRICE_CLOSE, 8};
```

Utilizando el operador de desreferenciación puede leer también el valor de un elemento de la estructura. Por ejemplo, utilizamos el número de barras para calcular el número de componentes.

```
s.components = (int)(MathSqrt(s.barNumber) + 1);
```

Aquí *MathSqrt* es la función raíz cuadrada integrada.

Hemos introducido un nuevo tipo, *Settings*, para facilitar el paso de un conjunto de parámetros a una función. Ahora puede utilizarse como único parámetro de la función actualizada *calculate*:

```
double calculate(Settings &settings);
```

Fíjese en el ampersand '&' delante del nombre del parámetro, que significa pasar por referencia. Las estructuras sólo pueden pasarse como parámetros por referencia.

Las estructuras también son útiles si necesita devolver un conjunto de valores de una función en lugar de un único valor. Imaginemos que la función *calculate* debe devolver, no un valor del tipo *double*, sino varios coeficientes y algunas recomendaciones de trading (dirección de la operación de trading y

probabilidad de éxito). Entonces, podemos definir el tipo de la estructura *Result* y utilizarlo en el prototipo de la función (*Structs.mq5*).

```
struct Result
{
   double probability;
   double coef[3];
   int direction;
   string status;
};

Result calculate(Settings &settings)
{
   if(settings.barNumber > 1000) // edit fields
   {
      settings.components = (int)(MathSqrt(settings.barNumber) + 1);
   }
   // ...
   // emulate getting the result
   Result r = {};
   r.direction = +1;
   for(int i = 0; i < 3; i++) r.coef[i] = i + 1;
   return r;
}
```

Las llaves vacías de la línea *Result r = {}* representan el inicializador agregado mínimo: rellena todos los campos de la estructura con ceros.

La definición y la declaración del tipo de estructura pueden hacerse por separado, si es necesario (por regla general, la declaración va en el archivo mqh de encabezado, y la definición, en el archivo mq5). Esta sintaxis ampliada se abordará en la sección Capítulo sobre las clases.

3.1.2 Funciones (métodos) de estructuras

Después de recibir un resultado de la función *calculate* sería deseable imprimirlo en el registro, pero la función *Print* no funciona con tipos definidos por el usuario: ellos mismos deben proporcionar una forma de dar salida a la información.

```
void OnStart()
{
   Settings s = {D'2021.01.01', 1000, PRICE_CLOSE, 8};
   Result r = calculate(s);
   // Print(r);  // error: 'r' - objects are passed by reference only
   // Print(&r); // error: 'r' - class type expected
}
```

Los comentarios muestran los intentos de llamar a la función *Print* para la estructura, así como lo que sigue después. El primer error se debe a que las instancias de estructura son objetos, y los objetos deben pasarse a las funciones por referencia. Al mismo tiempo, *Print* espera un valor (uno o varios). El uso de un ampersand delante del nombre de la variable en la segunda llamada *Print* significa en MQL5 que se ha recibido el puntero y no es una referencia como se podría pensar. Los punteros en MQL5 sólo

se admiten para objetos de clase (no estructuras), de ahí el segundo error «tipo de clase esperado». Aprenderemos más sobre los punteros en el próximo capítulo (véase Clases e interfaces).

Podríamos especificar en la llamada *Print* todos los miembros de la estructura por separado (utilizando la desreferenciación), pero esto es bastante problemático.

Para aquellos casos en los que sea necesario procesar el contenido de la estructura de una forma especial se pueden definir funciones dentro de la estructura. La sintaxis de la definición no difiere de las funciones de contexto global conocidas, pero la definición en sí se encuentra dentro del bloque de estructura.

Estas funciones se denominan métodos. Dado que están situados en el contexto del bloque correspondiente, se puede acceder desde ellos a los campos de la estructura sin el operador de desreferenciación. Escribamos a modo de ejemplo la implementación de la función *print* en la estructura *Result*.

```
struct Result
{
   ...
   void print()
   {
      Print(probability, " ", direction, " ", status);
      ArrayPrint(coef);
   }
};
```

Llamar a un método de la instancia de la estructura es tan sencillo como leer su campo: se utiliza el mismo operador '.'.

```
void OnStart()
{
   Settings s = {D'2021.01.01', 1000, PRICE_CLOSE, 8};
   Result r = calculate(s);
   r.print();
}
```

En el capítulo sobre las clases se tratarán los métodos con más detalle.

3.1.3 Copiar estructuras

Las estructuras del mismo tipo pueden copiarse completamente entre sí mediante el operador de asignación '='. Vamos a demostrar esta regla con un ejemplo de la estructura *Result*. Obtenemos la primera instancia de *r* a partir de la función *calculate*.

```
void OnStart()
{
   ...
   Result r = calculate(s);
   r.print();
   // will output to the log:
   // 0.5 1 ok
   // 1.00000 2.00000 3.00000
   ...
   Result r2;
   r2 = r;
   r2.print();
   // will output to the log the same values:
   // 0.5 1 ok
   // 1.00000 2.00000 3.00000
}
```

A continuación, se creó adicionalmente la variable *Result r2* y se duplicó en ella el contenido de la variable *r* , todos los campos al mismo tiempo. La exactitud de la operación puede verificarse mediante la salida al registro utilizando el método *print* (las líneas se indican en los comentarios).

Hay que tener en cuenta que definir dos tipos de estructuras con el mismo conjunto de campos no hace que los dos tipos sean iguales. No es posible asignar por completo una estructura a otra; en estos casos sólo se permite la asignación miembro a miembro.

Un poco más adelante hablaremos de la herencia de estructuras, que le ofrecerá más opciones para copiar. El hecho es que la copia no sólo funciona entre estructuras del mismo tipo, sino también entre tipos relacionados. No obstante, existen matices importantes, que abordaremos en la sección Disposición y herencia de estructuras.

3.1.4 Constructores y destructores

Entre los métodos que se pueden definir para una estructura existen unas funciones especiales: constructores y destructores.

Un constructor tiene el mismo nombre que el nombre de la estructura y no devuelve ningún valor (tipo *void*). El constructor, si está definido, se invocará en el momento de la inicialización para cada nueva instancia de la estructura. Debido a esto, en el constructor se puede calcular el estado inicial de la estructura de una manera especial.

Una estructura puede tener múltiples constructores con diferentes conjuntos de parámetros, y el compilador elegirá el apropiado basándose en el número y tipo de argumentos en el momento de definir la variable.

Por ejemplo, podemos describir un par de constructores en la estructura *Result*: uno sin parámetros y el segundo, con un parámetro de tipo cadena para establecer el estado.

```
struct Result
{
   ...
   void Result()
   {
      status = "ok";
   }
   void Result(string s)
   {
      status = s;
   }
};
```

Por cierto: un constructor sin parámetros se llama constructor por defecto. Si no hay constructores explícitos, el compilador crea implícitamente un constructor por defecto para cualquier estructura que contenga cadenas y arrays dinámicos para rellenar estos campos con ceros.

Es importante que los campos de otros tipos (por ejemplo, todos los numéricos) no se pongan a cero, independientemente de que la estructura tenga un constructor por defecto, por lo que los valores iniciales de los elementos tras la asignación de memoria serán aleatorios. Debería crear constructores o asegurarse de que los valores correctos se asignan en el código inmediatamente después de crear el objeto.

La presencia de constructores explícitos hace imposible utilizar la sintaxis de inicialización agregada. Por eso no se compilará la línea *Result r = {};* del método *calculate*. Ahora tenemos derecho a utilizar sólo uno de los constructores que hemos proporcionado nosotros mismos. Por ejemplo, las siguientes sentencias llaman al constructor sin parámetros:

```
Result r1;
Result r2();
```

Y crear una estructura con un estado relleno puede hacerse así:

```
Result r3("success");
```

También se llama al constructor por defecto (explícito o implícito) cuando se crea un array de estructuras. Por ejemplo, la siguiente sentencia asigna memoria para 10 estructuras con resultados y las inicializa con un constructor por defecto:

```
Result array[10];
```

Un destructor es una función que se invocará cuando el objeto estructura esté siendo eliminado. El destructor tiene el mismo nombre que el nombre de la estructura, pero va precedido del carácter de tilde '~'. El destructor, al igual que el constructor, no devuelve ningún valor, pero tampoco recibe parámetros.

Sólo puede haber un destructor.

No se puede llamar explícitamente al destructor: lo hace el programa en sí al salir de un bloque de código en el que se ha definido una variable de estructura local, o al liberar un array de estructuras.

El propósito del destructor es liberar cualquier recurso dinámico si la estructura asignó dichos recursos en el constructor. Por ejemplo, una estructura puede tener la propiedad *persistence*, es decir, guardar su estado en un archivo cuando se descarga de la memoria y restaurarlo cuando el programa la crea de

nuevo. En este caso se utiliza un descriptor que debe abrirse y cerrarse en las funciones de archivo integradas.

Vamos a definir un destructor en la estructura *Result* y a añadir constructores por el camino para que todos estos métodos hagan un seguimiento del número de instancias de objetos (a medida que se crean y se destruyen).

```
struct Result
{
   ...
   void Result()
   {
      static int count = 0;
      Print(__FUNCSIG__, " ", ++count);
      status = "ok";
   }

   void Result(string s)
   {
      static int count = 0;
      Print(__FUNCSIG__, " ", ++count);
      status = s;
   }

   void ~Result()
   {
      static int count = 0;
      Print(__FUNCSIG__, " ", ++count);
   }
};
```

Hay tres variables estáticas de nombre *count* que existen con independencia unas de otras: cada una de ellas cuenta en el contexto de su propia función.

Como resultado de la ejecución del script, obtendremos el siguiente registro:

```
Result::Result() 1
Result::Result() 2
Result::Result() 3
Result::~Result() 1
Result::~Result() 2
0.5 1 ok
1.00000 2.00000 3.00000
Result::Result(string) 1
0.5 1 ok
1.00000 2.00000 3.00000
Result::~Result() 3
Result::~Result() 4
```

Vamos a ver lo que significa.

La primera instancia de la estructura se crea en la función *OnStart*, en la misma línea en la que se llama a *calculate*. Al entrar en el constructor, el valor del contador *count* se inicializa una vez con cero

y luego se incrementa cada vez que se ejecuta el constructor, por lo que para la primera vez, el valor es 1.

Dentro de la función *calculate* se define una variable local de tipo *Result*; se registra con el número 2.

La tercera instancia de la estructura no es tan obvia. La cuestión es que, para pasar el resultado de la función, el compilador crea implícitamente una variable temporal en la que copia los datos de la variable local. Es probable que este comportamiento cambie en el futuro, y entonces la instancia local se «moverá» fuera de la función sin duplicarse.

La última llamada al constructor es en un método con un parámetro de cadena, por lo que el recuento de llamadas es 1.

Es importante que el número total de llamadas a ambos constructores sea el mismo que el número de llamadas al destructor: 4.

Hablaremos más sobre constructores y destructores en el capítulo sobre las clases.

3.1.5 Empaquetar estructuras en memoria e interactuar con DLLs

Para almacenar una instancia de la estructura se asigna en la memoria un área contigua suficiente para albergar todos los elementos.

A diferencia de C++, aquí los elementos de la estructura se suceden uno detrás de otro en la memoria y no se alinean en el límite de 2, 4, 8 o 16 bytes, dependiendo del tamaño de los propios elementos (los algoritmos de alineación difieren según los distintos compiladores y modos operativos). La alineación de elementos, cuyo tamaño es menor que el del bloque especificado, se realiza añadiendo variables ficticias no utilizadas a la composición de la estructura (el programa no tiene acceso directo a ellas). La alineación se utiliza para optimizar el rendimiento de la memoria.

MQL5 permite cambiar las reglas de alineación si es necesario, principalmente cuando se integran programas MQL con DLLs de terceros que describen tipos específicos de estructuras. Para ellos es necesario preparar una descripción equivalente en MQL5 (véase la sección sobre importación de bibliotecas). Es importante señalar que las estructuras destinadas a la integración sólo deben tener en su definición campos de un conjunto limitado de tipos. Por lo tanto, no pueden utilizar cadenas, arrays dinámicos, objetos de clase y punteros a objetos de clase.

La alineación se controla mediante la palabra clave *pack* añadida a la cabecera de la estructura. Existen dos opciones:

```
struct pack(size) identifier
struct identifier pack(size)
```

En ambos casos, el tamaño es un número entero 1, 2, 4, 8, 16. O bien, puede utilizar el operador *sizeof(built-in_type)* como tamaño; por ejemplo, *sizeof(double)*.

La opción *pack(1)*, es decir, la alineación de bytes, es idéntica al comportamiento por defecto sin el modificador *pack*.

El operador especial *offsetof()* permite averiguar el desplazamiento en bytes de un elemento concreto de la estructura desde su inicio. Tiene 2 parámetros: objeto de estructura e identificador de elemento. Por ejemplo:

```
Print(offsetof(Result, status)); // 36
```

Delante del campo *status* en la estructura *Result* hay 4 valores *double* y un valor *int*: 36 en total.

Cuando diseñe sus propias estructuras se recomienda que coloque primero los elementos más grandes y luego el resto, en orden decreciente de tamaño.

3.1.6 Herencia y disposición de estructuras

Las estructuras pueden tener como campos otras estructuras. Por ejemplo, vamos a definir la estructura *Inclosure* y a utilizar este tipo para el campo *data* en la estructura *Main* (*StructsComposition.mq5*):

```
struct Inclosure
{
   double X, Y;
};

struct Main
{
   Inclosure data;
   int code;
};

void OnStart()
{
   Main m = {{0.1, 0.2}, -1}; // aggregate initialization
   m.data.X = 1.0;            // assignment element by element
   m.data.Y = -1.0;
}
```

En la lista de inicialización, el campo *data* se representa mediante un nivel adicional de llaves con los valores de campo *Inclosure*. Para acceder a los campos de una estructura de este tipo es necesario utilizar dos operaciones de desreferenciación.

Si la estructura anidada no se utiliza en ningún otro lugar, puede declararse directamente dentro de la estructura externa.

```
struct Main2
{
   struct Inclosure2
   {
     double X, Y;
   }
   data;
   int code;
};
```

Otra forma de disponer estructuras es la herencia. Este mecanismo se utiliza normalmente para construir jerarquías de clases (y se tratará en detalle en la sección correspondiente), pero también está disponible para las estructuras.

A la hora de definir un nuevo tipo de estructura, el programador puede indicar el tipo de la estructura progenitora en su cabecera, después del signo de dos puntos (debe definirse antes en el código fuente). Como resultado, todos los campos de la estructura progenitora se añadirán a la estructura hija (en su inicio), y los campos propios de la nueva estructura se ubicarán en memoria detrás de los campos progenitores.

```
struct Main3 : Inclosure
{
   int code;
};
```

En este caso, la estructura progenitora no está anidada, sino que forma parte integrante de la estructura hija. Gracias a ello, rellenar campos no requiere llaves adicionales al inicializar, ni una cadena de múltiples operadores de desreferenciación.

```
Main3 m3 = {0.1, 0.2, -1};
m3.X = 1.0;
m3.Y = -1.0;
```

Las tres estructuras consideradas, *Main*, *Main2* y *Main3*, tienen la misma representación en memoria y un tamaño de 20 bytes, pero son tipos diferentes.

```
Print(sizeof(Main));   // 20
Print(sizeof(Main2));  // 20
Print(sizeof(Main3));  // 20
```

Como hemos dicho antes (véase Copiar estructuras), el operador de asignación '=' puede utilizarse para copiar tipos de estructuras relacionadas, más concretamente aquellas que están enlazadas por una cadena de herencia. En otras palabras: una estructura de tipo progenitor puede escribirse en una estructura de tipo hija (en este caso, los campos añadidos en la estructura derivada permanecerán intactos) o viceversa, una estructura de tipo hija puede escribirse en una estructura de tipo progenitor (en este caso, se cortarán los campos «extra»).

Por ejemplo:

```
Inclosure in = {10, 100};
m3 = in;
```

Aquí, la variable *m3* tiene un tipo *Main3* heredado de *Inclosure*. Como resultado de la asignación *m3* = *in*, los campos *X* y *Y* (la parte común para ambos tipos) se copiarán de la variable *in* del tipo base en los campos *X* y *Y* en la variable *m3* del tipo derivado. El campo *code* de la variable *m3* permanecerá inalterado.

No importa si la estructura hijo es descendiente directa o lejana del ancestro, es decir, la cadena de herencia puede ser larga. Esta copia de campos comunes funciona entre «hijos», «nietos» y otras combinaciones de tipos de distintas ramas del «árbol genealógico».

Si la estructura progenitora sólo tiene constructores con parámetros, debe ser invocada desde la lista de inicialización cuando se hereda el constructor de la estructura derivada. Por ejemplo:

```
struct Base
{
   const int mode;
   string s;
   Base(const int m) : mode(m) { }
};

struct Derived : Base
{
   double data[10];
   // if we remove the constructor, we get an error:
   Derived() : Base(1) { } // 'Base' - wrong parameters count
};
```

En el constructor *Base* rellenamos el campo *mode*. Dado que tiene el modificador *const*, el constructor es la única manera de establecer un valor para él, y esto debe hacerse en forma de una sintaxis de inicialización especial después de los dos puntos (ya no se puede asignar una constante en el cuerpo del constructor). Tener un constructor explícito hace que el compilador no genere un constructor implícito (sin parámetros). Sin embargo, no tenemos un constructor explícito sin parámetros en la estructura *Base*, y en su ausencia, cualquier clase derivada no sabe cómo llamar correctamente al constructor *Base* con un parámetro. Por lo tanto, en la estructura *Derived* se requiere inicializar explícitamente el constructor base: esto se hace también utilizando la sintaxis de inicialización en la cabecera del constructor, después del signo ':'; en este caso, llamamos a *Base(1)*.

Si eliminamos el constructor *Derived* obtenemos un error de «número de parámetros no válido» en el constructor base, porque el compilador intenta llamar al constructor de *Base* por defecto (que debería tener 0 parámetros).

Trataremos la sintaxis y el mecanismo de herencia con más detalle en el Capítulo Clase.

3.1.7 Derechos de acceso

Si es necesario, en la descripción de la estructura puede utilizar palabras clave especiales que representan modificadores de acceso que limitan la visibilidad de los campos desde fuera de la estructura. Hay tres modificadores: *public*, *protected* y *private*. Por defecto, todos los miembros de la estructura son públicos, lo que es equivalente a la siguiente entrada (utilizando la estructura *Result* como ejemplo):

```
struct Result
{
public:
   double probability;
   double coef[3];
   int direction;
   string status;
   ...
};
```

Todos los miembros por debajo del modificador reciben los derechos de acceso correspondientes hasta que se encuentra otro modificador o finaliza el bloque de estructura. Puede haber muchas secciones con diferentes derechos de acceso; no obstante, pueden modificarse de forma arbitraria.

Los miembros marcados como *protected* sólo están disponibles desde el código de esta estructura y estructuras descendientes; es decir, se supone que deben tener métodos públicos, de lo contrario, nadie podrá acceder a dichos campos.

Los miembros marcados como *private* sólo son accesibles desde dentro del código de la estructura. Por ejemplo, si añade *private* antes del campo *status*, lo más probable es que necesite un método para leer el estado por código externo (*getStatus*).

```
struct Result
{
public:
   double probability;
   double coef[3];
   int direction;

private:
   string status;

public:
   string getStatus()
   {
      return status;
   }
   ...
};
```

Sólo será posible establecer el estado a través del parámetro del segundo constructor. Si se accede directamente al campo se producirá el error «sin acceso al miembro privado 'status' de la estructura 'Result'»:

```
// error:
// cannot access to private member 'status' declared in structure 'Result'
r.status = "message";
```

En las clases, el acceso por defecto es *private*. Esto sigue el principio de encapsulación, que trataremos en el Capítulo sobre las clases.

3.1.8 Uniones

Una unión es un tipo definido por el usuario que se compone de campos situados en la misma zona de memoria, debido a lo cual se solapan entre sí. Esto permite escribir un valor de un tipo en una unión, y luego leer su representación interna (a nivel de bits) en la interpretación para otro tipo. Por consiguiente, es posible realizar conversiones no estándar de un tipo a otro.

Los campos de unión pueden ser de cualquier tipo integrado, excepto cadenas, arrays dinámicos y punteros. Además, en las uniones se pueden utilizar estructuras con los mismos tipos de campos simples y sin constructores o destructores.

El compilador asigna para la unión una celda de memoria con un tamaño igual al tamaño máximo entre los tipos de todos los elementos. Así, para la unión con campos como *long* (8 bytes) y *int* (4 bytes), se asignarán 8 bytes.

Todos los campos de la unión se encuentran en la misma dirección de memoria, es decir, están alineados al principio de la unión (tienen un desplazamiento de 0, lo que puede comprobarse utilizando offsetof; véase la sección Empaquetar estructuras).

La sintaxis para describir una unión es similar a la estructura pero utiliza la palabra clave *union*, seguida de un identificador y, a continuación, de un bloque de código con una lista de campos.

Por ejemplo, un algoritmo podría utilizar una matriz del tipo *double* para almacenar varios ajustes, simplemente porque el tipo *double* es uno de los que tienen un tamaño máximo en bytes igual a 8. Digamos que entre los ajustes hay números como *ulong*. Dado que no está garantizado que el tipo *double* reproduzca con precisión valores de *ulong* grandes, es necesario utilizar una unión para «empaquetar» *ulong* en *double* y «desempaquetarlo» de nuevo.

```
#define MAX_LONG_IN_DOUBLE       9007199254740992
// FYI: ULONG_MAX            18446744073709551615

union ulong2double
{
   ulong U;   // 8 bytes
   double D;  // 8 bytes
};
ulong2double converter;

void OnStart()
{
   Print(sizeof(ulong2double)); // 8

   const ulong value = MAX_LONG_IN_DOUBLE + 1;

   double d = value; // possible loss of data due to type conversion
   ulong result = d; // possible loss of data due to type conversion

   Print(d, " / ", value, " -> ", result);
   // 9007199254740992.0 / 9007199254740993 -> 9007199254740992

   converter.U = value;
   double r = converter.D;
   Print(r);               // 4.450147717014403e-308
   Print(offsetof(ulong2double, U), " ", offsetof(ulong2double, D)); // 0 0
}
```

El tamaño de la estructura *ulong2double* es igual a 8, ya que sus dos campos tienen este tamaño. Así, los campos U y D se solapan completamente.

En el ámbito de los números enteros, 9007199254740992 es el mayor valor que está garantizado con un almacenamiento robusto en *double*. En este ejemplo, intentamos almacenar un número más en *double*.

La conversión estándar de *ulong* en *double* tiene como resultado una pérdida de precisión: después de escribir 9007199254740993 en una variable *d* de tipo *double* leemos de su valor ya «redondeado» 9007199254740992 (para obtener información adicional sobre las sutilezas de almacenar números en el tipo *double*, véase la sección Números reales).

Al utilizar el conversor, el número 9007199254740993 se escribe en la unión «tal cual», sin conversiones, ya que lo estamos asignando a un campo U de tipo *ulong*. Su representación en términos de *double* está disponible, de nuevo sin conversiones, desde el campo D. Podemos copiarlo a otras variables y arrays como *double* sin preocuparnos.

Aunque el valor resultante *double* tiene un aspecto extraño, coincide exactamente con el entero original si es necesario extraerlo mediante conversión inversa: escribir en un campo D del tipo *double*, después leer desde un campo U del tipo *ulong*.

Una unión puede tener constructores y destructores, así como métodos. Por defecto, los miembros de la unión tienen derechos de acceso público, pero esto puede ajustarse utilizando modificadores de acceso, como en la estructura.

3.2 Clases e interfaces

Las clases son el principal bloque de construcción en el desarrollo de programas basados en la programación orientada a objetos. En un sentido global, el término clase se refiere a una colección de algo (cosas, personas, fórmulas, etc.) que tienen algunas características comunes. En el contexto de la programación orientada a objetos, esta lógica se mantiene: una clase genera objetos que tienen el mismo conjunto de propiedades y comportamiento.

En los capítulos anteriores de este libro nos familiarizamos con los tipos MQL5 integrados, como *double*, *int* o *string*. El compilador sabe cómo almacenar valores de estos tipos y qué operaciones se pueden realizar con ellos. Sin embargo, puede que no sea muy conveniente utilizar estos tipos a la hora de describir alguna área de aplicación. Por ejemplo, un operador de trading tiene que trabajar con entidades como una estrategia de trading, un filtro de señales, una cesta de divisas y una cartera de posiciones abiertas. Cada una de ellas consta de todo un conjunto de propiedades relacionadas, sujetas a normas específicas de tratamiento y coherencia.

Un programa para automatizar acciones con estos objetos podría consistir únicamente en tipos integrados y funciones sencillas, pero entonces habría que idear formas complicadas de almacenar y vincular propiedades. Aquí es donde la tecnología de programación orientada a objetos viene al rescate, proporcionando para ello mecanismos ya preparados, unificados e intuitivos.

La programación orientada a objetos propone escribir todas las instrucciones para almacenar propiedades, rellenarlas correctamente y realizar las operaciones permitidas en objetos de un tipo determinado definido por el usuario en un único contenedor con código fuente. Combina variables y funciones de una manera determinada. Los contenedores se dividen en clases, estructuras y asociaciones si se enumeran en orden descendente de capacidad y relevancia.

Ya nos encontramos con las estructuras y asociaciones en el capítulo anterior. Estos conocimientos serán útiles también para las clases, pero las clases ofrecen más herramientas del arsenal de la programación orientada a objetos.

Por analogía con una estructura, una clase es una descripción de un tipo definido por el usuario con un método de almacenamiento interno arbitrario y reglas para trabajar con él. Basándose en ella, el programa puede crear instancias de esta clase, los objetos que deben considerarse variables compuestas.

Todos los tipos definidos por el usuario comparten algunos de los conceptos básicos que podríamos llamar teoría de la programación orientada a objetos, pero son especialmente relevantes para las clases. Entre ellos figuran:

- abstracción
- encapsulación
- herencia
- polimorfismo
- composición (diseño)

A pesar de sus complicados nombres, aluden a normas bastante sencillas y familiares del mundo real, trasladadas al mundo de la programación. Empezaremos nuestra inmersión en la programación orientada a objetos analizando estos conceptos. En cuanto a la sintaxis para describir clases y cómo crear objetos, la abordaremos más adelante.

3.2.1 Fundamentos de la programación orientada a objetos: abstracción

A menudo utilizamos conceptos generalizadores en la vida cotidiana para transmitir la esencia de la información sin entrar en detalles. Por ejemplo, para responder a la pregunta «¿cómo has llegado hasta aquí?», una persona puede limitarse a decir «en coche». Y todo el mundo tendrá claro que estamos hablando de un vehículo de 4 ruedas, con un motor y un habitáculo para pasajeros; la marca concreta, el color o el año de fabricación del coche no nos importan.

Tampoco le importa al usuario realmente, a la hora de trabajar con el programa, qué tipo de algoritmo se implementa en él, siempre y cuando el programa realice correctamente su tarea. Por ejemplo, se puede ordenar una lista de una docena de maneras diferentes.

Por tanto, abstracción significa proporcionar una interfaz de programación sencilla que deje ocultas todas las complejidades y detalles de la implementación.

La interfaz de programación es un conjunto de funciones que se definen en el contexto de una clase y que realizan un conjunto de acciones en función de la finalidad de los objetos. Además de estas funciones de interfaz, puede haber funciones auxiliares más pequeñas, pero sólo están disponibles dentro de la clase. De forma similar a las estructuras, existe un nombre especial para todas las funciones de una clase: se denominan métodos.

La implementación, por regla general, utiliza variables o arrays pertenecientes al objeto (según la descripción de la clase) para almacenar información. Se denominan «campos» (este término procede del hecho de que las propiedades de los objetos suelen asociarse, en una relación 1:1, con campos de entrada en la interfaz de usuario, o con campos en bases de datos, en los que puede guardarse el estado actual del objeto para restaurarlo la próxima vez que se inicie el programa).

Los campos y métodos, aunque se describen en la clase, están relacionados con un objeto específico: cada instancia tiene su propio conjunto asignado de variables, tienen valores que son independientes del estado de otros objetos, y los métodos trabajan con los campos de su instancia.

> La interfaz y la aplicación deben ser independientes. Si se desea, un método de implementación debe poder sustituirse fácilmente por otro sin que ello afecte en modo alguno a la interfaz de programación. También es muy importante diseñar la interfaz en función de los requisitos de una tarea concreta, y no personalizarla específicamente para la aplicación. El desarrollador de la clase debe ser capaz de ver su creación desde dos puntos de vista diferentes: como autor de algoritmos internos y estructuras de datos, o como un exigente cliente potencial que utiliza la clase a modo de «caja negra» y su panel de control es la interfaz. Se recomienda empezar a desarrollar una clase desde la concepción de la interfaz de programación hasta la búsqueda y elección de los métodos de implementación.

3.2.2 Fundamentos de la programación orientada a objetos: encapsulación

Para entender qué es la encapsulación, volvamos por un momento al mundo real. Cuando compramos un electrodoméstico, suele venir «precintado» y en garantía. Podemos utilizarlo en modo normal, pero el fabricante no nos anima a abrir la carcasa y empezar a «trastear». Por ejemplo, se pueden usar utilidades especiales para mejorar el rendimiento del procesador del ordenador, pero ello también nos priva de la garantía, ya que estas acciones pueden provocar el fallo del equipo.

Lo mismo ocurre con el desarrollo de las clases: no se debe permitir a nadie el acceso a la implementación interna, a fin de no perturbar la clase. Esto e lo que se denomina encapsulación, es decir, incluir todo lo importante en una cápsula. En MQL5, como en C++, hay 3 niveles de derechos de acceso. Por defecto, la organización de la clase es privada, es decir, oculta a todos sus usuarios. Sólo el código fuente de la propia clase tiene acceso al contenido.

Los usuarios de la clase también son programadores. Incluso si está escribiendo una clase para Ud. mismo, tiene sentido aprovecharse de las máximas restricciones para no romper accidentalmente la clase (al fin y al cabo, la gente tiende a cometer errores y olvidar las características de su propio código al cabo de un tiempo, y los programas tienen tendencia a crecer indefinidamente).

El segundo nivel de acceso permite a los «familiares» (más concretamente, a los herederos; volveremos sobre ellos en un par de párrafos) echar un vistazo dentro.

Por último, el tercer nivel de acceso que se puede elegir es el público. Está pensado específicamente para interfaces de programación externas que permiten utilizar objetos desde cualquier parte del programa para su propósito principal.

Cada método o campo tiene uno de los tres niveles de acceso, lo cual está determinado por el desarrollador de la clase.

3.2.3 Fundamentos de la programación orientada a objetos: herencia

Cuando se construyen proyectos grandes y complejos, la gente busca formas de hacer más eficiente el proceso. Una de las formas más populares es aprovechar los desarrollos existentes. Por ejemplo, es mucho más fácil desarrollar un plan de construcción no desde cero, sino basándose en los planos anteriores.

También en programación es muy popular la reutilización de código. Ya conocemos una de estas técnicas: aislar un fragmento de código en una función y luego llamarla desde distintos lugares en los que se requiera la funcionalidad correspondiente. Pero la programación orientada a objetos ofrece un mecanismo más potente: al desarrollar una nueva clase, ésta puede heredar de otra, adquiriendo toda la estructura interna y la interfaz externa, lo que requiere sólo un ajuste mínimo para adaptarse al propósito. Así, partiendo de la clase padre, se puede «hacer crecer» rápidamente una clase derivada con capacidades adicionales o afinadas. Además, cualquier cambio posterior en la clase padre (como mejoras o correcciones de errores) afectará automáticamente a todas las clases hijas.

Cuando una clase es padre de otra, se denomina clase base. A su vez, la clase que hereda de la clase base se denomina derivada.

Por supuesto, la cadena de herencia (o, mejor dicho, el árbol genealógico) puede continuar: cada clase puede tener varios herederos y éstos, a su vez, sus propios herederos, y así sucesivamente. Lo único que no permiten las reglas de herencia son los ciclos en las relaciones de parentesco; por ejemplo, un nieto no puede ser padre de su abuelo.

La relación entre cualquier clase y su descendiente de cualquier generación se describe mediante las palabras «es un(a)», es decir, el descendiente puede actuar como antepasado, pero no viceversa. Esto se debe a que el objeto derivado contiene en realidad el modelo de datos del ancestro y lo complementa con nuevos campos y comportamientos.

Al heredar clases entre sí, tenemos la oportunidad de procesar objetos relacionados de forma unificada, ya que algunas de sus funciones son comunes.

Por ejemplo, un hipotético programa de dibujo puede utilizarse para implementar varios tipos de formas, como círculos, cuadrados, triángulos, etc. Cada objeto tiene coordenadas en la pantalla (para simplificar, supondremos que se especifica un par de valores X e Y del centro de la forma). Además, cada forma se representa con su propio color de fondo, color de borde y grosor de borde.

Esto significa que podemos implementar funciones para establecer las coordenadas y el estilo de dibujo sólo una vez en la clase padre que describe la forma abstracta, y estas funciones serán heredadas automáticamente por todos los descendientes.

Además, para simplificar el código fuente, es conveniente unificar de algún modo no sólo los ajustes, sino también el dibujo de las distintas formas. Esta frase encierra algún tipo de contradicción: puesto que las formas son diferentes y cada una debe mostrarse a su manera, ¿de qué tipo de unificación estamos hablando? Hablamos de una interfaz de software unificada. De hecho, según el concepto de abstracción, es necesario separar la interfaz externa de la implementación interna. Y la visualización de formas específicas es en esencia un detalle de implementación.

Una interfaz unificada y diferentes implementaciones para los tipos de forma nos llevan fácilmente al siguiente concepto: el polimorfismo.

3.2.4 Fundamentos de la programación orientada a objetos: polimorfismo

El término polimorfismo significa variabilidad o diversidad. Es lo contrario de la abstracción combinada con el mecanismo de herencia. Cuando tenemos una interfaz de programación común, esta puede implementarse por medio de diferentes clases vinculadas por relaciones de herencia. Llamar a los métodos de la interfaz hará que la tarea se realice de diferentes maneras.

Por ejemplo, imagine una familia de vehículos abstractos que incluya un par de tipos determinados, como un coche y un helicóptero. Ambos ejecutarán igual de bien la orden de desplazarse del punto A al punto B, pero el coche lo hará por tierra y el helicóptero, por el aire.

Continuemos el ejemplo con el programa de dibujo. Podemos decir que la diversidad en ella se establece a nivel de formas gráficas. El usuario es libre de dibujar cualquier combinación de círculos, cuadrados y triángulos. Cada uno de estos objetos debe poder mostrarse en la pantalla utilizando sus propias coordenadas y su propio estilo, pero lo más importante es hacerlo de manera que produzca una forma adecuada.

Lo más probable es que el programa tenga un array (u otro contenedor) que almacene todas las formas creadas por el usuario, y mostrar todo el dibujo en la pantalla debería consistir en dibujar secuencialmente cada forma. Si reducimos las instrucciones de dibujo para las formas a un método separado (llamémoslo *draw*), entonces cada clase tendrá su propia implementación. Sin embargo, los encabezados de estas funciones serán completamente idénticas, ya que realizan la misma tarea y toman los datos iniciales de los objetos.

Por lo tanto, tenemos la oportunidad de unificar el código fuente, ya que la misma llamada a *draw* dentro del bucle sobre formas presenta polimorfismo: la forma mostrada dependerá del tipo de objeto.

3.2.5 Fundamentos de la programación orientada a objetos: composición (diseño)

Cuando se diseñan programas utilizando la programación orientada a objetos, existe el problema de encontrar la división óptima (según unas características dadas) en clases y relaciones entre ellas. El término «composición» puede resultar ambiguo y a menudo se utiliza con distintos significados, incluido uno de los casos especiales de «componer» clases. Esta digresión es necesaria porque, cuando se lee otra literatura informática, pueden encontrarse diferentes interpretaciones del término «composiciones», tanto en un sentido generalizado como en un sentido más restringido. Intentaremos explicar este concepto, especificando el significado de los términos en cada caso (cuándo se refiere al «diseño o desarrollo del proyecto» en general de la interfaz de software, y cuándo se refiere a la «agregación compositiva»).

Así, la clase, como sabemos, consta de campos (propiedades) y métodos. Las propiedades, a su vez, pueden describirse mediante tipos personalizados, es decir, pueden ser objetos de otra clase. Hay varias formas de conectar lógicamente estos objetos:

- **Composición** (completa **inclusión** o agregación compositiva) de objetos-campos en un objeto propietario. La relación de tales objetos se describe mediante la relación «todo-parte», y la parte no puede existir fuera del todo. Se dice que el objeto propietario «tiene un» objeto de propiedad, y el objeto de propiedad es una «parte de» el objeto propietario. El propietario crea y destruye sus partes. Al borrar el propietario se eliminan todas sus partes; el propietario no puede existir sin partes.
- **Agregación** de objetos-campos por parte del objeto propietario; esto es una inclusión «más suave». Aunque la relación también se describe como «todo-parte», el propietario sólo contiene referencias a partes que pueden asignarse, modificarse y existir aisladas del todo. Además, una pieza puede utilizarse en varios «propietarios».
- **Asociación**, es decir, una conexión unidireccional o bidireccional de objetos independientes que tiene un significado aplicado arbitrario; se dice que un objeto «utiliza» a otro.

Otro tipo de relación a tener en cuenta es «es un(a)», abordada anteriormente en la herencia.

Un ejemplo de inclusión total es un coche y su motor. Aquí, un coche se entiende como un medio de transporte en toda regla. Pero no es así si no tiene motor. Y un motor determinado pertenece sólo a un coche a un mismo tiempo. Las situaciones en las que todavía no hay motor en el coche (en la fábrica) o este ya no existe (en el taller de reparación de coches) son equivalentes a que hayamos roto el código fuente del programa.

Un ejemplo de agregación es la composición de grupos de alumnos para los estudios de determinadas asignaturas: un grupo para cada asignatura incluye a varios alumnos, y cualquiera de ellos puede pertenecer a otros grupos (si estudia varias asignaturas). El grupo «tiene» estudiantes. La salida de un alumno del grupo no afecta al proceso educativo del grupo (el resto sigue estudiando).

Por último, para demostrar la idea de asociación, consideremos un ordenador y una impresora. Podemos decir que el ordenador utiliza la impresora para imprimir. La impresora puede encenderse o apagarse según sea necesario, y la misma impresora puede utilizarse desde distintos ordenadores. Todos los ordenadores e impresoras son independientes entre sí, pero pueden compartirse.

En cuanto a las características que suelen guiar el diseño de las clases, las más famosas son:

- DRY (Don't repeat yourself): en lugar de ello, mueva las partes comunes a clases padre (posiblemente abstractas).

- SRP (Single Responsibility Principle): una clase debe realizar una sola tarea, y si no es así, hay que dividirla en otras más pequeñas.
- OCP (Open-Closed Principle): «escribir código abierto a ampliación pero cerrado a modificación». Si hay varias opciones de cálculo codificadas en la clase X y pueden aparecer otras nuevas, cree una clase base (abstracta) para un cálculo independiente y cree opciones específicas («extensión» de la funcionalidad) a partir de ella,conectadas a la clase X sin modificarla.

Éstas son sólo algunas de las mejores prácticas del diseño de clase. Una vez que domine los fundamentos de la programación orientada a objetos en el ámbito de este libro, puede ser útil consultar otras fuentes de información especializadas en el tema, ya que proporcionan soluciones ya preparadas para la descomposición de objetos en muchas situaciones comunes.

3.2.6 Definición de clase

La declaración de definición de clase tiene muchos componentes opcionales que afectan a sus características. De forma generalizada, puede representarse del siguiente modo:

```
class class_name [: modifier_access name_parent_class ...]
{
  [ modifier_access:]
     [description_member...]
  ...
};
```

Para facilitar la presentación, empezaremos por la sintaxis mínima suficiente y la iremos ampliando a medida que avancemos en el material.

Como punto de partida, utilizamos una tarea con un programa de dibujo condicional que admite varios tipos de formas.

Para definir una nueva clase, utilice la palabra clave class seguida del identificador de la clase y un bloque de código entre llaves. Como todas las sentencias, una definición de este tipo debe terminar con un punto y coma.

El bloque de código puede estar vacío. Por ejemplo, una plantilla compilable de la clase *Shape* para un programa de dibujo tiene este aspecto:

```
class Shape
{
};
```

De los capítulos anteriores del libro, sabemos que las llaves denotan el contexto o ámbito de las variables. Cuando tales bloques aparecen en la definición de una función, definen su contexto local. Además, existe un contexto global en el que se definen las propias funciones, así como las variables globales.

Esta vez, los paréntesis de la definición de clase definen un nuevo tipo de contexto: el contexto de clase. Este es un contenedor tanto para variables como para funciones declaradas dentro de la clase.

La descripción de variables para almacenar propiedades de clase se realiza mediante las sentencias habituales dentro del bloque (*Shapes1.mq5*).

```
class Shape
{
   int x, y;              // center coordinates
   color backgroundColor; // fill color
};
```

Aquí hemos declarado algunos de los campos analizados en las secciones teóricas: las coordenadas del centro de la forma y el color de relleno.

Tras dicha descripción, el tipo definido por el usuario *Shape* pasa a estar disponible en el programa junto con los tipos integrados. En concreto, podemos crear una variable de este tipo, y ésta contendrá en su interior los campos especificados. Sin embargo, aún no podemos hacer nada con ellos, ni siquiera asegurarnos de que están ahí.

```
void OnStart()
{
   Shape s;
   // errors: cannot access private member declared in class 'Shape'
   Print(s.x, " ", s.y);
}
```

Los miembros de una clase son privados por defecto y, por lo tanto, no se puede acceder a ellos desde otras partes del código externas a la clase. Este es el principio de la encapsulación en acción.

Si intentamos dar salida a una forma en el registro, el resultado nos decepcionará por varias razones.

El enfoque más sencillo provocará el error «los objetos sólo se pasan por referencia» (también lo hemos visto con estructuras):

```
Print(s); // 's' - objects are passed by reference only
```

Los objetos pueden constar de muchos campos y, debido a su gran tamaño, resulta ineficiente pasarlos por valor. Por lo tanto, el compilador requiere que los parámetros de tipo objeto se pasen por referencia, mientras que *Print* toma valores.

De la sección sobre los parámetros de las funciones (véase la sección Parámetros de valor y parámetros de referencia), sabemos que el símbolo '&' se utiliza para describir referencias. Sería lógico suponer que, para obtener una referencia a una variable (en este caso, un objeto *s* de tipo *Shape*), es necesario anteponer el mismo signo a su nombre.

```
Print(&s);
```

Esta sentencia compila y se ejecuta sin problemas, pero no hace exactamente lo que se esperaba.

El programa produce un número entero durante la ejecución, por ejemplo, 1 o 2097152 (lo más probable es que sea diferente). Un signo ampersand delante de un nombre de variable significa obtener un puntero a esta variable, no una referencia (a diferencia de la descripción de un parámetro de función).

Los punteros se tratarán en detalle en otra sección. No obstante, tenga en cuenta que MQL5 no proporciona acceso directo a la memoria, y el puntero a un objeto es un descriptor, o de una manera sencilla, un número de objeto único (es asignado por el propio terminal). Pero incluso si el puntero apuntara a una dirección en memoria (como ocurre en C++), eso no proporcionaría una forma legal de leer el contenido del objeto.

Para enviar el contenido de los objetos *Shape* al registro o lo que sea, se necesita una función miembro de la clase que podemos llamar *toString* y debe devolver una cadena con alguna descripción del objeto. Podemos decidir más tarde qué mostrar en ella. Reservemos también el método *draw* para dibujar la forma. Por ahora, servirá de declaración de la futura interfaz de programación de objetos.

```
class Shape
{
   int x, y;              // center coordinates
   color backgroundColor; // fill color

   string toString()
   {
      ...
   }

   void draw() { /* future drawing interface stub */ }
};
```

La definición de las funciones de los métodos se realiza de la forma habitual, con la única diferencia de que se sitúan dentro del bloque de código que forma la clase.

> En el futuro aprenderemos a separar la declaración de una función dentro del bloque de clase y su definición fuera del bloque. Este enfoque se utiliza a menudo para poner las declaraciones en un archivo de encabezado y «ocultar» las definiciones en un archivo mq5. Esto hace que el código sea más comprensible (debido a que la interfaz de programación se presenta por separado, de forma compacta, sin implementación). También permite distribuir bibliotecas de software como archivos ex5 si es necesario (sin el código fuente principal pero proporcionando un archivo de encabezado que es suficiente para llamar a los métodos de la interfaz externa).

Dado que el método *toString* forma parte de la clase, tiene acceso a las variables y puede convertirlas en una cadena. Por ejemplo:

```
string toString()
{
  return (string)x + " " + (string)y;
}
```

No obstante, ahora *toString* y *draw* son privados, al igual que el resto de los campos. Tenemos que hacer que estén disponibles desde fuera de la clase.

3.2.7 Derechos de acceso

Se proporciona una sintaxis especial para editar el acceso a los miembros de la clase (ya la vimos en el capítulo sobre estructuras). En cualquier parte del bloque, antes de la descripción de los miembros de la clase, puede insertar un modificador: una de las tres palabras clave (*private*, *protected*, *public*) y un signo de dos puntos.

Todos los miembros que sigan al modificador, hasta que se encuentre otro modificador, o hasta el final de la clase, recibirán la restricción de visibilidad correspondiente.

Por ejemplo, la siguiente entrada es idéntica a la descripción anterior de la clase *Shape*, ya que se asume el modo *private* para las clases sin modificadores:

```
class Shape
{
private:
   int x, y;              // center coordinates
   color backgroundColor; // fill color
   ...
};
```

Si quisiéramos abrir el acceso a todos los campos, cambiaríamos el modificador a *public*

```
class Shape
{
public:
   int x, y;              // center coordinates
   ...
};
```

Pero eso infringiría el principio de encapsulación, y no lo vamos a hacer. En lugar de ello, insertamos el modificador *protected*: permite acceder a los miembros de las clases derivadas dejándolos ocultos al mundo exterior. Estamos planeando extender la clase *Shape* a varias otras clases de forma que necesitarán acceso a las variables del padre.

```
class Shape
{
protected:
   int x, y;              // center coordinates
   color backgroundColor; // fill color

public:
   string toString() const
   {
      return (string)x + " " + (string)y;
   }

   void draw() { /* shape drawing interface stub */ }
};
```

Por el camino, hemos hecho públicas ambas funciones.

Los modificadores pueden intercalarse en la descripción de la clase de forma arbitraria y repetirse muchas veces. Sin embargo, para mejorar la legibilidad del código, se recomienda hacer una sección de miembros *public*, *protected* y *private* , y mantener el mismo orden en todas las clases del proyecto.

Observe que hemos añadido la palabra clave *const* al final del encabezado de la función *toString*, lo que significa que la función no cambia el estado de los campos del objeto. Aunque no es necesario, ello ayuda a prevenir la corrupción accidental de variables y también permite a los usuarios de la clase y al compilador saber que la llamada a la función no dará lugar a ningún efecto secundario.

En la función *toString*, como en cualquier método de clase, los campos son accesibles por sus nombres. Más adelante veremos cómo declarar métodos como estáticos: están totalmente relacionados con la clase, no con las instancias del objeto, y por lo tanto no se puede acceder a los campos.

Ahora podemos llamar al método *toString* desde la variable de objeto *s*:

```
void OnStart()
{
   Shape s;
   Print(s.toString());
}
```

Aquí vemos el uso del carácter de punto '.' como operador de desreferenciación especial: proporciona acceso a los miembros del objeto, es decir, campos y métodos. A la izquierda debe haber un objeto, y a la derecha, el identificador de una de las propiedades disponibles.

El método *toString* es público, y por tanto accesible desde una función externa a la clase *OnStart*. Si intentáramos en *OnStart* «llegar» a los campos *s.x* o *s.y* mediante dereferenciación, obtendríamos un error de compilación «no se puede acceder al miembro protegido declarado en 'Shape'».

Para los profesionales de C++, señalamos que MQL5 no admite los llamados «amigos» (para el resto, expliquemos que en C++ es posible, si es necesario, hacer una especie de «lista blanca» de clases y métodos de terceros que tienen derechos extendidos, aunque no son «parientes»).

Cuando ejecutemos el programa, veremos que da un par de números. No obstante, los valores de las coordenadas serán aleatorios. Incluso si tiene la suerte de ver nulos, ello no garantiza que aparezcan la siguiente vez que ejecute el script. Por regla general, si la lista de programas MQL en ejecución no cambia en el terminal, los lanzamientos repetidos de cualquier script resultan en la asignación de la misma área de memoria al mismo, lo que puede dar la impresión engañosa de que el estado del objeto es estable. De hecho, los campos de un objeto, como en el caso de las variables locales, no se inicializan con nada por defecto (véase la sección Inicialización).

Para inicializarlos se utilizan funciones especiales de la clase: los constructores.

3.2.8 Constructores: por defecto, paramétricos y de copia

Ya hemos visto constructores en el capítulo sobre estructuras (véase la sección Constructores y destructores). Para las clases, funcionan de forma muy parecida. Volvamos a los puntos principales y analicemos otras características.

Un constructor es un método que tiene el mismo nombre que la clase y es de tipo void, lo que significa que no devuelve ningún valor. Normalmente, la palabra clave *void* se omite delante del nombre del constructor. Una clase puede tener varios constructores, que deben diferir en el número o tipo de parámetros. Cuando se crea un nuevo objeto, el programa llama al constructor para poder establecer los valores iniciales de los campos.

Una de las formas de crear un objeto que hemos utilizado es la descripción en el código de la variable de la clase correspondiente. Se llamará al constructor en esta cadena; ocurre automáticamente.

En función de la presencia y los tipos de parámetros, los constructores se dividen en:

- constructor por defecto: sin parámetros;
- constructor de copia: con un único parámetro que es el tipo de una referencia a un objeto de la misma clase;
- constructor paramétrico: con un conjunto arbitrario de parámetros, excepto una única referencia para la copia mostrada anteriormente.

Constructor por defecto

El constructor más sencillo, sin parámetros, se denomina constructor por defecto. A diferencia de C++, MQL5 no considera constructor por defecto a un constructor que tiene parámetros y todos ellos tienen valores por defecto (es decir, todos los parámetros son opcionales; véase la sección Parámetros opcionales).

Vamos a definir un constructor por defecto para la clase *Shape*.

```
class Shape
{
   ...
public:
   Shape()
   {
      ...
   }
};
```

Por supuesto, ello debe hacerse en la sección pública de la clase.

Los constructores a veces se hacen protegidos o privados de forma deliberada para controlar cómo se crean los objetos, por ejemplo, a través de métodos de fábrica. Pero en este caso estamos considerando la versión estándar de la composición de clases.

Para establecer los valores iniciales de las variables de los objetos podemos utilizar las sentencias de asignación habituales:

```
public:
   Shape()
   {
      x = 0;
      y = 0;
      ...
   }
```

No obstante, la sintaxis del constructor ofrece otra opción. Se denomina lista de inicialización y se escribe después del encabezado de la función, separada por dos puntos. La lista en sí es una secuencia de nombres de campo separados por comas, con el valor inicial deseado entre paréntesis a la derecha de cada nombre.

Por ejemplo, para el constructor *Shape* se puede escribir de la siguiente manera:

```
public:
   Shape() :
      x(0), y(0),
      backgroundColor(clrNONE)
   {
   }
```

Esta sintaxis es preferible a la asignación de variables en el cuerpo de un constructor por varias razones.

En primer lugar, la asignación en el cuerpo de la función se realiza después de haber creado la variable correspondiente. Dependiendo del tipo de variable, esto puede significar que primero se llamó al

constructor por defecto para ella y luego se sobrescribió el nuevo valor (y esto supone gastos adicionales). En el caso de una lista de inicialización, la variable se crea inmediatamente con el valor deseado. Es probable que el compilador pueda optimizar la asignación en ausencia de una lista de inicialización, pero en el caso general, esto no está garantizado.

En segundo lugar, algunos campos de clase pueden declararse con el modificador *const* . Entonces sólo se pueden establecer en la lista de inicialización.

En tercer lugar, las variables de campo de tipos definidos por el usuario pueden no tener un constructor por defecto (es decir, todos los constructores disponibles en su clase tienen parámetros). Esto significa que, cuando se crea una variable, es necesario pasarle parámetros reales, y la lista de inicialización le permite hacerlo: los valores de los argumentos se especifican dentro de paréntesis, como si se tratara de una llamada explícita al constructor. Se puede utilizar una lista de inicialización en las definiciones de los constructores, pero no en otros métodos.

Constructor paramétrico

Un constructor paramétrico, por definición, tiene múltiples parámetros (uno o más).

Por ejemplo, imaginemos que para las coordenadas *x* y *y* se describe una estructura especial con un constructor paramétrico:

```
struct Pair
{
   int x, y;
   Pair(int a, int b): x(a), y(b) { }
};
```

Entonces podemos utilizar el campo *coordinates* del nuevo tipo *Pair* en lugar de los dos campos de enteros *x* y *y* de la clase *Shape*. Esta construcción de objetos se denomina inclusión o agregación compositiva. El objeto *Pair* es parte integrante del objeto *Shape*. Un par de coordenadas se crea y destruye automáticamente junto con el objeto «anfitrión».

Dado que *Pair* no tiene constructor sin parámetros, el campo *coordinates* debe especificarse en la lista de inicialización del constructor *Shape*, con dos parámetros *(int, int)*:

```
class Shape
{
protected:
   // int x, y;
   Pair coordinates;  // center coordinates (object inclusion)
   ...
public:
   Shape() :
      // x(0), y(0),
      coordinates(0, 0), //object initialization
      backgroundColor(clrNONE)
   {
   }
};
```

Sin una lista de inicialización, estos objetos automáticos no pueden crearse.

Dado el cambio en cómo se almacenan las coordenadas en el objeto, necesitamos actualizar el método *toString*:

```
string toString() const
{
   return (string)coordinates.x + " " + (string)coordinates.y;
}
```

Pero esta no es la versión final: pronto haremos algunos cambios más.

Recordemos que las variables automáticas se describieron en la sección Instrucciones de definición y declaración . Se llaman automáticas porque el compilador las crea (asigna memoria) automáticamente, y también las borra automáticamente cuando la ejecución del programa abandona el contexto (bloque de código) en el que se creó la variable.

En el caso de las variables de objeto, la creación automática significa no sólo la asignación de memoria, sino también una llamada al constructor. La eliminación automática de un objeto va acompañada de una llamada a su destructor (véase más adelante la sección Destructores). Además, si el objeto forma parte de otro objeto, entonces su vida útil coincide con la vida útil de su «propietario», como en el caso del campo *coordinates*: una instancia de *Pair* en el objeto *Shape*.

Los objetos estáticos (incluidos los globales) también son gestionados automáticamente por el compilador.

Una alternativa a la asignación automática es la creación y manipulación dinámica de objetos mediante punteros.

En la sección sobre herencia descubriremos cómo una clase puede heredarse de otra. En este caso, la lista de inicialización es la única forma de llamar al constructor paramétrico de la clase base (el compilador no es capaz de generar de forma automática una llamada al constructor con parámetros, como hace implícitamente para el constructor por defecto).

Añadamos otro constructor a la clase *Shape* que permita establecer valores específicos para las variables. Será simplemente un constructor paramétrico (puede crear tantos como quiera, para diferentes propósitos y con un conjunto diferente de parámetros).

```
Shape(int px, int py, color back) :
   coordinates(px, py),
   backgroundColor(back)
{
}
```

La lista de inicialización asegura que cuando se ejecuta el cuerpo del constructor, todos los campos internos (incluyendo los objetos anidados, si los hay) ya han sido creados e inicializados.

El orden de inicialización de los miembros de la clase no se corresponde con la lista de inicialización, sino con la secuencia de su declaración en la clase.

Si se declara un constructor con parámetros en una clase y se requiere que permita la creación de objetos sin argumentos, el programador debe implementar explícitamente el constructor por defecto.

En el caso de que no haya ningún constructor en la clase, el compilador proporciona implícitamente un constructor por defecto en forma de función stub, que se encarga de inicializar los campos de los siguientes tipos: cadenas, arrays dinámicos y objetos automáticos con un constructor por defecto. Si no existen tales campos, el constructor implícito por defecto no hace nada. Los campos de otros tipos

no se ven afectados por el constructor implícito, por lo que contendrán «basura» aleatoria. Para evitarlo, el programador debe declarar explícitamente el constructor y establecer los valores iniciales.

Constructor de copia

El constructor de copia permite crear un objeto a partir de otro objeto pasado por referencia como único parámetro.

Por ejemplo, para la clase *Shape*, el constructor de copia podría tener este aspecto:

```
class Shape
{
   ...
   Shape(const Shape &source) :
      coordinates(source.coordinates.x, source.coordinates.y),
      backgroundColor(source.backgroundColor)
   {
   }
   ...
};
```

Tenga en cuenta que los miembros protegidos y privados de otro objeto están disponibles en el objeto actual porque los permisos funcionan a nivel de clase. En otras palabras: dos objetos de la misma clase pueden acceder a los datos del otro cuando se les da una referencia (o puntero).

Si existe tal constructor, puede crear objetos utilizando uno de los dos tipos de sintaxis:

```
void OnStart()
{
   Shape s;
   ...
   Shape s2(s);   // ok: syntax 1 - copying
   Shape s3 = s;  // ok: syntax 2 - copying via initialization
                  //                  (if there is copy constructor)
                  //                - or assignment
                  //                  (if there is no copy constructor,
                  //                   but there is default constructor)

   Shape s4;      // definition
   s4 = s;        // assignment, not copy constructor!
}
```

Es necesario distinguir entre la inicialización de un objeto durante la creación y la asignación.

La segunda opción (marcada con el comentario «sintaxis 2») funcionará aunque no exista constructor de copia, pero sí un constructor por defecto. En este caso, el compilador generará un código menos eficiente: primero, utilizando el constructor por defecto, creará una instancia vacía de la variable receptora (*s3*, en este caso), y luego copiará los campos de la muestra (*s*, en este caso) elemento a elemento. De hecho, se dará el mismo caso que con la variable *s4*, para la que la definición y la asignación se realizan mediante sentencias separadas.

Si no existe constructor de copia, al intentar utilizar la primera sintaxis se producirá un error de «conversión de parámetros no permitida», ya que el compilador intentará tomar algún otro constructor disponible con un conjunto diferente de parámetros.

Tenga en cuenta que si la clase tiene campos con el modificador *const*, la asignación de tales objetos está prohibida por razones obvias: un campo constante no se puede cambiar, sólo se puede establecer una vez al crear un objeto. Por lo tanto, el constructor de copia se convierte en la única forma de duplicar un objeto.

En concreto, en las secciones siguientes completaremos nuestro ejemplo *Shape1.mq5* y el campo siguiente aparecerá en la clase *Shape* (con una cadena de descripción *type*). El operador de asignación generará errores (en particular, para líneas tales como con la variable *s4*):

```
attempting to reference deleted function
   'void Shape::operator=(const Shape&)'
function 'void Shape::operator=(const Shape&)' was implicitly deleted
   because member 'type' has 'const' modifier
```

Gracias a la detallada redacción del compilador se puede comprender la esencia y las razones de lo que ocurre: en primer lugar, se menciona el operador de asignación ('='), y no el constructor de copia; en segundo lugar, se informa de que el operador de asignación se ha eliminado de forma implícita debido a la presencia del modificador *const*. Aquí nos encontramos con conceptos aún desconocidos que estudiaremos más adelante: sobrecarga de operadores en las clases, conversión de tipo de objetoy la posibilidad de marcar métodos como suprimidos.

En la sección herencia, después de aprender a describir las clases derivadas, necesitamos hacer algunas aclaraciones sobre los constructores de copia en las jerarquías de clases.

3.2.9 Destructores

En el capítulo sobre estructuras hablamos de los destructores (véase la sección sobre Constructores y destructores). Recapitulemos brevemente: un destructor es un método que se llama cuando se destruye un objeto. El destructor comparte el mismo nombre que la clase, pero va precedido de una tilde (~). Los destructores no devuelven valores y no tienen parámetros. Una clase sólo puede tener un destructor.

Incluso si la clase no tiene destructor o el destructor está vacío, el compilador realizará implícitamente la «recogida de basura» de los siguientes tipos de campos: cadenas, arrays dinámicos y objetos automáticos.

Normalmente, el destructor se coloca en la sección pública de la clase; sin embargo, en algunos casos específicos, el desarrollador puede moverlo a un grupo de miembros de *private* o *protected*. Un destructor privado o protegido no le permitirá declarar una variable automática de esta clase en el código. Sin embargo, más adelante veremos la creación dinámica de objetos , y para ellos, tal restricción podría tener sentido.

En particular, algunos objetos pueden implementarse de forma que deban borrarse a sí mismos cuando ya no se necesiten (el concepto de determinación de la demanda puede ser diferente). En otras palabras: mientras los objetos son utilizados por cualquier parte del programa, dichos objetos existen, y tan pronto como se completa la tarea, se autodestruyen (un destructor privado deja la posibilidad de eliminar el objeto de los métodos de la clase).

Para los programadores experimentados de C++, vale la pena señalar que los destructores son siempre virtuales en MQL5 (obtendrá más información acerca de los métodos virtuales en la sección sobre Métodos virtuales (virtual y override)). Este factor no afecta a la sintaxis de la descripción.

En el ejemplo del programa de dibujo, técnicamente, un destructor puede no ser necesario para las formas. Sin embargo, con el fin de trazar la secuencia de llamadas a constructores y destructores, incluiremos uno. Empecemos con un esquema simplificado que «imprima» el nombre completo del método:

```
class Shape
{
   ...
   ~Shape()
   {
      Print(__FUNCSIG__);
   }
};
```

Pronto añadiremos este y otros métodos para poder distinguir una instancia de un objeto de otra.

Veamos el siguiente ejemplo. Un par de objetos *Shape* se describen en dos contextos diferentes: global (fuera de las funciones) y local (dentro de *OnStart*). Se llamará al constructor del objeto global una vez cargado el script y antes de llamar a *OnStart* , y se llamará al destructor antes de que se descargue el script. Se llamará al constructor del objeto local en la línea con la definición de la variable, y el destructor se llamará cuando el bloque de código que contiene la definición de la variable salga, en este caso la función *OnStart*.

```
// the global constructor and destructor are related to script loading and unloading
Shape global;

// object reference does not create a copy and does not affect lifetime
void ProcessShape(Shape &shape)
{
   // ...
}

void OnStart()
{
   // ...
   Shape local; // <- local constructor call
   // ...
   ProcessShape(local);
   // ...
} // <- local destructor call
```

Pasar un objeto por referencia a otras funciones no crea copias del mismo y no llama al constructor ni al destructor.

3.2.10 Autorreferencia: esto

En el contexto de cada clase, en su código de métodos, hay una referencia especial al objeto actual: *this*.. Básicamente, se trata de una variable definida implícitamente y se le pueden aplicar todos los métodos de trabajo con variables de objeto. En concreto, puede desreferenciarse para que se refiera a

un campo de objeto o para llamar a un método. Por ejemplo, las siguientes sentencias en un método de la clase *Shape* son idénticas (utilizamos el método *draw* sólo a efectos de demostración):

```
class Shape
{
   ...
   void draw()
   {
      backgroundColor = clrBlue;
      this.backgroundColor = clrBlue;
   }
};
```

Podría ser necesario utilizar la forma larga si hay otras variables o parámetros con el mismo nombre en el mismo contexto. Esta práctica no suele ser bienvenida, pero si es necesario, la palabra clave *this* permite hacer referencia a los miembros anulados de un objeto.

El compilador emite un aviso si el nombre de cualquier variable local o parámetro de método se solapa con el nombre de una variable de miembro de clase.

En el siguiente ejemplo hipotético hemos implementado el método *draw*, que toma un parámetro de cadena opcional *backgroundColor* con el nombre del color. Como el nombre del parámetro es el mismo que el del miembro de la clase *Shape*, el compilador emite el primer aviso «la definición de 'backgroundColor' oculta el campo».

La consecuencia del solapamiento es que la posterior asignación errónea del valor *clrBlue* funciona en el parámetro y no en el miembro de la clase, y como los tipos del valor y del parámetro no coinciden, el compilador emitirá un segundo aviso: «conversión implícita de número a cadena» (el número aquí es una constante *clrBlue*). Pero la línea *this.backgroundColor = clrBlue* escribe el valor en el campo del objeto.

```
void draw(string backgroundColor = NULL) //warning 1:
             // declaration of 'backgroundColor' hides member
{
   ...
   backgroundColor = clrBlue; // warning 2:
     // implicit conversion from 'number' to 'string'
   this.backgroundColor = clrBlue; // ok

   {
      bool backgroundColor = false; // warning 3:
          // declaration of 'backgroundColor' hides local variable
      ...
      this.backgroundColor = clrRed; // ok
   }
   ...
}
```

La definición posterior de la variable booleana local *backgroundColor* (en el bloque anidado de llaves) anula de nuevo las definiciones anteriores de ese nombre (por eso recibimos el tercer aviso). Sin embargo, al anular la referencia a *this*, la sentencia *this.backgroundColor = clrRed* también hace referencia a un campo de objeto.

Sin *this* especificado, el compilador siempre elige la definición de nombre más cercana (por contexto).

También se necesita otro tipo de *this*: para pasar el objeto actual como parámetro a otra función. En concreto, se adopta un enfoque en el que objetos de la misma clase son responsables de crear o eliminar objetos de otra clase, y el objeto subordinado debe conocer a su «jefe». Seguidamente se crean los objetos dependientes en la clase «jefe» utilizando el constructor, y se le pasa *this* del objeto «jefe». Esta técnica suele utilizar la asignación dinámica de objetos y punteros, y debido a ello se mostrará un ejemplo relevante en la sección punteros.

Otro uso común de *this* es devolver un puntero al objeto actual desde una función miembro, lo que le permite organizar las llamadas a funciones miembro en una cadena. Como todavía tenemos que estudiar los punteros en detalle, bastará con saber que un puntero a un objeto de alguna clase se describe añadiendo el carácter '*' al nombre de la clase, y que se puede trabajar con un objeto a través de un puntero de la misma forma que se haría directamente.

Por ejemplo, podemos proporcionar al usuario varios métodos para establecer las propiedades de una forma individualmente: cambiar color, mover horizontal o mover verticalmente. Cada uno de ellos devolverá un puntero al objeto actual.

```
Shape *setColor(const color c)
{
   backgroundColor = c;
   return &this;
}

Shape *moveX(const int x)
{
   coordinates.x += x;
   return &this;
}

Shape *moveY(const int y)
{
   coordinates.y += y;
   return &this;
}
```

A continuación es posible organizar convenientemente las llamadas a estos métodos en una cadena.

```
Shape s;
s.setColor(clrWhite).moveX(80).moveY(-50);
```

Cuando hay muchas propiedades en una clase, este enfoque le permite configurar un objeto de forma compacta y selectiva.

En la sección Definición de clases intentamos registrar una variable de objeto, pero descubrimos que podíamos utilizar su nombre con sólo un ampersand (en una llamada a *Print*) para obtener un puntero o, de hecho, un número único (manejador o handle). En el contexto de un objeto, el mismo manejador está disponible a través de *&this*.

A efectos de depuración, puede identificar los objetos por su descriptor. Vamos a explorar la herencia de clases, y cuando hay más de una, la identificación nos resultará muy útil. Por ello, en todos los constructores y destructores, añadimos (y añadiremos en el futuro en las clases derivadas) la siguiente llamada a *Print*:

```
   ~Shape()
   {
      Print(__FUNCSIG__, " ", &this);
   }
```

Ahora todos los pasos de creación y eliminación se marcarán en el registro con el nombre de la clase y el número de objeto.

Implementamos constructores y destructores similares en la estructura *Pair*; sin embargo, en las estructuras, por desgracia, no se admiten los punteros, es decir, escribir *&this* es imposible. Por tanto, sólo podemos identificarlos por su contenido (en este caso, por sus coordenadas):

```
struct Pair
{
   int x, y;
   Pair(int a, int b): x(a), y(b)
   {
      Print(__FUNCSIG__, " ", x, " ", y);
   }
   ...
};
```

3.2.11 Herencia

Al definir una clase, un desarrollador puede heredarla de otra clase, incorporando así el concepto de herencia. Para ello, el nombre de la clase va seguido de un signo de dos puntos, un modificador opcional de derechos de acceso (una de las palabras clave *public*, *protected*, *private*) y el nombre de la clase padre. Por ejemplo, así es como podemos definir una clase *Rectangle* que derive de *Shape*:

```
class Rectangle : public Shape
{
};
```

Los modificadores de acceso de la cabecera de la clase controlan la «visibilidad» de los miembros de la clase padre incluidos en la clase hijo:

- *public* : todos los miembros heredados conservan sus derechos y limitaciones.
- *protected* : cambia los derechos de los miembros *public* heredados a *protected*
- p*rivate*: hace que todos los miembros heredados sean privados (*private*)

El modificador *public* se utiliza en la gran mayoría de las definiciones. Las otras dos opciones sólo tienen sentido en casos excepcionales, ya que infringen el principio básico de la herencia: los objetos de una clase derivada deben ser representantes de pleno derecho de la familia padre, y si «truncamos» sus derechos, pierden parte de sus características. Las estructuras también pueden heredarse entre sí de forma similar. Está prohibido heredar clases de estructuras o estructuras de clases.

> A diferencia de C++, MQL5 no es compatible con la herencia múltiple. Una clase puede tener como máximo un padre.

Un objeto de clase derivada tiene integrado un objeto de clase base. Teniendo en cuenta que la clase base puede, a su vez, ser heredada de alguna otra clase padre, el objeto creado puede compararse con muñecas matrioskas anidadas unas dentro de otras.

En la nueva clase necesitamos un constructor que rellene los campos del objeto de la misma forma que se hizo en la clase base.

```
class Rectangle : public Shape
{
public:
   Rectangle(int px, int py, color back) :
      Shape(px, py, back)
   {
      Print(__FUNCSIG__, " ", &this);
   }
};
```

En este caso, la lista de inicialización se ha convertido en una única llamada al constructor *Shape*. No se pueden establecer directamente variables de clase base en una lista de inicialización, ya que el constructor base es responsable de inicializarlas. No obstante, si fuera necesario, podríamos cambiar los campos *protected* de la clase base desde el cuerpo del constructor *Rectangle* (las sentencias del cuerpo de la función se ejecutan después de que el constructor base haya completado su llamada en la lista de inicialización).

El rectángulo tiene dos dimensiones, así que vamos a añadirlas como campos protegidos *dx* y *dy*. Para establecer sus valores es necesario completar la lista de parámetros del constructor.

```
class Rectangle : public Shape
{
protected:
   int dx, dy; // dimensions (width, height)

public:
   Rectangle(int px, int py, int sx, int sy, color back) :
      Shape(px, py, back), dx(sx), dy(sy)
   {
   }
};
```

Es importante señalar que los objetos *Rectangle* contienen implícitamente la función *toString* heredada de *Shape* (sin embargo, *draw* también está ahí, aunque sigue estando vacía). Por lo tanto, el siguiente código es correcto:

```
void OnStart()
{
   Rectangle r(100, 200, 50, 75, clrBlue);
   Print(r.toString());
};
```

Esto demuestra no sólo la llamada a *toString*, sino también la creación de un objeto rectángulo utilizando nuestro nuevo constructor.

No hay constructor por defecto (sin parámetros) en la clase *Rectangle*. Esto significa que el usuario de la clase no puede crear objetos rectángulo de forma sencilla, sin argumentos:

```
   Rectangle r; // 'Rectangle' - wrong parameters count
```

El compilador mostrará un error «Número de argumentos no válido».

Vamos a crear otra clase hija: *Ellipse*. Por ahora, no se diferenciará en nada de *Rectangle*, salvo en el nombre. Más adelante presentaremos las diferencias entre ellas.

```
class Ellipse : public Shape
{
protected:
   int dx, dy; // dimensions (large and small radii)
public:
   Ellipse(int px, int py, int rx, int ry, color back) :
      Shape(px, py, back), dx(rx), dy(ry)
   {
      Print(__FUNCSIG__, " ", &this);
   }
};
```

A medida que aumenta el número de clases, sería estupendo mostrar el nombre de la clase en el método *toString*. En la sección Operadores especiales sizeof y typename hemos descrito el operador *typename*. Vamos a intentar usarlo.

Recuerde que *typename* espera un parámetro, para el que se devuelve el nombre del tipo. Por ejemplo, si creamos un par de objetos *s* y *r* de las clases *Shape* y *Rectangle*, respectivamente, podemos averiguar su tipo de la siguiente manera:

```
void OnStart()
{
   Shape s;
   Rectangle r(100, 200, 75, 50, clrRed);
   Print(typename(s), " ", typename(r));      // Shape Rectangle
}
```

Pero necesitamos obtener este nombre dentro de la clase de alguna manera. Para ello, vamos a añadir un parámetro de cadena al constructor paramétrico *Shape* y a almacenarlo en un nuevo campo de cadena *type* (preste atención a la sección *protected* y al modificador *const*: este campo está oculto al mundo exterior y no puede editarse una vez creado el objeto):

```
class Shape
{
protected:
   ...
   const string type;

public:
   Shape(int px, int py, color back, string t) :
      coordinates(px, py),
      backgroundColor(back),
      type(t)
   {
      Print(__FUNCSIG__, " ", &this);
   }
   ...
};
```

En los constructores de clases derivadas, rellenamos este parámetro del constructor base utilizando *typename(this)*:

```
class Rectangle : public Shape
{
   ...
public:
   Rectangle(int px, int py, int sx, int sy, color back) :
      Shape(px, py, back, typename(this)), dx(sx), dy(sy)
   {
      Print(__FUNCSIG__, " ", &this);
   }
};
```

Ahora podemos mejorar el método *toString* utilizando el campo *type*.

```
class Shape
{
   ...
public:
   string toString() const
   {
      return type + " " + (string)coordinates.x + " " + (string)coordinates.y;
   }
};
```

Vamos a asegurarnos de que nuestra pequeña jerarquía de clases genera objetos según lo previsto e imprime entradas de registro de prueba cuando se llama a los constructores y destructores.

```
void OnStart()
{
   Shape s;
   //setting up an object by chaining calls via 'this'
   s.setColor(clrWhite).moveX(80).moveY(-50);
   Rectangle r(100, 200, 75, 50, clrBlue);
   Ellipse e(200, 300, 100, 150, clrRed);
   Print(s.toString());
   Print(r.toString());
   Print(e.toString());
}
```

Como resultado, obtenemos aproximadamente las siguientes entradas de registro (las líneas en blanco se añaden intencionadamente para separar la salida de diferentes objetos):

```
Pair::Pair(int,int) 0 0
Shape::Shape() 1048576

Pair::Pair(int,int) 100 200
Shape::Shape(int,int,color,string) 2097152
Rectangle::Rectangle(int,int,int,int,color) 2097152

Pair::Pair(int,int) 200 300
Shape::Shape(int,int,color,string) 3145728
Ellipse::Ellipse(int,int,int,int,color) 3145728

Shape 80 -50
Rectangle 100 200
Ellipse 200 300

Ellipse::~Ellipse() 3145728
Shape::~Shape() 3145728
Pair::~Pair() 200 300

Rectangle::~Rectangle() 2097152
Shape::~Shape() 2097152
Pair::~Pair() 100 200

Shape::~Shape() 1048576
Pair::~Pair() 80 -50
```

El registro deja claro en qué orden se llama a los constructores y destructores.

Para cada objeto, en primer lugar se crean los campos de objeto descritos en él (si los hay) y, a continuación, se llama al constructor base y a todos los constructores de las clases derivadas a lo largo de la cadena de herencia. Si hay campos propios (añadidos) de algunos tipos de objetos en una clase derivada, los constructores para ellos serán invocados inmediatamente antes del constructor de esta clase derivada. Cuando hay varios campos de objeto, estos se crean en el orden en que se describen en la clase.

Los destructores se invocan exactamente en el orden inverso.

En las clases derivadas se pueden definir constructores de copia, de los que hablamos en Constructores: predeterminado, paramétrico y de copia. Para tipos de forma específicos, como un rectángulo, su sintaxis es similar:

```
class Rectangle : public Shape
{
   ...
   Rectangle(const Rectangle &other) :
      Shape(other), dx(other.dx), dy(other.dy)
   {
   }
   ...
};
```

El ámbito de aplicación se amplía ligeramente. Un objeto de clase derivada puede utilizarse para copiar a una clase base (porque la clase derivada contiene todos los datos de la clase base). Sin embargo, en este caso, por supuesto, se ignoran los campos añadidos en la clase derivada.

```
void OnStart()
{
   Rectangle r(100, 200, 75, 50, clrBlue);
   Shape s2(r);         // ok: copy derived to base

   Shape s;
   Rectangle r4(s);     // error: no one of the overloads can be applied
                        // requires explicit constructor overloading
}
```

Para copiar en la dirección opuesta es necesario proporcionar una versión del constructor con una referencia a la clase derivada en la clase base (lo que, en teoría, contradice los principios de la programación orientada a objetos); de lo contrario se producirá el error de compilación «No se puede aplicar ninguna de las sobrecargas a la llamada a la función».

Ahora podemos incluir en el script un par o más de variables de forma para luego «pedirles» que se dibujen a sí mismas utilizando el método *draw*.

```
void OnStart()
{
   Rectangle r(100, 200, 50, 75, clrBlue);
   Ellispe e(100, 200, 50, 75, clrGreen);
   r.draw();
   e.draw();
};
```

Sin embargo, una entrada de este tipo significa que el número de formas, sus tipos y los parámetros están «programados» en el programa, mientras que el usuario debería ser capaz de elegir qué y dónde dibujar. De ahí la necesidad de crear formas de forma dinámica.

3.2.12 Creación dinámica de objetos: nuevo y suprimir

Hasta ahora sólo hemos intentado crear objetos automáticos, es decir, variables locales dentro de *OnStart*. Un objeto declarado en el contexto global (fuera de *OnStart* o de alguna otra función) también se crearía automáticamente (cuando se carga el script) y se eliminaría (cuando se descarga el script).

Además de estos dos modos, hemos mencionado la posibilidad de describir un campo de un tipo de objeto (en nuestro ejemplo, se trata de la estructura *Pair* utilizada para el campo *coordinates* dentro del

objeto *Shape*). Todos estos objetos son también automáticos: los crea para nosotros un compilador en un constructor de un objeto «anfitrión» y los elimina en su destructor.

Sin embargo, a menudo es imposible arreglárselas sólo con objetos automáticos en los programas. En el caso de un programa de dibujo, necesitaremos crear formas a petición del usuario. Además, las formas tendrán que almacenarse en un array, y para ello los objetos automáticos tendrían que tener un constructor por defecto (cosa que no ocurre en nuestro caso).

Para estas situaciones, MQL5 ofrece la posibilidad de crear y eliminar objetos de forma dinámica. La creación se realiza con el operador *new* y la eliminación, con el operador *delete*.

Operador*new*

La palabra clave *new* va seguida del nombre de la clase requerida y, entre paréntesis, una lista de argumentos para llamar a cualquiera de los constructores existentes. La ejecución del operador *new* conduce a la creación de una instancia de la clase.

El operador *new* devuelve un valor de un tipo especial: un puntero a un objeto. Para describir una variable de este tipo, añada un asterisco '*' después del nombre de la clase. Por ejemplo:

```
Rectangle *pr = new Rectangle(100, 200, 50, 75, clrBlue);
```

Aquí la variable *pr* tiene un tipo de puntero a un objeto de la clase *Rectangle*. Los punteros se tratarán con más detalle en una sección aparte.

Es importante señalar que la declaración de una variable de tipo puntero de objeto en sí no asigna memoria para un objeto y no invoca a su constructor. Por supuesto, un puntero ocupa espacio (8 bytes), pero se trata de un entero sin signo *ulong* que el sistema interpreta de una manera especial.

Puede trabajar con un puntero del mismo modo que con un objeto, es decir, puede invocar los métodos disponibles mediante el operador de desreferenciación y acceder a los campos.

```
Print(pr.toString());
```

La variable de un puntero a la que todavía no se le ha asignado un descriptor de objeto dinámico (por ejemplo, si el operador *new* no se invoca en el momento de la inicialización de una nueva variable, sino que se traslada a algunas líneas posteriores del código fuente), contiene un puntero nulo especial, que se denota como NULL (para distinguirlo de los números) pero que en realidad es igual a 0.

Operador*delete*

Los punteros recibidos a través de *new* deben liberarse al final de un algoritmo utilizando el operador *delete*. Por ejemplo:

```
delete pr;
```

Si no se hace así, la instancia asignada por el operador *new* permanecerá en memoria. Si se crean cada vez más objetos nuevos de esta forma y luego no se borran cuando ya no se necesitan, se producirá un consumo innecesario de memoria. El resto de objetos dinámicos no liberados hacen que se impriman avisos cuando el programa termina. Por ejemplo, si no borra el puntero *pr*, obtendrá algo como esto en el registro una vez descargado el script:

```
1 undeleted object left
1 object of type Rectangle left
168 bytes of leaked memory
```

El terminal informa de cuántos objetos y de qué clase fueron olvidados por el programador, así como de cuánta memoria ocupaban.

Una vez que se llama al operador *delete* para un puntero, éste se invalida porque el objeto ya no existe. Un intento posterior de acceder a sus propiedades provoca un error en tiempo de ejecución "Puntero no válido accedido":

```
Critical error while running script 'shapes (EURUSD,H1)'.
Invalid pointer access.
```

El programa MQL se interrumpe.

Sin embargo, esto no significa que ya no se pueda utilizar la misma variable de puntero. Basta con asignar un puntero a otra instancia recién creada del objeto.

MQL5 tiene una función integrada que le permite comprobar la validez de un puntero en una variable; se trata de *CheckPointer*:

```
ENUM_POINTER_TYPE CheckPointer(object *pointer);
```

Toma un parámetro de un puntero a una clase de tipo y devuelve un valor de la enumeración ENUM_POINTER_TYPE:

- POINTER_INVALID: puntero incorrecto;
- POINTER_DYNAMIC: puntero válido a un objeto dinámico;
- POINTER_AUTOMATIC: puntero válido a un objeto automático.

La ejecución de la sentencia *delete* sólo tiene sentido para un puntero para el que la función haya devuelto POINTER_DYNAMIC. Para un objeto automático no tendrá ningún efecto (tales objetos se borran automáticamente cuando el control vuelve del bloque de código en el que se ha definido la variable).

La siguiente macro simplifica y garantiza la correcta limpieza de un puntero:

```
#define FREE(P) if(CheckPointer(P) == POINTER_DYNAMIC) delete (P)
```

La necesidad de «limpiar» explícitamente es un precio inevitable que hay que pagar por la flexibilidad que proporcionan los objetos dinámicos y los punteros.

3.2.13 Punteros

Como dijimos en la sección Definición de clases, los punteros en MQL5 son ciertos descriptores (números únicos) de objetos, y no direcciones en la memoria, como en C++. Para un objeto automático, obtenemos un puntero anteponiendo un ampersand delante de su nombre (en este contexto, el carácter ampersand es el operador «obtener dirección»). Así, en el siguiente ejemplo, la variable *p* apunta al objeto automático *s*.

```
Shape s;         // automatic object
Shape *p = &s;   // a pointer to the same object
s.draw();        // calling an object method
p.draw();        // doing the same
```

En las secciones anteriores aprendimos a obtener un puntero a un objeto como resultado de crearlo dinámicamente con *new*. En este momento no se necesita un ampersand para obtener un descriptor: el valor del puntero es el descriptor.

La API de MQL5 proporciona la función *GetPointer* que realiza la misma acción que el operador ampersand '&', es decir, devuelve un puntero a un objeto:

```
void *GetPointer(Class object);
```

Utilizar una u otra opción es cuestión de preferencia.

Los punteros se utilizan a menudo para enlazar objetos. Vamos a ilustrar la idea de crear objetos subordinados que reciben un puntero a *this* de su creador de objetos (*ThisCallback.mq5*). Ya mencionamos este truco en la sección sobre la palabra clave *this*.

Vamos a intentar usarlo para implementar un esquema para notificar al «creador» de vez en cuando el porcentaje de cálculos realizados en el objeto subordinado: hemos hecho su análogo usando el puntero de función. La clase *Manager* controla los cálculos, y los cálculos en sí (muy probablemente, utilizando fórmulas diferentes) se realizan en clases separadas; en este ejemplo se muestra una de ellas, la clase *Element*.

```
class Manager; // preliminary announcement

class Element
{
   Manager *owner; // pointer

public:
   Element(Manager &t): owner(&t) { }

   void doMath()
   {
      const int N = 1000000;
      for(int i = 0; i < N; ++i)
      {
         if(i % (N / 20) == 0)
         {
            // we pass ourselves to the method of the control class
            owner.progressNotify(&this, i * 100.0f / N);
         }
         // ... massive calculations
      }
   }

   string getMyName() const
   {
      return typename(this);
   }
};

class Manager
{
   Element *elements[1]; // array of pointers (1 for demo)

public:
   Element *addElement()
   {
      // looking for an empty slot in the array
      // ...
      // passing to the constructor of the subclass
      elements[0] = new Element(this); // dynamic creation of an object
      return elements[0];
   }

   void progressNotify(Element *e, const float percent)
   {
      // Manager chooses how to notify the user:
      // display, print, send to the Internet
      Print(e.getMyName(), "=", percent);
   }
};
```

Un objeto subordinado puede utilizar el enlace recibido para notificar al «jefe» el progreso del trabajo. Al llegar al final del cálculo se envía una señal al objeto de control de que es posible borrar el objeto de calculadora, o dejar que trabaje otro. Por supuesto, el array fijo de un elemento de la clase *Manager* no tiene un aspecto muy impresionante, pero como demostración, sirve para entender el punto. El administrador no sólo gestiona la distribución de tareas informáticas, sino que también proporciona una capa abstracta para notificar al usuario: en lugar de enviar mensajes a un registro, puede escribirlos en un archivo independiente, mostrarlos en pantalla o enviarlos a Internet.

Por cierto: preste atención a la declaración preliminar de la clase *Manager* antes de la definición de la clase *Element*. Es necesario describir en la clase *Element* un puntero a la clase *Manager*, que se define a continuación en el código. Si se omite la declaración forward, obtendremos el error «'Manager': token inesperado, probablemente falta el tipo...».

La necesidad de una declaración forward surge cuando dos clases se refieren entre sí a través de sus miembros: en este caso, sea cual sea el orden en que dispongamos las clases, es imposible definir completamente ninguna de ellas. Una declaración forward permite reservar un nombre de tipo sin una definición completa.

> Una propiedad fundamental de los punteros es que un puntero a una clase base puede utilizarse para apuntar a un objeto de cualquier clase derivada. Esta es una de las manifestaciones de polimorfismo. Este comportamiento es posible porque los objetos derivados contienen «sub-objetos» integrados de clases progenitoras como matrioskas de muñecas anidadas.

En particular, para nuestra tarea con las formas, es fácil describir un array dinámico de punteros *Shape* y añadirle objetos de distintos tipos a petición del usuario.

El número de clases se ampliará a cinco (*Shapes2.mq5*). Además de *Rectangle* y *Ellipse*, vamos a añadir *Triangle* y a crear también una clase derivada de *Rectangle* para un cuadrado (*Square*), y una clase derivada de *Ellipse* para un círculo (*Circle*). Obviamente, un cuadrado es un rectángulo con lados iguales, y un círculo es una elipse de radios grande y pequeño iguales.

Para pasar un nombre de clase de cadena a lo largo de la cadena de herencia, vamos a añadir en las secciones *protected* de los constructores especiales de las clases *Rectangle* y *Ellipse* con un parámetro de cadena adicional *t*:

```
class Rectangle : public Shape
{
protected:
   Rectangle(int px, int py, int sx, int sy, color back, string t) :
      Shape(px, py, back, t), dx(sx), dy(sy)
   {
   }
   ...
};
```

Entonces, al crear un cuadrado, no sólo establecemos tamaños iguales para los lados, sino que también pasamos *typename(this)* de la clase *Square*:

```
class Square : public Rectangle
{
public:
   Square(int px, int py, int sx, color back) :
      Rectangle(px, py, sx, sx, back, typename(this))
   {
   }
};
```

Además, vamos a mover los constructores de la clase *Shape* a la sección *protected*. Esto prohibirá la creación del objeto *Shape* por sí mismo: sólo puede actuar como base para sus clases descendientes.

Vamos a asignar la función *addRandomShape* para generar formas, lo que devuelve un puntero a un objeto recién creado. Para fines de demostración, ahora implementará una generación aleatoria de formas: sus tipos, posiciones, tamaños y colores.

Los tipos de forma admitidos se resumen en la enumeración SHAPES: se corresponden con cinco clases implementadas.

La función *random* devuelve números aleatorios en un rango determinado (utiliza la función integrada *rand*, que devuelve un entero aleatorio en el rango de 0 a 32767 cada vez que es invocada. Los centros de las formas se generan en el rango de 0 a 500 píxeles, y los tamaños de las formas están en el rango de hasta 200. El color está formado por tres componentes RGB (véase la sección Color), cada uno de los cuales oscila entre 0 y 255.

```
int random(int range)
{
   return (int)(rand() / 32767.0 * range);
}

Shape *addRandomShape()
{
   enum SHAPES
   {
      RECTANGLE,
      ELLIPSE,
      TRIANGLE,
      SQUARE,
      CIRCLE,
      NUMBER_OF_SHAPES
   };

   SHAPES type = (SHAPES)random(NUMBER_OF_SHAPES);
   int cx = random(500), cy = random(500), dx = random(200), dy = random(200);
   color clr = (color)((random(256) << 16) | (random(256) << 8) | random(256));
   switch(type)
   {
      case RECTANGLE:
         return new Rectangle(cx, cy, dx, dy, clr);
      case ELLIPSE:
         return new Ellipse(cx, cy, dx, dy, clr);
      case TRIANGLE:
         return new Triangle(cx, cy, dx, clr);
      case SQUARE:
         return new Square(cx, cy, dx, clr);
      case CIRCLE:
         return new Circle(cx, cy, dx, clr);
   }
   return NULL;
}

void OnStart()
{
   Shape *shapes[];

   // simulate the creation of arbitrary shapes by the user
   ArrayResize(shapes, 10);
   for(int i = 0; i < 10; ++i)
   {
      shapes[i] = addRandomShape();
   }

   // processing shapes: for now, just output to the log
   for(int i = 0; i < 10; ++i)
   {
      Print(i, ": ", shapes[i].toString());
```

```
        delete shapes[i];
    }
}
```

Generamos 10 formas y las imprimimos en el registro (el resultado puede variar debido a la aleatoriedad de la elección de tipos y propiedades). No olvide borrar los objetos con *delete* porque fueron creados dinámicamente (aquí esto se hace en el mismo bucle porque las formas no se usan más; en un programa real, lo más probable es que el array de formas se almacene de alguna manera en un archivo para luego cargarla y seguir trabajando con una imagen).

```
0: Ellipse 241 38
1: Rectangle 10 420
2: Circle 186 38
3: Triangle 27 225
4: Circle 271 193
5: Circle 293 57
6: Rectangle 71 424
7: Square 477 46
8: Square 366 27
9: Ellipse 489 105
```

Las formas se crean con éxito e informan sobre sus propiedades.

Ya estamos listos para acceder a la API de nuestras clases, es decir, al método *draw*.

3.2.14 Métodos virtuales (virtual y override)

Las clases están pensadas para describir interfaces de programación externas y proporcionar su implementación interna. Dado que la funcionalidad de nuestro programa de prueba es dibujar varias formas, hemos descrito varias variables en la clase *Shape* y sus descendientes para una futura implementación, y también hemos reservado el método *draw* para la interfaz.

En la clase base *Shape* no debe ni puede hacer nada porque *Shape* no es una forma concreta: convertiremos *Shape* en una clase abstracta más adelante (hablaremos más sobre interfaces y clases abstractas más adelante).

Vamos a redefinir el método *draw* en las clases *Rectangle*, *Ellipse* y otras derivadas (*Shapes3.mq5*). Ello implica copiar el método y modificar su contenido en consonancia. Aunque muchos se refieren a este proceso como «overriding», vamos a distinguir entre los dos términos, reservando «overriding» exclusivamente para los métodos virtuales, que se abordarán más adelante.

En sentido estricto, redefinir un método sólo requiere que el nombre del método coincida. Sin embargo, para garantizar un uso coherente en todo el código, es esencial mantener la misma lista de parámetros y el mismo tipo de retorno.

```
class Rectangle : public Shape
{
   ...
   void draw()
   {
      Print("Drawing rectangle");
   }
};
```

Como aún no sabemos cómo dibujar en la pantalla, simplemente enviaremos el mensaje al registro.

Es importante señalar que, al proporcionar una nueva implementación del método en la clase derivada, obtenemos 2 versiones del método: una hace referencia al objeto base integrado (*Shape* interno) y la otra, al derivado (*Rectangle* externo).

La primera será invocada para una variable de tipo *Shape* y la segunda, para una variable de tipo *Rectangle*.

En una cadena de herencia más larga, un método puede redefinirse y propagarse todavía más veces.

Puede cambiar el tipo de acceso de un nuevo método; por ejemplo, hacerlo público si estaba protegido, o viceversa. Pero en este caso hemos dejado el método *draw* en la sección pública.

Si es necesario, el programador puede llamar a la implementación del método de cualquiera de las clases progenitoras: para ello, se utiliza un operador de resolución de contexto especial: dos signos de dos puntos '::'. En concreto, podríamos invocar la implementación *draw* de la clase *Rectangle* desde el método *draw* de la clase *Square*: para ello, especificamos el nombre de la clase deseada, '::' y el nombre del método, como por ejemplo *Rectangle::draw()*. Llamar a *draw* sin especificar el contexto implica que es un método de la clase actual, y por tanto, si lo hace desde el propio método *draw*, obtendrá una recursión infinita y, en última instancia, un desbordamiento de pila y una caída del programa.

```
class Square : public Rectangle
{
public:
   ...
   void draw()
   {
      Rectangle::draw();
      Print("Drawing square");
   }
};
```

Entonces, al invocar *draw* en el objeto *Square* se registrarían dos líneas:

```
Square s(100, 200, 50, clrGreen);
s.draw(); // Drawing rectangle
          // Drawing square
```

Vincular un método a una clase en la cual está declarado proporciona el envío estático (o vinculación estática): el compilador decide qué método invocar en la fase de compilación y «programa» la coincidencia encontrada en el código binario.

Durante el proceso de decisión, el compilador busca el método que se va a invocar en el objeto de la clase para el que se realiza la desreferenciación ('.'). Si el método está presente, es invocado, y si no,

el compilador comprueba la clase progenitora para la presencia del método, y así sucesivamente, a lo largo de la cadena de herencia hasta que se encuentra el método. Si el método no se encuentra en ninguna de las clases de la cadena, se producirá un error de compilación «identificador no declarado».

En concreto, el siguiente código llama al método *setColor* en el objeto *Rectangle*:

```
Rectangle r(100, 200, 75, 50, clrBlue);
r.setColor(clrWhite);
```

No obstante, este método se define sólo en la clase base *Shape* y se integra una vez en todas las clases descendientes, por lo que se ejecutará aquí.

Vamos a intentar empezar a dibujar formas arbitrarias desde un array en la función *OnStart* (recuerde que hemos duplicado y modificado el método *draw* en todas las clases descendientes).

```
for(int i = 0; i < 10; ++i)
{
   shapes[i].draw();
}
```

Curiosamente, no sale nada en el registro. Esto sucede porque el programa llama al método *draw* de la clase *Shape*.

Aquí hay un gran inconveniente del envío estático: cuando usamos un puntero a una clase base para almacenar un objeto de una clase derivada, el compilador elige un método basándose en el tipo del puntero, no del objeto. El hecho es que, en la fase de compilación, aún no se sabe a qué objeto de clase apuntará durante la ejecución del programa.

Así, es necesario un enfoque más flexible: un envío dinámico (o vinculación), que aplazaría la elección de un método (de entre todas las versiones sobrescritas (overridden) del método en la cadena descendiente) hasta el tiempo de ejecución. La elección debe basarse en el análisis de la clase real del objeto en el puntero. Es la expedición dinámica la que proporciona el principio de polimorfismo.

Este enfoque se implementa en MQL5 utilizando métodos virtuales. En la descripción de un método de este tipo debe añadirse la palabra clave *virtual* al principio del encabezamiento.

Vamos a declarar el método *draw* de la clase *Shape* (*Shapes4.mq5*) como virtual. Esto hará automáticamente que todas las versiones de la misma en las clases derivadas también sean virtuales.

```
class Shape
{
   ...
   virtual void draw()
   {
   }
};
```

Una vez que un método se virtualiza, modificarlo en las clases derivadas se denomina sobrescritura (overriding) en lugar de redefinición. La sobrescritura requiere que el nombre, los tipos de parámetros y el valor de retorno del método coincidan (teniendo en cuenta la presencia o ausencia de modificadores *const*).

> Tenga en cuenta que sobrescribir funciones virtuales es diferente a la sobrecarga de funciones. La sobrecarga utiliza el mismo nombre de función, pero con parámetros diferentes (en concreto, vimos la posibilidad de sobrecargar un constructor en el ejemplo de las estructuras; véase Constructores y destructores), y la sobrescritura requiere la coincidencia completa de las firmas de las funciones.

Las funciones sobrescritas (overridden) deben definirse en clases diferentes que estén relacionadas por relaciones de herencia. Las funciones sobrecargadas deben pertenecer a la misma clase; de lo contrario, no se tratará de una sobrecarga sino, muy probablemente, de una redefinición (y funcionará de forma diferente; véase un análisis más detallado del ejemplo *OverrideVsOverload.mq5*).

Si ejecuta un nuevo script, en el registro aparecerán las líneas esperadas, señalando las llamadas a versiones específicas del método *draw* en cada una de las clases.

```
Drawing square
Drawing circle
Drawing triangle
Drawing ellipse
Drawing triangle
Drawing rectangle
Drawing square
Drawing triangle
Drawing square
Drawing triangle
```

En las clases derivadas en las que se sobrescribe (override) un método virtual, se recomienda añadir la palabra clave *override* a su cabecera (aunque no es obligatorio).

```
class Rectangle : public Shape
{
   ...
   void draw() override
   {
      Print("Drawing rectangle");
   }
};
```

Esto le permite al compilador saber que estamos sobrescribiendo el método a propósito. Si en el futuro la API de la clase base cambia repentinamente y el método sobrescrito (overridden) deja de ser virtual (o simplemente se elimina), el compilador generará un mensaje de error: «el método se declara con el especificador 'override', pero no sobrescribe ningún método de la clase base». Tenga en cuenta que hasta la adición o eliminación del modificador *const* de un método cambia su firma, y la sobrescritura (overriding) puede romperse debido a ello.

La palabra clave *virtual* antes de un método anulado también está permitida, pero no es obligatoria.

Para que el envío dinámico funcione, el compilador genera una tabla de funciones virtuales para cada clase. Se añade un campo implícito a cada objeto con un enlace a la tabla dada de su clase. El compilador rellena la tabla basándose en la información sobre todos los métodos virtuales y sus versiones sobrescritas (overridden) a lo largo de la cadena de herencia de una clase concreta.

La llamada a un método virtual se codifica en la imagen binaria del programa de una forma especial: primero se consulta la tabla en busca de una versión para una clase de un objeto concreto (situada en el puntero) y, a continuación, se pasa a la función adecuada.

Como resultado, el envío dinámico es más lento que el estático.

En MQL5, las clases siempre contienen una tabla de funciones virtuales, independientemente de la presencia de métodos virtuales.

Si un método virtual devuelve un puntero a una clase, cuando se sobrescribe, es posible cambiar (hacerlo más específico, altamente especializado) el tipo de objeto del valor devuelto. En otras palabras: el tipo del puntero no sólo puede ser el mismo que en la declaración inicial del método virtual, sino también cualquiera de sus sucesores. Estos tipos se denominan «covariantes» o intercambiables.

Por ejemplo, si hiciéramos virtual el método *setColor* en la clase *Shape*,

```
class Shape
{
   ...
   virtual Shape *setColor(const color c)
   {
      backgroundColor = c;
      return &this;
   }
   ...
};
```

podríamos sobrescribirlo (override) en la clase *Rectangle* de esta manera (sólo a modo de demostración de la tecnología):

```
class Rectangle : public Shape
{
   ...
   virtual Rectangle *setColor(const color c) override
   {
      // call original method
      // (by pre-lightening the color,
      // no matter what for)
      Rectangle::setColor(c | 0x808080);
      return &this;
   }
};
```

Observe que el tipo de retorno es un puntero a *Rectangle* en lugar de *Shape*.

Tiene sentido utilizar un truco similar si la versión sobrescrita (overridden) del método cambia algo en esa parte del objeto que no pertenece a la clase base, de modo que el objeto, de hecho, ya no corresponde al estado permitido (invariante) de la clase base.

Nuestro ejemplo con formas de dibujo está casi listo; queda por llenar los métodos virtuales *draw* con contenidos reales. Lo haremos en el capítulo Gráficos (véase el ejemplo *ObjectShapesDraw.mq5*), pero lo mejoraremos después de estudiar los recursos gráficos.

> Teniendo en cuenta el concepto de herencia, el procedimiento por el que el compilador elige el método adecuado parece un poco confuso. A partir del nombre del método y de la lista específica de argumentos (sus tipos) en la instrucción de llamada, se compila una lista de todos los métodos candidatos disponibles.
>
> Para los métodos no virtuales, al principio sólo se analizan los métodos de la clase actual. Si ninguno de ellos coincide, el compilador continuará buscando en la clase base (y luego en los ancestros más lejanos hasta que encuentre una coincidencia). Si entre los métodos de la clase actual hay alguno adecuado (incluso si es necesaria la conversión implícita de los tipos de argumento), será elegido. Si la clase base tuviera un método con tipos de argumento más

apropiados (sin conversión o con menos conversiones), el compilador seguiría sin llegar a él. En otras palabras: los métodos no virtuales se analizan partiendo de la clase del objeto actual hacia los ancestros, hasta llegar a la primera coincidencia que «funcione».

En el caso de los métodos virtuales, el compilador busca primero el método requerido por su nombre en la clase del puntero y, a continuación, selecciona la implementación en la tabla de funciones virtuales de la clase que tiene más instancias (descendiente más lejana) en la que este método se sobrescribe (override) en la cadena entre el tipo de puntero y el tipo de objeto. En este caso se puede utilizar también la conversión implícita de argumentos si no hay coincidencia exacta entre los tipos de los argumentos.

Veamos el siguiente ejemplo (*OverrideVsOverload.mq5*). Hay 4 clases encadenadas: *Base*, *Derived*, *Concrete* y *Special*. Todas ellas contienen métodos con argumentos de tipo *int* y *float*. En la función *OnStart*, las variables entera *i* y real *f* se utilizan como argumentos para todas las llamadas a métodos.

```
class Base
{
public:
   void nonvirtual(float v)
   {
      Print(__FUNCSIG__, " ", v);
   }
   virtual void process(float v)
   {
      Print(__FUNCSIG__, " ", v);
   }
};

class Derived : public Base
{
public:
   void nonvirtual(int v)
   {
      Print(__FUNCSIG__, " ", v);
   }
   virtual void process(int v) // override
   // error: 'Derived::process' method is declared with 'override' specifier,
   // but does not override any base class method
   {
      Print(__FUNCSIG__, " ", v);
   }
};

class Concrete : public Derived
{
};

class Special : public Concrete
{
public:
   virtual void process(int v) override
   {
      Print(__FUNCSIG__, " ", v);
   }
   virtual void process(float v) override
   {
      Print(__FUNCSIG__, " ", v);
   }
};
```

En primer lugar, creamos un objeto de la clase *Concrete* y un puntero al mismo *Base *ptr*. Luego invocamos métodos virtuales y no virtuales para ellos. En la segunda parte, los métodos del objeto *Special* se invocan por medio de los punteros de clase *Base* y *Derived*.

```
void OnStart()
{
   float f = 2.0;
   int i = 1;

   Concrete c;
   Base *ptr = &c;

   // Static link tests

   ptr.nonvirtual(i); // Base::nonvirtual(float), conversion int -> float
   c.nonvirtual(i);   // Derived::nonvirtual(int)

   // warning: deprecated behavior, hidden method calling
   c.nonvirtual(f);   // Base::nonvirtual(float), because
                      // method selection ended in Base,
                      // Derived::nonvirtual(int) does not suit to f

   // Dynamic link tests

   // attention: there is no method Base::process(int), also
   // there are no process(float) overrides in classes up to and including Concrete
   ptr.process(i);    // Base::process(float), conversion int -> float
   c.process(i);      // Derived::process(int), because
                      // there is no override in Concrete,
                      // and the override in Special does not count

   Special s;
   ptr = &s;
   // attention: there is no method Base::process(int) in ptr
   ptr.process(i);    // Special::process(float), conversion int -> float
   ptr.process(f);    // Special::process(float)

   Derived *d = &s;
   d.process(i);      // Special::process(int)

   // warning: deprecated behavior, hidden method calling
   d.process(f);      // Special::process(float)
}
```

A continuación se muestra la salida del registro.

```
void Base::nonvirtual(float) 1.0
void Derived::nonvirtual(int) 1
void Base::nonvirtual(float) 2.0
void Base::process(float) 1.0
void Derived::process(int) 1
void Special::process(float) 1.0
void Special::process(float) 2.0
void Special::process(int) 1
void Special::process(float) 2.0
```

La llamada a *ptr.nonvirtual(i)* se realiza mediante enlace estático, y el número entero *i* se convierte de forma preliminar al tipo de parámetro, *float*.

La llamada *c.nonvirtual(i)* también es estática, y como no hay ningún método *void nonvirtual(int)* en la clase *Concrete*, el compilador encuentra dicho método en la clase padre *Derived*.

Invocar la función del mismo nombre sobre el mismo objeto con un valor del tipo *float* lleva al compilador al método *Base::nonvirtual(float)*, ya que *Derived::nonvirtual(int)* no es adecuado (la conversión provocaría una pérdida de precisión). Por el camino, el compilador emite un aviso «comportamiento obsoleto, llamada a método oculto».

Los métodos sobrecargados pueden parecerse a los sobreescritos (overridden) (tienen el mismo nombre pero parámetros diferentes) pero son diferentes porque se encuentran en clases diferentes. Cuando un método de una clase derivada sobrescribe (override) un método de una clase progenitora, sustituye el comportamiento del método de la clase progenitora, lo que a veces puede tener efectos inesperados. El programador podría esperar que el compilador eligiera otro método adecuado (como en la sobrecarga), pero en lugar de ello se invoca la subclase.

Para evitar posibles avisos, si la implementación de la clase progenitora es necesaria, debe escribirse exactamente como la misma función en la clase derivada, y la clase base debe ser invocada desde ella.

```
class Derived : public Base
{
public:
   ...
   // this override will suppress the warning
   // "deprecated behavior, hidden method calling"
   void nonvirtual(float v)
   {
      Base::nonvirtual(v);
      Print(__FUNCSIG__, " ", v);
   }
...
```

Volvamos a las pruebas en *OnStart*.

La llamada a *ptr.process(i)* demuestra la confusión entre sobrecarga y sobrescritura (override) descrita anteriormente. La clase *Base* tiene un método virtual *process(float)* y la clase *Derived* añade un nuevo método virtual *process(int),* que no es sobrescritura (overriding) en este caso porque los tipos de parámetros son diferentes. El compilador selecciona un método por su nombre en la clase base y comprueba en la tabla de funciones virtuales si hay sobrescrituras (overrides) en la cadena de herencia hasta la clase *Concrete* (incluida; ésta es la clase objeto por puntero). Como no se encontró ninguna sobrescritura (override), el compilador tomó *Base::process(float)* y aplicó la conversión de tipo del argumento al parámetro (*int* a *float*).

Si siguiéramos la regla de escribir siempre la palabra *override*, que lleva implícita la redefinición, y la añadiéramos a *Derived*, obtendríamos un error:

```
class Derived : public Base
{
   ...
   virtual void process(int v) override // error!
   {
      Print(__FUNCSIG__, " ", v);
   }
};
```

El compilador notificará «Método 'Derived::process' declarado con el especificador 'override', pero no sobrescribe (override) ningún método de la clase base». Esto serviría como pista para solucionar el problema.

La llamada a *process(i)* en el objeto *Concrete* se realiza con *Derived::process(int)*. Aunque tenemos una redefinición aún más extensa en la clase *Special*, es irrelevante porque se hace en la cadena de herencia después de la clase *Concrete*.

Cuando el puntero se asigna posteriormente al objeto *ptr*, las llamadas a *Special* y *process(i)* son resueltas por el compilador como *process(f)* porque *Special::process(float)* anula a *Special*. La elección del parámetro *float* se produce por la misma razón descrita anteriormente: el método *Base::process(float)* es sobrescrito (overridden) por *Special*.

Si aplicamos el puntero *d* de tipo *Derived*, obtendremos finalmente la llamada esperada *Special::process(int)* para la cadena *d.process(i)*. La cuestión es que *process(int)* está definido en *Derived*, y entra dentro del ámbito de búsqueda del compilador.

Observe que la clase *Special* no sólo sobrescribe (override) los métodos virtuales heredados, sino que también sobrecarga dos métodos en la clase en sí.

> No llame a una función virtual desde un constructor o destructor Aunque técnicamente es posible, el comportamiento virtual en el constructor y destructor se pierde por completo y podría obtener resultados inesperados. Deben evitarse no sólo las llamadas explícitas, sino también las indirectas (por ejemplo, cuando se llama a un método simple desde un constructor, que a su vez llama a uno virtual).
>
> Analicemos la situación con más detalle utilizando el ejemplo de un constructor. El hecho es que, en el momento en que funciona el constructor, el objeto aún no está completamente ensamblado a lo largo de toda la cadena de herencia, sino sólo hasta la clase actual. Todavía hay que «terminar» toda la parte derivada alrededor del núcleo existente. Por lo tanto, todas las sobrescrituras (overrides) de métodos virtuales posteriores (si las hay) aún no están disponibles en este punto. Como resultado, la versión actual del método será invocada desde el constructor.

3.2.15 Miembros estáticos

Hasta ahora, hemos considerado los campos y métodos de una clase que describen el estado y el comportamiento de los objetos de una clase determinada. Sin embargo, en los programas puede ser necesario almacenar ciertos atributos o realizar operaciones en toda la clase, en lugar de en sus objetos. Estas propiedades de clase se denominan estáticas y se describen utilizando la palabra clave *static* añadida antes del tipo. También se admiten en estructuras y uniones.

Por ejemplo, podemos contar el número de formas creadas por el usuario en un programa de dibujo. Para ello, en la clase *Shape* describiremos la variable estática *count* (*Shapes5.mq5*).

```
class Shape
{
private:
   static int count;

protected:
   ...
   Shape(int px, int py, color back, string t) :
      coordinates(px, py),
      backgroundColor(back),
      type(t)
   {
      ++count;
   }

public:
   ...
   static int getCount()
   {
      return count;
   }
};
```

Se define en la sección *private* y, por tanto, no es accesible desde el exterior.

Para leer el valor actual del contador se proporciona un método estático público *getCount()*. En teoría, dado que los miembros estáticos se definen en el contexto de una clase, reciben restricciones de visibilidad según el modificador de la sección en la que se encuentran.

Incrementaremos el contador en 1 en el constructor paramétrico *Shape* y eliminaremos el constructor por defecto. Así, se tendrá en cuenta cada instancia de una forma de cualquier tipo derivado.

Tenga en cuenta que una variable estática debe definirse explícitamente (y, opcionalmente, inicializarse) fuera del bloque de clase:

```
static int Shape::count = 0;
```

Las variables estáticas de clase son similares a las variables globales y a las variables estáticas dentro de funciones (véase la sección Variables estáticas) en el sentido de que se crean cuando se inicia el programa y se borran antes de que se descargue. Por lo tanto, a diferencia de las variables de objeto, deben existir desde el principio como una única instancia.

En este caso, la inicialización a cero puede omitirse porque, como sabemos, las variables globales y estáticas se ponen a cero por defecto. Las matrices también pueden ser estáticas.

En la definición de una variable estática vemos el uso del operador especial de selección de contexto '::'. Con él se forma un nombre de variable totalmente cualificado. A la izquierda de '::' está el nombre de la clase a la que pertenece la variable, y a la derecha su identificador. Obviamente, el nombre completamente cualificado es necesario, porque dentro de clases diferentes se pueden declarar variables estáticas con el mismo identificador, y se necesita una forma de referirse de forma única a cada una de ellas.

El mismo operador '::' se utiliza para acceder no sólo a las variables estáticas públicas de la clase, sino también a los métodos. En concreto, para llamar al método *getCount* en la función *OnStart*, utilizamos la sintaxis *Shape::getCount()*:

```
void OnStart()
{
   for(int i = 0; i < 10; ++i)
   {
      Shape *shape = addRandomShape();
      shape.draw();
      delete shape;
   }

   Print(Shape::getCount()); // 10
}
```

Como ahora se está generando el número de formas especificado (10), podemos comprobar que el contador funciona correctamente.

Si tiene un objeto de clase, puede hacer referencia a un método o propiedad estáticos mediante la desreferenciación habitual (por ejemplo, *shape.getCount()*), pero dicha notación puede ser engañosa (ya que oculta el hecho de que en realidad no se accede al objeto).

Tenga en cuenta que la creación de clases derivadas no afecta en modo alguno a las variables y métodos estáticos: estos siempre se asignan a la clase en la que se definieron. Nuestro contador es el mismo para todas las clases de formas derivadas de *Shape*.

No se puede utilizar *this* dentro de métodos estáticos porque se ejecutan sin estar vinculados a un objeto específico. Además, desde un método estático no se puede invocar directamente un método de clase normal o acceder a su campo sin desreferenciar ninguna variable de tipo objeto. Por ejemplo, si llama a *draw* desde *getCount*, obtendrá un error «acceso a función o miembro no estático»:

```
   static int getCount()
   {
      draw(); // error: 'draw' - access to non-static member or function
      return count;
   }
```

Por la misma razón, los métodos estáticos no pueden ser virtuales.

¿Es posible, utilizando variables estáticas, calcular, no el número total de formas, sino sus estadísticas por tipo? Sí, es posible. Esta tarea se deja para su estudio por separado. Los interesados pueden encontrar uno de los ejemplos de aplicación en el script *Shapes5stats.mq5*.

3.2.16 Tipos anidados, espacios de nombres y operador de contexto '::'

Las clases, estructuras y uniones pueden describirse no sólo en el contexto global, sino también dentro de otra clase o estructura. Es más: la definición puede hacerse dentro de la función. Esto permite describir todas las entidades necesarias para el funcionamiento de cualquier clase o estructura dentro del contexto adecuado y evitar así posibles conflictos de nombres.

En concreto, en el programa de dibujo, la estructura para almacenar las coordenadas *Pair* se ha definido hasta ahora de forma global. A medida que el programa crezca, es muy posible que se necesite

otra entidad llamada *Pair* (sobre todo, teniendo en cuenta el nombre, más bien genérico). Por lo tanto, es conveniente trasladar la descripción de la estructura al interior de la clase *Shape* (*Shapes6.mq5*).

```
class Shape
{
public:
   struct Pair
   {
      int x, y;
      Pair(int a, int b): x(a), y(b) { }
   };
   ...
};
```

Las descripciones anidadas tienen permisos de acceso de acuerdo con los modificadores de sección especificados. En este caso, hemos puesto el nombre *Pair* a disposición del público. Dentro de la clase *Shape*, el manejo del tipo de estructura *Pair* no cambia en modo alguno debido a la transferencia. No obstante, en el código externo debe especificar un nombre completo que incluya el nombre de la clase externa (contexto), el operador de selección de contexto '::' y el propio identificador de entidad interna. Por ejemplo, para describir una variable con un par de coordenadas, tendría que escribir:

```
Shape::Pair coordinates(0, 0);
```

El nivel de anidamiento al describir entidades no está limitado, por lo que un nombre totalmente cualificado puede contener identificadores de varios niveles (contextos) separados por '::'. Por ejemplo, podríamos envolver todas las clases de dibujo dentro de la clase externa *Drawing*, en la sección *public*.

```
class Drawing
{
public:
   class Shape
   {
   public:
      struct Pair
      {
         ...
      };
   };
   class Rectangle : public Shape
   {
      ...
   };
   ...
};
```

Entonces, los nombres de tipos totalmente cualificados (por ejemplo, para su uso en *OnStart* u otras funciones externas) se alargarían:

```
Drawing::Shape::Rect coordinates(0, 0);
Drawing::Rectangle rect(200, 100, 70, 50, clrBlue);
```

Por un lado, esto es una inconveniencia, pero por otro, a veces es una necesidad en proyectos grandes con un amplio número de clases. En nuestro pequeño proyecto, este enfoque se utiliza únicamente para demostrar la viabilidad técnica.

Para combinar clases y estructuras relacionadas lógicamente en grupos con nombre, MQL5 proporciona una forma más sencilla que incluirlas en una clase envoltorio «vacía».

Un espacio de nombres se declara utilizando la palabra clave *namespace* seguida del nombre y un bloque de llaves que incluye todas las definiciones necesarias. Este es el aspecto del mismo programa de pintura utilizando *namespace*:

```
namespace Drawing
{
   class Shape
   {
   public:
      struct Pair
      {
         ...
      };
   };
   class Rectangle : public Shape
   {
      ...
   };
   ...
}
```

Existen dos diferencias principales: el contenido interno del espacio está siempre disponible públicamente (los modificadores de acceso no son aplicables en él) y no hay punto y coma después de la llave de cierre.

Añadamos el método *move* a la clase *Shape*, que toma como parámetro la estructura *Pair*:

```
class Shape
{
public:
   ...
   Shape *move(const Pair &pair)
   {
      coordinates.x += pair.x;
      coordinates.y += pair.y;
      return &this;
   }
};
```

A continuación, en la función *OnStart*, puede organizar el desplazamiento de todas las formas en un valor dado invocando esta función:

```
void OnStart()
{
   //draw a random set of shapes
   for(int i = 0; i < 10; ++i)
   {
      Drawing::Shape *shape = addRandomShape();
      // move all shapes
      shape.move(Drawing::Shape::Pair(100, 100));
      shape.draw();
      delete shape;
   }
}
```

Tenga en cuenta que los tipos *Shape* y *Pair* deben describirse con nombres completos: *Drawing::Shape* y *Drawing::Shape::Pair*, respectivamente.

Puede haber varios bloques con el mismo nombre de espacio: todos sus contenidos entrarán en un contexto lógicamente unificado con el nombre especificado.

Los identificadores definidos en el contexto global, en particular todas las funciones integradas de la API de MQL5, también están disponibles a través del operador de selección de contexto no precedido de ninguna notación. Por ejemplo, este es el aspecto que podría tener una llamada a la función *Print*:

```
::Print("Done!");
```

Cuando la llamada se realiza desde cualquier función definida en el contexto global, no es necesaria dicha entrada.

La necesidad puede manifestarse dentro de cualquier clase o estructura si en ellas se define un elemento del mismo nombre (función, variable o constante). Por ejemplo, añadamos el método *Print* a la clase *Shape*:

```
   static void Print(string x)
   {
      // empty
      // (likely will output it to a separate log file later)
   }
```

Dado que las implementaciones de prueba del método *draw* en las clases derivadas llaman a *Print*, ahora se redirigen a este método *Print*: entre varios identificadores idénticos, el compilador elige el que está definido en un contexto más cercano. En este caso, la definición en la clase base está más cerca de las formas que del contexto global. Como resultado, se suprimirá la salida de registro de las clases de forma.

No obstante, llamar a *Print* desde la función *OnStart* sigue funcionando (porque está fuera del contexto de la clase *Shape*).

```
void OnStart()
{
   ...
   Print("Done!");
}
```

Para «arreglar» la impresión de depuración en las clases es necesario preceder todas las llamadas a *Print* con un operador global de selección de contexto:

```
class Rectangle : public Shape
{
   ...
   void draw() override
   {
      ::Print("Drawing rectangle"); // reprint via global Print(...)
   }
};
```

3.2.17 Dividir definición y declaración de clase

En los grandes proyectos de software es conveniente separar las clases en una breve descripción (declaración) y una definición, que incluye los principales detalles de implementación. En algunos casos, dicha separación se hace necesaria si las clases se refieren de alguna manera entre sí, es decir, ninguna puede definirse completamente sin declaraciones previas.

Hemos visto un ejemplo de declaración forward en la sección Indicadores (véase el archivo *ThisCallback.mq5*), donde las clases *Manager* y *Element* contienen punteros recíprocos. Allí, la clase se predeclaró de forma abreviada; es decir, forma de un encabezado con la palabra clave *class* y un nombre:

```
class Manager;
```

No obstante, esta es la declaración más breve posible. Registra sólo el nombre y permite posponer la descripción de la interfaz de programación hasta un momento dado, pero esta descripción debe encontrarse en algún punto posterior del código.

La mayoría de las veces, la declaración incluye una descripción completa de la interfaz: especifica todas las variables y cabeceras de métodos de la clase, pero sin sus cuerpos (bloques de código).

Las definiciones de métodos se escriben por separado: con cabeceras que utilizan nombres totalmente cualificados que incluyen el nombre de la clase (o varias clases y espacios de nombres si el contexto del método está muy anidado). Los nombres de todas las clases y el nombre del método se concatenan utilizando el operador de selección contextual '::'.

```
type class_name [:: nested_class_name...] :: method_name([parameters...])
{
}
```

En teoría, puede definir parte de los métodos directamente en el bloque de descripción de la clase (normalmente lo hacen con las funciones pequeñas), y algunos pueden sacarse por separado (por regla general, las funciones grandes). Pero un método debe tener una sola definición (es decir, no se puede definir un método en un bloque de clase y luego otra vez por separado) y una declaración (una definición en un bloque de clase es también una declaración).

La lista de parámetros, el tipo de devolución y los modificadores *const* (si existen) deben coincidir exactamente en la definición y declaración del método.

Veamos cómo podemos separar la descripción y definición de clases del script *ThisCallback.mq5* (ejemplo de la sección Punteros): vamos a crear su análogo con el nombre *ThisCallback2.mq5*.

La predeclaración *Manager* seguirá estando al principio. Además, ambas clases *Element* y *Manager* se declaran sin implementación: en lugar de un bloque de código con un cuerpo de método, hay un punto y coma.

```
class Manager; // preliminary announcement

class Element
{
   Manager *owner; // pointer
public:
   Element(Manager &t);
   void doMath();
   string getMyName() const;
};

class Manager
{
   Element *elements[1]; // array of pointers (replace with dynamic)
public:
   ~Manager();
   Element *addElement();
   void progressNotify(Element *e, const float percent);
};
```

La segunda parte del código fuente contiene las implementaciones de todos los métodos (las implementaciones en sí no se modifican).

```
Element::Element(Manager &t) : owner(&t)
{
}

void Element::doMath()
{
   ...
}

string Element::getMyName() const
{
   return typename(this);
}

Manager::~Manager()
{
   ...
}

Element *Manager::addElement()
{
   ...
}

void Manager::progressNotify(Element *e, const float percent)
{
   ...
}
```

Las estructuras también admiten definiciones y declaraciones de métodos independientes.

Tenga en cuenta que la lista de inicialización del constructor (después del nombre y ':') forma parte de la definición y, por tanto, debe preceder al cuerpo de la función (en otras palabras, la lista de inicialización no está permitida en una declaración de constructor en la que sólo esté presente la cabecera).

La redacción por separado de la declaración y la definición permite el desarrollo de bibliotecas cuyo código fuente debe estar cerrado. En este caso, las declaraciones se colocan en un archivo de cabecera independiente con la extensión *mqh*, mientras que las definiciones se colocan en un archivo del mismo nombre con la extensión *mq5*. El programa se compila y distribuye como un archivo ex5 con un archivo de cabecera que describe la interfaz externa.

En este caso puede plantearse la pregunta de por qué parte de la implementación interna, en concreto la organización de los datos (variables), es visible en la interfaz externa. En sentido estricto, esto indica un nivel insuficiente de abstracción en la jerarquía de clases. Todas las clases que proporcionan una interfaz externa no deben exponer ningún detalle de implementación.

En otras palabras: si nos marcamos el objetivo de exportar las clases anteriores de una determinada biblioteca, entonces tendríamos que separar sus métodos en clases base que proporcionaran una descripción de la API (sin campos de datos), y *Manager* y *Element* heredarían de ellas. Al mismo tiempo, en los métodos de las clases base, no podemos utilizar ningún dato de las clases derivadas y, en general, no pueden tener ninguna implementación. ¿Cómo es esto posible?

Para ello, existe una tecnología de métodos abstractos, clases abstractas e interfaces.

3.2.18 Clases abstractas e interfaces

Para explorar las clases abstractas y las interfaces vamos a volver a nuestro ejemplo de programa de dibujo de extremo a extremo. Para simplificar, su API consiste en un único método *draw* virtual. Hasta ahora, ha estado vacía, pero al mismo tiempo, incluso una implementación tan vacía es una implementación concreta. Sin embargo, los objetos de la clase *Shape* no se pueden dibujar: su forma no está definida. Por lo tanto, tiene sentido hacer que el método *draw* sea abstracto o, como también se denomina, puramente virtual.

Para ello, hay que eliminar el bloque con una implementación vacía y añadir «= 0» a la cabecera del método:

```
class Shape
{
public:
   virtual void draw() = 0;
   ...
```

Una clase que tiene al menos un método abstracto también se convierte en abstracta, ya que su objeto no se puede crear: no hay implementación. En concreto, nuestro constructor *Shape* estaba disponible para las clases derivadas (gracias al modificador *protected*), y sus desarrolladores podían, hipotéticamente, crear un objeto *Shape*. Pero era así antes, y después de la declaración del método abstracto detuvimos este comportamiento, ya que lo habíamos prohibido nosotros, los autores de la interfaz de dibujo. El compilador arrojará un error:

```
'Shape' -cannot instantiate abstract class
      'void Shape::draw()' is abstract
```

El mejor enfoque para describir una interfaz es crear una clase abstracta para ella que contenga sólo métodos abstractos. En nuestro caso, el método *draw* debería trasladarse a la nueva clase *Drawable*, y la clase *Shape* debería heredarse de ella (*Shapes.mq5*).

```
class Drawable
{
public:
   virtual void draw() = 0;
};

class Shape : public Drawable
{
public:
   ...
   // virtual void draw() = 0; // moved to base class
   ...
};
```

Por supuesto, los métodos de interfaz deben estar en la sección *public*.

MQL5 proporciona otra manera conveniente de describir las interfaces mediante el uso de la palabra clave *interface*. Todos los métodos de una interfaz se declaran sin implementación y se consideran

públicos y virtuales. La descripción de la interfaz *Drawable* que equivale a la clase anterior tiene el siguiente aspecto:

```
interface Drawable
{
   void draw();
};
```

En este caso no es necesario cambiar nada en las clases descendientes si no hubo campos en la clase abstracta (lo que supondría una violación del principio de abstracción).

Ahora es el momento de ampliar la interfaz y hacer que el trío de métodos *setColor*, *moveX*, *moveY*también formen parte de ella.

```
interface Drawable
{
   void draw();
   Drawable *setColor(const color c);
   Drawable *moveX(const int x);
   Drawable *moveY(const int y);
};
```

Tenga en cuenta que los métodos devuelven un objeto *Drawable* porque no sé nada de *Shape*. En la clase *Shape* tenemos ya implementaciones que son adecuadas para sobrescribir (override) estos métodos, porque *Shape* hereda de *Drawable* (*Shape* «son una especie de» objetos *Drawable*).

Ahora, los desarrolladores externos pueden añadir otras familias de clases *Drawable* al programa de dibujo, en concreto, no sólo formas, sino también texto, mapas de bits y también, sorprendentemente, recopilaciones de otros *Drawable*s, lo que permite anidar objetos unos dentro de otros y realizar composiciones complejas. Basta con heredar de la interfaz e implementar sus métodos.

```
class Text : public Drawable
{
public:
   Text(const string label)
   {
      ...
   }

   void draw()
   {
      ...
   }

   Text *setColor(const color c)
   {
      ...
      return &this;
   }
   ...
};
```

Si las clases de forma se distribuyeran como una biblioteca binaria ex5 (sin códigos fuente), proporcionaríamos un archivo de cabecera que sólo contuviera la descripción de la interfaz y ninguna pista sobre las estructuras de datos internas.

Dado que las funciones virtuales se vinculan dinámicamente (más tarde) a un objeto durante la ejecución del programa, es posible obtener un error fatal «Llamada a función virtual pura»: el programa termina. Esto ocurre si el programador «olvidó» sin darse cuenta proporcionar una implementación. El compilador no siempre es capaz de detectar tales omisiones en tiempo de compilación.

3.2.19 Sobrecarga de operadores

En el capítulo Expresiones descubrimos varias operaciones definidas para tipos integrados. Por ejemplo, para variables del tipo *double*, podríamos evaluar la siguiente expresión:

```
double a = 2.0, b = 3.0, c = 5.0;
double d = a * b + c;
```

Sería conveniente utilizar una sintaxis similar cuando se trabaja con tipos definidos por el usuario, como las matrices:

```
Matrix a(3, 3), b(3, 3), c(3, 3); // creating 3x3 matrices
// ... somehow fill in a, b, c
Matrix d = a * b + c;
```

MQL5 ofrece esta oportunidad gracias a la sobrecarga de operadores.

Esta técnica se organiza describiendo los métodos con un nombre que comienza por la palabra clave *operator* e incluyendo a continuación un símbolo (o secuencia de símbolos) de una de las operaciones admitidas. De forma generalizada, esto puede representarse de la siguiente manera:

```
result_type operator@ ( [type parameter_name] );
```

Aquí @, símbolo(s) de la operación.

La lista completa de operaciones en MQL5 se ha proporcionado en la sección Prioridades de las operaciones; no obstante, no todas ellas pueden sobrecargarse.

Prohibidas para sobrecarga:

- dos puntos '::', resolución de contexto;
- paréntesis '()', «llamada a función» o «agrupación»;
- punto '.', «desreferenciación»;
- ampersand '&', «dirección», operador unario (no obstante, el ampersand está disponible como operador binario «a nivel de bits AND»);
- ternario condicional '?:';
- coma ','.

Todos los demás operadores están disponibles para sobrecarga. Las prioridades de los operadores de sobrecarga no pueden cambiarse, siguen siendo iguales a la precedencia estándar, por lo que debe utilizarse la agrupación con paréntesis si es necesario.

No se puede crear una sobrecarga para un carácter nuevo que no esté incluido en la lista estándar.

Todos los operadores se sobrecargan teniendo en cuenta su condición de unarios o binarios, es decir, se preserva el número de operandos requeridos. Como cualquier método de clase, la sobrecarga de operadores puede devolver un valor de algún tipo. En este caso, el tipo mismo debe elegirse en función de la lógica prevista de utilización del resultado de la función en las expresiones (véase más adelante).

Los métodos de sobrecarga de operadores tienen la siguiente forma (en lugar del símbolo '@', se sustituye el símbolo o símbolos del operador requerido):

Nombre	Cabecera del método	Uso en una expresión	Función es equivalente a
prefijo unario	tipo operator@()	@object	object.operator@()
postfijo unario	tipo operator@(int)	object@	object.operator@(0)
binario	tipo operator@(tipo parameter_name)	object@argument	object.operator@(argument)
índice	tipo operator[](tipo index_name)	object[argument]	object.operator[](argument)

Los operadores unarios no admiten parámetros. De los operadores unarios, sólo los operadores de incremento '++' y decremento '--' admiten la forma postfija además de la forma prefija; todos los demás operadores unarios sólo admiten la forma prefija. Especificar un parámetro anónimo de tipo *int* se utiliza para denotar la forma postfija (para distinguirla de la forma prefija), pero el parámetro en sí se ignora.

Los operadores binarios deben tomar un parámetro. Para el mismo operador son posibles varias variantes sobrecargadas con un parámetro de tipo diferente, incluido el mismo tipo que la clase del objeto actual. En este caso, objetos como los parámetros sólo pueden pasarse por referencia o por puntero (esto último es sólo para objetos de clase, pero no para estructuras).

Los operadores sobrecargados pueden utilizarse tanto a través de la sintaxis de las operaciones como parte de las expresiones (que es la razón principal de la sobrecarga) como de la sintaxis de las llamadas a métodos; ambas opciones se muestran en la tabla anterior. El equivalente funcional hace más evidente que, técnicamente hablando, un operador no es más que una llamada a un método sobre un

objeto, con el objeto a la derecha del operador prefijo y a la izquierda del símbolo para todos los demás. Al método del operador binario se le pasará como argumento el valor o expresión que esté a la derecha del operador (puede ser, en particular, otro objeto o variable de un tipo integrado).

De ello se deduce que los operadores sobrecargados no tienen la propiedad conmutativa: *a@b* no es generalmente igual a *b@a*, porque para *a*, el operador @ puede estar sobrecargado, pero *b* no lo está. Además, si *b* es una variable o un valor de tipo integrado, entonces, en principio, no se puede sobrecargar el comportamiento estándar para ella.

Como primer ejemplo, consideremos la clase *Fibo* para generar números a partir de la serie de Fibonacci (ya hemos realizado una implementación de esta tarea utilizando funciones, véase Definición de funciones). En la clase proporcionaremos 2 campos para almacenar el número actual y anterior de la fila: *current* y *previous*, respectivamente. El constructor por defecto los inicializará con los valores 1 y 0. También proporcionaremos un constructor de copia (*FiboMonad.mq5*).

```
class Fibo
{
   int previous;
   int current;
public:
   Fibo() : current(1), previous(0) { }
   Fibo(const Fibo &other) : current(other.current), previous(other.previous) { }
   ...
};
```

El estado inicial del objeto: el número actual es 1, y el anterior es 0. Para encontrar el siguiente número de la serie, sobrecargamos los operadores de incremento prefijo y postfijo.

```
   Fibo *operator++() // prefix
   {
      int temp = current;
      current = current + previous;
      previous = temp;
      return &this;
   }

   Fibo operator++(int) // postfix
   {
      Fibo temp = this;
      ++this;
      return temp;
   }
```

Tenga en cuenta que el método prefijo no devuelve un puntero al objeto actual *Fibo* una vez modificado el número, sino que el método postfijo vuelve a un nuevo objeto con el contador anterior guardado, lo que corresponde a los principios del incremento postfijo.

Si es necesario, el programador, por supuesto, puede sobrecargar cualquier operación de forma arbitraria. Por ejemplo, es posible calcular el producto, enviar el número al registro o hacer algo más en la implementación del incremento. No obstante, se recomienda ceñirse al enfoque en el que la sobrecarga de operadores realiza acciones intuitivas.

Implementamos las operaciones de decremento de forma similar: devolverán el número anterior de la serie.

```
Fibo *operator--() // prefix
{
   int diff = current - previous;
   current = previous;
   previous = diff;
   return &this;
}

Fibo operator--(int) // postfix
{
   Fibo temp = this;
   --this;
   return temp;
}
```

Para obtener un número de una serie por un número dado, sobrecargaremos la operación de acceso al índice.

```
Fibo *operator[](int index)
{
   current = 1;
   previous = 0;
   for(int i = 0; i < index; ++i)
   {
      ++this;
   }
   return &this;
}
```

Para obtener el número actual contenido en la variable actual, vamos a sobrecargar el operador '~' (ya que rara vez se utiliza).

```
int operator~() const
{
   return current;
}
```

Sin esta sobrecarga, seguiría siendo necesario implementar algún método público para leer el campo privado *current*. Utilizaremos este operador para obtener números con *Print*.

También debe sobrecargar la asignación para mayor comodidad.

```
   Fibo *operator=(const Fibo &other)
   {
      current = other.current;
      previous = other.previous;
      return &this;
   }

   Fibo *operator=(const Fibo *other)
   {
      current = other.current;
      previous = other.previous;
      return &this;
   }
```

Veamos cómo funciona todo.

```
void OnStart()
{
   Fibo f1, f2, f3, f4;
   for(int i = 0; i < 10; ++i, ++f1) // prefix increment
   {
      f4 = f3++; // postfix increment and assignment overloading
   }

   // compare all values obtained by increments and by index [10]
   Print(~f1, " ", ~f2[10], " ", ~f3, " ", ~f4); // 89 89 89 55

   // counting in opposite direction, down to 0
   Fibo f0;
   Fibo f = f0[10]; // copy constructor (due to initialization)
   for(int i = 0; i < 10; ++i)
   {
      // prefix decrement
      Print(~--f); // 55, 34, 21, 13, 8, 5, 3, 2, 1, 1
   }
}
```

Los resultados son los esperados. Aun así, hay que tener en cuenta un detalle:

```
   Fibo f5;
   Fibo *pf5 = &f5;

   f5 = f4;   // call Fibo *operator=(const Fibo &other)
   f5 = &f4;  // call Fibo *operator=(const Fibo *other)
   pf5 = &f4; // calls nothing, assigns &f4 to pf5!
```

> La sobrecarga del operador de asignación para un puntero sólo funciona cuando se accede a través de un objeto. Si el acceso se realiza a través de un puntero, se produce una asignación estándar de un puntero a otro.

El tipo de devolución de un operador sobrecargado puede ser uno de los tipos integrados, un tipo de objeto (de una clase o estructura) o un puntero (sólo para objetos de clase).

Para devolver un objeto (una instancia, no una referencia), la clase debe implementar un constructor de copia. De esta forma se producirá una duplicación de instancias, lo que puede afectar a la eficiencia del código. Si es posible, debe devolver un puntero.

Sin embargo, cuando se devuelve un puntero, hay que asegurarse de que no se está devolviendo un objeto local automático (que se borrará cuando salga la función, y el puntero dejará de ser válido), sino alguno ya existente; por regla general, se devuelve *&this*.

Devolver un objeto o un puntero a un objeto permite «enviar» el resultado de un operador sobrecargado a otro, y construir así expresiones complejas del mismo modo que estamos acostumbrados a hacerlo con los tipos integrados. Si devuelve *void* será imposible utilizar el operador en expresiones. Por ejemplo, si el operador '=' se define con el tipo *void*, la asignación múltiple dejará de funcionar:

```
Type x, y, z = 1; // constructors and initialization of variables of a certain class
x = y = z; // assignments, compilation error
```

La cadena de asignación va de derecha a izquierda, y $y = z$ devolverá vacío.

Si los objetos sólo contienen campos de tipos integrados (incluidos los arrays), entonces no es necesario redefinir el operador de asignación o copia '=' de objetos de la misma clase: MQL5 proporciona copia «uno a uno» de todos los campos por defecto. El operador de asignación o copia no debe confundirse con el constructor de copia y la inicialización.

Pasemos ahora al segundo ejemplo: trabajar con matrices (*Matrix.mq5*).

> Tenga en cuenta, por cierto, que los tipos de objeto integrados matrices y vectores han aparecido recientemente en MQL5. Utilizar los tipos integrados o los suyos propios (o tal vez combinarlos) es decisión de cada desarrollador. La implementación rápida y lista para usar de muchos métodos populares en tipos integrados resulta cómoda y elimina la codificación rutinaria. Por otro lado, las clases personalizadas le permiten adaptar los algoritmos a sus tareas. Aquí proporcionamos la clase *Matrix* a modo de tutorial.

En la clase matriz, almacenaremos sus elementos en un array dinámico unidimensional *m*. En los tamaños, seleccione las variables *rows* y *columns*.

```
class Matrix
{
protected:
   double m[];
   int rows;
   int columns;
   void assign(const int r, const int c, const double v)
   {
      m[r * columns + c] = v;
   }

public:
   Matrix(const Matrix &other) : rows(other.rows), columns(other.columns)
   {
      ArrayCopy(m, other.m);
   }

   Matrix(const int r, const int c) : rows(r), columns(c)
   {
      ArrayResize(m, rows * columns);
      ArrayInitialize(m, 0);
   }
```

El constructor principal toma dos parámetros (dimensiones de la matriz) y asigna memoria para el array. También hay un constructor de copia de la otra matriz *other*. Aquí y más adelante, las funciones integradas para trabajar con arrays se utilizan de forma masiva (en particular, *ArrayCopy*, *ArrayResize*, *ArrayInitialize*); se abordarán en otro capítulo.

Organizamos el llenado de elementos desde un array externo sobrecargando el operador de asignación:

```
   Matrix *operator=(const double &a[])
   {
      if(ArraySize(a) == ArraySize(m))
      {
         ArrayCopy(m, a);
      }
      return &this;
   }
```

Para implementar la suma de dos matrices, sobrecargamos las operaciones '+=' y '+':

```
Matrix *operator+=(const Matrix &other)
{
   for(int i = 0; i < rows * columns; ++i)
   {
      m[i] += other.m[i];
   }
   return &this;
}

Matrix operator+(const Matrix &other) const
{
   Matrix temp(this);
   return temp += other;
}
```

Tenga en cuenta que el operador '+=' devuelve un puntero al objeto actual después de que este haya sido modificado, mientras que el operador '+' devuelve una nueva instancia por valor (se utilizará el constructor de copia), y el propio operador tiene el modificador *const*, por lo que el cómo no cambia el objeto actual.

El operador '+' es esencialmente un envoltorio que delega todo el trabajo al operador '+=', habiendo creado previamente una copia temporal de la matriz actual bajo el nombre *temp* para llamarla. Así, *temp* se añade a *other* mediante una llamada interna al operador '+=' (modificándose *temp*) y luego se devuelve como resultado del ' +'.

La multiplicación de matrices se sobrecarga de forma similar, con dos operadores '*=' y '*'.

```
Matrix *operator*=(const Matrix &other)
{
   // multiplication condition: this.columns == other.rows
  // the result will be a matrix of size this.rows by other.columns
   Matrix temp(rows, other.columns);

   for(int r = 0; r < temp.rows; ++r)
   {
      for(int c = 0; c < temp.columns; ++c)
      {
         double t = 0;
         //we add up the pairwise products of the i-th elements
        // row 'r' of the current matrix and column 'c' of the matrix other
         for(int i = 0; i < columns; ++i)
         {
            t += m[r * columns + i] * other.m[i * other.columns + c];
         }
         temp.assign(r, c, t);
      }
   }
   // copy the result to the current object of the matrix this
   this = temp; // calling an overloaded assignment operator
   return &this;
}

Matrix operator*(const Matrix &other) const
{
   Matrix temp(this);
   return temp *= other;
}
```

Ahora, multiplicamos la matriz por un número:

```
Matrix *operator*=(const double v)
{
   for(int i = 0; i < ArraySize(m); ++i)
   {
      m[i] *= v;
   }
   return &this;
}

Matrix operator*(const double v) const
{
   Matrix temp(this);
   return temp *= v;
}
```

Para comparar dos matrices, disponemos de los operadores '==' y '!=':

```
bool operator==(const Matrix &other) const
{
   return ArrayCompare(m, other.m) == 0;
}

bool operator!=(const Matrix &other) const
{
   return !(this == other);
}
```

A efectos de depuración, implementamos la salida del array de la matriz al registro.

```
void print() const
{
   ArrayPrint(m);
}
```

Además de las sobrecargas descritas, la clase *Matrix* dispone adicionalmente de una sobrecarga del operador []: devuelve un objeto de la clase anidada *MatrixRow*, es decir, una fila con un número dado.

```
MatrixRow operator[](int r)
{
   return MatrixRow(this, r);
}
```

La clase *MatrixRow* en sí proporciona un acceso más «profundo» a los elementos de la matriz sobrecargando el mismo operador [] (es decir, para una matriz, será posible especificar de forma natural dos índices *m[i][j]*).

```
class MatrixRow
{
protected:
   const Matrix *owner;
   const int row;

public:
   class MatrixElement
   {
   protected:
      const MatrixRow *row;
      const int column;

   public:
      MatrixElement(const MatrixRow &mr, const int c) : row(&mr), column(c) { }
      MatrixElement(const MatrixElement &other) : row(other.row), column(other.col

      double operator~() const
      {
         return row.owner.m[row.row * row.owner.columns + column];
      }

      double operator=(const double v)
      {
         row.owner.m[row.row * row.owner.columns + column] = v;
         return v;
      }
   };

   MatrixRow(const Matrix &m, const int r) : owner(&m), row(r) { }
   MatrixRow(const MatrixRow &other) : owner(other.owner), row(other.row) { }

   MatrixElement operator[](int c)
   {
      return MatrixElement(this, c);
   }

   double operator[](uint c)
   {
      return owner.m[row * owner.columns + c];
   }
};
```

El operador [] para un parámetro de tipo *int* devuelve un objeto de clase *MatrixElement*, a través del cual se puede escribir un elemento específico en el array. Para leer un elemento se utiliza el operador [] con un parámetro de tipo *uint*. Esto parece un truco, pero es una limitación del lenguaje: las sobrecargas deben diferir en el tipo de parámetro. Como alternativa a la lectura de un elemento, la clase *MatrixElement* proporciona una sobrecarga del operador '~'.

Cuando se trabaja con matrices, a menudo se necesita una matriz de identidad, así que vamos a crear una clase derivada para ello:

```
class MatrixIdentity : public Matrix
{
public:
   MatrixIdentity(const int n) : Matrix(n, n)
   {
      for(int i = 0; i < n; ++i)
      {
         m[i * rows + i] = 1;
      }
   }
};
```

Ahora vamos a probar las expresiones matriciales en acción.

```
void OnStart()
{
   Matrix m(2, 3), n(3, 2); // description
   MatrixIdentity p(2);     // identity matrix

   double ma[] = {-1,  0, -3,
                   4, -5,  6};
   double na[] = {7,  8,
                  9,  1,
                  2,  3};
   m = ma; // filling in data
   n = na;

   //we can read and write elements separately
   m[0][0] = m[0][(uint)0] + 2; // variant 1
   m[0][1] = ~m[0][1] + 2;      // variant 2

   Matrix r = m * n + p;                    // expression
   Matrix r2 = m.operator*(n).operator+(p); // equivalent
   Print(r == r2); // true

   m.print(); // 1.00000  2.00000 -3.00000  4.00000 -5.00000  6.00000
   n.print(); // 7.00000 8.00000 9.00000 1.00000 2.00000 3.00000
   r.print(); // 20.00000  1.00000 -5.00000  46.00000
}
```

Aquí hemos creado 2 matrices de 3 por 2 y 2 por 3 dimensiones, respectivamente; luego las hemos rellenado con valores de los arrays y hemos editado el elemento selectivo utilizando la sintaxis de dos índices [][]. Por último, calculamos la expresión *m * n + p*, donde todos los operandos son matrices. En la línea siguiente se muestra la misma expresión en forma de llamadas a métodos. Tenemos los mismos resultados.

> A diferencia de C++, MQL5 no admite la sobrecarga de operadores a nivel global. En MQL5, un operador sólo puede sobrecargarse en el contexto de una clase o estructura, es decir, utilizando su método. Además, MQL5 no admite la sobrecarga de conversión de tipos, operadores *new* y *delete*.

3.2.20 Conversión de tipos de objeto: dynamic_cast y puntero void *

Los tipos de objeto tienen reglas de conversión específicas que se aplican cuando los tipos de variable de origen y destino no coinciden. Las reglas para los tipos integrados ya se han abordado en el capítulo 2.6 Conversión de tipos. Los detalles específicos de la conversión de tipos de estructuras al copiar se describen en la sección Herencia y disposición de estructuras.

Tanto para las estructuras como para las clases, la condición principal para que pueda admitirse la conversión de tipos es que estén relacionadas a lo largo de la cadena de herencia. Los tipos de diferentes ramas de la jerarquía o no relacionados en absoluto no pueden convertirse entre sí.

Las reglas de conversión son diferentes para los objetos (valores) y los punteros.

Objetos

Un objeto de un tipo A puede asignarse a un objeto de otro tipo B si este último tiene un constructor que toma un parámetro de tipo A (con variaciones en valor, referencia o puntero, pero normalmente de la forma *B(const A &a)*). Un constructor de este tipo también se denomina constructor de conversión.

En ausencia de tal constructor explícito, el compilador intentará utilizar un operador de copia implícito, es decir, *B::operator=(const B &b)*, mientras que las clases A y B deben estar en la misma cadena de herencia para que la copia implícita funcione. Conversión de A a B. Si A se hereda de B (incluso no de forma directa, sino indirecta), entonces las propiedades añadidas a A desaparecerán cuando se copien a B. Si B se hereda de A, entonces sólo se copiará la parte de las propiedades que están en A. Estas conversiones no suelen ser bien recibidas.

Además, es posible que el compilador no siempre proporcione el operador de copia implícito. En concreto, si la clase tiene campos con el modificador *const*, la copia se considera prohibida (véase más adelante).

En el script *ShapesCasting.mq5* utilizamos la jerarquía de la clase Shape para demostrar las conversiones de tipo de objeto. En la clase *Shape*, el campo *type* se hace constante de forma deliberada, por lo que un intento de convertir (asignar) un objeto *Square* a un objeto *Rectangle* termina con un compilador de errores con explicaciones detalladas:

```
attempting to reference deleted function 'void Rectangle::operator=(const Rectangl
   function 'void Rectangle::operator=(const Rectangle&)' was implicitly deleted
   because it invokes deleted function 'void Shape::operator=(const Shape&)'
function 'void Shape::operator=(const Shape&)' was implicitly deleted
   because member 'type' has 'const' modifier
```

Según este mensaje, el método de copia *Rectangle::operator=(const Rectangle&)* fue eliminado implícitamente por el compilador (que proporciona su implementación por defecto) porque utiliza un método similar en la clase base *Shape::operator =(const Shape&)*, que a su vez fue eliminado debido a la presencia del campo *type* con el modificador *const*. Tales campos sólo se pueden establecer cuando se crea el objeto, y el compilador no sabe cómo copiar el objeto bajo tal restricción.

Por cierto: el efecto de «borrar» métodos está disponible no sólo para el compilador, sino también para el programador de la aplicación: se hablará más de esto en la sección Control de la herencia: final y supresión.

El problema podría resolverse eliminando el modificador *const* o proporcionando su propia implementación del operador de asignación (en ella, el campo *const* no interviene y guardará el contenido con una descripción del tipo «Rectángulo»):

```
Rectangle *operator=(const Rectangle &r)
{
   coordinates.x = r.coordinates.x;
   coordinates.y = r.coordinates.y;
   backgroundColor = r.backgroundColor;
   dx = r.dx;
   dy = r.dy;
   return &this;
}
```

Observe que esta definición devuelve un puntero al objeto actual, mientras que la implementación por defecto generada por el compilador era del tipo *void* (como se ve en el mensaje de error). Esto significa que los operadores de asignación por defecto proporcionados por el compilador no pueden utilizarse en la cadena $x = y = z$. Si necesita esta capacidad, sobrescriba *operator=* explícitamente y devuelva el tipo deseado distinto de *void*.

Punteros

Lo más práctico es convertir punteros a objetos de distintos tipos.

En teoría, todas las opciones para convertir punteros de tipo objeto pueden reducirse a tres:

- De base a derivado, la conversión de tipos hacia abajo (downcast), ya que es habitual dibujar una jerarquía de clases con un árbol invertido;
- De derivado a base, la conversión de tipos ascendente (upcast);
- Entre clases de diferentes ramas de la jerarquía o incluso de diferentes familias.

La última opción está prohibida (obtendremos un error de compilación). El compilador permite las dos primeras, pero si «upcast» es natural y segura, «downcast» puede provocar errores en tiempo de ejecución.

```
void OnStart()
{
   Rectangle *r = addRandomShape(Shape::SHAPES::RECTANGLE);
   Square *s = addRandomShape(Shape::SHAPES::SQUARE);
   Circle *c = NULL;
   Shape *p;
   Rectangle *r2;

   // OK
   p = c;   // Circle -> Shape
   p = s;   // Square -> Shape
   p = r;   // Rectangle -> Shape
   r2 = p;  // Shape -> Rectangle
   ...
};
```

Por supuesto, cuando se utiliza un puntero a un objeto de la clase base, no se puede llamar a métodos y propiedades de la clase derivada, aun cuando el objeto correspondiente se encuentre en el puntero. Obtendremos un error de compilación «identificador no declarado».

Sin embargo, la sintaxis de la conversión explícita está admitida para los punteros (véase estilo C), lo que permite la conversión «sobre la marcha» de un puntero al tipo requerido en expresiones y su desreferenciación sin crear una variable intermedia.

```
Base *b;
Derived d;
b = &d;
((Derived *)b).derivedMethod();
```

Aquí hemos creado un objeto de clase derivada (*Derived*) y un puntero de tipo base al mismo (*Base **). Para acceder al método *derivedMethod* de una clase derivada, el puntero se convierte temporalmente al tipo *Derived*.

Un tipo de puntero asterisco debe ir entre paréntesis. Además, la propia expresión de conversión, incluido el nombre de la variable, también está rodeada de otro par de paréntesis.

Otro error de compilación («type mismatch»: «falta de correspondencia de tipos») en nuestra prueba genera una línea en la que intentamos convertir un puntero a *Rectangle* en un puntero a *Circle*: proceden de ramas de herencia diferentes.

```
   c = r; // error: type mismatch
```

Las cosas son mucho peores cuando el tipo del puntero que se desea convertir no coincide con el objeto real (aunque sus tipos son compatibles, y por lo tanto el programa compila bien). Una operación de este tipo terminará con un error ya en la fase de ejecución del programa (es decir, el compilador no podrá detectarlo). A continuación, se descarga el programa.

Por ejemplo, en el script *ShapesCasting.mq5* hemos descrito un puntero a *Square* y le hemos asignado un puntero a *Shape*, que contiene el objeto *Rectangle*.

```
Square *s2;
// RUNTIME ERROR
s2 = p; // error: Incorrect casting of pointers
```

El terminal devuelve el error «conversión incorrecta de punteros». El puntero de tipo más específico *Square* no es capaz de apuntar al objeto progenitor *Rectangle*.

Para evitar problemas en tiempo de ejecución e impedir que el programa se bloquee, MQL5 proporciona un operador especial: *dynamic_cast*. Con esta construcción, puede comprobar «cuidadosamente» si es posible convertir un puntero en el tipo requerido. Si la conversión es posible, se llevará a efecto. Si no, obtendremos un puntero nulo (NULL) y podremos procesarlo de forma especial (por ejemplo, utilizando *if* para inicializar o interrumpir de alguna forma la ejecución de la función, pero no de todo el programa).

La sintaxis de *dynamic_cast* es la siguiente:

```
dynamic_cast< Class * >( pointer)
```

En nuestro caso, basta con escribir:

```
s2 = dynamic_cast<Square *>(p); // trying to cast type, and will get NULL if unsuc
Print(s2); // 0
```

El programa se ejecutará según lo esperado.

En concreto, podemos volver a intentar convertir un rectángulo en un círculo y asegurarnos de que obtenemos 0:

```
c = dynamic_cast<Circle *>(r); // trying to cast type, and will get NULL if unsucc
Print(c); // 0
```

Hay un tipo de puntero especial en MQL5 que puede almacenar cualquier objeto. Este tipo tiene la siguiente notación: *void **.

Vamos a demostrar cómo funciona la variable *void ** con *dynamic_cast*.

```
void *v;
v = s;   // set to the instance Square
PRT(dynamic_cast<Shape *>(v));
PRT(dynamic_cast<Rectangle *>(v));
PRT(dynamic_cast<Square *>(v));
PRT(dynamic_cast<Circle *>(v));
PRT(dynamic_cast<Triangle *>(v));
```

Las tres primeras líneas registrarán el valor del puntero (un descriptor del mismo objeto) y las dos últimas imprimirán 0.

Ahora, volvamos al ejemplo de la declaración forward del archivo Indicadores (véase el archivo *ThisCallback.mq5*), donde las clases *Manager* y *Element* contenían punteros mutuos.

El tipo de puntero *void ** le permite deshacerse de la declaración preliminar (*ThisCallbackVoid.mq5*). Vamos a comentar la línea que lo contiene, y cambiemos el tipo del campo *owner* con un puntero al objeto gestor a *void **. En el constructor, cambiamos también el tipo del parámetro.

```
// class Manager;
class Element
{
   void *owner; // looking forward to being compatible with the Manager type *
public:
   Element(void *t = NULL): owner(t) { } // was Element(Manager &t)
   void doMath()
   {
      const int N = 1000000;

      // get the desired type at runtime
      Manager *ptr = dynamic_cast<Manager *>(owner);
      // then everywhere you need to check ptr for NULL before using

      for(int i = 0; i < N; ++i)
      {
         if(i % (N / 20) == 0)
         {
            if(ptr != NULL) ptr.progressNotify(&this, i * 100.0f / N);
         }
         // ... lots of calculations
      }
      if(ptr != NULL) ptr.progressNotify(&this, 100.0f);
   }
   ...
};
```

Este enfoque puede proporcionar más flexibilidad, pero requiere más cuidado porque *dynamic_cast* puede devolver NULL. Se recomienda, siempre que sea posible, utilizar facilidades de envío estándar (estáticas y dinámicas) con control de los tipos proporcionados por el lenguaje.

Los punteros *void** suelen ser necesarios en casos excepcionales. Y no es el caso de la línea «extra» con una descripción preliminar: se ha utilizado aquí sólo como el ejemplo más sencillo de la universalidad del puntero *void**.

3.2.21 Punteros, referencias y const

Una vez descubiertos los tipos integrados y de objeto, así como los conceptos de referencia y puntero, probablemente tenga sentido hacer una comparación de todas las modificaciones de tipo disponibles.

Las referencias en MQL5 sólo se utilizan cuando se describen los parámetros de funciones y métodos. Además, los parámetros de tipo objeto deben pasarse por referencia.

```
void function(ClassOrStruct &object) { }          // OK
void function(ClassOrStruct object) { }           // wrong
void function(double &value) { }                  // OK
void function(double value) { }                   // OK
```

Aquí *ClassOrStruct* es el nombre de la clase o estructura.

Está permitido pasar sólo variables (LValue) como argumento de un parámetro de tipo referencia, pero no constantes o valores temporales obtenidos como resultado de la evaluación de una expresión.

No se puede crear una variable de tipo referencia ni devolver una referencia desde una función.

```
ClassOrStruct &function(void) { return Class(); } // wrong
ClassOrStruct &object;                            // wrong
double &value;                                    // wrong
```

Los punteros en MQL5 sólo están disponibles para objetos de clase. No se admiten punteros a variables de estructuras o tipos integrados.

Puede declarar una variable o parámetro de función de tipo puntero a un objeto, y también devolver un puntero a un objeto desde la función.

```
ClassOrStruct *pointer;                                 // OK
void function(ClassOrStruct *object) { }                // OK
ClassOrStruct *function() { return new ClassOrStruct(); } // OK
```

Sin embargo, no se puede devolver un puntero a un objeto local automático, ya que éste se liberará al salir la función y el puntero dejará de ser válido.

Si la función devuelve un puntero a un objeto asignado dinámicamente dentro de la función con new, entonces el código de llamada debe «recordar» liberar el puntero con delete.

Un puntero, a diferencia de una referencia, puede ser NULL. Los parámetros de puntero pueden tener un valor por defecto, pero las referencias no (error «la referencia no se puede inicializar»).

```
void function(ClassOrStruct *object = NULL) { }          // OK
void function(ClassOrStruct &object = NULL) { }          // wrong
```

Los enlaces y los punteros pueden combinarse en la descripción de un parámetro. Así, una función puede tomar una referencia a un puntero, y luego los cambios en el puntero en la función estará disponible en el código de llamada. En concreto, puede implementarse de este modo la función de fábrica, que es responsable de la creación de objetos.

```
void createObject(ClassName *&ref)
{
   ref = new ClassName();
   // further customization of ref
   ...
}
```

Es cierto que, para devolver un único puntero desde una función, suele ser habitual utilizar la sentencia return, por lo que este ejemplo es un tanto artificial. Sin embargo, en aquellos casos en los que es necesario pasar un array de punteros al exterior, una referencia al mismo en el parámetro se convierte en la opción preferida. Por ejemplo, en algunas clases de la biblioteca estándar para trabajar con clases contenedoras del tipo map con pares [clave, valor] (*MQL5/Include/Generic/SortedMap.mqh*, *MQL5/Include/Generic/HashMap.mqh*) existen métodos *CopyTo* para obtener arrays con elementos *CKeyValuePair*.

```
int CopyTo(CKeyValuePair<TKey,TValue> *&dst_array[], const int dst_start = 0);
```

El tipo de parámetro *dst_array* puede parecer poco familiar: se trata de una plantilla de clase. Hablaremos de las plantillas en el capítulo siguiente. Aquí, por ahora, lo único que nos importa es que se trata de una referencia a un array de punteros.

El modificador *const* impone un comportamiento especial para todos los tipos. En relación con los tipos integrados, esto se abordó en la sección sobre Variables constantes. Los tipos de objetos tienen sus propias características.

Si un parámetro de función o variable se declara como puntero o referencia a un objeto (sólo es una referencia en el caso de un parámetro), la presencia del modificador *const* sobre ellos limita el conjunto de métodos y propiedades a los que se puede acceder a sólo aquellos que también tengan el modificador *const*. En otras palabras: sólo las propiedades constantes son accesibles a través de punteros y referencias constantes.

Cuando intente llamar a un método no constante o cambiar un campo no constante, el compilador generará un error: «llamar a método no constante para objeto constante» o «no se puede modificar la constante».

Un parámetro de puntero no constante puede tomar cualquier argumento (constante o no constante).

Hay que tener presente que en la descripción del puntero se pueden establecer dos modificadores *const*, uno referido al objeto y el segundo, al puntero:

- *Class *pointer* es un puntero a un objeto; el objeto y el puntero funcionan sin limitaciones;
- *const Class *pointer* es un puntero a un objeto constante; para el objeto, sólo están disponibles los métodos constantes y las propiedades de lectura, pero el puntero se puede cambiar (asignándole la dirección de otro objeto);
- *const Class * const pointer* es un puntero constante a un objeto constante; para el objeto, sólo están disponibles los métodos constantes y las propiedades de lectura; el puntero no se puede modificar;
- *Class * const pointer* es un puntero constante a un objeto; el puntero no puede cambiarse, pero las propiedades del objeto sí.

Considere la siguiente clase *Counter* (*CounterConstPtr.mq5*) como ejemplo.

```
class Counter
{
public:
   int counter;

   Counter(const int n = 0) : counter(n) { }

   void increment()
   {
      ++counter;
   }

   Counter *clone() const
   {
      return new Counter(counter);
   }
};
```

Ha hecho artificialmente la variable pública *counter*. La clase también tiene dos métodos, uno de los cuales es constante (*clone*) y el segundo, no (*increment*). Recordemos que un método constante no tiene derecho a modificar los campos de un objeto.

La siguiente función con el parámetro de tipo *Counter *ptr* puede llamar a todos los métodos de la clase y cambiar sus campos.

```
void functionVolatile(Counter *ptr)
{
   // OK: everything is available
   ptr.increment();
   ptr.counter += 2;
   //remove the clone immediately so that there is no memory leak
   // the clone is only needed to demonstrate calling a constant method
   delete ptr.clone();
   ptr = NULL;
}
```

La siguiente función con el parámetro *const Counter *ptr* arrojará un par de errores.

```
void functionConst(const Counter *ptr)
{
   // ERRORS:
   ptr.increment(); // calling non-const method for constant object
   ptr.counter = 1; // constant cannot be modified

   // OK: only const methods are available, fields can be read
   Print(ptr.counter); // reading a const object
   Counter *clone = ptr.clone(); // calling a const method
   ptr = clone;     // changing a non-const pointer ptr
   delete ptr;      // cleaning memory
}
```

Por último, la siguiente función con el parámetro *const Counter * const ptr* hace aún menos.

```
void functionConstConst(const Counter * const ptr)
{
   // OK: only const methods are available, the pointer ptr cannot be changed
   Print(ptr.counter); // reading a const object
   delete ptr.clone(); // calling a const method

   Counter local(0);
   // ERRORS:
   ptr.increment(); // calling non-const method for constant object
   ptr.counter = 1; // constant cannot be modified
   ptr = &local;    // constant cannot be modified
}
```

En la función *OnStart*, donde hemos declarado dos objetos *Counter* (uno constante y el otro no), puede llamar a estas funciones con algunas excepciones:

```
void OnStart()
{
   Counter counter;
   const Counter constCounter;

   counter.increment();

   // ERROR:
   // constCounter.increment(); // call non-const method for constant object
   Counter *ptr = (Counter *)&constCounter; // trick: type casting without const
   ptr.increment();

   functionVolatile(&counter);

   // ERROR: cannot convert from a const pointer...
   // functionVolatile(&constCounter); // to a non-const pointer

   functionVolatile((Counter *)&constCounter); // type casting without const

   functionConst(&counter);
   functionConst(&constCounter);

   functionConstConst(&counter);
   functionConstConst(&constCounter);
}
```

En primer lugar, tenga en cuenta que las variables también generan un error cuando se intenta llamar a un método constante *increment* en un objeto no constante.

En segundo lugar, *constCounter* no puede pasarse a la función *functionVolatile*: obtenemos el error «no se puede convertir de puntero constante a puntero no constante».

Sin embargo, ambos errores pueden eludirse mediante la asignación explícita de tipos sin el modificador *const*, aunque esto no es recomendable.

3.2.22 Gestión de la herencia: final y delete

MQL5 permite imponer algunas restricciones a la herencia de clases y estructuras.

Palabra clave *final*

Añadiendo la palabra clave *final* después del nombre de la clase, el desarrollador puede desactivar la herencia de esa clase. Por ejemplo (*FinalDelete.mq5*):

```
class Base
{
};

class Derived final : public Base
{
};

class Concrete : public Derived // ERROR
{
};
```

El compilador devolverá el error «no se puede heredar de 'Derived' ya que ha sido declarado como 'final'».

Desgraciadamente, no hay consenso sobre las ventajas y las hipótesis de aplicación de tal restricción. La palabra clave permite a los usuarios de la clase saber que su autor, por una razón u otra, no recomienda tomarla como base (por ejemplo, su implementación actual es un borrador y cambiará mucho, lo que puede hacer que los posibles proyectos heredados dejen de compilar).

Algunas personas intentan fomentar el diseño de programas de esta manera, en la que se use la inclusión de objetos (composición) en lugar de la herencia. Una pasión excesiva por la herencia puede, en efecto, aumentar la cohesión de la clase (es decir, la influencia mutua), ya que todos los herederos pueden, de un modo u otro, modificar los datos o métodos padre (en particular, redefiniendo funciones virtuales). Como resultado, aumenta la complejidad de la lógica de funcionamiento del programa y la probabilidad de que se produzcan efectos secundarios imprevistos.

Una ventaja adicional del uso de *final* puede ser la optimización del código por parte del compilador: para punteros de tipos «finales», puede sustituir el envío dinámico de funciones virtuales por el estático.

Palabra clave *delete*

La palabra clave *delete* puede especificarse en el encabezado de un método para hacerlo inaccesible en la clase actual y sus descendientes. Los métodos virtuales de las clases padre no pueden eliminarse de esta forma (esto infringiría el «contrato» de la clase, es decir, los herederos dejarían de «ser» («es un») representantes de la misma clase).

```
class Base
{
public:
   void method() { Print(__FUNCSIG__); }
};

class Derived : public Base
{
public:
   void method() = delete;
};

void OnStart()
{
   Base *b;
   Derived d;

   b = &d;
   b.method();

   // ERROR:
   // attempting to reference deleted function 'void Derived::method()'
   //    function 'void Derived::method()' was explicitly deleted
   d.method();
}
```

Si se intenta llamarlo, se producirá un error de compilación.

Vimos un error similar en la sección Conversión de tipos de objeto porque el compilador tiene cierta inteligencia para «eliminar» también métodos en ciertas condiciones.

Se recomienda marcar como eliminados los siguientes métodos para los que el compilador proporciona implementaciones implícitas:

- constructor por defecto: *Class(void) = delete*;
- constructor de copias: *Class(const Class &object) = delete*;
- operador de copia/asignación: *void operator=(const Class &object) = delete*.

Si necesita alguno de ellos, debe definirlo explícitamente. En caso contrario, se considera una buena práctica abandonar la implementación implícita. La cuestión es que la implementación implícita es bastante sencilla y puede dar lugar a problemas difíciles de localizar, en particular, al convertir tipos de objetos.

3.3 Plantillas

En los lenguajes de programación modernos existen muchas funciones integradas que le permiten evitar la duplicación de código y, por tanto, minimizar el número de errores y aumentar la productividad del programador. En MQL5, dichas herramientas incluyen las ya conocidas funciones, tipos de objeto que admiten herencia (clases y estructuras), macros de preprocesador y la capacidad de incluir archivos. Pero esta lista estaría incompleta sin las plantillas.

Una plantilla es una definición genérica especialmente diseñada de una función o tipo de objeto a partir de la cual el compilador puede generar automáticamente instancias de trabajo de esa función o tipo de objeto. Las instancias resultantes contienen el mismo algoritmo pero operan sobre variables de distinto tipo, correspondientes a las condiciones de uso específicas de la plantilla en el código fuente.

Para los que conocen C++, señalamos que las plantillas de MQL5 no admiten muchas características de las plantillas de C++, en particular:

- parámetros que no son tipos;
- parámetros con valores por defecto;
- número variable de parámetros;
- especialización de clases, estructuras y asociaciones (total y parcial);
- plantillas para plantillas.

Por un lado, esto reduce el potencial de las plantillas en MQL5, pero, por otro, simplifica el aprendizaje del material para quienes no estén familiarizados con estas tecnologías.

3.3.1 Encabezado de plantilla

En MQL5, puede hacer con plantillas las funciones, los tipos de objetos (clases, estructuras, uniones) o los métodos independientes de dentro de ellos. En cualquier caso, la descripción de la plantilla tiene un título:

```
template <typename T [, typename Ti ... ]>
```

El encabezado comienza con la palabra clave *template*, seguida de una lista separada por comas de parámetros formales entre corchetes angulares: cada parámetro se señala mediante la palabra clave *typename* y un identificador. Los identificadores deben ser únicos dentro de una definición concreta.

La palabra clave *typename* del encabezado de la plantilla le indica al compilador que el siguiente identificador debe tratarse como un tipo. En el futuro, es probable que el compilador de MQL5 admita otros tipos de parámetros que no sean de tipo, como hace el compilador de C++.

Este uso de *typename* no debe confundirse con el operador integrado *typename*, que devuelve una cadena con el nombre del tipo del argumento pasado.

Un encabezado de plantilla va seguido de una definición habitual de una función (método) o clase (estructura, unión), en la que los parámetros formales de la plantilla (identificadores T, Ti) se utilizan en instrucciones y expresiones en aquellos lugares en los que la sintaxis requiere un nombre de tipo. Por ejemplo, para las funciones de plantilla, los parámetros de plantilla describen los tipos de los parámetros de la función o el valor de retorno, y en una clase de plantilla, un parámetro de plantilla puede designar un tipo de campo.

Una plantilla es una definición completa. Una plantilla termina con la definición de una entidad (función, método, clase, estructura, unión) precedida del encabezamiento *template*.

Para los nombres de parámetros de plantilla, es habitual tomar identificadores de uno o dos caracteres en mayúsculas.

El número mínimo de parámetros es 1 y el máximo, 64.

Los principales casos de uso de los parámetros (utilizando el parámetro T como ejemplo) incluyen:

- tipo al describir campos, variables locales en funciones/métodos, sus parámetros y valores de retorno (*T variable_name*; *T function(T parameter_name)*);
- uno de los componentes de un nombre de tipo totalmente cualificado, en concreto *T::SubType*, *T.StaticMember*;
- construcción de nuevos tipos con modificadores: *const T*, puntero *T **, referencia *T &*, matriz *T[]*, funciones *typedef T(*func)(T)*;
- construcción de nuevos tipos de plantillas: *T<Type>*, *Type<T>*, incluso al heredar de plantillas (véase la sección Especialización de plantillas, que no está presente);
- conversión de tipos *(T)* con la capacidad de añadir modificadores y crear objetos vía *new T()*;
- *sizeof(T)* como reemplazo primitivo para los parámetros de valor que están ausentes en las plantillas MQL (en el momento de escribir el libro).

3.3.2 Principios generales de funcionamiento de la plantilla

Recordemos la sobrecarga de funciones. Consiste en definir varias versiones de una función con distintos parámetros, incluidas las situaciones en las que el número de parámetros es el mismo, pero sus tipos son diferentes. A menudo, un algoritmo de este tipo de funciones es el mismo para parámetros de distintos tipos. Por ejemplo, MQL5 tiene una función integrada *MathMax* que devuelve el mayor de los dos valores que se le han pasado:

```
double MathMax(double value1, double value2);
```

Aunque sólo se proporciona un prototipo para el tipo *double*, en realidad la función es capaz de trabajar con pares de argumentos de otros tipos numéricos, como *int* o *datetime*. En otras palabras: la función es un núcleo sobrecargado para tipos numéricos integrados. Si quisiéramos conseguir el mismo efecto en nuestro código fuente, tendríamos que sobrecargar la función duplicándola con diferentes parámetros, de esta forma:

```
double Max(double value1, double value2)
{
   return value1 > value2 ? value1 : value2;
}

int Max(int value1, int value2)
{
   return value1 > value2 ? value1 : value2;
}

datetime Max(datetime value1, datetime value2)
{
   return value1 > value2 ? value1 : value2;
}
```

Todas las implementaciones (cuerpos de función) son iguales. Sólo cambian los tipos de parámetros.

Aquí es cuando las plantillas resultan útiles. Utilizándolas podemos describir una muestra del algoritmo con la implementación requerida, y el propio compilador generará varias instancias del mismo para los tipos específicos implicados en el programa. La generación se produce sobre la marcha durante la compilación y es imperceptible para el programador (a menos que haya un error en la plantilla). El código fuente obtenido de forma automática no se inserta en el texto del programa, sino que se convierte directamente en código binario (archivo ex5).

En la plantilla, uno o más parámetros son designaciones formales de tipos, para los cuales, en la etapa de compilación y de acuerdo con reglas especiales de inferencia de tipos, se seleccionarán tipos reales de entre los integrados o los definidos por el usuario. Por ejemplo, la función *Max* puede describirse utilizando la siguiente plantilla con el parámetro de tipo T:

```
template<typename T>
T Max(T value1, T value2)
{
   return value1 > value2 ? value1 : value2;
}
```

Y a continuación, se puede aplicar para variables de diversos tipos (véase *TemplatesMax.mq5*):

```
void OnStart()
{
   double d1 = 0, d2 = 1;
   datetime t1 = D'2020.01.01', t2 = D'2021.10.10';
   Print(Max(d1, d2));
   Print(Max(t1, t2));
   ...
}
```

En este caso, el compilador generará automáticamente variantes de la función *Max* para los tipos *double* y *datetime*.

La plantilla en sí no genera código fuente. Para ello es necesario crear una instancia de la plantilla de un modo u otro: llamar a una función de plantilla o mencionar el nombre de una clase de plantilla con tipos específicos para crear un objeto o una clase derivada.

Hasta que no se haga esto, el compilador ignorará todo el patrón. Por ejemplo, podemos escribir la siguiente función supuestamente de plantilla, que en realidad contiene código sintácticamente incorrecto. Sin embargo, la compilación de un módulo con esta función tendrá éxito siempre y cuando no sea invocada en ninguna parte.

```
template<typename T>
void function()
{
  it's not a comment, but it's not source code either
   !%^&*
}
```

Para cada uso de la plantilla, el compilador determina los tipos reales que coinciden con los parámetros formales de la plantilla. A partir de esta información se genera automáticamente el código fuente de la plantilla para cada combinación única de parámetros. Esta es la instancia.

Así, en el ejemplo dado de la función *Max*, hemos llamado a la función de plantilla dos veces: para el par de variables del tipo *double*, y para el par de variables del tipo *datetime*. El resultado fueron dos instancias de la función *Max* con código fuente para las coincidencias *T=double* y *T=datetime*. Por supuesto, si se llama a la misma plantilla en otras partes del código para los mismos tipos, no se generarán nuevas instancias. Sólo se necesita una nueva instancia de la plantilla si ésta se aplica a otro tipo (o conjunto de tipos, si hay más de 1 parámetro).

Tenga en cuenta que la plantilla *Max* tiene un parámetro, y establece el tipo para dos parámetros de entrada de la función y su valor de retorno a la vez. En otras palabras: la declaración de plantilla es capaz de imponer ciertas restricciones a los tipos de argumentos válidos.

Si llamáramos a *Max* en variables de tipos diferentes, el compilador no sería capaz de determinar el tipo para crear una nueva instancia de la plantilla y mostraría el error «parámetros de plantilla ambiguos, debe ser 'double' o 'datetime'»:

```
Print(Max(d1, t1)); // template parameter ambiguous,
                    // could be 'double' or 'datetime'
```

Este proceso de descubrimiento de los tipos reales de los parámetros de la plantilla basándose en el contexto en el que se utiliza la plantilla se denomina deducción de tipos. En MQL5, la inferencia de tipo está disponible sólo para las plantillas de métodos y funciones.

Para las clases, estructuras y uniones se utiliza una forma diferente de vincular tipos a los parámetros de la plantilla: los tipos requeridos se especifican explícitamente entre paréntesis angulares al crear una instancia de plantilla (si hay varios parámetros, el número correspondiente de tipos se indica como una lista separada por comas). Para obtener más información, consulte la sección Plantillas de tipos de objeto.

El mismo método explícito puede aplicarse a las funciones como alternativa a la inferencia automática de tipos.

Por ejemplo, podemos generar y llamar a una instancia de *Max* para el tipo *ulong*:

```
Print(Max<ulong>(1000, 10000000));
```

En este caso, si no fuera por la indicación explícita, la función de plantilla se asociaría al tipo *int* (basado en los valores de las constantes enteras).

3.3.3 Plantillas frente a macros de preprocesador

En algún momento puede surgir la pregunta de si es posible utilizar sustituciones de macros para generar código De hecho, sí es posible. Por ejemplo, el conjunto de funciones *Max* puede representarse fácilmente como una macro:

```
#define MAX(V1,V2) ((V1) > (V2) ? (V1) : (V2))
```

No obstante, las macros tienen capacidades más limitadas (nada más que la sustitución de texto) y, por tanto, sólo se utilizan en casos sencillos (como el anterior).

Cuando se comparan macros y plantillas deben tenerse en cuenta las diferencias que se indican a continuación.

El preprocesador «expande» las macros y las sustituye en el texto fuente antes de iniciar la compilación. Al mismo tiempo, no hay información sobre los tipos de parámetros y el contexto en el que se sustituye el contenido de la macro. En concreto, la macro MAX no puede proporcionar una comprobación de que los tipos de los parámetros V1 y V2 son iguales, y también de que el operador de comparación '>' está definido para ellos. Además, si en el texto de un programa se encuentra una variable con el nombre MAX, el preprocesador intentará sustituirla por la «llamada» de la macro MAX y estará «descontento» con la ausencia de argumentos. Peor aún, estas sustituciones ignoran en qué espacios de nombres o clases se encuentra el token MAX: básicamente, cualquiera sirve.

A diferencia de las macros, las plantillas son manejadas por el compilador en términos de tipos de argumentos específicos y dónde se utilizan, por lo que proporcionan comprobaciones de compatibilidad de tipos (y aplicabilidad general) para todas las expresiones de una plantilla, así como vinculación contextual. Por ejemplo, podemos definir una plantilla de método dentro de una clase concreta.

Una plantilla con el mismo nombre puede definirse de forma diferente para distintos tipos si es necesario, mientras que una macro con un nombre determinado siempre se sustituye por la misma «implementación». Por ejemplo, en el caso de una función como MAX, podríamos definir una comparación para cadenas que no distinga entre mayúsculas y minúsculas.

Los errores de compilación debidos a problemas en las macros son difíciles de diagnosticar, especialmente si la macro consta de varias líneas, ya que la línea problemática con la «llamada» de la macro se resalta «tal cual», sin la versión expandida del texto, tal y como llegó del preprocesador al compilador.

Al mismo tiempo, las plantillas son elementos del código fuente con una forma ya preparada, tal y como entran en el compilador, por lo que cualquier error en ellas tiene un número de línea y una posición en la línea específicos.

Las macros pueden tener efectos secundarios, de los que hablamos en la sección Forma de #define como pseudo-función: si los argumentos de la macro MAX son expresiones con incrementos/decrementos, se ejecutarán dos veces.

No obstante, las macros también tienen algunas ventajas. Las macros son capaces de generar cualquier texto, no sólo construcciones correctas del lenguaje. Por ejemplo, con unas pocas macros, puede simular la instrucción *switch* para cadenas (aunque este enfoque no es recomendable).

En la biblioteca estándar, las macros se utilizan, en concreto, para organizar el tratamiento de los eventos en los gráficos (véase *MQL5/Include/Controls/Defines.mqh*: EVENT_MAP_BEGIN, EVENT_MAP_END, ON_EVENT, etc.). No funcionará en plantillas, pero la forma de organizar un mapa de eventos en macros, por supuesto, está lejos de ser la única y no es la más conveniente para la depuración. Es difícil depurar paso a paso (línea a línea) la ejecución de código en macros. Las plantillas, por el contrario, admiten la depuración en su totalidad.

3.3.4 Características de tipos integrados y objetos en plantillas

Hay que tener en cuenta que tres aspectos importantes imponen restricciones a la posibilidad de aplicación de los tipos en una plantilla:

- Que el tipo sea integrado o definido por el usuario (los tipos definidos por el usuario requieren que los parámetros se pasen por referencia, y los integrados no permitirán que un literal se pase por referencia);
- Que el tipo de objeto sea una clase (sólo las clases admiten punteros);
- Un conjunto de operaciones realizadas sobre datos de los tipos apropiados en el algoritmo de plantilla.

Supongamos que tenemos una estructura Dummy (véase el script *TemplatesMax.mq5*):

```
struct Dummy
{
   int x;
};
```

Si intentamos llamar a la función Max para dos instancias de la estructura, obtendremos un ramillete de mensajes de error, siendo los principales los siguientes: «los objetos sólo se pueden pasar por referencia» y «no puede aplicar una plantilla».

```
// ERRORS:
// 'object1' - objects are passed by reference only
// 'Max' - cannot apply template
Dummy object1, object2;
Max(object1, object2);
```

La cúspide del problema es pasar parámetros de funciones de plantilla por valor, y este método es incompatible con cualquier tipo de objeto. Para solucionarlo, puede cambiar el tipo de parámetros a enlaces:

```
template<typename T>
T Max(T &value1, T &value2)
{
   return value1 > value2 ? value1 : value2;
}
```

El error anterior desaparecerá, pero entonces obtendremos un nuevo error: «'>' - uso de operación ilegal» («'>' - uso de operación ilegal»). La cuestión es que la plantilla Max tiene una expresión con el operador de comparación '>'. Por lo tanto, si se sustituye un tipo personalizado en la plantilla, el operador '>' debe estar sobrecargado en la plantilla (y la estructura *Dummy* no lo tiene; abordaremos eso en breve). Para funciones más complejas, es probable que necesite sobrecargar un número mucho mayor de operadores. Por suerte, el compilador le indica exactamente lo que falta.

Sin embargo, cambiar el método de pasar los parámetros de la función por referencia provocó a su vez que la llamada anterior dejara de funcionar como tal:

```
Print(Max<ulong>(1000, 10000000));
```

Ahora genera errores: «parámetro pasado como referencia, variable esperada». Así, nuestra plantilla de funciones dejó de funcionar con literales y otros valores temporales (en concreto, es imposible pasar directamente una expresión o el resultado de llamar a otra función).

Se podría pensar que la forma universal de salir de la situación sería la sobrecarga de funciones de plantilla, es decir, la definición de ambas opciones, que sólo difiere en el ampersand de los parámetros:

```
template<typename T>
T Max(T &value1, T &value2)
{
   return value1 > value2 ? value1 : value2;
}

template<typename T>
T Max(T value1, T value2)
{
   return value1 > value2 ? value1 : value2;
}
```

Pero no funcionará. Ahora el compilador muestra el error «sobrecarga de función ambigua con los mismos parámetros»:

```
'Max' - ambiguous call to overloaded function with the same parameters
could be one of 2 function(s)
   T Max(T&,T&)
   T Max(T,T)
```

La sobrecarga final en funcionamiento requeriría añadir el modificador *const* a los enlaces. Por el camino, hemos añadido el operador *Print* a la plantilla *Max* para que podamos ver en el registro a qué sobrecarga se está llamando y a qué tipo de parámetro corresponde T.

```
template<typename T>
T Max(const T &value1, const T &value2)
{
   Print(__FUNCSIG__, " T=", typename(T));
   return value1 > value2 ? value1 : value2;
}

template<typename T>
T Max(T value1, T value2)
{
   Print(__FUNCSIG__, " T=", typename(T));
   return value1 > value2 ? value1 : value2;
}

struct Dummy
{
   int x;
   bool operator>(const Dummy &other) const
   {
      return x > other.x;
   }
};
```

También hemos implementado una sobrecarga del operador '>' en la estructura Dummy. Por consiguiente, todas las llamadas a funciones *Max* del script de prueba se completan con éxito: tanto para tipos integrados como definidos por el usuario, así como para literales y variables. Las salidas que van al registro:

```
double Max<double>(double,double) T=double
1.0
datetime Max<datetime>(datetime,datetime) T=datetime
2021.10.10 00:00:00
ulong OnStart::Max<ulong>(ulong,ulong) T=ulong
10000000
Dummy Max<Dummy>(const Dummy&,const Dummy&) T=Dummy
```

Un lector atento se dará cuenta de que ahora tenemos dos funciones idénticas que sólo difieren en la forma en que se pasan los parámetros (por valor y por referencia), y ésta es exactamente la situación contra la que se dirige el uso de plantillas. Esta duplicación puede resultar costosa si el cuerpo de la función no es tan sencillo como el nuestro. Esto puede resolverse con los métodos habituales: separar la implementación en una función independiente y llamarla desde ambas «sobrecargas», o llamar a una «sobrecarga» desde la otra (se requería un parámetro opcional para evitar que la primera versión de *Max* se llamara a sí misma y, como consecuencia, se produjeran desbordamientos de pila):

```
template<typename T>
T Max(T value1, T value2)
{
   // calling a function with parameters by reference
   return Max(value1, value2, true);
}

template<typename T>
T Max(const T &value1, const T &value2, const bool ref = false)
{
   return (T)(value1 > value2 ? value1 : value2);
}
```

Todavía tenemos que considerar un punto más asociado a los tipos definidos por el usuario, a saber, el uso de punteros en plantillas (recuerde que sólo se aplican a los objetos de clase). Vamos a crear una sencilla clase *Data* y a intentar llamar a la función de plantilla *Max* para los punteros a sus objetos.

```
class Data
{
public:
   int x;
   bool operator>(const Data &other) const
   {
      return x > other.x;
   }
};

void OnStart()
{
   ...
   Data *pointer1 = new Data();
   Data *pointer2 = new Data();
   Max(pointer1, pointer2);
   delete pointer1;
   delete pointer2;
}
```

Veremos en el registro que 'T=Data*', es decir, el atributo de puntero, alcanza el tipo inline. Esto sugiere que, si es necesario, puede escribir otra sobrecarga de la función de plantilla, que será responsable sólo de los punteros.

```
template<typename T>
T *Max(T *value1, T *value2)
{
   Print(__FUNCSIG__, " T=", typename(T));
   return value1 > value2 ? value1 : value2;
}
```

En este caso, el atributo del puntero '*' ya está presente en los parámetros de la plantilla, por lo que la inferencia de tipo da como resultado 'T=Data'. Este enfoque le permite proporcionar una implementación de plantilla independiente para los punteros.

Si hay varias plantillas adecuadas para generar una instancia con tipos específicos, se elige la versión más especializada de la plantilla. En concreto, cuando se llama a la función *Max* con argumentos de puntero, aparecen dos plantillas con parámetros T (*T=Data**) y T* (*T=Data*), pero como la primera puede tomar tanto valores como punteros, es más general que la segunda, que sólo funciona con punteros. Por lo tanto, se elegirá la segunda para los punteros. En otras palabras: cuantos menos modificadores haya en el tipo real que se sustituye por T, más preferible será la variante de plantilla. Además del atributo del puntero '*', ello también incluye el modificador *const*. Los parámetros *const T** o *const T* son más especializados que *T** o *T*, respectivamente.

3.3.5 Plantillas de funciones

Una plantilla de función consta de un encabezado con parámetros de plantilla (la sintaxis se ha descrito antes) y una definición de función en la que los parámetros de la plantilla denotan tipos arbitrarios.

Como primer ejemplo, considere la función *Swap* para intercambiar dos elementos de una array (*TemplatesSorting.mq5*). El parámetro de plantilla T se utiliza como tipo de la variable de array de entrada, así como el tipo de la variable local *temp*.

```
template<typename T>
void Swap(T &array[], const int i, const int j)
{
   const T temp = array[i];
   array[i] = array[j];
   array[j] = temp;
}
```

Todas las sentencias y expresiones del cuerpo de la función deben ser aplicables a tipos reales, para los que se crearán seguidamente instancias de plantilla. En este caso, se utiliza el operador de asignación '='. Aunque siempre existe para los tipos integrados, puede ser necesario sobrecargarla explícitamente para los tipos definidos por el usuario.

El compilador genera por defecto la implementación del operador de copia para clases y estructuras, pero puede eliminarse implícita o explícitamente (véase la palabra clave *delete*). En particular, como vimos en la sección Conversión de tipos de objeto, tener un campo constante en una clase hace que el compilador elimine su opción de copia implícita. Seguidamente, la función *Swap* de la función anterior no puede utilizarse para objetos de esta clase: el compilador generará un error.

Para las clases/estructuras con las que trabaja la función *Swap*, es deseable tener no sólo un operador de asignación, sino también un constructor de copia, porque la declaración de la variable *temp* es en realidad una construcción con una inicialización, no una asignación. Con un constructor de copia, la primera línea de la función se ejecuta de una sola vez (se crea *temp* en base a *array[i]*), mientras que sin él, primero se llamará al constructor por defecto, y luego se ejecutará el operador '=' para *temp*.

Veamos cómo se puede utilizar la función de plantilla *Swap* en el algoritmo quicksort: lo implementa otra función de plantilla *QuickSort*.

```
template<typename T>
void QuickSort(T &array[], const int start = 0, int end = INT_MAX)
{
   if(end == INT_MAX)
   {
      end = start + ArraySize(array) - 1;
   }
   if(start < end)
   {
      int pivot = start;

      for(int i = start; i <= end; i++)
      {
         if(!(array[i] > array[end]))
         {
            Swap(array, i, pivot++);
         }
      }

      --pivot;

      QuickSort(array, start, pivot - 1);
      QuickSort(array, pivot + 1, end);
   }
}
```

Observe que el parámetro T de la plantilla *QuickSort* especifica el tipo del parámetro de entrada *array*, y este array se pasa a la plantilla *Swap*. Así, la inferencia de tipo T para la plantilla *QuickSort* determinará automáticamente el tipo T para la plantilla *Swap*.

La función integrada *ArraySize* (como muchas otras) es capaz de funcionar con arrays de tipos arbitrarios: en cierto sentido, se trata también de una plantilla, aunque está implementada directamente en el terminal.

La ordenación se realiza gracias al operador de comparación '>' de la sentencia *if*. Como hemos señalado antes, este operador debe definirse para cualquier tipo T que se esté ordenando, ya que se aplica a los elementos de un array de tipo T.

Comprobemos cómo funciona la ordenación para arrays de tipos integrados.

```
void OnStart()
{
   double numbers[] = {34, 11, -7, 49, 15, -100, 11};
   QuickSort(numbers);
   ArrayPrint(numbers);
   // -100.00000 -7.00000 11.00000 11.00000 15.00000 34.00000 49.00000

   string messages[] = {"usd", "eur", "jpy", "gbp", "chf", "cad", "aud", "nzd"};
   QuickSort(messages);
   ArrayPrint(messages);
   // "aud" "cad" "chf" "eur" "gbp" "jpy" "nzd" "usd"
}
```

Dos llamadas a la función de plantilla *QuickSort* deducen automáticamente el tipo de T basándose en los tipos de los arrays pasados. Como resultado, obtendremos dos instancias de *QuickSort* para los tipos *double* y *string*.

Para comprobar la ordenación de un tipo personalizado, vamos a crear una estructura ABC con un campo entero *x*, y la rellenaremos con números aleatorios en el constructor. También es importante sobrecargar el operador '>' en la estructura.

```
struct ABC
{
   int x;
   ABC()
   {
      x = rand();
   }
   bool operator>(const ABC &other) const
   {
      return x > other.x;
   }
};
void OnStart()
{
   ...
   ABC abc[10];
   QuickSort(abc);
   ArrayPrint(abc);
  /* Sample output:
            [x]
      [0]  1210
      [1]  2458
      [2] 10816
      [3] 13148
      [4] 15393
      [5] 20788
      [6] 24225
      [7] 29919
      [8] 32309
      [9] 32589
   */
}
```

Como los valores de la estructura se generan aleatoriamente, obtendremos resultados diferentes, pero siempre estarán ordenados de forma ascendente.

En este caso, el tipo T también se deduce automáticamente. Sin embargo, en algunos casos, la especificación explícita es la única forma de pasar un tipo a una plantilla de función. Así, si una función de plantilla debe devolver un valor de un tipo único (diferente de los tipos de sus parámetros) o si no hay parámetros, entonces sólo puede especificarse explícitamente.

Por ejemplo, la siguiente función de plantilla *createInstance* requiere que el tipo se especifique explícitamente en la instrucción de llamada, ya que no es posible «calcular» automáticamente el tipo T a partir del valor de retorno. Si no se hace así, el compilador genera un error de «desajuste de plantilla».

```
class Base
{
   ...
};

template<typename T>
T *createInstance()
{
   T *object = new T(); //calling the constructor
   ...                  //object setting
   return object;
}

void OnStart()
{
   Base *p1 = createInstance();       // error: template mismatch
   Base *p2 = createInstance<Base>(); // ok, explicit directive
   ...
}
```

Si hay varios parámetros de plantilla, y el tipo del valor de retorno no está vinculado a ninguno de los parámetros de entrada de la función, entonces también es necesario especificar un tipo concreto al realizar una llamada:

```
template<typename T,typename U>
T MyCast(const U u)
{
   return (T)u;
}

void OnStart()
{
   double d = MyCast<double,string>("123.0");
   string f = MyCast<string,double>(123.0);
}
```

Tenga en cuenta que, si los tipos para la plantilla se especifican de forma explícita, entonces esto es necesario para todos los parámetros, aun cuando el segundo parámetro U podría inferirse a partir del argumento pasado.

Después de que el compilador haya generado todas las instancias de la función de plantilla, éstas participan en el procedimiento estándar para elegir la mejor candidata de todas las sobrecargas de funciones con el mismo nombre y el número adecuado de parámetros. De todas las opciones de sobrecarga (incluidas las instancias de plantilla creadas), se selecciona la más cercana en términos de tipos (con el menor número de conversiones).

Si una función de plantilla tiene algunos parámetros de entrada de tipos específicos, sólo se considerará candidata si estos tipos coinciden completamente con los argumentos: cualquier necesidad de conversión hará que la plantilla sea «descartada» como inadecuada.

Las sobrecargas sin plantilla tienen prioridad sobre las sobrecargas con plantilla, las más especializadas («de enfoque estrecho») «ganan» frente a las sobrecargas con plantilla.

Si se especifica explícitamente el argumento de plantilla (tipo), se aplicarán las reglas para la conversión implícita de tipos para el argumento de función correspondiente (valor pasado), si es necesario, si estos tipos difieren.

Si varias variantes de una función coinciden por igual, obtendremos un error de «llamada ambigua a una función sobrecargada con los mismos parámetros».

Por ejemplo, si además de la plantilla *MyCast* se define una función para convertir una cadena en un tipo booleano:

```
bool MyCast(const string u)
{
   return u == "true";
}
```

entonces, al llamar a *MyCast<double,string>("123.0")* se producirá el error indicado, ya que las dos funciones sólo difieren en el valor de retorno:

```
'MyCast<double,string>' - ambiguous call to overloaded function with the same paramet
could be one of 2 function(s)
   double MyCast<double,string>(const string)
   bool MyCast(const string)
```

A la hora de describir funciones de plantilla se recomienda incluir todos los parámetros de la plantilla en los parámetros de la función. Los tipos sólo pueden deducirse de los argumentos, no del valor de retorno.

Si una función tiene un parámetro de tipo de plantilla T con un valor por defecto, y el argumento correspondiente se omite al llamarlo, entonces el compilador también fallará al inferir el tipo de T y mostrará un error de «no se puede aplicar plantilla».

```
class Base
{
public:
   Base(const Base *source = NULL) { }
   static Base *type;
};

static Base* Base::type;

template<typename T>
T *createInstanceFrom(T *origin = NULL)
{
   T *object = new T(origin);
   return object;
}

void OnStart()
{
   Base *p1 = createInstanceFrom();   // error: cannot to apply template
   Base *p2 = createInstanceFrom(Base::type); // ok, auto-detect from argument
   Base *p3 = createInstanceFrom<Base>();     // ok, explicit directive, an argument
}
```

3.3.6 Plantillas de tipos de objetos

Una definición de plantilla de tipo de objeto comienza con un encabezado que contiene parámetros con tipos (véase la sección Encabezado de plantilla) y la definición habitual de una clase, estructura o unión.

```
template <typename T [, typename Ti ...] >
class class_name
{
   ...
};
```

La única diferencia con respecto a la definición estándar es que los parámetros de plantilla pueden aparecer en un bloque de código, en todas las construcciones sintácticas del lenguaje en las que está permitido utilizar un nombre de tipo.

Una vez definida una plantilla, se crean instancias de trabajo de la misma cuando las variables del tipo de plantilla se declaran en el código, especificando los tipos específicos entre paréntesis angulares:

```
ClassName<Type1,Type2> object;
StructName<Type1,Type2,Type3> struct;
ClassName<Type1,Type2> *pointer = new ClassName<Type1,Type2>();
ClassName1<ClassName2<Type>> object;
```

A diferencia de lo que ocurre al llamar a funciones de plantilla, el compilador no es capaz de inferir por sí mismo los tipos reales de las plantillas de objetos.

Declarar una variable de clase/estructura de plantilla no es la única forma de crear una instancia de plantilla. El compilador también genera una instancia si se utiliza un tipo de plantilla como tipo base para otra clase o estructura específica (no de plantilla).

Por ejemplo, la siguiente clase *Worker*, aunque esté vacía, es una implementación de *Base* para el tipo *double*:

```
class Worker : Base<double>
{
};
```

Esta definición mínima es suficiente (con la posibilidad de añadir constructores si la clase *Base* los requiere) para empezar a compilar y validar el código de la plantilla.

En la sección sobre Creación dinámica de objetos nos hemos familiarizado con el concepto de puntero dinámico a un objeto obtenido mediante el operador *new*. Este mecanismo flexible tiene un inconveniente: hay que controlar los punteros y borrarlos «manualmente» cuando ya no se necesitan. En particular, al salir de una función o bloque de código, todos los punteros locales deben borrarse con una llamada *delete*.

Para simplificar la solución a este problema, vamos a crear una clase de plantilla *AutoPtr* (*TemplatesAutoPtr.mq5*, *AutoPtr.mqh*). Su parámetro T se utiliza para describir el campo *ptr*, que almacena un puntero a un objeto de una clase arbitraria. Recibiremos el valor del puntero a través del parámetro del constructor (*T *p*) o en el operador sobrecargado '='. Confiemos el trabajo principal al destructor: en el destructor, el puntero se borrará junto con el objeto *AutoPtr* (para ello se asigna el método del ayudante estático *free*).

El principio de funcionamiento de *AutoPtr* es sencillo: un objeto local de esta clase se destruirá automáticamente al salir del bloque en el que está descrito, y si previamente se le ordenó «seguir» algún puntero, entonces *AutoPtr* también lo liberará.

```
template<typename T>
class AutoPtr
{
private:
   T *ptr;

public:
   AutoPtr() : ptr(NULL) { }

   AutoPtr(T *p) : ptr(p)
   {
      Print(__FUNCSIG__, " ", &this, ": ", ptr);
   }

   AutoPtr(AutoPtr &p)
   {
      Print(__FUNCSIG__, " ", &this, ": ", ptr, " -> ", p.ptr);
      free(ptr);
      ptr = p.ptr;
      p.ptr = NULL;
   }

   ~AutoPtr()
   {
      Print(__FUNCSIG__, " ", &this, ": ", ptr);
      free(ptr);
   }

   T *operator=(T *n)
   {
      Print(__FUNCSIG__, " ", &this, ": ", ptr, " -> ", n);
      free(ptr);
      ptr = n;
      return ptr;
   }

   T* operator[](int x = 0) const
   {
      return ptr;
   }

   static void free(void *p)
   {
      if(CheckPointer(p) == POINTER_DYNAMIC) delete p;
   }
};
```

Además, la clase *AutoPtr* implementa un constructor de copia (más exactamente, un constructor de salto, ya que el objeto actual se convierte en el propietario del puntero), que permite devolver una instancia de *AutoPtr* junto con un puntero controlado desde una función.

Para probar el rendimiento de *AutoPtr* vamos a describir una clase *Dummy* ficticia.

```
class Dummy
{
   int x;
public:
   Dummy(int i) : x(i)
   {
      Print(__FUNCSIG__, " ", &this);
   }
   ...
   int value() const
   {
      return x;
   }
};
```

En el script, en la función *OnStart*, introduzca la variable *AutoPtr<Dummy>* y obtenga su valor de la función *generator*. En la propia función *generator* describiremos también el objeto *AutoPtr<Dummy>* y crearemos y le «adjuntaremos» secuencialmente dos objetos dinámicos *Dummy* (para comprobar la correcta liberación de memoria del objeto «antiguo»).

```
AutoPtr<Dummy> generator()
{
   AutoPtr<Dummy> ptr(new Dummy(1));
   // pointer to 1 will be freed after execution of '='
   ptr = new Dummy(2);
   return ptr;
}

void OnStart()
{
   AutoPtr<Dummy> ptr = generator();
   Print(ptr[].value());              // 2
}
```

Dado que todos los métodos principales registran descriptores de objetos (tanto *AutoPtr* como punteros controlados *ptr*), podemos realizar un seguimiento de todas las «transformaciones» de punteros (por comodidad, todas las líneas están numeradas).

```
01 Dummy::Dummy(int) 3145728
02  AutoPtr<Dummy>::AutoPtr<Dummy>(Dummy*) 2097152: 3145728
03  Dummy::Dummy(int) 4194304
04  Dummy*AutoPtr<Dummy>::operator=(Dummy*) 2097152: 3145728 -> 4194304
05  Dummy::~Dummy() 3145728
06  AutoPtr<Dummy>::AutoPtr<Dummy>(AutoPtr<Dummy>&) 5242880: 0 -> 4194304
07  AutoPtr<Dummy>::~AutoPtr<Dummy>() 2097152: 0
08  AutoPtr<Dummy>::AutoPtr<Dummy>(AutoPtr<Dummy>&) 1048576: 0 -> 4194304
09  AutoPtr<Dummy>::~AutoPtr<Dummy>() 5242880: 0
10  2
11  AutoPtr<Dummy>::~AutoPtr<Dummy>() 1048576: 4194304
12  Dummy::~Dummy() 4194304
```

Dejemos por un momento de lado las plantillas y describamos en detalle cómo funciona la utilidad, porque una clase así puede ser muy útil.

Inmediatamente después de iniciar *OnStart* se llama al generador de funciones. Éste debe devolver un valor para inicializar el objeto *AutoPtr* de *OnStart*, y por lo tanto su constructor aún no ha sido invocado. La línea 02 crea un objeto AutoPtr#2097152 dentro de la función *generator* y obtiene un puntero al primer Dummy#3145728. A continuación, se crea una segunda instancia de Dummy#4194304 (línea 03) que sustituye a la copia anterior con el descriptor 3145728 (línea 04) en AutoPtr#2097152, y se borra la copia antigua (línea 05). La línea 06 crea un AutoPtr#5242880 temporal para devolver el valor del *generator*, y borra el local (07). En la línea 08 se utiliza finalmente el constructor de copia para el objeto AutoPtr#1048576 en la función *OnStart* y se le transfiere el puntero del objeto temporal (que se borra inmediatamente en la línea 09). A continuación, llamamos a *Print* con el contenido del puntero. Cuando *OnStart* finaliza, el destructor *AutoPtr* (11) se dispara automáticamente, haciendo que también borremos el objeto de trabajo *Dummy* (12).

La tecnología de plantillas convierte la clase *AutoPtr* en un gestor parametrizado de objetos asignados dinámicamente. Pero como *AutoPtr* tiene un campo *T *ptr*, sólo se aplica a las clases (más concretamente, a los punteros a objetos de clase). Por ejemplo, intentar crear una instancia de una plantilla para una cadena (*AutoPtr<string> s*) dará como resultado un montón de errores en el texto de la plantilla, cuyo significado es que el tipo *string* no admite punteros.

Esto no es un problema aquí, ya que el propósito de esta plantilla se limita a las clases, pero es un matiz que debe tenerse en cuenta para plantillas más generales (véase la barra lateral).

Punteros y referencias

Tenga en cuenta que la construcción *T** no puede aparecer en plantillas que tenga previsto utilizar, incluidas las de estructuras o tipos integrados. La cuestión es que en MQL5, los punteros sólo se permiten para las clases. Esto no quiere decir que, en teoría, no se pueda escribir una plantilla que se aplique tanto a los tipos integrados como a los definidos por el usuario, pero puede requerir algunos ajustes. Probablemente será necesario abandonar parte de la funcionalidad o sacrificar un nivel de genericidad de la plantilla (hacer varias plantillas en lugar de una, sobrecargar funciones, etc.).

La forma más directa de «inyectar» un tipo de puntero en una plantilla es incluir el modificador '*' junto con el tipo real cuando se crea una instancia de la plantilla (es decir, debe coincidir con *T=Type**). Sin embargo, algunas funciones (como *CheckPointer*), operadores (como *delete*) y construcciones sintácticas (como conversión *((T)variable)*) son sensibles a si sus argumentos u operandos son punteros o no. Por ello, el mismo texto de plantilla no siempre es sintácticamente correcto tanto para punteros como para valores de tipo simple.

Otra diferencia significativa entre tipos que hay que tener en cuenta es que los objetos se pasan a los métodos sólo por referencia, pero los literales (constantes) de tipos simples no pueden pasarse por referencia. Debido a ello, la presencia o ausencia de un ampersand puede ser tratada como un error por el compilador, dependiendo del tipo inferido de T. Como una de las «soluciones», puede optar por «envolver» constantes de argumento en objetos o variables.

Otro truco consiste en utilizar métodos de plantilla. Lo veremos en la siguiente sección.

Cabe señalar que las técnicas orientadas a objetos combinan bien con los patrones. Dado que un puntero a una clase base puede utilizarse para almacenar un objeto de una clase derivada, *AutoPtr* es aplicable a objetos de cualquier clase derivada *Dummy*.

En teoría, este enfoque «híbrido» se utiliza ampliamente en las clases contenedoras (vector, cola, mapa, lista, etc.), que, por regla general, son plantillas. Las clases contenedoras pueden, dependiendo de la implementación, imponer requisitos adicionales al parámetro de plantilla; en concreto, que el tipo inline debe tener un constructor de copia y un operador de asignación (copia).

La biblioteca estándar MQL5 suministrada con MetaTrader 5 contiene muchas plantillas ya preparadas de esta serie: *Stack.mqh*, *Queue.mqh*, *HashMap.mqh*, *LinkedList.mqh*, *RedBlackTree.mqh,* y otras. Todas ellas se encuentran en el directorio MQL5/Include/Generic. Es cierto que no proporcionan control sobre objetos dinámicos (punteros).

Veremos nuestro propio ejemplo de una sencilla clase contenedora en Plantillas de métodos.

3.3.7 Plantillas de métodos

No sólo un tipo de objeto en su conjunto puede ser una plantilla, sino que su método por separado, simple o estático, también puede ser una plantilla. La excepción son los métodos virtuales: estos no pueden convertirse en plantillas. De ello se deduce que los métodos de plantilla no pueden declararse dentro de interfaces. No obstante, las propias interfaces pueden convertirse en plantillas, y los métodos virtuales pueden estar presentes en las plantillas de las clases.

Cuando una plantilla de método está contenida dentro de una plantilla de clase o estructura, los parámetros de ambas plantillas deben ser diferentes. Si hay varios métodos de plantilla, sus parámetros no están relacionados de ninguna manera y pueden tener el mismo nombre.

Una plantilla de método se declara de forma similar a una plantilla de funciones, pero sólo en el contexto de una clase, estructura o unión (que pueden o no ser plantillas).

```
[ template < typename T [, typename Ti ...] > ]
class class_name
{
   ...
   template < typename U [, typename Ui ...] >
  type method_name(parameters_with_types_T_and_U)
   {
   }
};
```

Los parámetros, el valor de retorno y el cuerpo del método pueden utilizar los tipos T (general para una clase) y U (específico para un método).

Una instancia de un método para una combinación específica de parámetros se genera sólo cuando es invocada en el código del programa.

En la sección anterior describimos la clase de plantilla *AutoPtr* para almacenar y liberar un único puntero. Cuando hay muchos punteros del mismo tipo es conveniente ponerlos en un objeto contenedor. Vamos a crear una plantilla simple con una funcionalidad similar: la clase *SimpleArray* (*SimpleArray.mqh*). Con el fin de no duplicar la funcionalidad para controlar la liberación de memoria dinámica, pondremos en el contrato de la clase que está pensada para almacenar valores y objetos, pero no punteros. Para almacenar los punteros, los colocaremos en objetos *AutoPtr*, y éstos en el contenedor.

Esto tiene otro efecto positivo: como el objeto *AutoPtr* es pequeño, es fácil de copiar (sin gastar demasiados recursos en ello), lo que suele ocurrir cuando se intercambian datos entre funciones. Los

objetos de esas clases de aplicación a los que apunta *AutoPtr* pueden ser grandes, y ni siquiera es necesario implementar en ellos su propio constructor de copia.

Por supuesto, es más barato devolver punteros desde las funciones, pero entonces hay que reinventar los medios de control de liberación de memoria. Por lo tanto, es más fácil utilizar una solución ya preparada en forma de *AutoPtr*.

Para los objetos de dentro del contenedor, crearemos el array *data* del tipo con plantilla T.

```
template<typename T>
class SimpleArray
{
protected:
   T data[];
   ...
```

Dado que una de las principales operaciones de un contenedor es añadir un elemento, vamos a proporcionar una función de ayuda para ampliar el array.

```
   int expand()
   {
      const int n = ArraySize(data);
      ArrayResize(data, n + 1);
      return n;
   }
```

Añadiremos elementos directamente a través del operador sobrecargado '<<'. Utiliza el parámetro genérico de plantilla T.

```
public:
   SimpleArray *operator<<(const T &r)
   {
      data[expand()] = (T)r;
      return &this;
   }
```

Esta opción toma un valor por referencia, es decir, una variable o un objeto. Por ahora, debe prestar atención a esto, y en unos instantes se verá por qué es importante.

La lectura de elementos se realiza sobrecargando el operador '[]' (tiene la precedencia más alta y, por tanto, no requiere el uso de paréntesis en las expresiones).

```
   T operator[](int i) const
   {
      return data[i];
   }
```

Primero, asegurémonos de que la clase funciona en el ejemplo de la estructura.

```
struct Properties
{
   int x;
   string s;
};
```

Para ello, describiremos un contenedor para la estructura en la función *OnStart* y colocaremos en él un objeto (*TemplatesSimpleArray.mq5*).

```
void OnStart()
{
   SimpleArray<Properties> arrayStructs;
   Properties prop = {12345, "abc"};
   arrayStructs << prop;
   Print(arrayStructs[0].x, " ", arrayStructs[0].s);
   ...
}
```

El registro de depuración permite verificar que la estructura se encuentra en un contenedor.

Ahora vamos a intentar almacenar algunos números en el contenedor.

```
SimpleArray<double> arrayNumbers;
arrayNumbers << 1.0 << 2.0 << 3.0;
```

Por desgracia, obtendremos errores de «parámetro pasado como referencia, variable esperada», que se producen exactamente en el operador sobrecargado '<<'.

Necesitamos una sobrecarga con paso de parámetros por valor. Sin embargo, no podemos escribir simplemente un método similar que no tenga *const* y '&':

```
SimpleArray *operator<<(T r)
{
   data[expand()] = (T)r;
   return &this;
}
```

Si lo hace, la nueva variante dará lugar a una plantilla no compilable para tipos de objeto: al fin y al cabo, los objetos sólo deben pasarse por referencia. Aunque la función no se utilice para objetos, sigue estando presente en la clase. Por lo tanto, definiremos el nuevo método como una plantilla con su propio parámetro.

```
template<typename T>
class SimpleArray
{
   ...
   template<typename U>
   SimpleArray *operator<<(U u)
   {
      data[expand()] = (T)u;
      return &this;
   }
```

Aparecerá en la clase sólo si se pasa algo por valor al operador '<<', lo que significa que definitivamente no es un objeto. Es cierto que no podemos garantizar que T y U sean iguales, así que se

realiza una conversión explícita *(T)u*. Para los tipos integrados (si los dos tipos no coinciden), en algunas combinaciones, la conversión con pérdida de precisión es posible, pero el código se compilará con seguridad. La única excepción es la prohibición de convertir una cadena en un tipo booleano, pero es poco probable que el contenedor se utilice para el array *bool*, por lo que esta restricción no es significativa. Quienes lo deseen pueden resolver este problema.

Con el nuevo método de plantilla, el contenedor *SimpleArray<double>* funciona como se esperaba y no entra en conflicto con *SimpleArray<Properties>* porque las dos instancias de plantilla tienen diferencias en el código fuente generado.

Por último, vamos a comprobar el contenedor con objetos *AutoPtr*. Para ello, prepararemos una clase sencilla *Dummy* que «suministrará» objetos para punteros dentro de *AutoPtr*.

```
class Dummy
{
   int x;
public:
   Dummy(int i) : x(i) { }
   int value() const
   {
      return x;
   }
};
```

Dentro de la función *OnStart* vamos a crear un contenedor *SimpleArray<AutoPtr<Dummy>>* y a llenarlo.

```
void OnStart()
{
   SimpleArray<AutoPtr<Dummy>> arrayObjects;
   AutoPtr<Dummy> ptr = new Dummy(20);
   arrayObjects << ptr;
   arrayObjects << AutoPtr<Dummy>(new Dummy(30));
   Print(arrayObjects[0][].value());
   Print(arrayObjects[1][].value());
}
```

Recuerde que en *AutoPtr* se utiliza el operador '[]' para devolver un puntero almacenado, por lo que *arrayObjects[0][]* significa: devolver el elemento 0 del array *data* en *SimpleArray*, es decir, el objeto *AutoPtr*, y a continuación se aplica el segundo par de corchetes al volumen, lo que da lugar a un puntero Dummy*. Seguidamente, podemos trabajar con todas las propiedades de este objeto: en este caso, recuperamos el valor actual del campo x.

Dado que *Dummy* no dispone de un constructor de copia, no puede utilizar un contenedor para almacenar estos objetos directamente sin *AutoPtr*.

```
// ERROR:
// object of 'Dummy' cannot be returned,
// copy constructor 'Dummy::Dummy(const Dummy &)' not found
SimpleArray<Dummy> bad;
```

No obstante, un usuario ingenioso puede averiguar cómo evitarlo.

```
SimpleArray<Dummy*> bad;
bad << new Dummy(0);
```

Este código se compilará y ejecutará, pero esta «solución» encierra un problema: *SimpleArray* no sabe controlar los punteros y, por lo tanto, cuando el programa sale, se detecta una fuga de memoria.

```
1 undeleted objects left
1 object of type Dummy left
24 bytes of leaked memory
```

Nosotros, como desarrolladores de *SimpleArray*, tenemos el deber de llenar ese vacío. Para ello, vamos a añadir otro método de plantilla a la clase con una sobrecarga del operador '<<', esta vez para punteros. Dado que se trata de una plantilla, sólo se incluye en el código fuente resultante «a demanda»: cuando el programador intenta utilizar esta sobrecarga, es decir, escribir un puntero al contenedor. En caso contrario, el método se ignora.

```
template<typename T>
class SimpleArray
{
   ...
   template<typename P>
   SimpleArray *operator<<(P *p)
   {
      data[expand()] = (T)*p;
      if(CheckPointer(p) == POINTER_DYNAMIC) delete p;
      return &this;
   }
```

Esta especialización arroja un error de compilación («puntero de objeto esperado») al crear una instancia de plantilla con un tipo de puntero. Por lo tanto, informamos al usuario de que este modo no es compatible.

```
SimpleArray<Dummy*> bad; // ERROR is generated in SimpleArray.mqh
```

Además, realiza otra acción protectora. Si la clase cliente todavía tiene un constructor de copia, entonces guardar objetos asignados dinámicamente en el contenedor ya no provocará una fuga de memoria: una copia del objeto en el puntero pasado *P *p* permanece en el contenedor, y el original se elimina. Cuando el contenedor se destruye al final de la función *OnStart*, su array interno *data* llamará automáticamente a los destructores para sus elementos.

```
void OnStart()
{
   ...
   SimpleArray<Dummy> good;
   good << new Dummy(0);
} // SimpleArray "cleans" its elements
  // no forgotten objects in memory
```

Las plantillas de métodos y los métodos «simples» pueden definirse fuera del bloque de la clase principal (o plantilla de clase), de forma similar a lo que vimos en el método Dividir declaración y definición de clase. Al mismo tiempo, todas van precedidas por el encabezado de la plantilla (*TemplatesExtended.mq5*):

```
template<typename T>
class ClassType
{
   ClassType() // private constructor
   {
      s = &this;
   }
   static ClassType *s; // object pointer (if it was created)
public:
   static ClassType *create() // creation (on first call only)
   {
      static ClassType single; //single pattern for every T
      return single;
   }

   static ClassType *check() // checking pointer without creating
   {
      return s;
   }

   template<typename U>
   void method(const U &u);
};

template<typename T>
template<typename U>
void ClassType::method(const U &u)
{
   Print(__FUNCSIG__, " ", typename(T), " ", typename(U));
}

template<typename T>
static ClassType<T> *ClassType::s = NULL;
```

También muestra la inicialización de una variable estática con plantilla, lo que denota el patrón de diseño singleton.

En la función *OnStart*, cree una instancia de la plantilla y pruébela:

```
void OnStart()
{
   ClassType<string> *object = ClassType<string>::create();
   double d = 5.0;
   object.method(d);
   // OUTPUT:
   // void ClassType<string>::method<double>(const double&) string double

   Print(ClassType<string>::check()); // 1048576 (an example of an instance id)
    Print(ClassType<long>::check());   // 0 (there is no instance for T=long)
}
```

3.3.8 Plantillas anidadas

Las plantillas pueden anidarse dentro de clases/estructuras o dentro de otras plantillas de clases/estructuras. Lo mismo ocurre con las uniones.

En la sección Uniones vimos la posibilidad de «convertir» los valores *long* en *double* y viceversa sin pérdida de precisión.

Ahora podemos utilizar plantillas para escribir un «conversor» universal (*TemplatesConverter.mq5*). La clase de plantilla *Converter* tiene dos parámetros T1 y T2 que indican los tipos entre los que se realizará la conversión. Para escribir un valor según las reglas de un tipo y leerlo según las reglas de otro necesitamos de nuevo una unión. Lo convertiremos también en una plantilla (*DataOverlay*) con los parámetros U1 y U2, y lo definiremos dentro de la clase.

La clase proporciona una práctica transformación sobrecargando los operadores [], en cuya implementación se escriben y leen los campos de unión.

```
template<typename T1,typename T2>
class Converter
{
private:
   template<typename U1,typename U2>
   union DataOverlay
   {
      U1 L;
      U2 D;
   };

   DataOverlay<T1,T2> data;

public:
   T2 operator[](const T1 L)
   {
      data.L = L;
      return data.D;
   }

   T1 operator[](const T2 D)
   {
      data.D = D;
      return data.L;
   }
};
```

La unión se utiliza para describir el campo *DataOverlay<T1,T2>data* de dentro de la clase. Podríamos utilizar T1 y T2 directamente en *DataOverlay* y no hacer de esta unión una plantilla, pero para demostrar la técnica en sí, los parámetros de la plantilla externa se pasan a la plantilla interna cuando se genera el campo *data*. Dentro de *DataOverlay*, el mismo par de tipos se conocerá como U1 y U2 (además de T1 y T2).

Veamos la plantilla en acción.

```
#define MAX_LONG_IN_DOUBLE       9007199254740992

void OnStart()
{
   Converter<double,ulong> c;

   const ulong value = MAX_LONG_IN_DOUBLE + 1;

   double d = value; // possible loss of data due to type conversion
   ulong result = d; // possible loss of data due to type conversion

   Print(value == result); // false

   double z = c[value];
   ulong restored = c[z];

   Print(value == restored); // true
}
```

3.3.9 Ausencia de especialización de plantillas

En algunos casos, puede ser necesario proporcionar una implementación de plantilla para un tipo particular (o conjunto de tipos) de forma que difiera de la genérica. Por ejemplo, suele tener sentido preparar una versión especial de la función swap para punteros o arrays. En tales casos, C++ permite realizar lo que se denomina especialización de plantilla, es decir, definir una versión de la plantilla en la que el parámetro de tipo genérico T se sustituye por el tipo concreto requerido.

A la hora de especializar plantillas de funciones y métodos, deben especificarse tipos concretos para todos los parámetros. Esto se denomina especialización completa.

En el caso de las plantillas de tipos de objetos de C++, la especialización puede ser no sólo completa, sino también parcial: especifica el tipo de sólo algunos de los parámetros (y el resto se inferirá o especificará cuando se creen instancias de la plantilla). Puede haber varias especializaciones parciales: la única condición para ello es que cada especialización describa una combinación única de tipos.

Por desgracia, en MQL5 no existe la especialización en el sentido estricto de la palabra.

La especialización de funciones de plantilla no es diferente de la sobrecarga. Por ejemplo, dada la siguiente plantilla *func*:

```
template<typename T>
void func(T t) { ... }
```

está permitido proporcionar su implementación personalizada para un tipo determinado (como *string*) en uno de los formularios:

```
// explicit specialization
template<>
void func(string t) { ... }
```

o bien,

```
// normal overload
void func(string t) { ... }
```

Sólo debe seleccionarse uno de los formularios. De lo contrario, obtendremos un error de compilación «'func' - la función ya se ha definido y tiene cuerpo».

En cuanto a la especialización de las clases, la herencia a partir de plantillas con indicación de tipos específicos para algunos de los parámetros de la plantilla puede considerarse equivalente a su especialización parcial. Los métodos de plantilla se pueden sobrescribir (overridden) en una clase derivada.

En el siguiente ejemplo (*TemplatesExtended.mq5*) se muestran varias opciones para utilizar parámetros de plantilla como tipos padre, incluidos casos en los que uno de ellos se explicita como específico.

```
#define RTTI Print(typename(this))

class Base
{
public:
   Base() { RTTI; }
};

template<typename T>
class Derived : public T
{
public:
   Derived() { RTTI; }
};

template<typename T>
class Base1
{
   Derived<T> object;
public:
   Base1() { RTTI; }
};

template<typename T>                // complete "specialization"
class Derived1 : public Base1<Base> // 1 of 1 parameter is set
{
public:
   Derived1() { RTTI; }
};

template<typename T,typename E>
class Base2 : public T
{
public:
   Base2() { RTTI; }
};

template<typename T>                    // partial "specialization"
class Derived2 : public Base2<T,string> // 1 of 2 parameters is set
{
public:
   Derived2() { RTTI; }
};
```

Proporcionaremos una creación de instancia de un objeto según una plantilla utilizando una variable:

```
Derived2<Derived1<Base>> derived2;
```

El registro de tipo de depuración utilizando la macro RTTI produce el siguiente resultado:

```
Base
Derived<Base>
Base1<Base>
Derived1<Base>
Base2<Derived1<Base>,string>
Derived2<Derived1<Base>>
```

Cuando se desarrollan bibliotecas que vienen como binarios cerrados, debe asegurarse de que se crean explícitamente instancias de plantillas para todos los tipos con los que se espera que trabajen los futuros usuarios de la biblioteca. Puede hacer esto llamando explícitamente a plantillas de funciones y creando objetos con parámetros de tipo en alguna función auxiliar, por ejemplo, ligada a la inicialización de una variable global.

Parte 4. API de MQL5 comunes

En anteriores partes del libro hemos estudiado los conceptos básicos, la sintaxis y las reglas de uso de las construcciones del lenguaje MQL5. Sin embargo, esto es sólo una base para escribir programas reales que satisfagan los requisitos de los operadores de trading, como el tratamiento analítico de datos y el trading automático. La resolución de tales tareas no sería posible sin una amplia gama de funciones integradas y medios de interacción con el terminal MetaTrader 5, los cuales conforman la API de MQL5.

En este capítulo comenzaremos a dominar la API de MQL5 y seguiremos haciéndolo hasta el final del libro, familiarizándonos gradualmente con todos los subsistemas especializados.

La lista de tecnologías y capacidades proporcionadas a cualquier programa MQL por el núcleo (el entorno de ejecución de los programas MQL dentro del terminal) es muy amplia. Por eso tiene sentido empezar por las cosas más sencillas que pueden ser útiles en la mayoría de los programas. En concreto, aquí veremos funciones especializadas para trabajar con arrays, cadenas, archivos, transformación de datos, interacción con el usuario, funciones matemáticas y control del entorno.

Anteriormente aprendimos a describir nuestras funciones en MQL5 y a llamarlas. Las funciones integradas de la API de MQL5 están disponibles en el código fuente, como se suele decir, «out of the box», es decir, sin ninguna descripción preliminar.

Es importante señalar que, a diferencia de lo que ocurre en C++, no se necesitan directivas de preprocesador adicionales para incluir un conjunto específico de funciones integradas en un programa. Los nombres de todas las funciones de la API de MQL5 están presentes en el contexto global (espacio de nombres), siempre y sin condiciones.

Por un lado, esto resulta conveniente, pero por otro, exige que usted sea consciente de la posible existencia de un conflicto de nombres. Si accidentalmente intenta utilizar alguno de los nombres de las funciones integradas, ello anulará la implementación estándar, lo que puede tener consecuencias inesperadas: en el mejor de los casos, obtendrá un error del compilador sobre una sobrecarga ambigua, y en el peor, todas las llamadas habituales serán redirigidas a la «nueva» implementación, sin ninguna advertencia.

En teoría, se pueden utilizar nombres similares en otros contextos; por ejemplo, como nombre de método de una clase o en un espacio de nombres dedicado (de usuario). En tales casos, la llamada a una función global puede realizarse utilizando el operador de resolución de contexto: ya hemos abordado esta situación en la sección Tipos anidados, espacios de nombres y operador de contexto '::'.

Programación en MQL5 para Traders: códigos fuente del libro. Parte 4

Los ejemplos del libro también están disponibles en el proyecto público \MQL5\Shared Projects\MQL5Book

4.1 Conversiones de tipos integrados

Los programas a menudo operan con distintos tipos de datos. Ya hemos encontrado mecanismos de conversión explícita e implícita de tipos integrados en la sección Conversión de tipos. Proporcionan métodos de conversión universales que no siempre son adecuados, por una razón u otra. La API de MQL5 proporciona un conjunto de funciones de conversión con las cuales un programador puede gestionar las conversiones de datos de un tipo a otro y configurar los resultados de la conversión.

Entre las funciones más utilizadas están las que convierten varios tipos en cadenas o viceversa. En concreto, ello incluye conversiones de números, fechas y horas, colores, estructuras y enums. Algunos tipos tienen operaciones específicas adicionales.

En esta sección se analizan varios métodos de conversión de datos, proporcionando a los programadores las herramientas necesarias para trabajar con una variedad de tipos de datos en los robots de trading. Se incluyen las siguientes subsecciones:

De números a cadenas y viceversa:

- En esta subsección se exploran métodos para convertir valores numéricos en cadenas y viceversa. Abarca aspectos importantes tales como el formato de los números y el manejo de diversos sistemas numéricos.

Normalización de doubles:

- La normalización de los números de tipo double es un aspecto importante cuando se trabaja con datos financieros. En esta sección se tratan los métodos de normalización, las formas de evitar la pérdida de precisión y el procesamiento de valores de punto flotante.

Fecha y hora:

- La conversión de fecha y hora desempeña un papel fundamental en las estrategias de trading. En esta subsección se abordan métodos para trabajar con fechas, intervalos de tiempo y tipos de datos especiales como datetime.

Color:

- En MQL5, los colores se representan mediante un tipo de datos especial. En esta subsección se examina la conversión de los valores de color, su representación y su uso en los elementos gráficos de los robots de trading.

Estructuras:

- La conversión de datos dentro de estructuras es un tema importante cuando se trabaja con datos estructurados complejos. Veremos métodos para interactuar con las estructuras y sus elementos.

Enumeraciones:

- Las enumeraciones proporcionan constantes con nombre y mejoran la legibilidad del código. En esta subsección se explica cómo convertir valores de enumeración y utilizarlos eficazmente en un programa.

Tipo complejo:

- El tipo complejo está diseñado para trabajar con números complejos. En esta sección se estudian métodos para convertir y utilizar números complejos.

En este capítulo estudiaremos todas estas funciones.

4.1.1 De números a cadenas y viceversa

La conversión de números a cadenas y viceversa, de cadenas a números, puede realizarse mediante el operador conversión explícita de tipos. Por ejemplo, para los tipos *double* y *string*, podría tener este aspecto:

```
double number = (double)text;
string text = (string)number;
```

Las cadenas pueden convertirse a otros tipos numéricos, como *float*, *long*, *int*, etc.

Observe que la conversión a un tipo real *(float)* proporciona menos dígitos significativos, lo que en algunas aplicaciones puede considerarse una ventaja, ya que proporciona una representación más compacta y fácil de leer.

En términos estrictos, esta conversión de tipos no es obligatoria, ya que, aunque no exista ningún operador de conversión explícito, el compilador producirá la conversión de tipos de forma implícita. No obstante, en este caso recibirá una advertencia del compilador, por lo que se recomienda hacer siempre explícitas las conversiones de tipos.

La API de MQL5 proporciona algunas otras funciones útiles, que se describen a continuación. Las descripciones van seguidas de un ejemplo general.

double StringToDouble(string text)

La función *StringToDouble* convierte una cadena en un número *double*.

Esto es completamente análogo a la conversión de tipos a (*double*). En realidad, su finalidad práctica se limita a preservar la compatibilidad con los códigos fuente heredados. El método preferido es la conversión de tipos, ya que es más compacta y se implementa dentro de la sintaxis del lenguaje.

Según el proceso de conversión, una cadena debe contener una secuencia de caracteres que cumplan las reglas de escritura de literales de tipo numérico (tanto float como integer). En particular, una cadena puede comenzar con un signo '+' o '-' , seguido de un dígito, y puede continuar como una secuencia de dígitos.

Los números reales pueden contener un único carácter de punto '.' que separa la parte fraccionaria y un exponente opcional con el siguiente formato: carácter 'e' o 'E' seguido de una secuencia de dígitos para el grado (también puede ir precedido de un '+' o '-').

Para los números enteros se admite la notación hexadecimal, es decir, el prefijo «0x» puede ir seguido no sólo de dígitos decimales, sino también de 'A', 'B', 'C', 'D', 'E', 'F' (en cualquier posición).

Cuando se encuentra en la cadena cualquier carácter no esperado (como una letra, un signo de puntuación, un segundo punto o un espacio intermedio), la conversión finaliza. En este caso, si había caracteres permitidos antes de esta posición, se interpretan como un número, y si no, el resultado será 0.

Los caracteres vacíos iniciales (espacios, tabuladores, nuevas líneas) se omiten y no afectan a la conversión. Si van seguidos de números y otros caracteres que cumplan las normas, el número se recibirá correctamente.

En la siguiente tabla se ofrecen algunos ejemplos de conversiones válidas con explicaciones.

cadena	double	Resultado
"123.45"	123.45	Un punto decimal
"\t 123"	123.0	Los espacios en blanco al principio no se tienen en cuenta.
"-12345"	-12345.0	Un número con signo

cadena	double	Resultado
"123e-5"	0.00123	Notación científica con exponente
"0x12345"	74565.0	Notación hexadecimal

En la siguiente tabla se muestran ejemplos de conversiones incorrectas.

cadena	double	Resultado
"x12345"	0.0	Empieza por un carácter no resuelto (letra)
"123x45"	123.0	La letra después de 123 rompe la conversión
" 12 3"	12.0	El espacio después de 12 rompe la conversión
"123.4.5"	123.4	El segundo punto decimal después de 123.4 rompe la conversión
"1,234.50"	1.0	La coma después de 1 rompe la conversión
"-+12345"	0.0	Demasiados signos (dos)

string DoubleToString(double number, int digits = 8)

La función *DoubleToString* convierte un número en una cadena con la precisión especificada (número de dígitos de -16 a 16).

Realiza un trabajo similar al de convertir un número a (*string*) pero permite elegir, mediante el segundo parámetro, la precisión numérica de la cadena resultante.

El operador *(string)* aplicado a *double* muestra 16 dígitos significativos (total, incluyendo mantisa y parte fraccionaria). El equivalente completo de esto no puede conseguirse con una función.

Si el parámetro *digits* es mayor o igual que 0, indica el número de decimales. En este caso, el número de caracteres antes de la marca decimal viene determinado por el propio número (lo grande que es), y si el número total de caracteres en la mantisa y el indicado en *digits* resulta ser mayor que 16, entonces los dígitos menos significativos contendrán «basura» (debido a cómo se almacenan los números reales). 16 caracteres representan la precisión media máxima para el tipo *double*, es decir, si se establece *digits* en 16 (máximo), sólo se obtendrá una representación precisa de los valores inferiores a 10.

Si el parámetro *digits* es menor que 0, especifica el número de dígitos significativos, y este número se mostrará en formato científico con un exponente. En términos de precisión (pero no de formato de grabación), la configuración de *digits=-16* en la función genera un resultado cercano a la conversión *(string)*.

La función, por regla general, se utiliza para dar un formato uniforme a los conjuntos de datos (incluida la alineación a la derecha de una columna de una tabla determinada), en los que los valores tienen la misma precisión decimal (por ejemplo, el número de decimales del precio de un instrumento financiero o el tamaño de un lote).

> Tenga en cuenta que pueden producirse errores durante los cálculos matemáticos, lo que provocaría que el resultado no sea un número válido aunque tenga el tipo *double* (o *float*). Por ejemplo, una variable podría contener el resultado de calcular la raíz cuadrada de un número

negativo.

Tales valores se denominan «Not a Number» (NaN, «No es un número») y se muestran al convertirlos a *(string)* como una breve indicación del tipo de error, por ejemplo, -nan(ind) (ind - indefinido), nan(inf) (inf - infinito). Al utilizar la función *DoubleToString* obtendrá un número grande que no tiene sentido.

Es especialmente importante que todos los cálculos posteriores con NaN también den NaN. Para comprobar estos valores, existe la función *MathIsValidNumber*.

long StringToInteger(string text)

La función convierte una cadena en un número del tipo *long*. Observe que el tipo de resultado es definitivamente *long*, y no *int* (a pesar del nombre) y no *ulong*.

Una forma alternativa es convertir el tipo utilizando el operador *(long)*. Además, se puede utilizar cualquier otro tipo de entero de su elección para la conversión:*(int)*, *(uint)*, *(ulong)*, etc.

Las reglas de conversión son similares a las del tipo *double*, pero excluyen el carácter de punto y el exponente de los caracteres permitidos.

string IntegerToString(long number, int length = 0, ushort filling = ' ')

La función *IntegerToString* convierte un entero del tipo *long* en una cadena de la longitud especificada. Si la representación numérica ocupa menos de un carácter, se rellena a la izquierda con un carácter *filling* (con un espacio por defecto). De lo contrario, el número se muestra en su totalidad, sin restricciones. Llamar a una función con parámetros por defecto equivale a hacer una conversión a *(string)*.

Por supuesto, los tipos enteros más pequeños (por ejemplo, *int*, *short*) serán procesados por la función sin problemas.

En el script *ConversionNumbers.mq5* se ofrecen ejemplos de uso de todas las funciones anteriores.

```
void OnStart()
{
   const string text = "123.4567890123456789";
   const string message = "-123e-5 buckazoid";
   const double number = 123.4567890123456789;
   const double exponent = 1.234567890123456789e-5;

   // type casting
   Print((double)text);    // 123.4567890123457
   Print((double)message); // -0.00123
   Print((string)number);  // 123.4567890123457
   Print((string)exponent);// 1.234567890123457e-05
   Print((long)text);      // 123
   Print((long)message);   // -123

   // converting with functions
   Print(StringToDouble(text)); // 123.4567890123457
   Print(StringToDouble(message)); // -0.00123

   // by default, 8 decimal digits
   Print(DoubleToString(number)); // 123.45678901

   // custom precision
   Print(DoubleToString(number, 5));  // 123.45679
   Print(DoubleToString(number, -5)); // 1.23457e+02
   Print(DoubleToString(number, -16));// 1.2345678901234568e+02
   Print(DoubleToString(number, 16)); // 123.4567890123456807
   // last 2 digits are not accurate!
   Print(MathSqrt(-1.0));                  // -nan(ind)
   Print(DoubleToString(MathSqrt(-1.0))); // 9223372129088496176.54775808

   Print(StringToInteger(text));      // 123
   Print(StringToInteger(message));   // -123

   Print(IntegerToString(INT_MAX));         // '2147483647'
   Print(IntegerToString(INT_MAX, 5));      // '2147483647'
   Print(IntegerToString(INT_MAX, 16));     // '      2147483647'
   Print(IntegerToString(INT_MAX, 16, '0'));// '0000002147483647'
}
```

4.1.2 Normalización de doubles

La API de MQL5 proporciona una función para redondear números de coma flotante a una precisión especificada (el número de dígitos significativos en la parte fraccionaria).

double NormalizeDouble(double number, int digits)

El redondeo es necesario en los algoritmos de trading para fijar volúmenes y precios en órdenes. El redondeo se realiza según las reglas estándar: el último dígito visible se incrementa en 1 si el siguiente dígito (descartado) es mayor o igual que 5.

Valores válidos del parámetro *digits*: 0 a 8.

Encontrará ejemplos de utilización de la función en el archivo *ConversionNormal.mq5*.

```
void OnStart()
{
   Print(M_PI);                      // 3.141592653589793
   Print(NormalizeDouble(M_PI, 16)); // 3.14159265359
   Print(NormalizeDouble(M_PI, 8));  // 3.14159265
   Print(NormalizeDouble(M_PI, 5));  // 3.14159
   Print(NormalizeDouble(M_PI, 1));  // 3.1
   Print(NormalizeDouble(M_PI, -1)); // 3.14159265359
   ...
```

Debido al hecho de que cualquier número real tiene una precisión de representación interna, el número puede mostrarse de forma aproximada aunque esté normalizado:

```
   ...
   Print(512.06);                    // 512.0599999999999
   Print(NormalizeDouble(512.06, 5));// 512.0599999999999
   Print(DoubleToString(512.06, 5)); // 512.06000000
   Print((float)512.06);             // 512.06
}
```

Esto es normal e inevitable. Para un formato más compacto, utilice las funciones *DoubleToString*, *StringFormat* o conversión intermedia a *(float)*.

Para redondear un número hacia arriba o hacia abajo al número entero más próximo, utilice las funciones *MathRound*, *MathCeil*, *MathFloor* (véase la sección Funciones de redondeo).

4.1.3 Fecha y hora

Los valores del tipo *datetime* destinados a almacenar fecha y/o hora suelen sufrir varios tipos de conversión:

- en líneas y viceversa para mostrar datos al usuario y leer datos de fuentes externas;
- en estructuras *MqlDateTime* especiales (véase más abajo) para trabajar con componentes individuales de fecha y hora;
- al número de segundos transcurridos desde el 01/01/1970, que corresponde a la representación interna de *datetime* y equivale al tipo entero. *long*

Para el último elemento, utilice la conversión de *datetime* a *(long)*, o viceversa, *long* a *(datetime)*, pero tenga en cuenta que el intervalo de fechas admitido va del 1 de enero de 1970 (valor 0) al 31 de diciembre de 3000 (32535215999 segundos).

Para las dos primeras opciones, la API de MQL5 proporciona las siguientes funciones:

string TimeToString(datetime value, int mode = TIME_DATE | TIME_MINUTES)

La función *TimeToString* convierte un valor de tipo *datetime* en una cadena con componentes de fecha y hora, de acuerdo con el parámetro *mode* en el que se puede establecer una combinación arbitraria de banderas:

- TIME_DATE - fecha en el formato «AAAA.MM.DD».
- TIME_MINUTES - hora en el formato «hh:mm», es decir, con horas y minutos
- TIME_SECONDS - hora en formato «hh:mm:ss», es decir, con horas, minutos y segundos

Para obtener los datos completos de fecha y hora, puede configurar *mode* como TIME_DATE | TIME_SECONDS (la opción TIME_DATE | TIME_MINUTES | TIME_SECONDS también funciona, pero es redundante). Esto equivale a pasar un valor de tipo *datetime* a *(string)*.

En el archivo *ConversionTime.mq5* se ofrecen ejemplos de uso.

```
#define PRT(A) Print(#A, "=", (A))

void OnStart()
{
   datetime time = D'2021.01.21 23:00:15';
   PRT((string)time);
   PRT(TimeToString(time));
   PRT(TimeToString(time, TIME_DATE | TIME_MINUTES | TIME_SECONDS));
   PRT(TimeToString(time, TIME_MINUTES | TIME_SECONDS));
   PRT(TimeToString(time, TIME_DATE | TIME_SECONDS));
   PRT(TimeToString(time, TIME_DATE));
   PRT(TimeToString(time, TIME_MINUTES));
   PRT(TimeToString(time, TIME_SECONDS));
}
```

El script imprimirá el siguiente registro:

```
(string)time=2021.01.21 23:00:15
TimeToString(time)=2021.01.21 23:00
TimeToString(time,TIME_DATE|TIME_MINUTES|TIME_SECONDS)=2021.01.21 23:00:15
TimeToString(time,TIME_MINUTES|TIME_SECONDS)=23:00:15
TimeToString(time,TIME_DATE|TIME_SECONDS)=2021.01.21 23:00:15
TimeToString(time,TIME_DATE)=2021.01.21
TimeToString(time,TIME_MINUTES)=23:00
TimeToString(time,TIME_SECONDS)=23:00:15
```

datetime StringToTime(string value)

La función *StringToTime* convierte una cadena que contiene una fecha y/o una hora en un valor del tipo *datetime*. La cadena puede contener sólo la fecha, sólo la hora, o la fecha y la hora juntas.

Se reconocen los siguientes formatos para las fechas:

- «AAAA.MM.DD»
- «AAAAMMDD»
- «AAAA/MM/DD»
- «AAAA-MM-DD»
- «DD.MM.AAAA»
- «DD/MM/AAAA»
- «DD-MM-AAAA»

Se admiten los siguientes formatos para la hora:

- «hh:mm»
- «hh:mm:ss»

- «hhmmss»

Debe haber al menos un espacio entre la fecha y la hora.

Si en la cadena sólo aparece la hora, la fecha actual se sustituirá en el resultado. Si en la cadena sólo está presente la fecha, la hora se fijará en 00:00:00.

Si se rompe la sintaxis admitida en la cadena, el resultado es la fecha actual.

Los ejemplos de uso de las funciones figuran en el script *ConversionTime.mq5*.

```
void OnStart()
{
   string timeonly = "21:01";   // time only
   PRT(timeonly);
   PRT((datetime)timeonly);
   PRT(StringToTime(timeonly));

   string date = "2000-10-10";  // date only
   PRT((datetime)date);
   PRT(StringToTime(date));
   PRT((long)(datetime)date);
   long seconds = 60;
   PRT((datetime)seconds); // 1 minute from the beginning of 1970

   string ddmmyy = "15/01/2012 01:02:03"; // date and time, and the date in
   PRT(StringToTime(ddmmyy));             // in "forward" order, still ok

   string wrong = "January 2-nd";
   PRT(StringToTime(wrong));
}
```

En el registro, veremos algo como lo siguiente (####.##.## es la fecha actual en que se lanzó el script):

```
timeonly=21:01
(datetime)timeonly=####.##.## 21:01:00
StringToTime(timeonly)=####.##.## 21:01:00
(datetime)date=2000.10.10 00:00:00
StringToTime(date)=2000.10.10 00:00:00
(long)(datetime)date=971136000
(datetime)seconds=1970.01.01 00:01:00
StringToTime(ddmmyy)=2012.01.15 01:02:03
(datetime)wrong=####.##.## 00:00:00
```

Además de *StringToTime*, puede utilizar el operador de conversión *(datetime)* para convertir cadenas en fechas y horas. Sin embargo, la ventaja de la función es que cuando se detecta una cadena fuente incorrecta, la función establece una variable interna con un código de error *_LastError* (que también está disponible a través de la función *GetLastError*). Dependiendo de qué parte de la cadena contenga datos no interpretados, el código de error podría ser ERR_WRONG_STRING_DATE (5031), ERR_WRONG_STRING_TIME (5032) u otra opción de la lista relacionada con la obtención de la fecha y la hora a partir de la cadena.

bool TimeToStruct(datetime value, MqlDateTime &struct)

Para analizar sintácticamente los componentes de fecha y hora por separado, la API de MQL5 proporciona la función *TimeToStruct*, que convierte un valor de tipo *datetime* en la estructura *MqlDateTime*:

```
struct MqlDateTime
{
   int year;           // year
   int mon;            // month
   int day;            // day
   int hour;           // hour
   int min;            // minutes
   int sec;            // seconds
   int day_of_week;    // day of the week
   int day_of_year;    // the number of the day in a year (January 1 has number 0)
};
```

Los días de la semana están numerados a la manera americana: 0 para el domingo, 1 para el lunes, y así sucesivamente hasta 6 para el sábado. Pueden identificarse utilizando la enumeración integrada ENUM_DAY_OF_WEEK.

La función devuelve *true* si tiene éxito y *false* en caso de error, en particular si se pasa una fecha incorrecta.

Comprobemos el rendimiento de la función utilizando el script *ConversionTimeStruct.mq5*. Para ello, vamos a crear el array *time* de tipo *datetime* con valores de prueba. Llamaremos a *TimeToStruct* para cada uno de ellos en un bucle.

Los resultados se añadirán a un array de estructuras *MqlDateTime mdt[]*. Primero lo inicializaremos con ceros, pero como la función integrada *ArrayInitialize* no sabe manejar estructuras, tendremos que escribir una sobrecarga para ella (más adelante descubriremos una forma más fácil de rellenar un array con ceros: en la sección Puesta a cero de objetos y arrays se introducirá la función *ZeroMemory*).

```
int ArrayInitialize(MqlDateTime &mdt[], MqlDateTime &init)
{
   const int n = ArraySize(mdt);
   for(int i = 0; i < n; ++i)
   {
      mdt[i] = init;
   }
   return n;
}
```

Tras el proceso, enviaremos el array de estructuras al registro utilizando la función integrada *ArrayPrint*. Esta es la forma más fácil de proporcionar un formato de datos agradable (se puede utilizar incluso si sólo hay una estructura: basta con ponerla en un array de tamaño 1).

```
void OnStart()
{
   // fill the array with tests
   datetime time[] =
   {
      D'2021.01.28 23:00:15', // valid datetime value
      D'3000.12.31 23:59:59', // the largest supported date and time
      LONG_MAX // invalid date: will cause an error ERR_INVALID_DATETIME (4010)
   };

   // calculate the size of the array at compile time
   const int n = sizeof(time) / sizeof(datetime);

   MqlDateTime null = {}; // example with zeros
   MqlDateTime mdt[];

   // allocating memory for an array of structures with results
   ArrayResize(mdt, n);

   // call our ArrayInitialize overload
   ArrayInitialize(mdt, null);

   // run tests
   for(int i = 0; i < n; ++i)
   {
      PRT(time[i]); // displaying initial data

      if(!TimeToStruct(time[i], mdt[i])) // if an error occurs, output its code
      {
         Print("error: ", _LastError);
         mdt[i].year = _LastError;
      }
   }

   // output the results to the log
   ArrayPrint(mdt);
   ...
}
```

Como resultado, obtenemos las siguientes cadenas en el registro:

```
time[i]=2021.01.28 23:00:15
time[i]=3000.12.31 23:59:59
time[i]=wrong datetime
wrong datetime -> 4010
    [year] [mon] [day] [hour] [min] [sec] [day_of_week] [day_of_year]
[0]   2021     1    28     23     0    15             4            27
[1]   3000    12    31     23    59    59             3           364
[2]   4010     0     0      0     0     0             0             0
```

Puede asegurarse de que todos los campos han recibido los valores adecuados. Para las fechas iniciales incorrectas, almacenamos el código de error en el campo *year*(en este caso, sólo hay un error de este tipo: 4010, ERR_INVALID_DATETIME).

Recordemos que para el valor máximo de fecha en MQL5 se introduce la constante DATETIME_MAX, igual al valor entero 0x793406fff, que corresponde a las 23:59:59 horas del 31 de diciembre de 3000.

El problema más común que se resuelve utilizando la función *TimeToStruct* es obtener el valor de un determinado componente fecha/hora. Por lo tanto, tiene sentido preparar un archivo de encabezado auxiliar (*MQL5Book/DateTime.mqh*) con una opción de implementación lista. El archivo tiene la clase *datetime*.

```
class DateTime
{
private:
   MqlDateTime mdtstruct;
   datetime origin;

   DateTime() : origin(0)
   {
      TimeToStruct(0, mdtstruct);
   }

   void convert(const datetime &dt)
   {
      if(origin != dt)
      {
         origin = dt;
         TimeToStruct(dt, mdtstruct);
      }
   }

public:
   static DateTime *assign(const datetime dt)
   {
      _DateTime.convert(dt);
      return &_DateTime;
   }
   ENUM_DAY_OF_WEEK timeDayOfWeek() const
   {
      return (ENUM_DAY_OF_WEEK)mdtstruct.day_of_week;
   }
   int timeDayOfYear() const
   {
      return mdtstruct.day_of_year;
   }
   int timeYear() const
   {
      return mdtstruct.year;
   }
   int timeMonth() const
   {
      return mdtstruct.mon;
   }
   int timeDay() const
   {
      return mdtstruct.day;
   }
   int timeHour() const
   {
      return mdtstruct.hour;
   }
   int timeMinute() const
```

```
      {
         return mdtstruct.min;
      }
      int timeSeconds() const
      {
         return mdtstruct.sec;
      }

      static DateTime _DateTime;
   };

   static DateTime DateTime::_DateTime;
```

La clase viene con varias macros que facilitan la llamada a sus métodos.

```
   #define TimeDayOfWeek(T) DateTime::assign(T).timeDayOfWeek()
   #define TimeDayOfYear(T) DateTime::assign(T).timeDayOfYear()
   #define TimeYear(T) DateTime::assign(T).timeYear()
   #define TimeMonth(T) DateTime::assign(T).timeMonth()
   #define TimeDay(T) DateTime::assign(T).timeDay()
   #define TimeHour(T) DateTime::assign(T).timeHour()
   #define TimeMinute(T) DateTime::assign(T).timeMinute()
   #define TimeSeconds(T) DateTime::assign(T).timeSeconds()

   #define _TimeDayOfWeek DateTime::_DateTime.timeDayOfWeek
   #define _TimeDayOfYear DateTime::_DateTime.timeDayOfYear
   #define _TimeYear DateTime::_DateTime.timeYear
   #define _TimeMonth DateTime::_DateTime.timeMonth
   #define _TimeDay DateTime::_DateTime.timeDay
   #define _TimeHour DateTime::_DateTime.timeHour
   #define _TimeMinute DateTime::_DateTime.timeMinute
   #define _TimeSeconds DateTime::_DateTime.timeSeconds
```

La clase tiene el campo *mdtstruct* del tipo de estructura *MqlDateTime*. Este campo se utiliza en todas las conversiones internas. Los campos de estructura se leen mediante métodos getter: a cada campo se le asigna un método correspondiente.

Se define una instancia estática dentro de la clase: *_DateTime* (un objeto es suficiente, porque todos los programas MQL son de un solo hilo). El constructor es privado, por lo que intentar crear otros objetos *datetime* fallará.

Utilizando macros podemos recibir cómodamente componentes separados de *datetime*, como por ejemplo el año (*TimeYear(T)*), el mes (*TimeMonth(T)*), el número (*TimeDay(T)*), o el día de la semana (*TimeDayOfWeek(T)*).

Si a partir de un valor de *datetime* es necesario recibir varios campos, entonces es mejor utilizar macros similares en todas las llamadas excepto en la primera sin parámetro y empezando por el símbolo de subrayado: leen el campo deseado de la estructura sin reajustar la fecha/hora y llamar a la función *TimeToStruct*. Por ejemplo:

```
// use the DateTime class from MQL5Book/DateTime.mqh:
// first get the day of the week for the specified datetime value
PRT(EnumToString(TimeDayOfWeek(time[0])));
// then read year, month and day for the same value
PRT(_TimeYear());
PRT(_TimeMonth());
PRT(_TimeDay());
```

En el registro deberían aparecer las siguientes cadenas.

```
EnumToString(DateTime::_DateTime.assign(time[0]).__TimeDayOfWeek())=THURSDAY
DateTime::_DateTime.__TimeYear()=2021
DateTime::_DateTime.__TimeMonth()=1
DateTime::_DateTime.__TimeDay()=28
```

La función integrada *EnumToString* convierte un elemento de cualquier enumeración en una cadena. Se describirá en una sección aparte.

datetime StructToTime(MqlDateTime &struct)

La función *StructToTime* realiza una conversión de la estructura *MqlDateTime* (véase más arriba la descripción de la función *TimeToStruct*), que contiene componentes de fecha y hora, en un valor de tipo *datetime*. Los campos *day_of_week* y *day_of_year* no se utilizan.

Si el estado de los campos restantes no es válido (correspondiente a una fecha inexistente o no admitida), la función puede devolver un valor corregido, o WRONG_VALUE (-1 en la representación de tipo *long*), dependiendo del problema. Por lo tanto, debe comprobar si se produce un error según el estado de la variable global *_LastError*. Una conversión correcta se completa con el código 0. Antes de convertir, debe restablecer un posible estado fallido en *_LastError* (conservado como artefacto de la ejecución de algunas instrucciones anteriores) utilizando la función *ResetLastError*.

La prueba de la función *StructToTime* también se proporciona en el script *ConversionTimeStruct.mq5*. El array de estructuras *parts* se convierte en *datetime* en el bucle.

```
MqlDateTime parts[] =
{
   {0, 0, 0, 0, 0, 0, 0, 0},
   {100, 0, 0, 0, 0, 0, 0, 0},
   {2021, 2, 30, 0, 0, 0, 0, 0},
   {2021, 13, -5, 0, 0, 0, 0, 0},
   {2021, 50, 100, 0, 0, 0, 0, 0},
   {2021, 10, 20, 15, 30, 155, 0, 0},
   {2021, 10, 20, 15, 30, 55, 0, 0},
};
ArrayPrint(parts);
Print("");

// convert all elements in the loop
for(int i = 0; i < sizeof(parts) / sizeof(MqlDateTime); ++i)
{
   ResetLastError();
   datetime result = StructToTime(parts[i]);
   Print("[", i, "] ", (long)result, " ", result, " ", _LastError);
}
```

Para cada elemento se muestra el valor resultante y un código de error.

```
     [year] [mon] [day] [hour] [min] [sec] [day_of_week] [day_of_year]
[0]      0     0     0      0     0     0             0             0
[1]    100     0     0      0     0     0             0             0
[2]   2021     2    30      0     0     0             0             0
[3]   2021    13    -5      0     0     0             0             0
[4]   2021    50   100      0     0     0             0             0
[5]   2021    10    20     15    30   155             0             0
[6]   2021    10    20     15    30    55             0             0

[0] -1 wrong datetime 4010
[1] 946684800 2000.01.01 00:00:00 4010
[2] 1614643200 2021.03.02 00:00:00 0
[3] 1638316800 2021.12.01 00:00:00 4010
[4] 1640908800 2021.12.31 00:00:00 4010
[5] 1634743859 2021.10.20 15:30:59 4010
[6] 1634743855 2021.10.20 15:30:55 0
```

Observe que la función corrige algunos valores sin levantar la bandera de error. Así, en el elemento número 2, pasamos la fecha, 30 de febrero de 2021, a la función, que fue convertida a 2 de marzo de 2021, y *_LastError = 0*.

4.1.4 Color

La API de MQL5 contiene 3 funciones integradas para trabajar con el color: dos de ellas sirven para la conversión del tipo *color* hacia y desde una cadena, y la tercera proporciona una representación especial del color con transparencia (ARGB).

string ColorToString(color value, bool showName = false)

La función *ColorToString* convierte el color pasado *value* en una cadena como «R,G,B» (donde R, G, B son números de 0 a 255, correspondientes a la intensidad de los componentes rojo, verde y azul del color) o en el nombre del color de la lista de colores web predefinidos si el parámetro *showName* es igual a *true*. El nombre del color sólo se devuelve si el valor del color coincide exactamente con uno de webset.

En el script *ConversionColor.mq5* se ofrecen ejemplos de uso de la función.

```
void OnStart()
{
   Print(ColorToString(clrBlue));            // 0,0,255
   Print(ColorToString(C'0, 0, 255', true)); // clrBlue
   Print(ColorToString(C'0, 0, 250'));       // 0,0,250
   Print(ColorToString(C'0, 0, 250', true)); // 0,0,250 (no name for this color)
   Print(ColorToString(0x34AB6821, true));   // 33,104,171 (0x21,0x68,0xAB)
}
```

color StringToColor(string text)

La función *StringToColor* convierte una cadena como «R,G,B» o una cadena que contenga el nombre de un color web estándar en un valor de tipo *color*. Si la cadena no contiene un triplete de números con el formato adecuado o un nombre de color, la función devolverá 0 (*clrBlack*).

Se pueden ver ejemplos en el script *ConversionColor.mq5*.

```
void OnStart()
{
   Print(StringToColor("0,0,255")); // clrBlue
   Print(StringToColor("clrBlue")); // clrBlue
   Print(StringToColor("Blue"));    // clrBlack (no color with that name)
   // extra text will be ignored
   Print(StringToColor("255,255,255 more text"));      // clrWhite
   Print(StringToColor("This is color: 128,128,128")); // clrGray
}
```

uint ColorToARGB(color value, uchar alpha = 255)

La función *ColorToARGB* convierte un valor de tipo *color* y un valor de un byte *alpha* (que especifica la transparencia) en una representación ARGB de un color (un valor de tipo *uint*). El formato de color ARGB se utiliza al crear recursos gráficos como dibujo de texto en gráficos.

El valor *alpha* puede variar de 0 a 255. «0» corresponde a la transparencia total del color (al mostrar un píxel de este color, deja inalterada la imagen del gráfico existente en este punto), 255 significa aplicar la densidad total del color (al mostrar un píxel de este color, sustituye completamente el color del gráfico en el punto correspondiente). El valor 128 (0x80) es translúcido.

> Como sabemos, el tipo *color* describe un color utilizando tres componentes de color: rojo (Red), verde (Green) y azul (Blue), que se almacenan en el formato 0x00BBGGRR en un entero de 4 bytes (*uint*). Cada componente es un byte que especifica la saturación de ese color en el rango de 0 a 255 (0x00 a 0xFF en hexadecimal). El byte más alto está vacío. Por ejemplo, el color blanco

contiene todos los colores y, por tanto, tiene un significado *color* igual a 0xFFFFFF.

Sin embargo, en determinadas tareas es necesario especificar la transparencia del color para describir el aspecto que tendrá la imagen cuando se superponga sobre algún fondo (sobre otra imagen ya existente). Para estos casos se introduce el concepto de canal alfa, que se codifica mediante un byte adicional.

La representación de color ARGB, junto con el canal alfa (AA), es 0xAARRGGBB. Por ejemplo, el valor 0x80FFFF00 significa color amarillo (una mezcla de los componentes rojo y verde) translúcido.

Al superponer una imagen con un canal alfa sobre algún fondo, se obtiene el color resultante:

```
Cresult = (Cforeground * alpha + Cbackground * (255 - alpha)) / 255
```

donde C toma el valor de cada uno de los componentes R, G, B, respectivamente. Esta fórmula se facilita como referencia. Cuando se utilizan funciones integradas con colores ARGB, la transparencia se aplica automáticamente.

Un ejemplo de aplicación de *ColorToARGB* figura en *ConversionColor.mq5*. Se han añadido al script una estructura auxiliar *Argb* y la unión ColorARGB para mayor comodidad a la hora de analizar los componentes de color.

```
struct Argb
{
   uchar BB;
   uchar GG;
   uchar RR;
   uchar AA;
};

union ColorARGB
{
   uint value;
   uchar channels[4]; // 0 - BB, 1 - GG, 2 - RR, 3 - AA
   Argb split[1];
   ColorARGB(uint u) : value(u) { }
};
```

La estructura se utiliza como campo de tipo *split* en la unión y proporciona acceso a los componentes ARGB por nombre. La unión también tiene un array de bytes *channels*, que le permite acceder a los componentes por índice.

```
void OnStart()
{
   uint u = ColorToARGB(clrBlue);
   PrintFormat("ARGB1=%X", u); // ARGB1=FF0000FF
   ColorARGB clr1(u);
   ArrayPrint(clr1.split);
   /*
       [BB] [GG] [RR] [AA]
   [0]  255    0    0  255
   */

   u = ColorToARGB(clrDeepSkyBlue, 0x40);
   PrintFormat("ARGB2=%X", u); // ARGB2=4000BFFF
   ColorARGB clr2(u);
   ArrayPrint(clr2.split);
   /*
       [BB] [GG] [RR] [AA]
   [0]  255  191    0   64
   */
}
```

Veremos la función *print format* un poco más adelante, en la sección correspondiente.

No existe ninguna función integrada para convertir ARGB de nuevo a *color* (ya que no suele ser necesario), pero quienes deseen hacerlo, pueden utilizar la siguiente macro:

```
#define ARGBToColor(U) (color) \
   ((((U) & 0xFF) << 16) | ((U) & 0xFF00) | (((U) >> 16) & 0xFF))
```

4.1.5 Estructuras

Al integrar programas MQL con sistemas externos, en concreto, al enviar o recibir datos a través de Internet, se hace necesario convertir las estructuras de datos en arrays de bytes. Para ello, la API de MQL5 ofrece dos funciones: *StructToCharArray* y *CharArrayToStruct*.

En ambos casos, se supone que una estructura contiene sólo tipos integrados sencillos, es decir, todos los tipos integrados excepto líneas y arrays dinámicos. Una estructura también puede contener otras estructuras simples. Los objetos de clase y los punteros no están permitidos. Estas estructuras también se denominan POD (Plain Old Data).

bool StructToCharArray(const void &object, uchar &array[], uint pos = 0)

La función *StructToCharArray* copia la estructura POD *object* en el array *array* de tipo *uchar*. Opcionalmente, utilizando el parámetro *pos* puede especificar la posición en el array, a partir de la cual se colocarán los bytes. Por defecto, la copia va al principio del array, y el array dinámico aumentará automáticamente de tamaño si su tamaño actual no es suficiente para toda la estructura.

La función devuelve un indicador de éxito (*true*) o errores (*false*).

Comprobemos su rendimiento con el script *ConversionStruct.mq5*. Vamos a crear una nueva estructura de tipo *DateTimeMsc*, que incluye la estructura estándar *MqlDateTime* (campo *mdt*) y un campo adicional *msc* de tipo *int* para almacenar milisegundos.

```
struct DateTimeMsc
{
   MqlDateTime mdt;
   int msc;
   DateTimeMsc(MqlDateTime &init, int m = 0) : msc(m)
   {
      mdt = init;
   }
};
```

Dentro de la función *OnStart* , vamos a convertir un valor de prueba *datetime* a nuestra estructura, y luego al array de bytes.

```
MqlDateTime TimeToStructInplace(datetime dt)
{
   static MqlDateTime m;
   if(!TimeToStruct(dt, m))
   {
      // the error code, _LastError, can be displayed
      // but here we just return zero time
      static MqlDateTime z = {};
      return z;
   }
   return m;
}

#define MDT(T) TimeToStructInplace(T)

void OnStart()
{
   DateTimeMsc test(MDT(D'2021.01.01 10:10:15'), 123);
   uchar a[];
   Print(StructToCharArray(test, a));
   Print(ArraySize(a));
   ArrayPrint(a);
}
```

Obtendremos el siguiente resultado en el registro (el array está reformateado con saltos de línea adicionales para enfatizar la correspondencia de bytes a cada uno de los campos):

```
true
36
229   7   0   0
  1   0   0   0
  1   0   0   0
 10   0   0   0
 10   0   0   0
 15   0   0   0
  5   0   0   0
  0   0   0   0
123   0   0   0
```

bool CharArrayToStruct(void &object, const uchar &array[], uint pos = 0)

La función *CharArrayToStruct* copia el array *array* del tipo *uchar* en la estructura POD *object*. Utilizando el parámetro *pos*, puede especificar la posición en el array desde la que empezar a leer bytes.

La función devuelve un indicador de éxito (*true*) o errores (*false*).

Continuando con el mismo ejemplo (*ConversionStruct.mq5*), podemos restaurar la fecha y hora originales desde el array de bytes.

```
void OnStart()
{
   ...
   DateTimeMsc receiver;
   Print(CharArrayToStruct(receiver, a));                // true
   Print(StructToTime(receiver.mdt), "'", receiver.msc); // 2021.01.01 10:10:15'123
}
```

4.1.6 Enumeraciones

En la API de MQL5, un valor de enumeración se puede convertir en una cadena utilizando la función *EnumToString*. No existe ninguna transformación inversa preparada.

string EnumToString(enum value)

La función convierte el valor (es decir, el ID del elemento pasado) de una enumeración de cualquier tipo en una cadena.

Vamos a utilizarlo para resolver una de las tareas más populares: averiguar el tamaño de la enumeración (cuántos elementos contiene) y qué valores corresponden exactamente a todos los elementos. Para ello, en el archivo de encabezado *EnumToArray.mqh* implementamos la función de plantilla especial (debido al tipo de plantilla E, funcionará para cualquier enum):

```
template<typename E>
int EnumToArray(E dummy, int &values[],
   const int start = INT_MIN,
   const int stop = INT_MAX)
{
   const static string t = "::";

   ArrayResize(values, 0);
   int count = 0;

   for(int i = start; i < stop && !IsStopped(); i++)
   {
      E e = (E)i;
      if(StringFind(EnumToString(e), t) == -1)
      {
         ArrayResize(values, count + 1);
         values[count++] = i;
      }
   }
   return count;
```

```
}
```

El concepto de su funcionamiento se basa en lo siguiente. Dado que las enumeraciones en MQL5 se almacenan como enteros de tipo *int*, se admite una conversión implícita de cualquier enumeración a *(int)*, y también se permite una conversión explícita *int* de vuelta a cualquier tipo de enumeración. En este caso, si el valor corresponde a uno de los elementos de la enumeración, la función *EnumToString* devuelve una cadena con el ID de este elemento. En caso contrario, la función devuelve una cadena de la forma ENUM_TYPE::value.

Así, recorriendo los enteros en el rango aceptable y convirtiéndolos explícitamente a un tipo enum, se puede analizar la cadena de salida *EnumToString* en busca de la presencia de '::' para determinar si el entero dado es un miembro enum o no.

La función *StringFind* utilizada aquí se presentará en el capítulo siguiente, al igual que otras funciones de cadena.

Vamos a crear el script *ConversionEnum.mq5* para probar el concepto. En él, implementamos una función auxiliar *process*, que llamará a la plantilla *EnumToArray*, informará del número de elementos de enum, e imprimirá el array resultante con las coincidencias entre los elementos de enum y sus valores.

```
template<typename E>
void process(E a)
{
  int result[];
  int n = EnumToArray(a, result, 0, USHORT_MAX);
  Print(typename(E), " Count=", n);
  for(int i = 0; i < n; i++)
  {
    Print(i, " ", EnumToString((E)result[i]), "=", result[i]);
  }
}
```

Como enumeración con fines de investigación, utilizaremos la enumeración integrada con los tipos de precio ENUM_APPLIED_PRICE. Dentro de la función *OnStart*, asegurémonos primero de que *EnumToString* produce cadenas como se ha descrito anteriormente. Así, para el elemento PRICE_CLOSE, la función devolverá la cadena «PRICE_CLOSE», y para el valor (ENUM_APPLIED_PRICE)10, que obviamente está fuera de rango, devolverá «ENUM_APPLIED_PRICE::10».

```
void OnStart()
{
   PRT(EnumToString(PRICE_CLOSE));             // PRICE_CLOSE
   PRT(EnumToString((ENUM_APPLIED_PRICE)10)); // ENUM_APPLIED_PRICE::10

   process((ENUM_APPLIED_PRICE)0);
}
```

A continuación, llamamos a la función *process* para cualquier valor convertido a ENUM_APPLIED_PRICE (o a una variable de ese tipo) y obtenemos el siguiente resultado:

```
ENUM_APPLIED_PRICE Count=7
0 PRICE_CLOSE=1
1 PRICE_OPEN=2
2 PRICE_HIGH=3
3 PRICE_LOW=4
4 PRICE_MEDIAN=5
5 PRICE_TYPICAL=6
6 PRICE_WEIGHTED=7
```

Aquí vemos que se definen 7 elementos en la enumeración, y la numeración no empieza por 0, como es habitual, sino por 1 (PRICE_CLOSE). Conocer los valores asociados a los elementos permite en algunos casos optimizar la escritura de algoritmos.

4.1.7 Tipo complejo

El tipo *complex* integrado es una estructura con dos campos de tipo *double*:

```
struct complex
{
   double      real;   // real part
   double      imag;   // imaginary part
};
```

Esta estructura se describe en la sección de conversión de tipos porque «convierte» dos números *double* en una nueva entidad, de forma similar a como las estructuras se convierten en arrays de bytes y viceversa. Además, sería bastante difícil presentar este tipo sin describir antes las estructuras.

La estructura *complex* no tiene constructor, por lo que los números complejos deben crearse utilizando una lista de inicialización.

```
complex c = {re, im};
```

Para los números complejos, actualmente sólo se dispone de operaciones aritméticas y de comparación sencillas: =, +, -, *, /, +=, -=, *=, /=, ==, !=. Más adelante se añadirá ayuda para las funciones matemáticas.

> ¡Atención! Las variables complejas no se pueden declarar como entradas (utilizando la palabra clave *input*) para un programa MQL.

El sufijo 'i' se utiliza para describir constantes complejas (partes imaginarias), por ejemplo:

```
const complex x = 1 - 2i;
const complex y = 0.5i;
```

En el siguiente ejemplo (script *Complex.mq5*) se crea un número complejo y se eleva al cuadrado.

```
input double r = 1;
input double i = 2;

complex c = {r, i};

complex mirror(const complex z)
{
   complex result = {z.imag, z.real}; // swap real and imaginary parts
   return result;
}

complex square(const complex z)
{
   return (z * z);
}

void OnStart()
{
   Print(c);
   Print(square(c));
   Print(square(mirror(c)));
}
```

Con los parámetros predeterminados, el script mostrará lo siguiente:

```
c=(1,2) / ok
square(c)=(-3,4) / ok
square(mirror(c))=(3,4) / ok
```

Aquí, los pares de números entre paréntesis son la representación de cadena del número complejo.

El tipo *complex* se puede pasar por valor como parámetro de funciones MQL (a diferencia de las estructuras ordinarias, que se pasan sólo por referencia). Para las funciones importadas de DLL, el tipo *complex* sólo debe pasarse por referencia.

4.2 Trabajar con cadenas y símbolos

Aunque su nombre procede del verbo «computar», los ordenadores (o computadores) tienen el mismo éxito a la hora de procesar también, además de números, cualquier información no estructurada, como por ejemplo, texto. En los programas MQL, el texto también se utiliza en todas partes, desde los nombres de los programas en sí hasta los comentarios en las órdenes de trading. Para trabajar con el texto en MQL5 existe un tipo de cadena integrado que permite operar con secuencias de caracteres de longitud arbitraria.

Para realizar acciones típicas con cadenas, la API de MQL5 proporciona una amplia gama de funciones que pueden dividirse condicionalmente en grupos según su finalidad, como la inicialización de cadenas, su adición, la búsqueda y sustitución de fragmentos dentro de cadenas, la conversión de cadenas en arrays de caracteres, el acceso a caracteres individuales, así como el formateo.

La mayoría de las funciones de este capítulo devuelven una indicación del estado de ejecución: éxito o error. Para funciones con tipo de resultado *bool*, *true* suele ser un éxito y *false*, un error. Para las funciones con tipo de resultado *int*, un valor de 0 o -1 puede considerarse un error: esto se indica en la

descripción de cada función. En todos estos casos, el desarrollador puede averiguar la esencia del problema. Para ello, llame a la función *GetLastError* y obtenga el código de error específico: en la documentación encontrará una lista de todos los códigos con explicaciones. Es importante llamar a *GetLastError* inmediatamente después de recibir la bandera de error porque llamar a cada instrucción siguiente en el algoritmo puede dar lugar a otro error.

4.2.1 Inicialización y medición de cadenas

Como sabemos por la sección Tipo de cadena, basta con describir en el código una variable de tipo *string*, y estará listo para funcionar.

Para cualquier variable de tipo *string* se asignan 12 bytes para la estructura de servicio que es la representación interna de la cadena. La estructura contiene la dirección de memoria (puntero) donde se almacena el texto, junto con alguna otra meta-información. El texto en sí también requiere memoria suficiente, pero este búfer se asigna con algunas optimizaciones menos obvias.

En particular, podemos describir una cadena junto con una inicialización explícita, incluyendo un literal vacío:

```
string s = ""; // pointer to the literal containing '\0'
```

En ese caso, el puntero se establecerá directamente en el literal, y no se asignará memoria para el búfer (aunque el literal sea largo). Obviamente, ya se ha asignado memoria estática para el literal, y puede utilizarse de forma directa. La memoria para el búfer se asignará sólo si alguna instrucción del programa cambia el contenido de la línea. Por ejemplo (observe que la operación de suma '+' está permitida para las cadenas):

```
int n = 1;
s += (string)n;    // pointer to memory containing "1"'\0'[plus reserve]
```

A partir de este momento, la cadena contiene en realidad el texto «1» y, en sentido estricto, requiere memoria para dos caracteres: el dígito «1» y el cero terminal implícito '\0' (terminador de cadena). No obstante, el sistema asignará un búfer mayor, con algo de espacio reservado.

Cuando declaramos una variable sin valor inicial, el compilador la inicializa implícitamente, aunque en este caso con un valor especial NULL:

```
string z; // memory for the pointer is not allocated, pointer = NULL
```

Una cadena de este tipo sólo requiere 12 bytes por estructura, y el puntero no apunta a ninguna parte: eso es lo que significa NULL.

> En futuras versiones del compilador MQL5, este comportamiento puede cambiar, y siempre se asignará inicialmente una pequeña área de memoria para una cadena vacía, proporcionando algo de espacio reservado.

Además de estas características internas, las variables del tipo *string* no difieren de las variables de otros tipos. Sin embargo, debido a que las cadenas pueden tener una longitud variable y, lo que es más importante, pueden cambiar de longitud durante el algoritmo, esto puede afectar negativamente a la eficiencia de la asignación de memoria y al rendimiento.

Por ejemplo, si en algún momento el programa necesita añadir una nueva palabra a una cadena, puede resultar que no haya suficiente memoria asignada a la cadena. El entorno de ejecución del programa MQL, de forma imperceptible para el usuario, encontrará un nuevo bloque de memoria libre de mayor

tamaño y copiará el valor antiguo junto con la palabra añadida. Después, la dirección antigua se sustituye por otra nueva en la estructura de servicios de la línea.

Si hay muchas operaciones de este tipo, la ralentización debida a la copia puede llegar a ser notable y, además, la memoria del programa está sujeta a fragmentación: las pequeñas áreas de memoria antiguas liberadas tras la copia forman vacíos que no son adecuados en tamaño para cadenas grandes y, por lo tanto, conducen al desperdicio de memoria. Por supuesto, el terminal es capaz de controlar estas situaciones y reorganizar la memoria, pero esto también tiene un coste.

La forma más eficaz de resolver este problema es indicar explícitamente por adelantado el tamaño del búfer para la cadena e inicializarlo utilizando las funciones integradas de la API de MQL5, que veremos más adelante en esta sección.

La base de esta optimización es simplemente que el tamaño de la memoria asignada puede exceder la longitud actual (y, potencialmente, la futura) de la cadena, que viene determinada por el primer carácter nulo del texto. Así, podemos asignar un búfer para 100 caracteres, pero desde el principio poner '\0' al principio, lo que dará una cadena de longitud cero ("").

Naturalmente, se supone que en estos casos el programador puede calcular de forma aproximada y de antemano la longitud prevista de la cadena o su tasa de crecimiento.

Dado que las cadenas en MQL5 se basan en caracteres de doble byte (lo que garantiza la compatibilidad Unicode), el tamaño de la cadena y del búfer en caracteres debe multiplicarse por 2 para obtener la cantidad de memoria ocupada y asignada en bytes.

Al final de la sección se ofrecerá un ejemplo general de utilización de todas las funciones (*StringInit.mq5*).

bool StringInit(string &variable, int capacity = 0, ushort character = 0)

La función *StringInit* se utiliza para inicializar (asignar y llenar memoria) y desinicializar (liberar memoria) cadenas. La variable que se va a procesar se pasa en el primer parámetro.

Si el parámetro *capacity* es mayor que 0, se asigna un búfer (área de memoria) del tamaño especificado para la cadena y se rellena con el símbolo *character*. Si *character* es 0, la longitud de la cadena será cero, porque el primer carácter es terminal.

Si el parámetro *capacity* es 0, se libera la memoria previamente asignada. El estado de la variable pasa a ser idéntico al que tenía si se acababa de declarar sin inicializar (el puntero al búfer es NULL). Más sencillamente, se puede hacer lo mismo poniendo la variable cadena en NULL.

La función devuelve un indicador de éxito (*true*) o errores (*false*).

bool StringReserve(string &variable, uint capacity)

La función *StringReserve* aumenta o disminuye el tamaño del búfer de la cadena *variable*, al menos hasta el número de caracteres especificado en el parámetro *capacity*. Si el valor de *capacity* es menor que la longitud actual de la cadena, la función no hace nada. De hecho, el tamaño del búfer puede ser mayor que el solicitado: el entorno hace esto por razones de eficiencia en futuras manipulaciones con la cadena. Así, si se llama a la función con un valor reducido para el búfer, puede ignorar la petición y seguir devolviendo *true* («sin errores»).

El tamaño actual del búfer puede obtenerse mediante la función *StringBufferLen* (véase más abajo).

En caso de éxito, la función devuelve *true*, en caso contrario — *false*.

A diferencia de *StringInit*, la función *StringReserve* no modifica el contenido de la cadena y no la rellena con caracteres.

bool StringFill(string &variable, ushort character)

La función *StringFill* rellena la cadena *variable* especificada con el carácter *character* en toda su longitud actual (hasta el primer cero). Si se asigna un búfer a una cadena, la modificación se realiza in situ, sin operaciones intermedias de nueva línea y copia.

La función devuelve un indicador de éxito (*true*) o errores (*false*).

int StringBufferLen(const string &variable)

La función devuelve el tamaño del búfer asignado a la cadena *variable*.

Tenga en cuenta que, para una cadena inicializada con un literal, no se asigna inicialmente ningún búfer porque el puntero apunta al literal. Por lo tanto, la función devolverá 0 aunque la longitud de la cadena *StringLen* (véase más abajo) pueda ser mayor.

El valor -1 significa que la línea pertenece al terminal cliente y no puede modificarse.

bool StringSetLength(string &variable, uint length)

La función establece la longitud especificada en caracteres *length* para la cadena *variable*. El valor de *length* no debe ser mayor que la longitud actual de la cadena. En otras palabras, la función sólo permite acortar la cadena, pero no alargarla. La longitud de la cadena se incrementa automáticamente cuando se llama a la función *StringAdd* o se realiza la operación de suma '+'.

El equivalente de la función *StringSetLength* es la llamada *StringSetCharacter(variable, length, 0)* (véase la sección Trabajar con símbolos y páginas de códigos).

Si ya se ha asignado un búfer para la cadena antes de la llamada a la función, ésta no lo modifica. Si la cadena no tenía un búfer (apuntaba a un literal), al disminuir la longitud se asigna un nuevo búfer y se copia en él la cadena acortada.

La función devuelve *true* o *false* en caso de éxito o fracaso, respectivamente.

int StringLen(const string text)

La función devuelve el número de caracteres de la cadena *text*. El terminal cero no se tiene en cuenta.

Tenga en cuenta que el parámetro se pasa por valor, por lo que puede calcular la longitud de las cadenas no sólo en variables, sino también para cualquier otro valor intermedio: literales o resultados de cálculo.

El script *StringInit.mq5* se ha creado para demostrar las funciones anteriores. Utiliza una versión especial de la macro PRT, PRTE, que analiza el resultado de una expresión en *true* o *false* y, en el caso de esta última, devuelve además un código de error:

```
#define PRTE(A) Print(#A, "=", (A) ? "true" : "false:" + (string)GetLastError())
```

Para la salida de depuración al registro de una cadena y sus métricas actuales (longitud de línea y tamaño del búfer), se implementa la función *StrOut*:

```
void StrOut(const string &s)
{
   Print("'", s, "' [", StringLen(s), "] ", StringBufferLen(s));
}
```

Utiliza las funciones integradas *StringLen* y *StringBufferLen*.

El script de prueba realiza una serie de acciones sobre una cadena en *OnStart*:

```
void OnStart()
{
   string s = "message";
   StrOut(s);
   PRTE(StringReserve(s, 100)); // ok, but we get a buffer larger than requested: 260
   StrOut(s);
   PRTE(StringReserve(s, 500)); // ok, buffer is increased to 500
   StrOut(s);
   PRTE(StringSetLength(s, 4)); // ok: string is shortened
   StrOut(s);
   s += "age";
   PRTE(StringReserve(s, 100)); // ok: buffer remains at 500
   StrOut(s);
   PRTE(StringSetLength(s, 8)); // no: string lengthening is not supported
   StrOut(s);                   //     via StringSetLength
   PRTE(StringInit(s, 8, '$')); // ok: line increased by padding
   StrOut(s);                   //     buffer remains the same
   PRTE(StringFill(s, 0));      // ok: string collapsed to empty because
   StrOut(s);                   //     was filled with 0s, the buffer is the same
   PRTE(StringInit(s, 0));      // ok: line is zeroed, including buffer
                                // we could just write s = NULL;
   StrOut(s);
}
```

El script registrará los siguientes mensajes:

```
'message' [7] 0
StringReserve(s,100)=true
'message' [7] 260
StringReserve(s,500)=true
'message' [7] 500
StringSetLength(s,4)=true
'mess' [4] 500
StringReserve(s,10)=true
'message' [7] 500
StringSetLength(s,8)=false:5035
'message' [7] 500
StringInit(s,8,'$')=true
'$$$$$$$$' [8] 500
StringFill(s,0)=true
'' [0] 500
StringInit(s,0)=true
'' [0] 0
```

Tenga en cuenta que la llamada *StringSetLength* con longitud de cadena aumentada terminó con el error 5035 (ERR_STRING_SMALL_LEN).

4.2.2 Concatenación de cadenas

La concatenación de cadenas es probablemente la operación de cadena más común. En MQL5, se puede hacer utilizando los operadores '+' o '+='. El primer operador concatena dos cadenas (los operandos a izquierda y derecha del signo '+') y crea una cadena concatenada temporal que puede asignarse a una variable de destino o pasarse a otra parte de una expresión (como una llamada a función). El segundo operador añade la cadena situada a la derecha del operador '+=' a la cadena (variable) situada a la izquierda de este operador.

Además, la API de MQL5 proporciona un par de funciones para componer cadenas a partir de otras cadenas o elementos de otros tipos.

En el script *StringAdd.mq5*, que se considera después de su descripción, se ofrecen ejemplos de utilización de las funciones.

bool StringAdd(string &variable, const string addition)

La función añade la cadena *addition* especificada al final de una variable de cadena *variable*. Siempre que sea posible, el sistema utiliza el búfer disponible de la cadena *variable* (si su tamaño es suficiente para el resultado combinado) sin reasignar memoria ni copiar cadenas.

La función es equivalente al operador *variable* += *addition*. Los costes de tiempo y el consumo de memoria son prácticamente los mismos.

La función devuelve *true* en caso de éxito y *false* en caso de error.

int StringConcatenate(string &variable, void argument1, void argument2 [, void argumentI...])

La función convierte dos o más argumentos de tipos integrados a una representación de cadena y las concatena en la cadena *variable*. Los argumentos se pasan empezando por el segundo parámetro de la función. No se admiten como argumentos arrays, estructuras, objetos ni punteros.

El número de argumentos debe estar comprendido entre 2 y 63.

Los argumentos de cadena se añaden a la variable resultante tal cual.

Los argumentos de tipo *double* se convierten con la máxima precisión (hasta 16 dígitos significativos) y puede elegirse la notación científica con exponente si resulta más compacta. Los argumentos de tipo *float* se muestran con 5 caracteres.

Los valores del tipo *datetime* se convierten en una cadena con todos los campos de fecha y hora («AAAA.MM.DD hh:mm:ss»).

Las enumeraciones y los caracteres de uno y dos bytes se emiten como números enteros.

Los valores del tipo *color* se muestran como un trío de componentes «R,G,B» o un nombre de color (si está disponible en la lista de colores web estándar).

Al convertir el tipo *bool* se utilizan las cadenas «true» o «false».

La función *StringConcatenate* devuelve la longitud de la cadena resultante.

> *StringConcatenate* ha sido diseñada para construir una cadena a partir de otras fuentes (variables, expresiones) distintas de la variable receptora. No se recomienda utilizar *StringConcatenate* para concatenar nuevos trozos de datos en la misma fila llamando a *StringConcatenate(variable, variable, ...)*. Esta llamada a la función no está optimizada y es extremadamente lenta en comparación con el operador '+' y StringAdd.

Las funciones *StringAdd* y *StringConcatenate* se prueban en la secuencia de comandos *StringAdd.mq5*, que utiliza la macro PRTE y la función de ayuda *StrOut* de la sección anterior.

```
void OnStart()
{
   string s = "message";
   StrOut(s);
   PRTE(StringAdd(s, "r"));
   StrOut(s);
   PRTE(StringConcatenate(s, M_PI * 100, " ", clrBlue, PRICE_CLOSE));
   StrOut(s);
}
```

Como resultado de su ejecución, se muestran las siguientes líneas en el registro:

```
'message' [7] 0
StringAdd(s,r)=true
'messager' [8] 260
StringConcatenate(s,M_PI*100, ,clrBlue,PRICE_CLOSE)=true
'314.1592653589793 clrBlue1' [26] 260
```

El script también incluye el archivo de encabezado *StringBenchmark.mqh* con la clase *benchmark*. Proporciona un marco para que las clases derivadas implementadas en el script midan el rendimiento de varios métodos de adición de cadenas. En concreto, se aseguran de que la suma de cadenas mediante el operador '+' y la función *StringAdd* sean comparables. Este material se deja para el estudio independiente.

Además, el libro viene con el script *StringReserve.mq5*: hace una comparación visual de la velocidad de adición de cadenas dependiendo del uso o no del búfer (*StringReserve*).

4.2.3 Comparación de cadenas

Para comparar cadenas en MQL5, puede utilizar los operadores de comparación estándar, en particular '==', '!=', '>', '<'. Todos estos operadores realizan comparaciones carácter por carácter, distinguiendo entre mayúsculas y minúsculas.

Cada carácter tiene un código Unicode que es un entero del tipo *ushort*. En consecuencia, primero se comparan los códigos de los primeros caracteres de dos cadenas, luego los códigos de los segundos, y así sucesivamente hasta que se alcanza la primera falta de coincidencia o el final de una de las cadenas.

Por ejemplo, la cadena «ABC» es menor que «abc», ya que los códigos de las letras mayúsculas en la tabla de caracteres son menores que los códigos de las letras minúsculas correspondientes (en el primer carácter ya obtenemos que «A» < «a»). Si las cadenas tienen caracteres coincidentes al principio, pero una de ellas es más larga que la otra, se considera que la cadena más larga es mayor («ABCD» > «ABC»).

Estas relaciones entre cadenas forman el orden lexicográfico. Cuando la cadena «A» es menor que la cadena «B» («A» < «B»), se dice que «A» precede a «B».

> Para familiarizarse con los códigos de caracteres, puede utilizar la aplicación estándar de Windows «Tabla de caracteres». En ella, los caracteres están dispuestos en orden de códigos crecientes. Además de la tabla general de Unicode, que incluye muchas lenguas nacionales, existen páginas de códigos: las tablas estándar ANSI con códigos de caracteres de un byte, difieren para cada lengua o grupo de lenguas. Analizaremos esta cuestión con más detalle en la sección Trabajar con símbolos y páginas de códigos.

La parte inicial de las tablas de caracteres con códigos de 0 a 127 es la misma para todas las lenguas. Esta parte se muestra en la siguiente tabla.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	non-printable control codes								BS	HT	LF	VT	FF	CR		
1									back space	horiz. tab	line feed	vertical tab	form feed	carriage return		
2	␣	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Tabla de códigos de caracteres ASCII

Para obtener el código del carácter, tome el dígito hexadecimal de la izquierda (el número de línea en el que se encuentra el carácter) y sume el número de arriba (el número de columna en el que se encuentra el carácter): el resultado es un número hexadecimal. Por ejemplo, para '!' hay 2 a la izquierda y 1 arriba, lo que significa que el código del carácter es 0x21, o 33 en decimal.

Los códigos hasta 32 son códigos de control. Entre ellos se encuentran, en particular, la tabulación (código 0x9), el avance de línea (código 0xA) y el retorno de carro (código 0xD).

En los archivos de texto de Windows se utiliza un par de caracteres 0xD 0xA seguidos para saltar a una nueva línea. Nos hemos familiarizado con los literales de MQL5 correspondientes en la sección Tipos de caracteres. 0xA puede denotarse como '\n' y 0xD como '\r'. La tabulación 0x9 también tiene su propia representación: '\t'.

La API de MQL5 proporciona la función *StringCompare*, que permite desactivar la distinción entre mayúsculas y minúsculas al comparar cadenas.

int StringCompare(const string &string1, const string &string2, const bool case_sensitive = true)

La función compara dos cadenas y devuelve uno de tres valores: +1 si la primera cadena es «mayor que» la segunda; 0 si las cadenas son «iguales»; -1 si la primera cadena es «menor que» la segunda. Los conceptos «mayor que», «menor que» e «igual a» dependen del parámetro *case_sensitive* .

Cuando el parámetro *case_sensitive* es igual a *true* (que es el valor predeterminado), la comparación distingue entre mayúsculas y minúsculas, considerándose las mayúsculas mayores que las minúsculas similares. Es el orden inverso al orden lexicográfico estándar según los códigos de caracteres.

> Cuando se distingue entre mayúsculas y minúsculas, la función *StringCompare* utiliza un orden de mayúsculas y minúsculas diferente del orden lexicográfico. Por ejemplo, sabemos que la relación «A» < «a» es verdadera, en la que el operador '<' se guía por códigos de caracteres. Por lo tanto, las palabras en mayúscula deben aparecer en el diccionario hipotético (array) antes de las palabras con la misma letra minúscula. No obstante, al comparar «A» y «a» utilizando la función *StringCompare("A", "a")*, obtenemos +1, lo que significa que «A» es mayor que «a». Así, en el diccionario ordenado, las palabras que empiecen por minúscula irán primero, y sólo después vendrán las palabras con mayúscula.

En otras palabras: la función ordena las cadenas alfabéticamente. Además, en el modo de distinción entre mayúsculas y minúsculas, se aplica una regla adicional: si hay cadenas que sólo difieren en mayúsculas o minúsculas, las que tienen letras mayúsculas siguen a sus homólogas con letras minúsculas (en las mismas posiciones de la palabra).

Si el parámetro *case_sensitive* es igual a *false*, las letras no distinguen entre mayúsculas de minúsculas, por lo que las cadenas «A» y «a» se consideran iguales, y la función devuelve 0.

Puede comprobar diferentes resultados de comparación mediante la función *StringCompare* y con el operador utilizando el script *StringCompare.mq5*.

```
void OnStart()
{
   PRT(StringCompare("A", "a"));        // 1, which means "A" > "a" (!)
   PRT(StringCompare("A", "a", false)); // 0, which means "A" == "a"
   PRT("A" > "a");                      // false,   "A" < "a"

   PRT(StringCompare("x", "y"));        // -1, which means "x" < "y"
   PRT("x" > "y");                      // false,    "x" < "y"
   ...
}
```

En la sección Plantillas de funciones hemos creado un algoritmo de ordenación rápida. Vamos a transformarlo en una clase de plantilla y a utilizarlo para varias opciones de ordenación: utilizando operadores de comparación, así como la función *StringCompare* con y sin la distinción de mayúsculas y

minúsculas activada. Pongamos la nueva clase *QuickSortT* en el archivo de encabezado *QuickSortT.mqh* y conectémosla al script de prueba *StringCompare.mq5*.

La API de clasificación se ha mantenido prácticamente sin cambios.

```
template<typename T>
class QuickSortT
{
public:
   void Swap(T &array[], const int i, const int j)
   {
      ...
   }

   virtual int Compare(T &a, T &b)
   {
      return a > b ? +1 : (a < b ? -1 : 0);
   }

   void QuickSort(T &array[], const int start = 0, int end = INT_MAX)
   {
      ...
         for(int i = start; i <= end; i++)
         {
            //if(!(array[i] > array[end]))
            if(Compare(array[i], array[end]) <= 0)
            {
               Swap(array, i, pivot++);
            }
         }
      ...
   }
};
```

La principal diferencia es que hemos añadido un método virtual *Compare*, que por defecto contiene una comparación utilizando los operadores '>' y '<', y devuelve +1, -1, o 0 de la misma forma que *StringCompare*. El método *Compare* se utiliza ahora en el método *QuickSort* en lugar de una simple comparación y debe sobrescribirse en las clases hijas para poder utilizar la función *StringCompare* o cualquier otra forma de comparación.

En concreto, en el archivo *StringCompare.mq5*, implementamos la siguiente clase «comparador» derivada de *QuickSortT<string>*:

```
class SortingStringCompare : public QuickSortT<string>
{
   const bool caseEnabled;
public:
   SortingStringCompare(const bool sensitivity = true) :
      caseEnabled(sensitivity) { }

   virtual int Compare(string &a, string &b) override
   {
      return StringCompare(a, b, caseEnabled);
   }
};
```

El constructor recibe 1 parámetro, que especifica el signo de comparación de la cadena teniendo en cuenta (*true*) o ignorando (*false*) el registro. La comparación de cadenas en sí se realiza en el método virtual redefinido *Compare* que llama a la función *StringCompare* con los argumentos y la configuración dados.

Para probar la ordenación, necesitamos un conjunto de cadenas que combine mayúsculas y minúsculas. Podemos generarlo nosotros mismos: basta con desarrollar una clase que realice permutaciones (con repetición) de caracteres de un conjunto predefinido (alfabeto) para una longitud de conjunto dada (cadena). Por ejemplo, puede limitarse al pequeño alfabeto «abcABC», es decir, tres letras inglesas de puño y letra en ambos casos, y generar a partir de ellas todas las cadenas posibles de 2 caracteres.

La clase *PermutationGenerator* se suministra en el archivo *PermutationGenerator.mqh* y se deja para estudio independiente. Aquí sólo presentamos su interfaz pública.

```
class PermutationGenerator
{
public:
   struct Result
   {
      int indices[]; // indexes of elements in each position of the set, i.e.
   };                // for example, the numbers of the letters of the "alphabet" in
   PermutationGenerator(const int length, const int elements);
   SimpleArray<Result> *run();
};
```

Al crear un objeto generador, debe especificar la longitud de los conjuntos generados *length* (en nuestro caso, será la longitud de las cadenas, es decir, 2) y el número de elementos diferentes de los que se compondrán los conjuntos (en nuestro caso, es el número de letras únicas, es decir, 6). Con estos datos de entrada, deberían obtenerse 6 * 6 = 36 variantes de líneas.

El proceso en sí se lleva a cabo con el método *run*. Se utiliza una clase de plantilla para devolver un array con los resultados *SimpleArray*, de lo que hablamos en la sección Plantillas de métodos. En este caso, se parametriza mediante el tipo de estructura *result*.

La llamada del generador y la creación real de cadenas de acuerdo con el array de permutaciones recibido de él (en forma de índices de letras en cada posición para todas las cadenas posibles) se realiza en la función auxiliar *GenerateStringList*.

```
void GenerateStringList(const string symbols, const int len, string &result[])
{
   const int n = StringLen(symbols); // alphabet length, unique characters
   PermutationGenerator g(len, n);
   SimpleArray<PermutationGenerator::Result> *r = g.run();
   ArrayResize(result, r.size());
   // loop through all received character permutations
   for(int i = 0; i < r.size(); ++i)
   {
      string element;
      // loop through all characters in the string
      for(int j = 0; j < len; ++j)
      {
         // add a letter from the alphabet (by its index) to the string
         element += ShortToString(symbols[r[i].indices[j]]);
      }
      result[i] = element;
   }
}
```

Aquí utilizamos varias funciones que todavía nos son desconocidas (*ArrayResize*, *ShortToString*), pero llegaremos a ellas pronto. Por ahora, sólo debemos saber que la función *ShortToString* devuelve una cadena formada por ese único carácter basado en el código de caracteres de tipo *ushort*. Utilizando el operador '+=' concatenamos cada cadena resultante de dichas cadenas de un solo carácter. Recuerde que el operador [] está definido para cadenas, por lo que la expresión *symbols[k]* devolverá el *k*-ésimo carácter de la cadena *symbols*. Por supuesto, *k* puede ser a su vez una expresión entera, y aquí *r[i].indices[j]* se refiere al *i*-ésimo elemento del array *r* del que se lee el índice del carácter «alfabeto» para la *j*-ésima posición de la cadena.

Cada cadena recibida se almacena en un array de parámetros *result*.

Vamos a aplicar esta información en la función *OnStart*.

```
void OnStart()
{
   ...
   string messages[];
   GenerateStringList("abcABC", 2, messages);
   Print("Original data[", ArraySize(messages), "]:");
   ArrayPrint(messages);

   Print("Default case-sensitive sorting:");
   QuickSortT<string> sorting;
   sorting.QuickSort(messages);
   ArrayPrint(messages);

   Print("StringCompare case-insensitive sorting:");
   SortingStringCompare caseOff(false);
   caseOff.QuickSort(messages);
   ArrayPrint(messages);

   Print("StringCompare case-sensitive sorting:");
   SortingStringCompare caseOn(true);
   caseOn.QuickSort(messages);
   ArrayPrint(messages);
}
```

El script primero obtiene todas las opciones de cadena en el array de mensajes y luego las ordena en tres modos: usando los operadores de comparación integrados, usando la función *StringCompare* en el modo que no distingue entre mayúsculas y minúsculas, y usando la misma función en el modo que sí distingue entre mayúsculas y minúsculas.

Obtendremos la siguiente salida de registro:

```
Original data[36]:
[ 0] "aa" "ab" "ac" "aA" "aB" "aC" "ba" "bb" "bc" "bA" "bB" "bC" "ca" "cb" "cc" "cA"
[18] "Aa" "Ab" "Ac" "AA" "AB" "AC" "Ba" "Bb" "Bc" "BA" "BB" "BC" "Ca" "Cb" "Cc" "CA"
Default case-sensitive sorting:
[ 0] "AA" "AB" "AC" "Aa" "Ab" "Ac" "BA" "BB" "BC" "Ba" "Bb" "Bc" "CA" "CB" "CC" "Ca"
[18] "aA" "aB" "aC" "aa" "ab" "ac" "bA" "bB" "bC" "ba" "bb" "bc" "cA" "cB" "cC" "ca"
StringCompare case-insensitive sorting:
[ 0] "AA" "Aa" "aA" "aa" "AB" "aB" "Ab" "ab" "aC" "AC" "Ac" "ac" "BA" "Ba" "bA" "ba"
[18] "Bb" "bb" "bC" "BC" "Bc" "bc" "CA" "Ca" "cA" "ca" "CB" "cB" "Cb" "cb" "cC" "CC"
StringCompare case-sensitive sorting:
[ 0] "aa" "aA" "Aa" "AA" "ab" "aB" "Ab" "AB" "ac" "aC" "Ac" "AC" "ba" "bA" "Ba" "BA"
[18] "Bb" "BB" "bc" "bC" "Bc" "BC" "ca" "cA" "Ca" "CA" "cb" "cB" "Cb" "CB" "cc" "cC"
```

La salida muestra las diferencias en estos tres modos.

4.2.4 Cambiar mayúsculas y minúsculas y recortar espacios

Trabajar con textos implica a menudo el uso de algunas operaciones estándar, como convertir todos los caracteres a mayúsculas o minúsculas y eliminar los caracteres vacíos sobrantes (por ejemplo, espacios) al principio o al final de una cadena. Para estos fines, la API de MQL5 proporciona cuatro

funciones correspondientes. Todas ellas modifican la cadena in situ, es decir, directamente en el búfer disponible (si ya está asignado).

El parámetro de entrada de todas las funciones es una referencia a una cadena, es decir, sólo se les pueden pasar variables (no expresiones), y no variables constantes, ya que las funciones implican la modificación del argumento.

El script de prueba de todas las funciones sigue las descripciones correspondientes.

bool StringToLower(string &variable)
bool StringToUpper(string &variable)

Las funciones convierten todos los caracteres de la cadena especificada al caso apropiado: *StringToLower* a letras minúsculas y *StringToUpper*, a mayúsculas. Esto incluye la compatibilidad con los idiomas nacionales disponibles a nivel del sistema Windows.

Si tiene éxito, devuelve *true*. En caso de error, devuelve *false*.

int StringTrimLeft(string &variable)
int StringTrimRight(string &variable)

La función elimina el retorno de carro ('\r'), el salto de línea ('\n'), los espacios (' '), los tabuladores ('\t') y algunos otros caracteres no visualizables al principio (para *StringTrimLeft*) o al final (para *StringTrimRight*) de una cadena. Si hay espacios vacíos dentro de la cadena (entre los caracteres mostrados), se conservarán.

La función devuelve el número de caracteres eliminados.

El archivo *StringModify.mq5* demuestra el funcionamiento de las funciones anteriores.

```
void OnStart()
{
   string text = "  \tAbCdE F1  ";
               // ↑        ↑  ↑
               // |        |  └2 spaces
               // |        └space
               // └2 spaces and tab
   PRT(StringToLower(text));   // 'true'
   PRT(text);                  // '  \tabcde f1  '
   PRT(StringToUpper(text));   // 'true'
   PRT(text);                  // '  \tABCDE F1  '
   PRT(StringTrimLeft(text));  // '3'
   PRT(text);                  // 'ABCDE F1  '
   PRT(StringTrimRight(text)); // '2'
   PRT(text);                  // 'ABCDE F1'
   PRT(StringTrimRight(text)); // '0'  (there is nothing else to delete)
   PRT(text);                  // 'ABCDE F1'
                               //       ↑
                               //       └the space inside remains

   string russian = "Russian text";
   PRT(StringToUpper(russian));  // 'true'
   PRT(russian);                 // 'RUSSIAN TEXT'
   string german = "straßenführung";
   PRT(StringToUpper(german));   // 'true'
   PRT(german);                  // 'STRAßENFÜHRUNG'
}
```

4.2.5 Encontrar, reemplazar y extraer fragmentos de cadena

Quizá las operaciones más populares cuando se trabaja con cadenas sean encontrar y reemplazar fragmentos, así como extraerlos. En esta sección, estudiaremos las funciones de la API de MQL5 que ayudarán a resolver estos problemas. En el archivo *StringFindReplace.mq5* se resumen ejemplos de su uso.

int StringFind(string value, string wanted, int start = 0)

La función busca la subcadena *wanted* en la cadena *value*, a partir de la posición *start*. Si se encuentra la subcadena, la función devolverá la posición donde comienza, con los caracteres de la cadena numerados empezando por 0. En caso contrario, la función devolverá -1. Ambos parámetros se pasan por valor, lo que permite procesar no sólo variables, sino también resultados intermedios de cálculos (expresiones, llamadas a funciones).

La búsqueda se realiza a partir de una coincidencia estricta de caracteres, es decir, distingue entre mayúsculas y minúsculas. Si desea buscar sin distinguir entre mayúsculas y minúsculas, primero debe convertir la cadena de origen a sólo mayúsculas o sólo minúsculas con *StringToLoweroStringToUpper*.

Intentemos contar el número de apariciones de la subcadena deseada en el texto utilizando *StringFind*. Para ello, vamos a escribir una función auxiliar *CountSubstring* que llamará a *StringFind* en un bucle, desplazando gradualmente la posición de inicio de la búsqueda en el último parámetro *start*. El bucle continuará en tanto se encuentren nuevas ocurrencias de la subcadena.

```
int CountSubstring(const string value, const string wanted)
{
   // indent back because of the increment at the beginning of the loop
   int cursor = -1;
   int count = -1;
   do
   {
      ++count;
      ++cursor; // search continues from the next position
      // get the position of the next substring, or -1 if there are no matches
      cursor = StringFind(value, wanted, cursor);
   }
   while(cursor > -1);
   return count;
}
```

Es importante señalar que la implementación presentada busca subcadenas que puedan solaparse. Esto se debe a que la posición actual se cambia en 1 (++*cursor*) antes de empezar a buscar la siguiente ocurrencia. Como resultado, al buscar, digamos, la subcadena «AAA» en la cadena «AAAAA» se encontrarán 3 coincidencias. Los requisitos técnicos de búsqueda pueden diferir de este comportamiento. En concreto, existe la práctica de continuar la búsqueda después de la posición en la que terminó el fragmento encontrado anteriormente. En este caso, será necesario modificar el algoritmo para que el cursor se desplace con un paso igual a *StringLen(wanted)*.

Llamemos a *CountSubstring* para diferentes argumentos en la función *OnStart*.

```
void OnStart()
{
   string abracadabra = "ABRACADABRA";
   PRT(CountSubstring(abracadabra, "A"));    // 5
   PRT(CountSubstring(abracadabra, "D"));    // 1
   PRT(CountSubstring(abracadabra, "E"));    // 0
   PRT(CountSubstring(abracadabra, "ABRA")); // 2
   ...
}
```

int StringReplace(string &variable, const string wanted, const string replacement)

La función sustituye todas las subcadenas *wanted* encontradas por la subcadena *replacement* de la cadena *variable*.

La función devuelve el número de sustituciones realizadas o -1 en caso de error. El código de error puede obtenerse llamando a la función *GetLastError*. En concreto, puede tratarse de errores de falta de memoria o del uso de una cadena no inicializada (NULL) como argumento. Los parámetros *variables* y *wanted* deben ser cadenas de longitud distinta de cero.

Cuando se da una cadena vacía "" como argumento *replacement*, todas las apariciones de *wanted* simplemente se cortan de la cadena original.

Si no hubo sustituciones, el resultado de la función es 0.

Vamos a utilizar el ejemplo de *StringFindReplace.mq5* para comprobar *StringReplace* en acción.

```
string abracadabra = "ABRACADABRA";
...
PRT(StringReplace(abracadabra, "ABRA", "-ABRA-")); // 2
PRT(StringReplace(abracadabra, "CAD", "-"));      // 1
PRT(StringReplace(abracadabra, "", "XYZ"));      // -1, error
PRT(GetLastError());      // 5040, ERR_WRONG_STRING_PARAMETER
PRT(abracadabra);                              // '-ABRA---ABRA-'
...
```

A continuación, utilizando la función *StringReplace*, vamos a intentar ejecutar una de las tareas que se encuentran en el tratamiento de textos arbitrarios. Intentaremos que un determinado carácter separador se utilice siempre como un único carácter, es decir, que las secuencias de varios caracteres de este tipo se sustituyan por uno. Normalmente, se trata de espacios entre palabras, pero puede haber otros separadores en los datos técnicos. Vamos a probar nuestro programa para el separador '-'.

Implementamos el algoritmo como una función independiente *NormalizeSeparatorsByReplace*:

```
int NormalizeSeparatorsByReplace(string &value, const ushort separator = ' ')
{
   const string single = ShortToString(separator);
   const string twin = single + single;
   int count = 0;
   int replaced = 0;
   do
   {
      replaced = StringReplace(value, twin, single);
      if(replaced > 0) count += replaced;
   }
   while(replaced > 0);
   return count;
}
```

El programa intenta reemplazar una secuencia de dos separadores por uno en un bucle *do-while*, y el bucle continúa siempre que la función *StringReplace* devuelve valores mayores que 0 (es decir, todavía hay algo que reemplazar). La función devuelve el número total de sustituciones realizadas.

En la función *OnStart* vamos a «limpiar» nuestra inscripción de múltiples caracteres '-'.

```
...
string copy1 = "-" + abracadabra + "-";
string copy2 = copy1;
PRT(copy1);                                    // '--ABRA---ABRA--'
PRT(NormalizeSeparatorsByReplace(copy1, '-')); // 4
PRT(copy1);                                    // '-ABRA-ABRA-'
PRT(StringReplace(copy1, "-", ""));            // 1
PRT(copy1);                                    // 'ABRAABRA'
...
```

int StringSplit(const string value, const ushort separator, string &result[])

La función divide la cadena *value* pasada en subcadenas basadas en el separador dado y las coloca en el array *result*. La función devuelve el número de subcadenas recibidas o -1 en caso de error.

Si no hay separador en la cadena, el array tendrá un elemento igual a toda la cadena.

Si la cadena de origen está vacía o es NULL, la función devolverá 0.

Para demostrar el funcionamiento de esta función, vamos a resolver el problema anterior de una nueva forma utilizando *StringSplit*. Para ello, escribamos la función *NormalizeSeparatorsBySplit*.

```
int NormalizeSeparatorsBySplit(string &value, const ushort separator = ' ')
{
   const string single = ShortToString(separator);

   string elements[];
   const int n = StringSplit(value, separator, elements);
   ArrayPrint(elements); // debug

   StringFill(value, 0); // result will replace original string

   for(int i = 0; i < n; ++i)
   {
      // empty strings mean delimiters, and we only need to add them
      // if the previous line is not empty (i.e. not a separator either)
      if(elements[i] == "" && (i == 0 || elements[i - 1] != ""))
      {
         value += single;
      }
      else // all other lines are joined together "as is"
      {
         value += elements[i];
      }
   }

   return n;
}
```

Cuando los separadores aparecen uno tras otro en el texto fuente, el elemento correspondiente en el array de salida *StringSplit* resulta ser una cadena vacía "". Además, una cadena vacía estará al principio del array si el texto comienza con un separador, y al final del array si el texto termina con el separador.

Para obtener un texto «despejado», hay que añadir todas las cadenas no vacías del array, «pegándolos» con caracteres separadores simples. Además, sólo deben convertirse en separadores aquellos elementos vacíos en los que el elemento anterior del array tampoco esté vacío.

Por supuesto, ésta es sólo una de las posibles opciones para implementar esta funcionalidad. Comprobémoslo en la función *OnStart*.

```
   ...
   string copy2 = "-" + abracadabra + "-";         // '--ABRA---ABRA--'
   PRT(NormalizeSeparatorsBySplit(copy2, '-'));    // 8
   // debug output of split array (inside function):
   // ""     ""     "ABRA" ""     ""     "ABRA" ""     ""
   PRT(copy2);                                     // '-ABRA-ABRA-'
```

string StringSubstr(string value, int start, int length = -1)

La función extrae del texto pasado *value* una subcadena que comienza en la posición especificada *start*, de la longitud *length*. La posición inicial puede ser desde 0 hasta la longitud de la cadena menos 1. Si la longitud *length* es -1 o superior al número de caracteres desde *start* hasta el final de la cadena, el resto de la cadena se extraerá completa.

La función devuelve una subcadena o una cadena vacía si los parámetros son incorrectos.

Veamos cómo funciona.

```
PRT(StringSubstr("ABRACADABRA", 4, 3));         // 'CAD'
PRT(StringSubstr("ABRACADABRA", 4, 100));       // 'CADABRA'
PRT(StringSubstr("ABRACADABRA", 4));            // 'CADABRA'
PRT(StringSubstr("ABRACADABRA", 100));          // ''
```

4.2.6 Trabajar con símbolos y páginas de códigos

Dado que las cadenas están formadas por caracteres, a veces es necesario o simplemente más conveniente manipular caracteres individuales o grupos de caracteres de una cadena a nivel de los códigos de sus enteros. Por ejemplo: usted necesita leer o sustituir caracteres de uno en uno o convertirlos en arrays de códigos de enteros para su transmisión a través de protocolos de comunicación o en interfaces de programación de terceros de bibliotecas dinámicas DLL. En todos estos casos, pasar cadenas como texto puede ir acompañado de diversas dificultades:

- garantizar la codificación correcta (de las que hay muchas, y su elección depende de la configuración regional del sistema operativo, los ajustes del programa, la configuración de los servidores con los que se comunica, etc.);
- la conversión de los caracteres de lenguas nacionales desde la codificación de texto local a Unicode y viceversa;
- la asignación y anulación de la asignación de memoria de forma unificada.

El uso de arrays con códigos de enteros (aunque tal uso produce en realidad una representación binaria y no textual de la cadena) simplifica estos problemas.

La API de MQL5 proporciona un conjunto de funciones para operar con caracteres individuales o sus grupos, teniendo en cuenta las características de codificación.

Las cadenas en MQL5 contienen caracteres en codificación Unicode de dos bytes, lo que proporciona una compatibilidad universal para toda la variedad de alfabetos nacionales en una única (pero muy grande) tabla de caracteres. Dos bytes permiten codificar 65535 elementos.

El tipo de carácter por defecto es *ushort*. No obstante, si es necesario, la cadena puede convertirse en una secuencia de caracteres *uchar* de un solo byte en una codificación lingüística específica. Esta conversión puede ir acompañada de la pérdida de cierta información (en particular, las letras que no figuran en la tabla de caracteres localizados pueden «perder» diéresis o incluso «convertirse» en algún tipo de carácter sustitutivo: según el contexto, puede mostrarse de forma diferente, pero normalmente aparece como ' ?' o un carácter cuadrado).

> Para evitar problemas con textos que puedan contener caracteres arbitrarios, se recomienda utilizar siempre Unicode. Se puede hacer una excepción si algunos servicios o programas externos que deben integrarse con su programa MQL no admiten Unicode, o si el texto ha sido concebido desde el principio para almacenar un conjunto limitado de caracteres (por ejemplo, sólo números y letras latinas).

Al convertir a/desde caracteres de un solo byte, la API de MQL5 utiliza la codificación ANSI por defecto, dependiendo de la configuración actual de Windows. No obstante, el desarrollador puede especificar una tabla de códigos diferente (véanse otras funciones *CharArrayToString*, *StringToCharArray*).

En el archivo *StringSymbols.mq5* se ofrecen ejemplos de utilización de las funciones descritas más abajo.

bool StringSetCharacter(string &variable, int position, ushort character)

La función cambia el carácter en *position* al valor *character* en la cadena *variable* pasada. El número debe estar comprendido entre 0 y la longitud de la cadena (*StringLen*) menos 1.

Si el carácter que se va a escribir es 0, especifica un nuevo final de línea (actúa como un cero terminal), es decir, la longitud de la línea pasa a ser igual a *position*. El tamaño del búfer asignado a la línea no cambia.

Si el parámetro *position* es igual a la longitud de la cadena y el carácter que se está escribiendo no es igual a 0, entonces el carácter se añade a la cadena y su longitud se incrementa en 1. Esto equivale a la expresión *variable += ShortToString(character)*.

La función devuelve *true* si se completa con éxito, o *false* en caso de error.

```
void OnStart()
{
   string numbers = "0123456789";
   PRT(numbers);
   PRT(StringSetCharacter(numbers, 7, 0));   // cut off at the 7th character
   PRT(numbers);                             // 0123456
   PRT(StringSetCharacter(numbers, StringLen(numbers), '*')); // add '*'
   PRT(numbers);                             // 0123456*
   ...
}
```

ushort StringGetCharacter(string value, int position)

La función devuelve el código del carácter situado en la posición especificada de la cadena. El número de posición debe estar comprendido entre 0 y la longitud de la cadena (*StringLen*) menos 1. En caso de error, la función devolverá 0.

La función equivale a escribir utilizando el operador '[]': *value[position]*.

```
   string numbers = "0123456789";
   PRT(StringGetCharacter(numbers, 5));      // 53 = code '5'
   PRT(numbers[5]);                          // 53 - is the same
```

string CharToString(uchar code)

La función convierte el código ANSI de un carácter en una cadena de un solo carácter. Dependiendo de la página de códigos de Windows configurada, la mitad superior de los códigos (superior a 127) puede generar letras diferentes (el estilo de los caracteres es diferente, mientras que el código sigue siendo el mismo). Por ejemplo, el símbolo con el código 0xB8 (184 en decimal) indica una cedilla (gancho

inferior) en los idiomas de Europa occidental, mientras que en la lengua rusa aquí se encuentra la letra «ë». He aquí otro ejemplo:

```
PRT(CharToString(0xA9));    // "©"
PRT(CharToString(0xE6));    // "æ", "ж", or another character
                            // depending on your Windows locale
```

string ShortToString(ushort code)

La función convierte el código Unicode de un carácter en una cadena de un solo carácter. Para el parámetro *code* puede utilizar un literal o un entero. Por ejemplo, la letra griega mayúscula «sigma» (el signo de la suma en las fórmulas matemáticas) puede especificarse como 0x3A3 o 'Σ'.

```
PRT(ShortToString(0x3A3)); // "Σ"
PRT(ShortToString('Σ'));   // "Σ"
```

int StringToShortArray(const string text, ushort &array[], int start = 0, int count = -1)

La función convierte una cadena en una secuencia de códigos de caracteres *ushort* que se copian en la ubicación especificada del array: empezando por el elemento numerado *start* (0 por defecto, es decir, el principio del array) y en la cantidad de *count*.

> Nota: el parámetro *start* se refiere a la posición en el array, no en la cadena. Si desea convertir parte de una cadena, primero debe extraerla utilizando la función *StringSubstr*.

Si el parámetro *count* es igual a -1 (o WHOLE_ARRAY), se copian todos los caracteres hasta el final de la cadena (incluido el nulo terminal) o los caracteres de acuerdo con el tamaño del array, si es de tamaño fijo.

En el caso de un array dinámico, su tamaño se incrementará automáticamente si es necesario. Si el tamaño de un array dinámico es mayor que la longitud de la cadena, no se reduce el tamaño del array.

Para copiar caracteres sin un nulo de terminación debe llamar explícitamente a *StringLen* como argumento *count*. En caso contrario, la longitud del array será 1 más que la longitud de la cadena (y 0 en el último elemento).

La función devuelve el número de caracteres copiados.

```
...
ushort array1[], array2[]; // dynamic arrays
ushort text[5];            // fixed size array
string alphabet = "ABCDEAБВГД";
// copy with the terminal '0'
PRT(StringToShortArray(alphabet, array1)); // 11
ArrayPrint(array1); // 65   66   67   68   69 1040 1041 1042 1043 1044    0
// copy without the terminal '0'
PRT(StringToShortArray(alphabet, array2, 0, StringLen(alphabet))); // 10
ArrayPrint(array2); // 65   66   67   68   69 1040 1041 1042 1043 1044
// copy to a fixed array
PRT(StringToShortArray(alphabet, text)); // 5
ArrayPrint(text); // 65 66 67 68 69
// copy beyond the previous limits of the array
// (elements [11-19] will be random)
PRT(StringToShortArray(alphabet, array2, 20)); // 11
ArrayPrint(array2);
/*
[ 0]    65    66    67    68    69  1040  1041  1042
      1043  1044     0     0     0     0     0 14245
[16] 15102 37754 48617 54228    65    66    67    68
        69  1040  1041  1042  1043  1044     0
*/
```

Tenga en cuenta que, si la posición para copiar supera el tamaño del array, los elementos intermedios se asignarán pero no se inicializarán. En consecuencia, pueden contener datos aleatorios (resaltados en amarillo más arriba).

string ShortArrayToString(const ushort &array[], int start = 0, int count = -1)

La función convierte parte del array con códigos de caracteres en una cadena. El rango de los elementos del array se establece mediante los parámetros *start* y *count*, es decir, la posición inicial y la cantidad, respectivamente. El parámetro *start* debe estar comprendido entre 0 y el número de elementos del array menos 1. Si *count* es igual a -1 (o WHOLE_ARRAY) se copian todos los elementos hasta el final del array o hasta el primer elemento nulo.

Utilizando el mismo ejemplo de *StringSymbols.mq5*, vamos a intentar convertir un array en la cadena *array2*, que tiene un tamaño de 30.

```
...
string s = ShortArrayToString(array2, 0, 30);
PRT(s); // "ABCDEAБВГД", additional random characters may appear here
```

Dado que en el array *array2* la cadena «ABCDEABCD» se copió dos veces, y concretamente, la primera vez al principio y la segunda vez en el desplazamiento 20, los caracteres intermedios serán aleatorios y podrán formar una cadena más larga que la nuestra.

int StringToCharArray(const string text, uchar &array[], int start = 0, int count = -1, uint codepage = CP_ACP)

La función convierte la cadena *text* en una secuencia de caracteres de un solo byte que se copian en la ubicación especificada del array: empezando por el elemento numerado *start* (0 por defecto, es decir, el principio del array) y en la cantidad de *count*. El proceso de copia convierte los caracteres de

Unicode a la página de códigos seleccionada *codepage* (por defecto, CP_ACP, que significa el idioma del sistema operativo Windows (más adelante encontrará más información al respecto)).

Si el parámetro *count* es igual a -1 (o WHOLE_ARRAY), se copian todos los caracteres hasta el final de la cadena (incluido el nulo terminal) o de acuerdo con el tamaño del array, si es de tamaño fijo.

En el caso de un array dinámico, su tamaño se incrementará automáticamente si es necesario. Si el tamaño de un array dinámico es mayor que la longitud de la cadena, no se reduce el tamaño del array.

Para copiar caracteres sin un nulo de terminación debe llamar explícitamente a *StringLen* como argumento *count*.

La función devuelve el número de caracteres copiados.

Consulte la lista de páginas de códigos válidas para el parámetro *codepage* en la documentación. Estas son algunas de las páginas de códigos ANSI más utilizadas:

Idioma	Código
Alfabeto latino de Europa central	1250
Cirílico	1251
Alfabeto latino de Europa occidental	1252
Griego	1253
Turco	1254
Hebreo	1255
Árabe	1256
Báltico	1257

Así, en ordenadores con idiomas de Europa occidental, CP_ACP es 1252, y, por ejemplo, en ordenadores con ruso, es 1251.

Durante el proceso de conversión, algunos caracteres pueden convertirse con pérdida de información, ya que la tabla Unicode es mucho mayor que ANSI (cada tabla de códigos ANSI tiene 256 caracteres).

En este sentido, CP_UTF8 reviste especial importancia entre todas las constantes CP_***. Permite conservar adecuadamente los caracteres nacionales mediante la codificación de longitud variable: el array resultante sigue almacenando bytes, pero cada carácter nacional puede abarcar varios bytes, escritos en un formato especial. Por ello, la longitud del array puede ser significativamente mayor que la longitud de la cadena. La codificación UTF-8 se utiliza ampliamente en Internet y en diversos programas informáticos. Por cierto: UTF significa «Unicode Transformation Format» (formato de transformación Unicode), y existen otras modificaciones, en particular UTF-16 y UTF-32.

Veremos un ejemplo para *StringToCharArray* una vez que nos familiaricemos con la función «inversa» *CharArrayToString*: su trabajo debe demostrarse conjuntamente.

string CharArrayToString(const uchar &array[], int start = 0, int count = -1, uint codepage = CP_ACP)

La función convierte un array de bytes o parte de él en una cadena. El array debe contener caracteres en una codificación específica. El rango de los elementos del array se establece mediante los parámetros *start* y *count*, es decir, la posición inicial y la cantidad, respectivamente. El parámetro *start* debe estar comprendido entre 0 y el número de elementos del array. Cuando *count* es igual a -1 (o WHOLE_ARRAY) se copian todos los elementos hasta el final del array o hasta el primer elemento nulo.

Veamos cómo funcionan las funciones *StringToCharArray* y *CharArrayToString* con diferentes caracteres nacionales con diferentes configuraciones de página de códigos. Para ello se ha preparado un script de prueba *StringCodepages.mq5*.

Se utilizarán dos líneas como sujetos de prueba, en ruso y en alemán:

```
void OnStart()
{
   Print("Locales");
   uchar bytes1[], bytes2[];

   string german = "straßenführung";
   string russian = "Russian text";
   ...
```

Las copiaremos en los arrays *bytes1* y *bytes2*, y luego las recuperaremos como cadenas.

En primer lugar, convirtamos el texto alemán utilizando la página de códigos europeos 1252.

```
   ...
   StringToCharArray(german, bytes1, 0, WHOLE_ARRAY, 1252);
   ArrayPrint(bytes1);
   // 115 116 114  97 223 101 110 102 252 104 114 117 110 103   0
```

En las copias europeas de Windows, esto equivale a la llamada a una función más sencilla con parámetros por defecto, porque allí CP_ACP = 1252:

```
   StringToCharArray(german, bytes1);
```

A continuación, restauramos el texto del array con la siguiente llamada y nos aseguramos de que todo coincide con el original:

```
   ...
   PRT(CharArrayToString(bytes1, 0, WHOLE_ARRAY, 1252));
   // CharArrayToString(bytes1,0,WHOLE_ARRAY,1252)='straßenführung'
```

Ahora vamos a intentar convertir el texto ruso en la misma codificación europea (o puede llamar a *StringToCharArray(english, bytes2)* en el entorno de Windows donde CP_ACP está configurado en 1252 como página de códigos por defecto):

```
   ...
   StringToCharArray(russian, bytes2, 0, WHOLE_ARRAY, 1252);
   ArrayPrint(bytes2);
   // 63 63 63 63 63 63 63 32 63 63 63 63 63  0
```

Aquí ya puede ver que hubo un problema durante la conversión porque 1252 no tiene cirílico. Restaurar una cadena a partir de un array muestra claramente la esencia:

```
...
PRT(CharArrayToString(bytes2, 0, WHOLE_ARRAY, 1252));
// CharArrayToString(bytes2,0,WHOLE_ARRAY,1252)='??????? ?????'
```

Repitamos el experimento en un entorno ruso condicional, es decir, convertiremos ambas cadenas de ida y vuelta utilizando la página de códigos cirílicos 1251.

```
...
StringToCharArray(russian, bytes2, 0, WHOLE_ARRAY, 1251);
// on Russian Windows, this call is equivalent to a simpler one
// StringToCharArray(russian, bytes2);
// because CP_ACP = 1251
ArrayPrint(bytes2); // this time the character codes are meaningful
// 208 243 241 241 234 232 233  32 210 229 234 241 242   0

// restore the string and make sure it matches the original
PRT(CharArrayToString(bytes2, 0, WHOLE_ARRAY, 1251));
// CharArrayToString(bytes2,0,WHOLE_ARRAY,1251)='Русский Текст'

// and for the German text...
StringToCharArray(german, bytes1, 0, WHOLE_ARRAY, 1251);
ArrayPrint(bytes1);
// 115 116 114  97  63 101 110 102 117 104 114 117 110 103   0
// if we compare this content of bytes1 with the previous version,
// it's easy to see that a couple of characters are affected; here's what happened
// 115 116 114  97 223 101 110 102 252 104 114 117 110 103   0

// restore the string to see the differences visually:
PRT(CharArrayToString(bytes1, 0, WHOLE_ARRAY, 1251));
// CharArrayToString(bytes1,0,WHOLE_ARRAY,1251)='stra?enfuhrung'
// specific German characters were corrupted
```

Se hace así evidente la fragilidad de las codificaciones de un solo byte.

Por último, activemos la codificación CP_UTF8 para ambas cadenas de prueba. Esta parte del ejemplo funcionará de forma estable independientemente de la configuración de Windows.

```
...
StringToCharArray(german, bytes1, 0, WHOLE_ARRAY, CP_UTF8);
ArrayPrint(bytes1);
// 115 116 114  97 195 159 101 110 102 195 188 104 114 117 110 103   0
PRT(CharArrayToString(bytes1, 0, WHOLE_ARRAY, CP_UTF8));
// CharArrayToString(bytes1,0,WHOLE_ARRAY,CP_UTF8)='straßenführung'

StringToCharArray(russian, bytes2, 0, WHOLE_ARRAY, CP_UTF8);
ArrayPrint(bytes2);
// 208 160 209 131 209 129 209 129 208 186 208 184 208 185
//  32 208 162 208 181 208 186 209 129 209 130   0
PRT(CharArrayToString(bytes2, 0, WHOLE_ARRAY, CP_UTF8));
// CharArrayToString(bytes2,0,WHOLE_ARRAY,CP_UTF8)='Русский Текст'
```

Observe que ambas cadenas codificadas en UTF-8 requieren arrays más grandes que las ANSI. Además, el array con el texto ruso se ha hecho 2 veces más largo, porque ahora todas las letras

ocupan 2 bytes. Quienes lo deseen pueden encontrar detalles en fuentes abiertas sobre cómo funciona exactamente la codificación UTF-8. En el contexto de este libro, es importante para nosotros que la API de MQL5 proporcione funciones ya hechas con las que trabajar.

4.2.7 Salida universal de datos formateados a una cadena

Al generar una cadena para mostrarla al usuario, guardarla en un archivo o enviarla por Internet, puede ser necesario incluir en ella los valores de varias variables de distintos tipos. Este problema se puede resolver mediante la conversión explícita de todas las variables al tipo *(string)* y la adición de las cadenas resultantes, pero en este caso, la instrucción de código MQL será larga y difícil de entender. Probablemente sería más conveniente utilizar la función *StringConcatenate*, pero este método no resuelve completamente el problema.

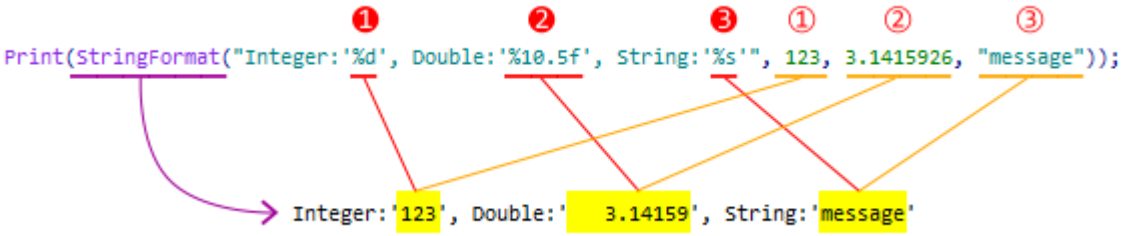
El hecho es que una cadena suele contener no sólo variables, sino también algunas inserciones de texto que actúan como enlaces de conexión y proporcionan la estructura correcta del mensaje global. Resulta que trozos de texto de formato se mezclan con variables. Este tipo de código es difícil de mantener, lo que va en contra de uno de los principios más conocidos de la programación: la separación entre contenido y presentación.

Existe una solución especial para este problema: la función *StringFormat*.

El mismo esquema se aplica a otra función de la API de MQL5: *PrintFormat*.

string StringFormat(const string format, ...)

La función convierte argumentos de tipo integrado arbitrarios en una cadena según el formato especificado. El primer parámetro es la plantilla de la cadena que se desea preparar, en la que se indican de forma especial los lugares para insertar variables y se determina el formato de su salida. Estos comandos de control pueden intercalarse con texto sin formato, que se copia en la cadena de salida sin cambios. Los siguientes parámetros de función, separados por comas, enumeran todas las variables en el orden y los tipos que se han reservado para ellas en la plantilla.



Interacción de los argumentos de cadena de formato y StringFormat

Cada punto de inserción de una variable en una cadena se marca con un especificador de formato, el carácter '%', tras el cual se pueden especificar varios ajustes.

La cadena de formato se analiza de izquierda a derecha. Cuando se encuentra el primer especificador (si existe), el valor del primer parámetro después de la cadena de formato se convierte y se añade a la cadena resultante de acuerdo con la configuración especificada. El segundo especificador hace que el segundo parámetro se convierta y se imprima, y así sucesivamente, hasta el final de la cadena de formato. Todos los demás caracteres del patrón entre los especificadores se copian sin cambios en la cadena resultante.

La plantilla puede no contener ningún especificador, es decir, puede ser una simple cadena. En este caso, debe pasar un argumento ficticio a la función además de la cadena (el argumento no se colocará en la cadena).

Si desea mostrar el signo de porcentaje en la plantilla, deberá escribirlo dos veces seguidas: %%. Si el signo % no se duplica, los caracteres siguientes a % se analizan siempre como un especificador.

Un atributo obligatorio de un especificador es un símbolo que indica el tipo esperado y la interpretación del siguiente argumento de la función. Llamemos condicionalmente a este símbolo T. Entonces, en el caso más simple, un especificador de formato se parece a %T.

De forma generalizada, el especificador puede constar de varios campos más (los campos opcionales se indican entre corchetes):

%[Z][W][.P][M]T

Cada campo cumple su función y adopta uno de los valores permitidos. A continuación, vamos a examinar gradualmente todos los campos.

Tipo T

Para los números enteros se pueden utilizar los siguientes caracteres como T, con una explicación de cómo se muestran los números correspondientes en la cadena:

- c - carácter Unicode
- C - carácter ANSI
- d, i - decimal con signo
- o - octal sin signo
- u - decimal sin signo
- x - hexadecimal sin signo (minúsculas)
- X - hexadecimal sin signo (mayúsculas)

Recordemos que, según el método de almacenamiento interno de datos, los tipos enteros también incluyen los tipos MQL5 integrados *datetime*, *color*, *bool* y las enumeraciones.

Para los números reales, los siguientes símbolos son aplicables como T:

- e - formato científico con exponente ('e' minúscula)
- E - formato científico con exponente ('E' mayúscula)
- f - formato normal
- g - análogo de f o e (se elige la forma más compacta)
- G - análogo de f o E (se elige la forma más compacta)
- a - formato científico con exponente, hexadecimal (minúsculas)
- A - formato científico con exponente, hexadecimal (mayúsculas)

Por último, sólo hay una versión del carácter T disponible para las cadenas: s.

Tamaño de los enteros M

Para los tipos enteros, se puede especificar además de forma explícita el tamaño de la variable en bytes anteponiendo a T uno de los siguientes caracteres o combinaciones de ellos (los hemos generalizado bajo la letra M):

- h - 2 bytes (short, ushort)
- l (L minúscula) - 4 bytes (int, uint)
- I32 (i mayúscula) - 4 bytes (int, uint)
- ll (dos L minúsculas) - 8 bytes (long)
- I64 (i mayúscula) - 8 bytes (long, ulong)

Anchura W

El campo W es un número decimal no negativo que especifica el número mínimo de espacios de caracteres disponibles para el valor formateado. Si el valor de la variable cabe en menos caracteres, se añade el número correspondiente de espacios a la izquierda o a la derecha. Se selecciona el lado izquierdo o derecho en función de la alineación (véase la bandera '-' en el campo Z). Si la bandera '0' está presente, se añade el número correspondiente de ceros delante del valor de salida. Si el número de caracteres de salida es mayor que la anchura especificada, se ignora el ajuste de anchura y el valor de salida no se trunca.

Si se especifica un asterisco '*' como anchura, entonces la anchura del valor de salida debe especificarse en la lista de parámetros pasados. Debe ser un valor del tipo *int* en la posición que precede a la variable que se está formateando.

Precisión P

El campo P también contiene un número decimal no negativo y siempre va precedido de un punto '.'. Para T entero, este campo especifica el número mínimo de dígitos significativos. Si el valor cabe en menos dígitos, se le anteponen ceros.

Para los números reales, P especifica el número de posiciones decimales (por defecto es 6), excepto para los especificadores g y G, para los que P es el número total de dígitos significativos (mantisa y decimal).

Para una cadena, P especifica el número de caracteres que se van a mostrar. Si la longitud de la cadena supera el valor de precisión, la cadena se mostrará truncada.

Si se especifica el asterisco '*' como la precisión, se trata de la misma manera que para la anchura pero controla la precisión.

La anchura y/o precisión fijas, junto con la alineación a la derecha, permiten mostrar los valores en una columna ordenada.

Banderas Z

Por último, el campo Z describe las banderas:

- - (menos) - alineación a la izquierda dentro de la anchura especificada (en ausencia de la bandera, se realiza la alineación a la derecha);
- + (más) - visualización incondicional de un signo+' o '-' antes del valor (sin esta bandera sólo se muestra '-' para los valores negativos);
- 0 - se añaden ceros antes del valor de salida si es menor que la anchura especificada;
- (espacio) - se antepone un espacio al valor visualizado si es con signo y positivo;
- # - controla la visualización de prefijos de números octales y hexadecimales en los formatos *o*, *x* o *X* (por ejemplo, para el formato *x* se añade el prefijo «0x» delante del número visualizado; para el formato *X*, el prefijo «0X»), punto decimal en números reales (formatos *e*, *E*, *a* o *A*) con parte fraccionaria cero, y algunos otros matices.

Puede obtener más información sobre las posibilidades de la salida formateada a una cadena en la documentación.

El número total de parámetros de función no puede ser superior a 64.

Si el número de argumentos pasados a la función es mayor que el número de especificadores, se omiten los argumentos adicionales.

Si el número de especificadores en la cadena de formato es mayor que los argumentos, el sistema intentará mostrar ceros en lugar de los datos que faltan, pero se incrustará una advertencia de texto («parámetro de cadena que falta») para los especificadores de cadena.

Si el tipo del valor no coincide con el tipo del especificador correspondiente, el sistema intentará leer los datos de la variable de acuerdo con el formato y mostrará el valor resultante (puede tener un aspecto extraño debido a una mala interpretación de la representación de bits interna de los datos reales). En el caso de cadenas, puede que se incluya una advertencia («no cadena pasada») en el resultado.

Probemos la función con el script *StringFormat.mq5*.

En primer lugar, vamos a probar diferentes opciones para T y el especificador de tipo de datos.

```
PRT(StringFormat("[Infinity Sign] Unicode (ok): %c; ANSI (overflow): %C",
   '∞', '∞'));
PRT(StringFormat("short (ok): %hi, short (overflow): %hi",
   SHORT_MAX, INT_MAX));
PRT(StringFormat("int (ok): %i, int (overflow): %i",
   INT_MAX, LONG_MAX));
PRT(StringFormat("long (ok): %lli, long (overflow): %i",
   LONG_MAX, LONG_MAX));
PRT(StringFormat("ulong (ok): %llu, long signed (overflow): %lli",
   ULONG_MAX, ULONG_MAX));
```

Aquí se representan tanto los especificadores correctos como los incorrectos (los incorrectos aparecen en segundo lugar en cada instrucción y están marcados con la palabra «desbordamiento», ya que el valor pasado no cabe en el tipo de formato).

Esto es lo que ocurre en el registro (las interrupciones de líneas largas aquí y abajo se han hecho para su publicación):

```
StringFormat(Plain string,0)='Plain string'
StringFormat([Infinity Sign] Unicode: %c; ANSI: %C,'∞','∞')=
   '[Infinity Sign] Unicode (ok): ∞; ANSI (overflow):  '
StringFormat(short (ok): %hi, short (overflow): %hi,SHORT_MAX,INT_MAX)=
   'short (ok): 32767, short (overflow): -1'
StringFormat(int (ok): %i, int (overflow): %i,INT_MAX,LONG_MAX)=
   'int (ok): 2147483647, int (overflow): -1'
StringFormat(long (ok): %lli, long (overflow): %i,LONG_MAX,LONG_MAX)=
   'long (ok): 9223372036854775807, long (overflow): -1'
StringFormat(ulong (ok): %llu, long signed (overflow): %lli,ULONG_MAX,ULONG_MAX)=
   'ulong (ok): 18446744073709551615, long signed (overflow): -1'
```

Todas las instrucciones siguientes son correctas:

```
PRT(StringFormat("ulong (ok): %I64u", ULONG_MAX));
PRT(StringFormat("ulong (HEX): %I64X, ulong (hex): %I64x",
   1234567890123456, 1234567890123456));
PRT(StringFormat("double PI: %f", M_PI));
PRT(StringFormat("double PI: %e", M_PI));
PRT(StringFormat("double PI: %g", M_PI));
PRT(StringFormat("double PI: %a", M_PI));
PRT(StringFormat("string: %s", "ABCDEFGHIJ"));
```

El resultado de su trabajo se muestra a continuación:

```
StringFormat(ulong (ok): %I64u,ULONG_MAX)=
   'ulong (ok): 18446744073709551615'
StringFormat(ulong (HEX): %I64X, ulong (hex): %I64x,1234567890123456,1234567890123456
   'ulong (HEX): 462D53C8ABAC0, ulong (hex): 462d53c8abac0'
StringFormat(double PI: %f,M_PI)='double PI: 3.141593'
StringFormat(double PI: %e,M_PI)='double PI: 3.141593e+00'
StringFormat(double PI: %g,M_PI)='double PI: 3.14159'
StringFormat(double PI: %a,M_PI)='double PI: 0x1.921fb54442d18p+1'
StringFormat(string: %s,ABCDEFGHIJ)='string: ABCDEFGHIJ'
```

Veamos ahora los distintos modificadores.

Con la alineación a la derecha (por defecto) y un ancho de campo fijo (número de caracteres), podemos utilizar diferentes opciones para rellenar la cadena resultante a la izquierda: con un espacio o con ceros. Además, para cualquier alineación, puede activar o desactivar la indicación explícita del signo del valor (de modo que no sólo se muestre menos para negativo, sino también más para positivo).

```
PRT(StringFormat("space padding: %10i", SHORT_MAX));
PRT(StringFormat("0-padding: %010i", SHORT_MAX));
PRT(StringFormat("with sign: %+10i", SHORT_MAX));
PRT(StringFormat("precision: %.10i", SHORT_MAX));
```

Obtenemos lo siguiente en el registro:

```
StringFormat(space padding: %10i,SHORT_MAX)='space padding:      32767'
StringFormat(0-padding: %010i,SHORT_MAX)='0-padding: 0000032767'
StringFormat(with sign: %+10i,SHORT_MAX)='with sign:     +32767'
StringFormat(precision: %.10i,SHORT_MAX)='precision: 0000032767'
```

Para alinear a la izquierda, debe utilizar la bandera '-' (menos), la adición de la cadena a la anchura especificada se produce a la derecha:

```
PRT(StringFormat("no sign (default): %-10i", SHORT_MAX));
PRT(StringFormat("with sign: %+-10i", SHORT_MAX));
```

Resultado:

```
StringFormat(no sign (default): %-10i,SHORT_MAX)='no sign (default): 32767      '
StringFormat(with sign: %+-10i,SHORT_MAX)='with sign: +32767    '
```

Si es necesario, podemos mostrar u ocultar el signo del valor (por defecto, sólo se muestra el signo menos para los valores negativos), añadir un espacio para los valores positivos, y así garantizar el mismo formato cuando necesite mostrar variables en una columna:

```
PRT(StringFormat("default: %i", SHORT_MAX));  // standard
PRT(StringFormat("default: %i", SHORT_MIN));
PRT(StringFormat("space  : % i", SHORT_MAX)); // extra space for positive
PRT(StringFormat("space  : % i", SHORT_MIN));
PRT(StringFormat("sign   : %+i", SHORT_MAX)); // force sign output
PRT(StringFormat("sign   : %+i", SHORT_MIN));
```

Este es el aspecto que tiene en el registro:

```
StringFormat(default: %i,SHORT_MAX)='default: 32767'
StringFormat(default: %i,SHORT_MIN)='default: -32768'
StringFormat(space  : % i,SHORT_MAX)='space  :  32767'
StringFormat(space  : % i,SHORT_MIN)='space  : -32768'
StringFormat(sign   : %+i,SHORT_MAX)='sign   : +32767'
StringFormat(sign   : %+i,SHORT_MIN)='sign   : -32768'
```

Ahora comparemos cómo afectan la anchura y la precisión a los números reales.

```
PRT(StringFormat("double PI: %15.10f", M_PI));
PRT(StringFormat("double PI: %15.10e", M_PI));
PRT(StringFormat("double PI: %15.10g", M_PI));
PRT(StringFormat("double PI: %15.10a", M_PI));

// default precision = 6
PRT(StringFormat("double PI: %15f", M_PI));
PRT(StringFormat("double PI: %15e", M_PI));
PRT(StringFormat("double PI: %15g", M_PI));
PRT(StringFormat("double PI: %15a", M_PI));
```

Resultado:

```
StringFormat(double PI: %15.10f,M_PI)='double PI:    3.1415926536'
StringFormat(double PI: %15.10e,M_PI)='double PI: 3.1415926536e+00'
StringFormat(double PI: %15.10g,M_PI)='double PI:     3.141592654'
StringFormat(double PI: %15.10a,M_PI)='double PI: 0x1.921fb54443p+1'
StringFormat(double PI: %15f,M_PI)='double PI:        3.141593'
StringFormat(double PI: %15e,M_PI)='double PI:    3.141593e+00'
StringFormat(double PI: %15g,M_PI)='double PI:         3.14159'
StringFormat(double PI: %15a,M_PI)='double PI: 0x1.921fb54442d18p+1'
```

Si no se especifica la anchura explícita, los valores se emiten sin rellenar con espacios.

```
PRT(StringFormat("double PI: %.10f", M_PI));
PRT(StringFormat("double PI: %.10e", M_PI));
PRT(StringFormat("double PI: %.10g", M_PI));
PRT(StringFormat("double PI: %.10a", M_PI));
```

Resultado:

```
StringFormat(double PI: %.10f,M_PI)='double PI: 3.1415926536'
StringFormat(double PI: %.10e,M_PI)='double PI: 3.1415926536e+00'
StringFormat(double PI: %.10g,M_PI)='double PI: 3.141592654'
StringFormat(double PI: %.10a,M_PI)='double PI: 0x1.921fb54443p+1'
```

El establecimiento de la anchura y la precisión de los valores mediante el signo '*' y basándose en argumentos de función adicionales se realiza del siguiente modo:

```
PRT(StringFormat("double PI: %*.*f", 12, 5, M_PI));
PRT(StringFormat("string: %*s", 15, "ABCDEFGHIJ"));
PRT(StringFormat("string: %-*s", 15, "ABCDEFGHIJ"));
```

Tenga en cuenta que se pasan 1 o 2 valores de tipo entero antes del valor de salida, según el número de asteriscos '*' del especificador: puede controlar la precisión y la anchura por separado o ambas juntas.

```
StringFormat(double PI: %*.*f,12,5,M_PI)='double PI:      3.14159'
StringFormat(string: %*s,15,ABCDEFGHIJ)='string:      ABCDEFGHIJ'
StringFormat(string: %-*s,15,ABCDEFGHIJ)='string: ABCDEFGHIJ     '
```

Por último, veamos algunos errores de formato comunes.

```
PRT(StringFormat("string: %s %d %f %s", "ABCDEFGHIJ"));
PRT(StringFormat("string vs int: %d", "ABCDEFGHIJ"));
PRT(StringFormat("double vs int: %d", M_PI));
PRT(StringFormat("string vs double: %s", M_PI));
```

La primera instrucción tiene más especificadores que argumentos. En otros casos, los tipos de especificadores y valores pasados no coinciden. Como resultado, obtenemos la siguiente salida:

```
StringFormat(string: %s %d %f %s,ABCDEFGHIJ)=
   'string: ABCDEFGHIJ 0 0.000000 (missed string parameter)'
StringFormat(string vs int: %d,ABCDEFGHIJ)='string vs int: 0'
StringFormat(double vs int: %d,M_PI)='double vs int: 1413754136'
StringFormat(string vs double: %s,M_PI)=
   'string vs double: (non-string passed)'
```

Disponer de una única cadena de formato en cada llamada a la función *StringFormat* permite utilizarla, en especial, para traducir la interfaz externa de programas y mensajes a distintos idiomas: basta con descargar y sustituir en *StringFormat* varias cadenas de formato (preparadas de antemano) en función de las preferencias del usuario o de la configuración del terminal.

4.3 Trabajar con arrays

Es difícil imaginar cualquier programa, y especialmente uno relacionado con el trading, sin arrays. Ya hemos estudiado los principios generales de la descripción y el uso de arrays en el capítulo sobre arrays. Se complementan orgánicamente con un conjunto de funciones integradas para trabajar con arrays.

Algunos de ellos proporcionan implementaciones listas para usar de las operaciones más habituales con arrays, como encontrar el máximo y el mínimo, ordenar, insertar y eliminar elementos.

Sin embargo, hay una serie de funciones sin las cuales es imposible utilizar arrays de tipos específicos. En particular, un array dinámico debe primero asignar memoria antes de trabajar con él, y los arrays

con datos para búferes indicadores (estudiaremos este tipo de programa MQL en la Parte 5 del libro) utilizan un orden especial de indexación de elementos, establecido por una función especial.

Y empezaremos a ver funciones para trabajar con arrays con la operación de salida al registro. Ya lo vimos en capítulos anteriores del libro y será útil en muchos posteriores.

> Dado que los arrays de MQL5 pueden ser multidimensionales (de 1 a 4 dimensiones), tendremos que referirnos a los números de dimensión más adelante en el texto. Los llamaremos números, empezando por el primero, que es más familiar desde el punto de vista geométrico y que subraya el hecho de que un array debe tener al menos una dimensión (aunque esté vacía). No obstante, los elementos del array para cada dimensión se numeran, como es habitual en MQL5 (y en muchos otros lenguajes de programación), partiendo de cero. Así, para un array descrito como *array[5][10]*, la primera dimensión es 5 y la segunda, 10.

4.3.1 Registrar arrays

La impresión de variables, arrays y mensajes sobre el estado de un programa MQL en el registro es el medio más sencillo para informar al usuario, depurar y diagnosticar problemas. En cuanto al array, podemos implementar la impresión por elementos utilizando la función *Print* que ya conocemos de los scripts de demostración. Lo describiremos formalmente un poco más adelante, en la sección sobre interacción con el usuario.

Sin embargo, es más conveniente confiar toda la rutina relacionada con la iteración sobre los elementos y su formateo preciso al entorno MQL5. Para ello, la API ofrece una función *ArrayPrint* especial.

Ya hemos visto ejemplos de cómo trabajar con esta función en la sección Utilización de arrays. Hablemos ahora de sus capacidades con más detalle.

```
void ArrayPrint(const void &array[], uint digits = _Digits, const string separator = NULL,
 ulong start = 0, ulong count = WHOLE_ARRAY,
 ulong flags = ARRAYPRINT_HEADER | ARRAYPRINT_INDEX | ARRAYPRINT_LIMIT | ARRAYPRINT_DATE |
ARRAYPRINT_SECONDS)
```

La función registra un array utilizando la configuración especificada. El array debe ser uno de los tipos integrados o un tipo de estructura simple. Una estructura simple es una estructura con campos de tipos integrados, a excepción de las cadenas y los arrays dinámicos. La presencia de objetos de clase y punteros en la composición de la estructura la saca de la categoría simple.

El array debe tener una dimensión de 1 o 2. El formato se ajusta automáticamente a la configuración del array y, si es posible, lo muestra de forma visual (véase más abajo). A pesar de que MQL5 admite arrays con dimensiones de hasta 4, la función no muestra arrays con 3 o más dimensiones, ya que es difícil representarlos de forma «plana». Esto ocurre sin generar errores en el paso de compilación o ejecución del programa.

Todos los parámetros, excepto el primero, pueden omitirse, y para ellos se definen valores por defecto.

El parámetro *digits* se utiliza para arrays de números reales y para campos numéricos de estructuras. Establece el número de caracteres mostrados en la parte fraccionaria de los números. El valor por defecto es una de las variables de gráfico predefinidas, *_Digits*, que es el número de decimales del precio del símbolo del gráfico actual.

El carácter de separación *separator* se utiliza para designar columnas cuando se muestran campos en un array de estructuras. Con el valor por defecto (NULL), la función utiliza un espacio como separador.

Los parámetros *start* y *count* establecen el número del elemento inicial y el número de elementos que se van a imprimir, respectivamente. Por defecto, la función imprime todo el array, pero el resultado puede verse afectado de forma adicional por la presencia de la bandera ARRAYPRINT_LIMIT (véase más abajo).

El parámetro *flags* acepta una combinación de banderas que controlan varias características de visualización. He aquí algunas de ellas:

- ARRAYPRINT_HEADER imprime el encabezado con los nombres de los campos de la estructura delante del array de estructuras; no afecta a los arrays de no estructuras.
- ARRAYPRINT_INDEX muestra los índices de los elementos por dimensiones (para arrays unidimensionales, los índices se muestran a la izquierda; para arrays bidimensionales se muestran a la izquierda y arriba).
- ARRAYPRINT_LIMIT se utiliza para arrays grandes, y la salida se limita a los cien primeros y los cien últimos registros (este límite está activado por defecto).
- ARRAYPRINT_DATE se utiliza para valores del tipo *datetime* para mostrar la fecha.
- ARRAYPRINT_MINUTES se utiliza para valores del tipo *datetime* para mostrar la hora al minuto más cercano.
- ARRAYPRINT_SECONDS se utiliza para valores del tipo *datetime* para mostrar la hora al segundo más cercano.

Los valores del tipo *datetime* semuestran por defecto en el formato ARRAYPRINT_DATE | ARRAYPRINT_SECONDS.

Los valores del tipo *color* se muestran en formato hexadecimal.

Los valores de enumeración se muestran como números enteros.

La función no produce arrays anidados, estructuras ni punteros a objetos. En lugar de ellos aparecen tres puntos.

El script *ArrayPrint.mq5* demuestra cómo funciona.

La función *OnStart* proporciona definiciones de varios arrays (unidimensionales, bidimensionales y tridimensionales), cuya salida se realiza mediante *ArrayPrint* (con la configuración predeterminada).

```
void OnStart()
{
   int array1D[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   double array2D[][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}};
   double array3D[][3][5] =
   {
      {{ 1,  2,  3,  4,  5}, { 6,  7,  8,  9, 10}, {11, 12, 13, 14, 15}},
      {{16, 17, 18, 19, 20}, {21, 22, 23, 24, 25}, {26, 27, 28, 29, 30}},
   };

   Print("array1D");
   ArrayPrint(array1D);
   Print("array2D");
   ArrayPrint(array2D);
   Print("array3D");
   ArrayPrint(array3D);
   ...
}
```

Obtendremos las siguientes líneas en el registro:

```
array1D
 1  2  3  4  5  6  7  8  9 10
array2D
         [,0]     [,1]     [,2]     [,3]     [,4]
[0,]  1.00000  2.00000  3.00000  4.00000  5.00000
[1,]  6.00000  7.00000  8.00000  9.00000 10.00000
array3D
```

El array *array1D* no es lo suficientemente grande (cabe en una fila), por lo que no se muestran los índices correspondientes.

El array *array2D* tiene varias filas (índices), y por lo tanto se muestran sus índices (ARRAYPRINT_INDEX está activado por defecto).

Tenga en cuenta que el script se ejecutó en el gráfico EURUSD con precios de cinco dígitos, *_Digits=5*, lo que afecta al formato de los valores de tipo *double*.

El array *array3D* se ignora: no se ha generado ninguna fila para él.

Además, las estructuras *Pair* y *SimpleStruct* están definidas en el script:

```
struct Pair
{
   int x, y;
};

struct SimpleStruct
{
   double value;
   datetime time;
   int count;
   ENUM_APPLIED_PRICE price;
   color clr;
   string details;
   void *ptr;
   Pair pair;
};
```

SimpleStruct contiene campos de tipos integrados, un puntero a *void*, así como un campo de tipo *Pair*.

En la función *OnStart* se crea un array del tipo *SimpleStruct* y se emite utilizando *ArrayPrint* en dos modos: con la configuración predeterminada y con la personalizada (el número de dígitos después de la «coma» es 3, el separador es «;», el formato para *datetime* es sólo fecha).

```
void OnStart()
{
   ...
   SimpleStruct simple[] =
   {
      { 12.57839, D'2021.07.23 11:15', 22345, PRICE_MEDIAN, clrBlue, "text message"},
      {135.82949, D'2021.06.20 23:45', 8569, PRICE_TYPICAL, clrAzure},
      { 1087.576, D'2021.05.15 10:01:30', -3298, PRICE_WEIGHTED, clrYellow, "note"},
   };
   Print("SimpleStruct (default)");
   ArrayPrint(simple);

   Print("SimpleStruct (custom)");
   ArrayPrint(simple, 3, ";", 0, WHOLE_ARRAY, ARRAYPRINT_DATE);
}
```

El resultado es el siguiente:

```
SimpleStruct (default)
       [value]              [time] [count] [type]    [clr]      [details] [ptr] [pair
[0]   12.57839 2021.07.23 11:15:00   22345      5 00FF0000 "text message"   ...    ..
[1]  135.82949 2021.06.20 23:45:00    8569      6 00FFFFF0 null             ...    ..
[2] 1087.57600 2021.05.15 10:01:30   -3298      7 0000FFFF "note"           ...    ..
SimpleStruct (custom)
  12.578;2021.07.23;  22345;     5;00FF0000;"text message";  ...;   ...
 135.829;2021.06.20;   8569;     6;00FFFFF0;null          ;  ...;   ...
1087.576;2021.05.15;  -3298;     7;0000FFFF;"note"        ;  ...;   ...
```

Tenga en cuenta que el registro que utilizamos en este caso y en las secciones anteriores se genera en el terminal y está a disposición del usuario en la pestaña *Experts* de la ventana *Toolbox*. No obstante, en el futuro nos familiarizaremos con el probador, que proporciona el mismo entorno

de ejecución para ciertos tipos de programas MQL (indicadores y Asesores Expertos) que el propio terminal. Si se lanzan en el probador, la función *ArrayPrint* y otras funciones relacionadas, que se describen en la sección Interacción con el usuario enviarán mensajes al registro de agentes de prueba.

Hasta ahora hemos trabajado, y seguiremos haciéndolo durante algún tiempo, sólo con scripts, y éstos sólo pueden ejecutarse en el terminal.

4.3.2 Arrays dinámicos

Los arrays dinámicos pueden cambiar de tamaño durante la ejecución del programa a petición del programador. Recordemos que para describir un array dinámico, debe dejar vacío el primer par de corchetes después del identificador del array. MQL5 requiere que todas las dimensiones subsiguientes (si hay más de una) tengan un tamaño fijo especificado con una constante.

Es imposible aumentar dinámicamente el número de elementos para cualquier dimensión «más antigua» que la primera. Además, debido a la estricta descripción del tamaño, los arrays tienen una forma «cuadrada», es decir, por ejemplo, es imposible construir un array bidimensional con columnas o filas de longitudes diferentes. Si alguna de estas restricciones es crítica para la implementación del algoritmo, no debería usar arrays MQL5 estándar, sino sus propias estructuras o clases escritas en MQL5.

Tenga en cuenta que si un array no tiene un tamaño en la primera dimensión, pero sí tiene una lista de inicialización que le permite determinar el tamaño, entonces dicho array es un array de tamaño fijo, no dinámico.

Por ejemplo, en la sección anterior, utilizamos el array *array1D*:

```
int array1D[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Debido a la lista de inicialización, su tamaño es conocido por el compilador, y por lo tanto el array es fijo.

A diferencia de este sencillo ejemplo, no siempre es fácil determinar si un array concreto de un programa real es dinámico. En particular, un array puede pasarse como parámetro a una función. Sin embargo, puede ser importante saber si un array es dinámico porque la memoria se puede asignar manualmente llamando a *ArrayResize* sólo para este tipo de arrays.

En estos casos, la función *ArrayIsDynamic* le permite determinar el tipo de array.

Veamos algunas descripciones técnicas de funciones para trabajar con arrays dinámicos y, a continuación, probémoslas utilizando el script *ArrayDynamic.mq5*.

bool ArrayIsDynamic(const void &array[])

La función comprueba si el array pasado es dinámico. Un array puede tener cualquier dimensión permitida de 1 a 4. Los elementos del array pueden ser de cualquier tipo.

La función devuelve *true* para un array dinámico, o *false* en otros casos (array fijo, o array con series temporales controlado por el propio terminal o por el indicador).

int ArrayResize(void &array[], int size, int reserve = 0)

La función establece el nuevo *size* en la primera dimensión del *array* dinámico. Un array puede tener cualquier dimensión permitida de 1 a 4. Los elementos del array pueden ser de cualquier tipo.

Si el parámetro *reserve* es mayor que cero, se asigna memoria para el array con una reserva para el número de elementos especificado. Esto hace que pueda aumentar la velocidad del programa que tiene muchas llamadas a funciones consecutivas. Hasta que el nuevo tamaño solicitado del array no supere el actual teniendo en cuenta la reserva, no habrá reasignación de memoria física y los nuevos elementos se tomarán de la reserva.

La función devuelve el nuevo tamaño del array si su modificación se ha realizado correctamente, o -1 en caso de error.

Si la función se aplica a un array o serie temporal fija, su tamaño no cambia. En estos casos, si el tamaño solicitado es menor o igual que el tamaño actual del array, la función devolverá el valor del parámetro *size*; en caso contrario, devolverá -1.

Al aumentar el tamaño de un array ya existente se conservan todos los datos de sus elementos. Los elementos añadidos no se inicializan con nada y pueden contener datos arbitrarios incorrectos («basura»).

Estableciendo el tamaño del array a 0, *ArrayResize(array, 0),* no libera la memoria realmente asignada para ello, incluida una posible reserva. Esta llamada sólo restablecerá los metadatos del array. Esto se hace con el fin de optimizar futuras operaciones con el array. Para forzar la liberación de la memoria, utilice *ArrayFree* (véase más abajo).

Es importante entender que el parámetro *reserve* no se utiliza cada vez que se llama a la función, sino sólo en aquellos momentos en los que realmente se realiza la reasignación de memoria, es decir, cuando el tamaño solicitado supera la capacidad actual del array incluyendo la reserva. Para mostrar visualmente cómo funciona esto, crearemos una copia incompleta del objeto array interno e implementaremos la función gemela *ArrayResize* para él, y también los análogos *ArrayFree* y *ArraySize*, a fin de tener un conjunto de herramientas completo.

```
template<typename T>
struct DynArray
{
   int size;
   int capacity;
   T memory[];
};

template<typename T>
int DynArraySize(DynArray<T> &array)
{
   return array.size;
}

template<typename T>
void DynArrayFree(DynArray<T> &array)
{
   ArrayFree(array.memory);
   ZeroMemory(array);
}

template<typename T>
int DynArrayResize(DynArray<T> &array, int size, int reserve = 0)
{
   if(size > array.capacity)
   {
      static int temp;
      temp = array.capacity;
      long ul = (long)GetMicrosecondCount();
      array.capacity = ArrayResize(array.memory, size + reserve);
      array.size = MathMin(size, array.capacity);
      ul -= (long)GetMicrosecondCount();
      PrintFormat("Reallocation: [%d] -> [%d], done in %d µs",
         temp, array.capacity, -ul);
   }
   else
   {
      array.size = size;
   }
   return array.size;
}
```

Una ventaja de la función *DynArrayResize* frente a *ArrayResize* integrada es que aquí insertamos una impresión de depuración para aquellas situaciones en las que se reasigna la capacidad interna del array.

Ahora podemos tomar el ejemplo estándar para la función *ArrayResize* de la documentación MQL5 y reemplazar las llamadas a la función integrada por análogas «hechas a sí mismas» con el prefijo «Dyn». El resultado modificado se presenta en el script *ArrayCapacity.mq5*.

```
void OnStart()
{
   ulong start = GetTickCount();
   ulong now;
   int   count = 0;

   DynArray<double> a;

 // fast option with memory reservation
   Print("--- Test Fast: ArrayResize(arr,100000,100000)");

   DynArrayResize(a, 100000, 100000);

   for(int i = 1; i <= 300000 && !IsStopped(); i++)
   {
 // set the new size and reserve to 100000 elements
      DynArrayResize(a, i, 100000);
 // on "round" iterations, show the size of the array and the elapsed time
      if(DynArraySize(a) % 100000 == 0)
      {
         now = GetTickCount();
         count++;
         PrintFormat("%d. ArraySize(arr)=%d Time=%d ms",
            count, DynArraySize(a), (now - start));
         start = now;
      }
   }
   DynArrayFree(a);

 // now this is a slow option without redundancy (with less redundancy)
   count = 0;
   start = GetTickCount();
   Print("---- Test Slow: ArrayResize(slow,100000)");

   DynArrayResize(a, 100000, 100000);

   for(int i = 1; i <= 300000 && !IsStopped(); i++)
   {
 // set new size but with 100 times smaller margin: 1000
      DynArrayResize(a, i, 1000);
 // on "round" iterations, show the size of the array and the elapsed time
      if(DynArraySize(a) % 100000 == 0)
      {
         now = GetTickCount();
         count++;
         PrintFormat("%d. ArraySize(arr)=%d Time=%d ms",
            count, DynArraySize(a), (now - start));
         start = now;
      }
   }
}
```

La única diferencia significativa es la siguiente: en la versión lenta, la llamada *ArrayResize(a, i)* se sustituye por la más moderada *DynArrayResize(a, i, 1000)*, es decir, la redistribución se solicita no en cada iteración, sino cada 1000 (de lo contrario, el registro se llenaría en exceso de mensajes).

Después de ejecutar el script, veremos los siguientes tiempos en el registro (los intervalos de tiempo absolutos dependen de su ordenador, pero nos interesa la diferencia entre las variantes de rendimiento con y sin la reserva):

```
--- Test Fast: ArrayResize(arr,100000,100000)
Reallocation: [0] -> [200000], done in 17 µs
1. ArraySize(arr)=100000 Time=0 ms
2. ArraySize(arr)=200000 Time=0 ms
Reallocation: [200000] -> [300001], done in 2296 µs
3. ArraySize(arr)=300000 Time=0 ms
---- Test Slow: ArrayResize(slow,100000)
Reallocation: [0] -> [200000], done in 21 µs
1. ArraySize(arr)=100000 Time=0 ms
2. ArraySize(arr)=200000 Time=0 ms
Reallocation: [200000] -> [201001], done in 1838 µs
Reallocation: [201001] -> [202002], done in 1994 µs
Reallocation: [202002] -> [203003], done in 1677 µs
Reallocation: [203003] -> [204004], done in 1983 µs
Reallocation: [204004] -> [205005], done in 1637 µs
...
Reallocation: [295095] -> [296096], done in 2921 µs
Reallocation: [296096] -> [297097], done in 2189 µs
Reallocation: [297097] -> [298098], done in 2152 µs
Reallocation: [298098] -> [299099], done in 2767 µs
Reallocation: [299099] -> [300100], done in 2115 µs
3. ArraySize(arr)=300000 Time=219 ms
```

La ganancia de tiempo es significativa. Además, vemos en qué iteraciones y cómo se modifica la capacidad real del array (reserva).

void ArrayFree(void &array[])

La función libera toda la memoria del array dinámico pasado (incluida la posible reserva establecida mediante el tercer parámetro de la función *ArrayResize*) y pone a cero el tamaño de su primera dimensión.

En teoría, los arrays en MQL5 liberan memoria automáticamente cuando finaliza la ejecución del algoritmo en el bloque actual. No importa si un array está definido localmente (dentro de funciones) o globalmente, si es fijo o dinámico, ya que el sistema liberará la memoria por sí mismo en cualquier caso, sin requerir acciones explícitas del programador.

Por lo tanto, no es necesario llamar a esta función. No obstante, hay situaciones en las que un array se utiliza en un algoritmo para volver a rellenarse con algo desde cero, es decir, es necesario liberarlo antes cada vez que se vuelva a rellenar. Esta función puede así resultarle útil.

Tenga en cuenta que si los elementos del array contienen punteros a objetos asignados dinámicamente, la función no los borra: el programador debe llamar a *delete* para ellos (véase más adelante).

Vamos a probar las funciones comentadas anteriormente: *ArrayIsDynamic*, *ArrayResize*, *ArrayFree*.

En el script *ArrayDynamic.mq5* se escribe la función *ArrayExtend*, que incrementa el tamaño del array dinámico en 1 y escribe el valor pasado en el nuevo elemento.

```
template<typename T>
void ArrayExtend(T &array[], const T value)
{
   if(ArrayIsDynamic(array))
   {
      const int n = ArraySize(array);
      ArrayResize(array, n + 1);
      array[n] = (T)value;
   }
}
```

La función *ArrayIsDynamic* se utiliza para asegurarse de que el array sólo se actualiza si es dinámico. Esto se hace en una sentencia condicional. La función *ArrayResize* le permite cambiar el tamaño del array, y la función *ArraySize* se utiliza para averiguar el tamaño actual (esto se abordará en la siguiente sección).

En la función principal del script, aplicaremos *ArrayExtend* para arrays de diferentes categorías: dinámicos y fijos.

```
void OnStart()
{
   int dynamic[];
   int fixed[10] = {}; // padding with zeros

   PRT(ArrayResize(fixed, 0)); // warning: not applicable for fixed array

   for(int i = 0; i < 10; ++i)
   {
      ArrayExtend(dynamic, (i + 1) * (i + 1));
      ArrayExtend(fixed, (i + 1) * (i + 1));
   }

   Print("Filled");
   ArrayPrint(dynamic);
   ArrayPrint(fixed);

   ArrayFree(dynamic);
   ArrayFree(fixed); // warning: not applicable for fixed array

   Print("Free Up");
   ArrayPrint(dynamic); // outputs nothing
   ArrayPrint(fixed);
   ...
}
```

En las líneas de código que llaman a las funciones que no pueden utilizarse para arrays fijos, el compilador genera un aviso de «no puede utilizarse para arrays asignados estáticamente». Es importante señalar que no hay avisos de este tipo dentro de la función *ArrayExtend* porque se puede pasar a la función un array de cualquier categoría. Por eso lo comprobamos en *ArrayIsDynamic*.

Después de un bucle en *OnStart*,, el array *dynamic* se expandirá hasta 10 y obtendrá los elementos iguales a los índices al cuadrado. El array *fixed* permanecerá lleno de ceros y no cambiará de tamaño.

Liberar un array fijo con *ArrayFree* no tendrá ningún efecto, y el array dinámico de hecho se borrará. En este caso, el último intento de imprimirlo no producirá ninguna línea en el registro.

Veamos el resultado de la ejecución del script.

```
ArrayResize(fixed,0)=0
Filled
  1   4   9  16  25  36  49  64  81 100
0 0 0 0 0 0 0 0 0 0
Free Up
0 0 0 0 0 0 0 0 0 0
```

De particular interés son los arrays dinámicos con punteros a objetos. Definamos una clase ficticia sencilla *Dummy* y creemos un array de punteros a dichos objetos.

```
class Dummy
{
};

void OnStart()
{
   ...
   Dummy *dummies[] = {};
   ArrayExtend(dummies, new Dummy());
   ArrayFree(dummies);
}
```

Después de extender el array *dummy* con un nuevo puntero, lo liberamos con *ArrayFree*, pero hay entradas en el registro del terminal que indican que el objeto se quedó en memoria.

```
1 undeleted objects left
1 object of type Dummy left
24 bytes of leaked memory
```

El hecho es que la función sólo gestiona la memoria asignada al array. En este caso, esta memoria contenía un puntero, pero lo que señala no pertenece al array. En otras palabras: si el array contiene punteros a objetos «externos», tendrá que ocuparte de ellos usted mismo. Por ejemplo:

```
for(int i = 0; i < ArraySize(dummies); ++i)
{
   delete dummies[i];
}
```

Este borrado debe iniciarse antes de llamar a *ArrayFree*.

Para abreviar la entrada, puede utilizar las siguientes macros (bucle sobre los elementos, llamada a *delete* para cada uno de ellos):

```
#define FORALL(A) for(int _iterator_ = 0; _iterator_ < ArraySize(A); ++_iterator_)
#define FREE(P) { if(CheckPointer(P) == POINTER_DYNAMIC) delete (P); }
#define CALLALL(A, CALL) FORALL(A) { CALL(A[_iterator_]) }
```

El borrado de punteros se simplifica entonces a la siguiente notación:

```
...
CALLALL(dummies, FREE);
ArrayFree(dummies);
```

Como solución alternativa, puede utilizar una clase envolvente de punteros como *AutoPtr*, de la que hablamos en la sección Plantillas de tipos de objeto. El array debe entonces declararse con el tipo *AutoPtr*. Dado que el array almacenará objetos envoltorio, no punteros, cuando se vacíe el array se llamará automáticamente a los destructores de cada «envoltorio» y se liberará a su vez la memoria de punteros de los mismos.

4.3.3 Medición de arrays

Una de las principales características de un array es su tamaño, es decir, el número total de elementos que lo componen. Es importante tener en cuenta que, para los arrays multidimensionales, el tamaño es el producto de las longitudes de todas sus dimensiones.

Para arrays fijos, puede calcular su tamaño en la fase de compilación utilizando la siguiente construcción del lenguaje basada en operadores sizeof:

```
sizeof(array) / sizeof(type)
```

donde *array* es un identificador y *type* es el tipo de array.

Por ejemplo, si se define un array en el código *fixed*:

```
int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};
```

entonces su tamaño es:

```
int n = sizeof(fixed) / sizeof(int); // 8
```

Para arrays dinámicos, esta regla no funciona, ya que el operador *sizeof* siempre genera el mismo tamaño del objeto array dinámico interno: 52 bytes.

> Tenga en cuenta que en las funciones, todos los parámetros de array se representan internamente como objetos envoltorio de array dinámicos. Esto se hace para que se pueda pasar a la función un array con cualquier método de asignación de memoria, incluido uno fijo. Por eso *sizeof(array)* devolverá 52 para el array de parámetros, aunque se le haya pasado un array de tamaño fijo.
>
> La presencia de «envoltorios» sólo afecta a *sizeof*. La función *ArrayIsDynamic* siempre determina correctamente la categoría del argumento real pasado a través del array de parámetros.

Para obtener el tamaño de cualquier array en la fase de ejecución del programa, utilice la función *ArraySize*.

int ArraySize(const void &array[])

La función devuelve el número total de elementos del array. La dimensión y el tipo de array pueden ser cualesquiera. Para un array unidimensional, la llamada a la función es similar a *ArrayRange(array, 0)* (véase más abajo).

Si el array se distribuyó con una reserva (el tercer parámetro de la función *ArrayResize*), su valor no se tiene en cuenta.

Hasta que se asigne memoria para el array dinámico utilizando *ArrayResize*, la función *ArraySize* devolverá 0. Además, el tamaño pasa a ser cero después de llamar a *ArrayFree* para el array

int ArrayRange(const void &array[], int dimension)

La función *ArrayRange* devuelve el número de elementos de la dimensión de array especificada. La dimensión y el tipo de array pueden ser cualesquiera. El parámetro *dimension* debe estar comprendido entre 0 y el número de dimensiones del array menos 1. El índice 0 corresponde a la primera dimensión, el índice 1 a la segunda, y así sucesivamente.

El producto de todos los valores de *ArrayRange(array, i)* con *i* ejecutándose en todas las dimensiones da *ArraySize(array)*.

Veamos los ejemplos de las funciones descritas anteriormente (véase el archivo *ArraySize.mq5*).

```
void OnStart()
{
   int dynamic[];
   int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};

   PRT(sizeof(fixed) / sizeof(int));   // 8
   PRT(ArraySize(fixed));              // 8

   ArrayResize(dynamic, 10);

   PRT(sizeof(dynamic) / sizeof(int)); // 13 (incorrect)
   PRT(ArraySize(dynamic));            // 10

   PRT(ArrayRange(fixed, 0));          // 2
   PRT(ArrayRange(fixed, 1));          // 4

   PRT(ArrayRange(dynamic, 0));        // 10
   PRT(ArrayRange(dynamic, 1));        // 0
   int size = 1;
   for(int i = 0; i < 2; ++i)
   {
      size *= ArrayRange(fixed, i);
   }
   PRT(size == ArraySize(fixed));      // true
}
```

4.3.4 Inicializar y rellenar arrays

Describir un array con una lista de inicialización sólo es posible para arrays de tamaño fijo. Los arrays dinámicos sólo pueden poblarse después de asignarles memoria mediante la función *ArrayResize*. Se rellenan con las funciones *ArrayInitialize* o *ArrayFill*, que también son útiles en un programa cuando se desea reemplazar valores en bloque en arrays fijos o series temporales.

Tras la descripción de las funciones se ofrecen ejemplos de uso de las mismas.

int ArrayInitialize(type &array[], type value)

La función establece todos los elementos del array en el valor especificado. Sólo se admiten arrays de tipos numéricos integrados (*char*, *uchar*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *bool*, *color*, *datetime*, *float*, *double*). Los arrays de cadenas, estructuras y punteros no pueden rellenarse de esta forma: tendrán que implementar sus propias funciones de inicialización. Un array puede ser multidimensional.

La función devuelve el número de elementos.

Si el array dinámico se asigna con una reserva (el tercer parámetro de la función *ArrayResize*), entonces la reserva no se inicializa.

Si, una vez inicializado el array, se aumenta su tamaño mediante *ArrayResize*, los elementos añadidos no se ajustarán automáticamente a *value*. Pueden rellenarse con la función *ArrayFill*.

void ArrayFill(type &array[], int start, int count, type value)

La función rellena un array numérico o parte de él con un valor especificado. Parte del array viene dado por los parámetros *start* y *count*, que indican el número inicial del elemento y el número de elementos que se van a rellenar, respectivamente.

A la función no le importa si se establece el orden de numeración de los elementos del array como en series temporales o no: esta propiedad se ignora. En otras palabras: los elementos de un array se cuentan siempre desde su principio hasta su final.

Para un array multidimensional, el parámetro *start* puede obtenerse convirtiendo las coordenadas en todas las dimensiones en un índice de paso para un array unidimensional equivalente. Así, para un array bidimensional, los elementos con el índice 0 en la primera dimensión se ubican primero en memoria, después estarán los elementos con el índice 1 en la primera dimensión, y así sucesivamente. La fórmula para calcular *start* es la siguiente:

```
start = D1 * N2 + D2
```

donde D1 y D2 son los índices de la primera y segunda dimensión, respectivamente, y N2 es el número de elementos de la segunda dimensión. D2 pasa de 0 a (N2-1), D1 pasa de 0 a (N1-1). Por ejemplo, en un array *array[3][4]*, el elemento con índices [1][3] es el séptimo de una fila, por lo que la llamada *ArrayFill(array, 7, 2, ...)* llenará dos elementos:*array[1][3]* y a continuación *array[2][0]*. En el diagrama, esto puede representarse de la siguiente manera (cada celda contiene un índice de paso del elemento):

```
       [][0]  [][1]  [][2]  [][3]
[0][]    0      1      2      3
[1][]    4      5      6      7
[2][]    8      9     10     11
```

El script *ArrayFill.mq5* proporciona ejemplos de uso de las funciones mencionadas.

```
void OnStart()
{
   int dynamic[];
   int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};

   PRT(ArrayInitialize(fixed, -1));
   ArrayPrint(fixed);
   ArrayFill(fixed, 3, 4, +1);
   ArrayPrint(fixed);

   PRT(ArrayResize(dynamic, 10, 50));
   PRT(ArrayInitialize(dynamic, 0));
   ArrayPrint(dynamic);
   PRT(ArrayResize(dynamic, 50));
   ArrayPrint(dynamic);
   ArrayFill(dynamic, 10, 40, 0);
   ArrayPrint(dynamic);
}
```

A continuación se muestra un posible resultado (los datos aleatorios en elementos no inicializados de un array dinámico serán diferentes):

```
ArrayInitialize(fixed,-1)=8
    [,0][,1][,2][,3]
[0,]  -1  -1  -1  -1
[1,]  -1  -1  -1  -1
    [,0][,1][,2][,3]
[0,]  -1  -1  -1   1
[1,]   1   1   1  -1
ArrayResize(dynamic,10,50)=10
ArrayInitialize(dynamic,0)=10
0 0 0 0 0 0 0 0 0 0
ArrayResize(dynamic,50)=50
[ 0]           0           0           0           0           0
               0           0           0           0           0
[10] -1402885947  -727144693   699739629   172950740 -1326090126
           47384           0           0     4194184           0
[20]           2           0           2           0           0
               0           0  1765933056  2084602885 -1956758056
[30]    73910037 -1937061701          56           0          56
               0     1048601  1979187200       10851           0
[40]           0           0           0  -685178880 -1720475236
       782716519 -1462194191  1434596297   415166825 -1944066819
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

4.3.5 Copiar y editar arrays

En esta sección aprenderemos a utilizar las funciones integradas para insertar y eliminar elementos de arrays, cambiar su orden y copiar arrays enteros.

bool ArrayInsert(void &target[], const void &source[], uint to, uint from = 0, uint count = WHOLE_ARRAY)

La función inserta el número especificado de elementos del array de origen 'source' en el array de destino *target*. La posición de inserción en el array *target* se establece mediante el índice del parámetro *to*. El índice inicial del elemento en el que empezar a copiar del array *source* viene dado por el índice *from*. La constante WHOLE_ARRAY ((uint)-1) del parámetro *count* especifica la transferencia de todos los elementos del array de origen.

Todos los índices y recuentos son relativos a la primera dimensión de los arrays. En otras palabras, en el caso de los arrays multidimensionales, la inserción no se realiza por elementos individuales, sino por toda la configuración descrita por las dimensiones «superiores». Por ejemplo, para un array bidimensional, el valor 1 en el parámetro *count* significa insertar un vector de longitud igual a la segunda dimensión (véase el ejemplo).

Por ello, el array de destino y el array de origen deben tener las mismas configuraciones. De lo contrario, se producirá un error y la copia fallará. Para arrays unidimensionales, esto no es una limitación, pero para arrays multidimensionales, es necesario observar la igualdad de tamaños en dimensiones superiores a la primera. En particular, los elementos del array [][4] no pueden insertarse en el array [][5] y viceversa.

La función sólo es aplicable a arrays de tamaño fijo o dinámico. Edición de series temporales (arrays con series temporales) no puede realizarse con esta función. Está prohibido especificar en los parámetros *target* y *source* el mismo array.

Cuando se insertan en un array fijo, los nuevos elementos desplazan los existentes hacia la derecha y desplazan *count* de los elementos situados más a la derecha hacia el exterior del array. El parámetro *to* debe tener un valor comprendido entre 0 y el tamaño del array menos 1.

Cuando se insertan en un array dinámico, los elementos antiguos también se desplazan a la derecha, pero no desaparecen, porque el propio array se amplía en *count* elementos. El parámetro *to* debe tener un valor comprendido entre 0 y el tamaño del array. Si es igual al tamaño del array, se añaden nuevos elementos al final del array.

Los elementos especificados se copian de un array a otro, es decir, permanecen inalterados en el array original, y sus «dobles» en el nuevo array se convierten en instancias independientes que no guardan relación alguna con los «originales».

La función devuelve *true* si tiene éxito o *false* en caso de error.

Veamos algunos ejemplos (*ArrayInsert.mq5*). La función *OnStart* proporciona descripciones de varios arrays de diferentes configuraciones, tanto fijos como dinámicos.

```
#define PRTS(A) Print(#A, "=", (string)(A) + " / status:" + (string)GetLastError())

void OnStart()
{
   int dynamic[];
   int dynamic2Dx5[][5];
   int dynamic2Dx4[][4];
   int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};
   int insert[] = {10, 11, 12};
   int array[1] = {100};
   ...
```

Para empezar, por comodidad, se introduce una macro que muestra el código de error (obtenido mediante la función *GetLastError*) inmediatamente después de llamar a la instrucción bajo prueba: PRTS. Se trata de una versión ligeramente modificada de la conocida macro PRT.

Los intentos de copiar elementos entre arrays de configuraciones diferentes terminan con el error 4006 (ERR_INVALID_ARRAY).

```
   // you can't mix 1D and 2D arrays
   PRTS(ArrayInsert(dynamic, fixed, 0)); // false:4006, ERR_INVALID_ARRAY
   ArrayPrint(dynamic); // empty
   // you can't mix 2D arrays of different configurations by the second dimension
   PRTS(ArrayInsert(dynamic2Dx5, fixed, 0)); // false:4006, ERR_INVALID_ARRAY
   ArrayPrint(dynamic2Dx5); // empty
   // even if both arrays are fixed (or both are dynamic),
   // size by "higher" dimensions must match
   PRTS(ArrayInsert(fixed, insert, 0)); // false:4006, ERR_INVALID_ARRAY
   ArrayPrint(fixed); // not changed
   ...
```

El índice de destino debe estar dentro del array.

```
   // target index 10 is out of the range or the array 'insert',
   // could be 0, 1, 2, because its size = 3
   PRTS(ArrayInsert(insert, array, 10)); // false:5052, ERR_SMALL_ARRAY
   ArrayPrint(insert); // not changed
   ...
```

Las siguientes son modificaciones de arrays realizadas con éxito:

```
   // copy second row from 'fixed', 'dynamic2Dx4' is allocated
   PRTS(ArrayInsert(dynamic2Dx4, fixed, 0, 1, 1)); // true
   ArrayPrint(dynamic2Dx4);
   // both rows from 'fixed' are added to the end of 'dynamic2Dx4', it expands
   PRTS(ArrayInsert(dynamic2Dx4, fixed, 1)); // true
   ArrayPrint(dynamic2Dx4);
   // memory is allocated for 'dynamic' for all elements 'insert'
   PRTS(ArrayInsert(dynamic, insert, 0)); // true
   ArrayPrint(dynamic);
   // 'dynamic' expands by 1 element
   PRTS(ArrayInsert(dynamic, array, 1)); // true
   ArrayPrint(dynamic);
   // new element will push the last one out of 'insert'
   PRTS(ArrayInsert(insert, array, 1)); // true
   ArrayPrint(insert);
}
```

Esto es lo que aparecerá en el registro:

```
ArrayInsert(dynamic2Dx4,fixed,0,1,1)=true
    [,0][,1][,2][,3]
[0,]   5   6   7   8
ArrayInsert(dynamic2Dx4,fixed,1)=true
    [,0][,1][,2][,3]
[0,]   5   6   7   8
[1,]   1   2   3   4
[2,]   5   6   7   8
ArrayInsert(dynamic,insert,0)=true
10 11 12
ArrayInsert(dynamic,array,1)=true
 10 100  11  12
ArrayInsert(insert,array,1)=true
 10 100  11
```

bool ArrayCopy(void &target[], const void &source[], int to = 0, int from = 0, int count = WHOLE_ARRAY)

La función copia parte o la totalidad del array *source* en el array *target*. El lugar del array *target* en el que se escriben los elementos se especifica mediante el índice del parámetro *to*. El índice inicial del elemento desde el que empezar a copiar del array *source* viene dado por el índice *from*. La constante WHOLE_ARRAY (-1) en el parámetro *count* especifica la transferencia de todos los elementos del array de origen. Si *count* es menor que cero o mayor que el número de elementos que quedan desde la posición *from* hasta el final del array *source*, se copia todo el resto del array.

A diferencia de la función *ArrayInsert*, la función *ArrayCopy* no desplaza los elementos existentes del array receptor, sino que escribe nuevos elementos en las posiciones especificadas sobre los antiguos.

Todos los índices y el número de elementos se establecen teniendo en cuenta la numeración continua de los elementos, independientemente del número de dimensiones de los arrays y de su configuración. En otras palabras, se pueden copiar elementos de arrays multidimensionales a arrays unidimensionales

y viceversa, o entre arrays multidimensionales con diferentes tamaños según las dimensiones «superiores» (véase el ejemplo).

La función funciona con arrays fijos y dinámicos, así como con arrays de series temporales designadas como búferes indicadores.

Se permite copiar elementos de un array a sí mismo. Pero si las áreas *target* y *source* se solapan, debe tener en cuenta que la iteración se realiza de izquierda a derecha.

Un array de destino dinámico se amplía automáticamente según sea necesario. Los arrays fijos conservan sus dimensiones, y lo que se copia debe caber en el array, de lo contrario se producirá un error.

Se admiten arrays de tipos integrados y arrays de estructuras con campos de tipo simple. Para los tipos numéricos, la función intentará convertir los datos si los tipos de origen y destino difieren. Un array de cadenas sólo puede copiarse en otro array de cadenas. Los objetos de clase no están permitidos, pero los punteros a objetos sí pueden copiarse.

La función devuelve el número de elementos copiados (0 en caso de error).

En el script *ArrayCopy.mq5* hay varios ejemplos de uso de la función.

```
class Dummy
{
   int x;
};

void OnStart()
{
   Dummy objects1[5], objects2[5];
 // error: structures or classes with objects are not allowed
   PRTS(ArrayCopy(objects1, objects2));
   ...
```

Los arrays con objetos generan un error de compilación que indica que «no se permiten estructuras o clases que contengan objetos», pero los punteros pueden copiarse.

```
   Dummy *pointers1[5], *pointers2[5];
   for(int i = 0; i < 5; ++i)
   {
      pointers1[i] = &objects1[i];
   }
   PRTS(ArrayCopy(pointers2, pointers1)); // 5 / status:0
   for(int i = 0; i < 5; ++i)
   {
      Print(i, " ", pointers1[i], " ", pointers2[i]);
   }
 // it outputs some pairwise identical object descriptors
   /*
   0 1048576 1048576
   1 2097152 2097152
   2 3145728 3145728
   3 4194304 4194304
   4 5242880 5242880
   */
```

Los arrays de estructuras con campos de tipos simples también se copian sin problemas.

```
struct Simple
{
   int x;
};

void OnStart()
{
   ...
   Simple s1[3] = {{123}, {456}, {789}}, s2[];
   PRTS(ArrayCopy(s2, s1)); // 3 / status:0
   ArrayPrint(s2);
   /*
       [x]
   [0] 123
   [1] 456
   [2] 789
   */
   ...
```

Para seguir demostrando cómo trabajar con arrays de diferentes tipos y configuraciones, se definen los siguientes arrays (incluyendo arrays fijos y dinámicos, así como arrays con un número diferente de dimensiones):

```
int dynamic[];
int dynamic2Dx5[][5];
int dynamic2Dx4[][4];
int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};
int insert[] = {10, 11, 12};
double array[1] = {M_PI};
string texts[];
string message[1] = {"ok"};
...
```

Cuando se copia un elemento del array *fixed* desde la posición 1 (número 2), se asigna una fila entera de 4 elementos en el array dinámico receptor *dynamic2Dx4*, y como sólo se copia 1 elemento, los tres restantes contendrán «basura» aleatoria (resaltada en amarillo).

```
PRTS(ArrayCopy(dynamic2Dx4, fixed, 0, 1, 1)); // 1 / status:0
ArrayPrint(dynamic2Dx4);
/*
    [,0][,1][,2][,3]
[0,]   2   1   2   3
*/
...
```

A continuación, copiamos todos los elementos del array *fixed*, empezando por el tercero, en el mismo array *dynamic2Dx4*, pero empezando por la posición 1. Dado que se copian 5 elementos (el número total del array *fixed* es 8 menos la posición inicial 3) y se colocan en el índice 1, en total, 1 + 5 estarán ocupados en el array receptor, para un total de 6 elementos. Y como el array *dynamic2Dx4* tiene 4 elementos en cada fila (en la segunda dimensión), es posible asignarle memoria sólo para el número de elementos que sea múltiplo de 4, es decir, se distribuirán 2 elementos más, en los que quedarán datos aleatorios.

```
PRTS(ArrayCopy(dynamic2Dx4, fixed, 1, 3)); // 5 / status:0
ArrayPrint(dynamic2Dx4);
/*
    [,0][,1][,2][,3]
[0,]   2   4   5   6
[1,]   7   8   3   4
*/
```

Al copiar un array multidimensional en un array unidimensional, los elementos se presentarán de forma «plana».

```
PRTS(ArrayCopy(dynamic, fixed)); // 8 / status:0
ArrayPrint(dynamic);
/*
1 2 3 4 5 6 7 8
*/
```

Cuando se copia un array unidimensional en otro multidimensional, los elementos se «expanden» según las dimensiones del array receptor.

```
PRTS(ArrayCopy(dynamic2Dx5, insert)); // 3 / status:0
ArrayPrint(dynamic2Dx5);
/*
    [,0][,1][,2][,3][,4]
[0,]  10  11  12   4   5
*/
```

En este caso, se han copiado 3 elementos que caben en una fila de 5 elementos (según la configuración del array receptor). La memoria para los dos elementos restantes de la serie se asignó, pero no se llenó (contiene «basura»).

Podemos sobrescribir el array *dynamic2Dx5* desde otra fuente, incluso desde un array multidimensional de una configuración diferente. Dado que se asignaron dos filas de 5 elementos cada una en el array receptor y 2 filas de 4 elementos cada una en el array de origen, han quedado 2 elementos adicionales sin rellenar.

```
PRTS(ArrayCopy(dynamic2Dx5, fixed)); // 8 / status:0
ArrayPrint(dynamic2Dx5);
/*
    [,0][,1][,2][,3][,4]
[0,]   1   2   3   4   5
[1,]   6   7   8   0   0
*/
```

Mediante *ArrayCopy* es posible cambiar los elementos de los arrays receptores fijos.

```
PRTS(ArrayCopy(fixed, insert)); // 3 / status:0
ArrayPrint(fixed);
/*
    [,0][,1][,2][,3]
[0,]  10  11  12   4
[1,]   5   6   7   8
*/
```

Aquí hemos sobrescrito los tres primeros elementos del array *fixed*. Y luego vamos a sobrescribir los tres últimos.

```
PRTS(ArrayCopy(fixed, insert, 5)); // 3 / status:0
ArrayPrint(fixed);
/*
    [,0][,1][,2][,3]
[0,]  10  11  12   4
[1,]   5  10  11  12
*/
```

Copiar a una posición igual a la longitud del array fijo no funcionará (el array dinámico de destino se expandiría en este caso).

```
PRTS(ArrayCopy(fixed, insert, 8)); // 4006, ERR_INVALID_ARRAY
ArrayPrint(fixed); // no changes
```

Los arrays de cadenas combinados con arrays de otros tipos producirán un error:

```
PRTS(ArrayCopy(texts, insert)); // 5050, ERR_INCOMPATIBLE_ARRAYS
ArrayPrint(texts); // empty
```

Pero entre arrays de cadenas, la copia es posible:

```
PRTS(ArrayCopy(texts, message));
ArrayPrint(texts); // "ok"
```

Los arrays de distintos tipos numéricos se copian con la conversión necesaria.

```
PRTS(ArrayCopy(insert, array, 1)); // 1 / status:0
ArrayPrint(insert); // 10  3 12
```

Aquí hemos escrito el número Pi en un array de enteros, y por lo tanto recibimos el valor 3 (sustituyó a 11).

bool ArrayRemove(void &array[], uint start, uint count = WHOLE_ARRAY)

La función elimina el número especificado de elementos del array a partir del índice *start*. Un array puede ser multidimensional y tener cualquier tipo de estructura o integrado con campos de tipos integrados, a excepción de las cadenas.

El índice *start* y la cantidad *count* se refieren a la primera dimensión de los arrays. En otras palabras: en el caso de los arrays multidimensionales, el borrado no se realiza por elementos individuales, sino por toda la configuración descrita por dimensiones «superiores». Por ejemplo, para un array bidimensional, el valor 1 en el parámetro *count* significa borrar toda una serie de longitud igual a la segunda dimensión (véase el ejemplo).

El valor *start* debe estar comprendido entre 0 y el tamaño de la primera dimensión menos 1.

La función no puede aplicarse a arrays con series temporales (series temporales o búferes indicadores integrados).

Para probar la función, preparamos el script *ArrayRemove.mq5*. En concreto, define dos estructuras:

```
struct Simple
{
   int x;
};

struct NotSoSimple
{
   int x;
   string s; // a field of type string causes the compiler to make an implicit destru
};
```

Los arrays con una estructura simple pueden ser procesados con éxito por la función *ArrayRemove*, mientras que los arrays de objetos con destructores (incluso con destructores implícitos, como en *NotSoSimple*) provocan un error:

```
void OnStart()
{
   Simple structs1[10];
   PRTS(ArrayRemove(structs1, 0, 5)); // true / status:0

   NotSoSimple structs2[10];
   PRTS(ArrayRemove(structs2, 0, 5)); // false / status:4005,
                                      // ERR_STRUCT_WITHOBJECTS_ORCLASS
   ...
```

A continuación, se definen e inicializan arrays de distintas configuraciones.

```
int dynamic[];
int dynamic2Dx4[][4];
int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};

// make 2 copies
ArrayCopy(dynamic, fixed);
ArrayCopy(dynamic2Dx4, fixed);

// show initial data
ArrayPrint(dynamic);
/*
1 2 3 4 5 6 7 8
*/
ArrayPrint(dynamic2Dx4);
/*
    [,0][,1][,2][,3]
[0,]   1   2   3   4
[1,]   5   6   7   8
*/
```

Al borrar de un array fijo, todos los elementos posteriores al fragmento que se elimina se desplazan hacia la izquierda. Es importante que el tamaño del array no cambie, por lo que las copias de los elementos desplazados aparecen por duplicado.

```
PRTS(ArrayRemove(fixed, 0, 1));
ArrayPrint(fixed);
/*
ArrayRemove(fixed,0,1)=true / status:0
    [,0][,1][,2][,3]
[0,]   5   6   7   8
[1,]   5   6   7   8
*/
```

Aquí eliminamos un elemento de la primera dimensión de un array bidimensional *fixed* por el offset 0, es decir, la fila inicial. Los elementos de la fila siguiente se desplazan hacia arriba y permanecen en la misma fila.

Si realizamos la misma operación con un array dinámico (idéntico en contenido al array *fixed*), su tamaño se reducirá automáticamente en el número de elementos eliminados.

```
PRTS(ArrayRemove(dynamic2Dx4, 0, 1));
ArrayPrint(dynamic2Dx4);
/*
ArrayRemove(dynamic2Dx4,0,1)=true / status:0
    [,0][,1][,2][,3]
[0,]   5   6   7   8
*/
```

En un array unidimensional, cada elemento eliminado corresponde a un único valor. Por ejemplo, en el array *dynamic*, al eliminar tres elementos a partir del índice 2, obtenemos el siguiente resultado:

```
PRTS(ArrayRemove(dynamic, 2, 3));
ArrayPrint(dynamic);
/*
ArrayRemove(dynamic,2,3)=true / status:0
1 2 6 7 8
*/
```

Se han eliminado los valores 3, 4 y 5, el tamaño del array se ha reducido en 3.

bool ArrayReverse(void &array[], uint start = 0, uint count = WHOLE_ARRAY)

La función invierte el orden de los elementos especificados en el array. Los elementos que deben invertirse vienen determinados por una posición inicial *start* y una cantidad *count*. Si *start = 0* y *count = WHOLE_ARRAY*, se accede a todo el array.

Se admiten arrays de dimensiones y tipos arbitrarios, tanto fijos como dinámicos (incluidas series temporales en búferes indicadores). Un array puede contener objetos, punteros o estructuras. En el caso de los arrays multidimensionales, sólo se invierte la primera dimensión.

El valor de *count* debe estar comprendido entre 0 y el número de elementos de la primera dimensión. Tenga en cuenta que *count* menos de 2 no producirá un efecto perceptible, pero puede utilizarse para unificar bucles en algoritmos.

La función devuelve *true* si tiene éxito o *false* en caso de error.

El script *ArrayReverse.mq5* puede utilizarse para probar la función. Al principio, se define una clase para generar objetos almacenados en un array.. La presencia de cadenas y otros campos «complejos» no supone ningún problema.

```
class Dummy
{
   static int counter;
   int x;
   string s; // a field of type string causes the compiler to create an implicit dest
public:
   Dummy() { x = counter++; }
};

static int Dummy::counter;
```

Los objetos se identifican mediante un número de serie (asignado en el momento de su creación).

```
void OnStart()
{
   Dummy objects[5];
   Print("Objects before reverse");
   ArrayPrint(objects);
   /*
       [x]  [s]
   [0]   0 null
   [1]   1 null
   [2]   2 null
   [3]   3 null
   [4]   4 null
   */
```

Tras aplicar *ArrayReverse* obtenemos el orden inverso esperado de los objetos.

```
   PRTS(ArrayReverse(objects)); // true / status:0
   Print("Objects after reverse");
   ArrayPrint(objects);
   /*
       [x]  [s]
   [0]   4 null
   [1]   3 null
   [2]   2 null
   [3]   1 null
   [4]   0 null
   */
```

A continuación, se preparan arrays numéricos de distintas configuraciones y se despliegan con diferentes parámetros.

```
   int dynamic[];
   int dynamic2Dx4[][4];
   int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};

   ArrayCopy(dynamic, fixed);
   ArrayCopy(dynamic2Dx4, fixed);

   PRTS(ArrayReverse(fixed)); // true / status:0
   ArrayPrint(fixed);
   /*
       [,0][,1][,2][,3]
   [0,]   5   6   7   8
   [1,]   1   2   3   4
   */

   PRTS(ArrayReverse(dynamic, 4, 3)); // true / status:0
   ArrayPrint(dynamic);
   /*
   1 2 3 4 7 6 5 8
   */

   PRTS(ArrayReverse(dynamic, 0, 1)); // does nothing (count = 1)
   PRTS(ArrayReverse(dynamic2Dx4, 2, 1)); // false / status:5052, ERR_SMALL_ARRAY
}
```

En este último caso, el valor de *start* (2) supera el tamaño en la primera dimensión, por lo que se produce un error.

4.3.6 Mover (intercambiar) arrays

MQL5 ofrece la posibilidad de intercambiar el contenido de dos arrays de forma eficiente (sin asignación física de memoria ni copia de datos). En algunos otros lenguajes de programación, se admite una operación similar no sólo para los arrays, sino también para otras variables, y se denomina desplazamiento.

bool ArraySwap(void &array1[], void &array2[])

La función intercambia el contenido de dos arrays dinámicos del mismo tipo. Se admiten arrays de cualquier tipo. Sin embargo, la función no es aplicable a los arrays de series temporales ni a los búferes de indicadores, así como tampoco a ningún array con el modificador *const*.

Para arrays multidimensionales, el número de elementos de todas las dimensiones excepto la primera debe coincidir.

La función devuelve *true* si tiene éxito o *false* en caso de error.

El principal uso de la función es acelerar el programa eliminando la copia física del array cuando éste se pasa a la función o se devuelve de ella, y se sabe que el array de origen ya no es necesario. El hecho es que el intercambio se produce casi instantáneamente, ya que los datos de la aplicación no se mueven de ninguna manera. En lugar de ello hay un intercambio de metadatos sobre arrays almacenados en estructuras de servicio que describen arrays dinámicos (y esto sólo ocupa 52 bytes).

Supongamos que existe una clase destinada a procesar un array mediante determinados algoritmos. El mismo array puede ser sometido a diferentes operaciones y, por lo tanto, tiene sentido mantenerlo como un miembro de la clase. Pero entonces surge la pregunta: ¿cómo transferirlo a un objeto? En MQL5, los métodos (así como las funciones en general) permiten pasar arrays sólo por referencia. Dejando a un lado todo tipo de clases de envoltorios que contengan un array y se pasen por puntero, la única solución sencilla parece ser la siguiente: describir, por ejemplo, un parámetro de array en el constructor de la clase y copiarlo al array interno utilizando *ArrayCopy*. Pero es más eficaz utilizar *ArraySwap*.

```
template<typename T>
class Worker
{
   T array[];

public:
   Worker(T &source[])
   {
      // ArrayCopy(array, source); // memory and time consuming
      ArraySwap(source, array);
   }
   ...
};
```

Dado que el array *array* estaba vacío antes del intercambio, después de la operación el array utilizado como argumento *source* quedará vacío, mientras que *array* se llenará con datos de entrada con poca o ninguna sobrecarga.

Una vez que el objeto de la clase se convierte en el «propietario» del array, podemos modificarlo con los algoritmos requeridos; por ejemplo, mediante un método especial *process*, que toma como parámetro el código del algoritmo solicitado. Puede ordenar, suavizar, mezclar, añadir ruido y mucho más. Pero antes, probemos la idea en una operación sencilla de inversión de arrays mediante la función *ArrayReverse* (véase el archivo *ArraySwapSimple.mq5*).

```
bool process(const int mode)
{
   if(ArraySize(array) == 0) return false;
   switch(mode)
   {
   case -4:
      // example: shuffling
      break;
   case -3:
      // example: logarithm
      break;
   case -2:
      // example: adding noise
      break;
   case -1:
      ArrayReverse(array);
      break;
   ...
   }
   return true;
}
```

Puede proporcionar acceso a los resultados del trabajo utilizando dos métodos: elemento por elemento (sobrecargando el operador '[]') o por un array completo (de nuevo utilizamos *ArraySwap* en el método correspondiente *get*, pero también puede proporcionar un método para copiar a través de *ArrayCopy*).

```
T operator[](int i)
{
   return array[i];
}

void get(T &result[])
{
   ArraySwap(array, result);
}
```

A efectos de universalidad, la clase se hace plantilla. Esto permitirá adaptarla en el futuro para arrays de estructuras arbitrarias, pero por ahora se puede comprobar la inversión de un array simple del tipo *double*:

```
void OnStart()
{
   double data[];
   ArrayResize(data, 3);
   data[0] = 1;
   data[1] = 2;
   data[2] = 3;

   PRT(ArraySize(data));        // 3
   Worker<double> simple(data);
   PRT(ArraySize(data));        // 0
   simple.process(-1);  // reversing array

   double res[];
   simple.get(res);
   ArrayPrint(res); // 3.00000 2.00000 1.00000
}
```

La tarea de ordenar es más realista, y para un array de estructuras, puede ser necesario ordenar por cualquier campo. En la sección siguiente estudiaremos en detalle la función *ArraySort*, que permite ordenar en orden ascendente un array de cualquier tipo integrado, pero no estructuras. Ahí intentaremos eliminar esta «laguna», dejando *ArraySwap* en acción.

4.3.7 Comparar, ordenar y buscar en arrays

La API de MQL5 contiene varias funciones que permiten comparar y ordenar arrays, así como buscar en ellos el máximo, el mínimo o cualquier valor concreto.

int ArrayCompare(const void &array1[], const void &array2[], int start1 = 0, int start2 = 0, int count = WHOLE_ARRAY)

La función devuelve el resultado de comparar dos arrays de tipos integrados o estructuras con campos de tipos integrados, excluidas las cadenas. No se admiten arrays de objetos de clase. Tampoco se pueden comparar arrays de estructuras que contengan arrays dinámicos, objetos de clase o punteros.

Por defecto, la comparación se realiza para arrays completos pero, si es necesario, puede especificar partes de arrays, para lo cual existen los parámetros *start1* (posición inicial en el primer array), *start2* (posición inicial en el segundo array) y *count*.

Los arrays pueden ser fijos o dinámicos, así como multidimensionales. Durante la comparación, los arrays multidimensionales se representan como arrays unidimensionales equivalentes (por ejemplo, para arrays bidimensionales, los elementos de la segunda fila siguen a los elementos de la primera, los elementos de la tercera fila siguen a los de la segunda, etc.). Por esta razón, los parámetros *start1*, *start2* y *count* para arrays multidimensionales se especifican mediante la numeración de elementos, y no con un índice a lo largo de la primera dimensión.

Utilizando varios desplazamientos *start1* y *start2* puede comparar diferentes partes del mismo array.

Los arrays se comparan elemento por elemento hasta que se encuentra la primera discrepancia o se llega al final de alguno de los arrays La relación entre dos elementos (que se encuentran en las mismas posiciones en ambos arrays) depende del tipo: para los números, se utilizan los operadores '>', '<', '==', y para las cadenas, la función *StringCompare*. Las estructuras se comparan byte a byte, lo que equivale a ejecutar el siguiente código para cada par de elementos:

```
uchar bytes1[], bytes2[];
StructToCharArray(array1[i], bytes1);
StructToCharArray(array2[i], bytes2);
int cmp = ArrayCompare(bytes1, bytes2);
```

Basándose en la relación de los primeros elementos que difieren, se obtiene el resultado de la comparación a granel de los arrays *array1* y *array2*. Si no se encuentran diferencias y la longitud de los arrays es igual, se considera que los arrays son iguales. Si la longitud es diferente, se considera mayor el array más largo.

La función devuelve -1 si *array1* es «menor que» *array2*, +1 si *array1* es «mayor que» *array2*, y 0 si son «iguales».

En caso de error, el resultado es -2.

Veamos algunos ejemplos en el script *ArrayCompare.mq5*.

Vamos a crear una estructura sencilla para rellenar los arrays que se van a comparar.

```
struct Dummy
{
   int x;
   int y;

   Dummy()
   {
      x = rand() / 10000;
      y = rand() / 5000;
   }
};
```

Los campos de clase se rellenan con números aleatorios (cada vez que se ejecute el script, recibiremos nuevos valores).

En la función *OnStart*, describimos un pequeño array de estructuras y comparamos los elementos sucesivos entre sí (como si desplazáramos fragmentos vecinos de un array con la longitud de 1 elemento).

```
#define LIMIT 10

void OnStart()
{
   Dummy a1[LIMIT];
   ArrayPrint(a1);

   // pairwise comparison of neighboring elements
   // -1: [i] < [i + 1]
   // +1: [i] > [i + 1]
   for(int i = 0; i < LIMIT - 1; ++i)
   {
      PRT(ArrayCompare(a1, a1, i, i + 1, 1));
   }
   ...
```

A continuación se muestran los resultados de una de las opciones del array (para facilitar el análisis, la columna con los signos «mayor que» (+1) / «menor que» (-1) se añade directamente a la derecha del contenido del array):

```
        [x] [y]   // result
[0]   0   3   // -1
[1]   2   4   // +1
[2]   2   3   // +1
[3]   1   6   // +1
[4]   0   6   // -1
[5]   2   0   // +1
[6]   0   4   // -1
[7]   2   5   // +1
[8]   0   5   // -1
[9]   3   6
```

Al comparar entre sí las dos mitades del array se obtiene -1:

```
// compare first and second half
  PRT(ArrayCompare(a1, a1, 0, LIMIT / 2, LIMIT / 2)); // -1
```

A continuación, compararemos arrays de cadenas con datos predefinidos.

```
   string s[] = {"abc", "456", "$"};
   string s0[][3] = {{"abc", "456", "$"}};
   string s1[][3] = {{"abc", "456", ""}};
   string s2[][3] = {{"abc", "456"}}; // last element omitted: it is null
   string s3[][2] = {{"abc", "456"}};
   string s4[][2] = {{"aBc", "456"}};

   PRT(ArrayCompare(s0, s));  // s0 == s, 1D and 2D arrays contain the same data
   PRT(ArrayCompare(s0, s1)); // s0 > s1 since "$" > ""
   PRT(ArrayCompare(s1, s2)); // s1 > s2 since "" > null
   PRT(ArrayCompare(s2, s3)); // s2 > s3 due to different lengths: [3] > [2]
   PRT(ArrayCompare(s3, s4)); // s3 < s4 since "abc" < "aBc"
```

Por último, vamos a comprobar la proporción de fragmentos del array:

```
   PRT(ArrayCompare(s0, s1, 1, 1, 1)); // second elements (with index 1) are equal
   PRT(ArrayCompare(s1, s2, 0, 0, 2)); // the first two elements are equal
```

bool ArraySort(void &array[])

La función ordena un array numérico (incluyendo posiblemente un array multidimensional) por la primera dimensión. El orden de clasificación es ascendente. Para ordenar un array en orden descendente, aplique la función *ArrayReverse* al array resultante o procéselo en orden inverso.

La función no admite arrays de cadenas, estructuras o clases.

La función devuelve *true* si tiene éxito o *false* en caso de error.

Si se establece la propiedad «timeseries» (series temporales) para un array, los elementos que lo componen se indexan en orden inverso (véanse los detalles en la sección Dirección de indexación de arrays como en las series temporales), y esto tiene un efecto inverso «externo» en el orden de clasificación: cuando procese directamente un array de este tipo, obtendrá valores descendentes. A nivel físico, el array siempre se ordena de forma ascendente, y así es como se almacena.

En el script *ArraySort.mq5* se genera un array bidimensional de 10 por 3 y se ordena utilizando *ArraySort*:

```
#define LIMIT 10
#define SUBLIMIT 3

void OnStart()
{
   // generating random data
   int array[][SUBLIMIT];
   ArrayResize(array, LIMIT);
   for(int i = 0; i < LIMIT; ++i)
   {
      for(int j = 0; j < SUBLIMIT; ++j)
      {
         array[i][j] = rand();
      }
   }

   Print("Before sort");
   ArrayPrint(array);    // source array

   PRTS(ArraySort(array));

   Print("After sort");
   ArrayPrint(array);    // ordered array
   ...
}
```

Según el registro, la primera columna está ordenada de forma ascendente (los números concretos variarán debido a la generación aleatoria):

```
Before sort
      [,0]  [,1]  [,2]
[0,]  8955  2836 20011
[1,]  2860  6153 25032
[2,] 16314  4036 20406
[3,] 30366 10462 19364
[4,] 27506  5527 21671
[5,]  4207  7649 28701
[6,]  4838   638 32392
[7,] 29158 18824 13536
[8,] 17869 23835 12323
[9,] 18079  1310 29114
ArraySort(array)=true / status:0
After sort
      [,0]  [,1]  [,2]
[0,]  2860  6153 25032
[1,]  4207  7649 28701
[2,]  4838   638 32392
[3,]  8955  2836 20011
[4,] 16314  4036 20406
[5,] 17869 23835 12323
[6,] 18079  1310 29114
[7,] 27506  5527 21671
[8,] 29158 18824 13536
[9,] 30366 10462 19364
```

Los valores de las columnas siguientes se han movido de forma sincronizada con los valores «principales» de la primera columna. En otras palabras: se permutan todas las filas, a pesar de que sólo la primera columna es el criterio de ordenación.

Pero, ¿qué ocurre si se desea ordenar un array bidimensional por una columna distinta de la primera? Puede escribir un algoritmo especial para ello. Una de las opciones se incluye en el archivo *ArraySort.mq5* como función de plantilla:

```
template<typename T>
bool ArraySort(T &array[][], const int column)
{
   if(!ArrayIsDynamic(array)) return false;

   if(column == 0)
   {
      return ArraySort(array); // standard function
   }

   const int n = ArrayRange(array, 0);
   const int m = ArrayRange(array, 1);

   T temp[][2];

   ArrayResize(temp, n);
   for(int i = 0; i < n; ++i)
   {
      temp[i][0] = array[i][column];
      temp[i][1] = i;
   }

   if(!ArraySort(temp)) return false;

   ArrayResize(array, n * 2);
   for(int i = n; i < n * 2; ++i)
   {
      ArrayCopy(array, array, i * m, (int)(temp[i - n][1] + 0.1) * m, m);
      /* equivalent
      for(int j = 0; j < m; ++j)
      {
         array[i][j] = array[(int)(temp[i - n][1] + 0.1)][j];
      }
      */
   }

   return ArrayRemove(array, 0, n);
}
```

La función dada sólo funciona con arrays dinámicos porque el tamaño de *array* se duplica para reunir los resultados intermedios en la segunda mitad del array y, por último, la primera mitad (original) se elimina con *ArrayRemove*. Por eso, el array de prueba original de la función *OnStart* se distribuyó a través de *ArrayResize*.

Le animamos a que estudie el principio de clasificación por su cuenta (o a que pase un par de páginas).

Algo similar debería implementarse para arrays con un gran número de dimensiones (por ejemplo, *array[][][]*).

Recordemos que en la sección anterior planteamos la cuestión de ordenar un array de estructuras por un campo arbitrario. Como sabemos, la función estándar *ArraySort* no es capaz de hacerlo. Intentemos

idear un «rodeo». Tomemos como base la clase del archivo *ArraySwapSimple.mq5* de la sección anterior. Vamos a copiarla en *ArrayWorker.mq5* y a añadir el código necesario.

En el método *Worker::process*, proporcionaremos una llamada al método auxiliar de ordenación *arrayStructSort*, y el campo que se desea ordenar se especificará por número (más adelante describiremos cómo hacerlo):

```
   ...
   bool process(const int mode)
   {
      ...
      switch(mode)
      {
      ...
      case -1:
         ArrayReverse(array);
         break;
      default: // sorting by field number 'mode'
         arrayStructSort(mode);
         break;
      }
      return true;
   }

private:
   bool arrayStructSort(const int field)
   {
      ...
   }
```

Ahora queda claro por qué todos los modos anteriores (valores del parámetro *mode*) en el método *process* eran negativos: los valores cero y positivo están reservados para la clasificación y corresponden al número de «columna».

La idea de ordenar un array de estructuras está tomada de ordenar un array bidimensional. Sólo necesitamos mapear de algún modo una única estructura a un array unidimensional (que representa una fila de un array bidimensional). Para ello, en primer lugar hay que decidir de qué tipo debe ser el array.

Dado que la clase *worker* ya es una plantilla, añadiremos un parámetro más a la plantilla para que el tipo de array se pueda establecer de forma flexible.

```
template<typename T, typename R>
class Worker
{
   T array[];
   ...
```

Ahora, volvamos a las asociaciones, que le permiten superponer variables de distintos tipos. Así, obtenemos la siguiente construcción complicada:

```
union Overlay
{
   T r;
   R d[sizeof(T) / sizeof(R)];
};
```

En esta unión, el tipo de la estructura se combina con un array de tipo R, y su tamaño es calculado automáticamente por el compilador basándose en la relación de los tamaños de los dos tipos, T y R.

Ahora, dentro del método *arrayStructSort*, podemos duplicar parcialmente el código de ordenación de arrays bidimensionales.

```
bool arrayStructSort(const int field)
{
   const int n = ArraySize(array);

   R temp[][2];
   Overlay overlay;

   ArrayResize(temp, n);
   for(int i = 0; i < n; ++i)
   {
      overlay.r = array[i];
      temp[i][0] = overlay.d[field];
      temp[i][1] = i;
   }
   ...
```

En lugar de un array con las estructuras originales, preparamos el array *temp[][2]* de tipo R, lo ampliamos al número de registros en *array*, y escribimos lo siguiente en el bucle: la «visualización» del campo *field* requerido de la estructura en el índice 0 de cada fila, y el índice original de este elemento en el índice 1.

La «visualización» se basa en el hecho de que los campos de las estructuras suelen estar alineados de alguna manera, ya que utilizan tipos estándar. Por lo tanto, con un tipo R adecuadamente elegido, es posible proporcionar una coincidencia total o parcial de los campos en los elementos del array en la «superposición».

Por ejemplo, en la estructura estándar *MqlRates*, los 6 primeros campos tienen un tamaño de 8 bytes y, por lo tanto, se asignan correctamente al array *double* o *long* (son candidatos de tipo de plantilla R).

```
struct MqlRates
{
   datetime time;
   double   open;
   double   high;
   double   low;
   double   close;
   long     tick_volume;
   int      spread;
   long     real_volume;
};
```

Con los dos últimos campos, la situación es más complicada. Si el campo *spread* todavía se puede alcanzar utilizando el tipo *int* como R, entonces el campo *real_volume* resulta estar en un desplazamiento que no es múltiplo de su propio tamaño (debido al tipo de campo *int*, es decir, 4 bytes, antes de él). Estos son problemas de un método concreto. Se puede mejorar, o inventar otro método.

Pero volvamos al algoritmo de clasificación. Una vez rellenado el array *temp*, se puede ordenar con la función habitual *ArraySort*, y luego se pueden utilizar los índices originales para formar un nuevo array con el orden de estructura correcto.

```
      ...
      if(!ArraySort(temp)) return false;
      T result[];

      ArrayResize(result, n);
      for(int i = 0; i < n; ++i)
      {
         result[i] = array[(int)(temp[i][1] + 0.1)];
      }

      return ArraySwap(result, array);
   }
```

Antes de salir de la función, volvemos a utilizar *ArraySwap* para sustituir el contenido de un array intra-objeto *array* de forma eficiente en recursos por algo nuevo y ordenado, que se recibe en el array local *result*.

Comprobemos la clase *worker* en acción: en la función *OnStart* vamos a definir un array de estructuras *MqlRates* y a pedir al terminal varios miles de registros.

```
#define LIMIT 5000

void OnStart()
{
   MqlRates rates[];
   int n = CopyRates(_Symbol, _Period, 0, LIMIT, rates);
   ...
```

La función *CopyRates* se describirá en una sección aparte. Por ahora, nos basta con saber que rellena el array pasado *rates* con cotizaciones del símbolo y el marco temporal del gráfico actual en el que se está ejecutando el script. La macro LÍMITE especifica el número de barras solicitadas: debe asegurarse de que este valor no sea superior al configurado en su terminal para el número de barras de cada ventana.

Para procesar los datos recibidos, crearemos un objeto *worker* con los tipos *T=MqlRates* y *R=double*:

```
Worker<MqlRates, double> worker(rates);
```

La clasificación puede iniciarse con una instrucción de la siguiente forma:

```
worker.process(offsetof(MqlRates, open) / sizeof(double));
```

Aquí utilizamos el operador *offsetof* para obtener el desplazamiento de byte del campo *open* dentro de la estructura. Se divide a su vez por el tamaño *double* y da el número de «columna» correcto para ordenar por el precio *open* . Puede leer el resultado de la ordenación elemento por elemento u obtener el array completo:

```
Print(worker[i].open);
...
worker.get(rates);
ArrayPrint(rates);
```

Tenga en cuenta que obtener un array mediante el método *get* la desplaza del array interno *array* al externo (pasado como argumento) con *ArraySwap*. Por lo tanto, después de eso, las llamadas *worker.process()* no tienen sentido: no hay más datos en el objeto *worker*.

Para simplificar el inicio de la ordenación por diferentes campos se ha implementado una función auxiliar *sort*:

```
void sort(Worker<MqlRates, double> &worker, const int offset, const string title)
{
   Print(title);
   worker.process(offset);
   Print("First struct");
   StructPrint(worker[0]);
   Print("Last struct");
   StructPrint(worker[worker.size() - 1]);
}
```

Envía un encabezado y los elementos primero y último del array ordenado al registro. Con su ayuda, las pruebas en *OnStart* para tres campos tienen este aspecto:

```
voidOnStart()
{
...
Worker<MqlRates,double>worker(rates);
sort(worker,offsetof(MqlRates,open) /sizeof(double),"Sorting by open price...");
sort(worker,offsetof(MqlRates,tick_volume) /sizeof(double),"Sorting by tick volume...
sort(worker,offsetof(MqlRates,time) /sizeof(double),"Sorting by time...");
}
```

Por desgracia, la función estándar *print* no admite la impresión de estructuras individuales, y no hay ninguna función integrada *StructPrint* en MQL5. Por lo tanto, tuvimos que escribirla nosotros mismos, basándonos en *ArrayPrint*: de hecho, basta con poner la estructura en un array de tamaño 1.

```
template<typename S>
void StructPrint(const S &s)
{
   S temp[1];
   temp[0] = s;
   ArrayPrint(temp);
}
```

Como resultado de la ejecución del script, podemos obtener algo como lo siguiente (dependiendo de la configuración del terminal, es decir, en qué símbolo/marco temporal se ejecuta):

```
Sorting by open price...
First struct
                [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.07.21 10:30:00 1.17557 1.17584 1.17519 1.17561          1073        0             0
Last struct
                [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.05.25 15:15:00 1.22641 1.22664 1.22592 1.22618           852        0             0
Sorting by tick volume...
First struct
                [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.05.24 00:00:00 1.21776 1.21811 1.21764 1.21794            52       20             0
Last struct
                [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.06.16 21:30:00 1.20436 1.20489 1.20149 1.20154          4817        0             0
Sorting by time...
First struct
                [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.05.14 16:15:00 1.21305 1.21411 1.21289 1.21333           888        0             0
Last struct
                [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_volume]
[0] 2021.07.27 22:45:00 1.18197 1.18227 1.18191 1.18225           382        0             0
```

Aquí están los datos para EURUSD,M15.

La implementación anterior de la ordenación es potencialmente una de las más rápidas porque utiliza la función *ArraySort*.

Sin embargo, si las dificultades para alinear los campos de la estructura o el escepticismo ante el propio planteamiento de «mapear» la estructura en un array nos obligan a abandonar este método (y, por tanto, la función *ArraySort*), sigue a nuestra disposición el acreditado método del «hágalo usted mismo».

Existe un gran número de algoritmos de ordenación que son fáciles de adaptar a MQL5. Una de las opciones de clasificación rápida se presenta en el archivo *QuickSortStructT.mqh* adjunto al libro. Se trata de una versión *QuickSortT.mqh* mejorada, que utilizamos en la sección Comparación de cadenas. Tiene el método *Compare* de la clase de plantilla *QuickSortStructT* que se hace puramente virtual y debe redefinirse en la clase descendiente para devolver un análogo del operador de comparación '>' para el tipo requerido y sus campos. Para comodidad del usuario, se ha creado una macro en el archivo de encabezado:

```
#define SORT_STRUCT(T, A, F)                                                  \
{                                                                             \
   class InternalSort : public QuickSortStructT<T>                            \
   {                                                                          \
      virtual bool Compare(const T &a, const T &b) override                   \
      {                                                                       \
         return a.##F > b.##F;                                                \
      }                                                                       \
   } sort;                                                                    \
   sort.QuickSort(A);                                                         \
}
```

Utilizándolo, para ordenar un array de estructuras por un campo determinado basta con escribir una instrucción. Por ejemplo:

```
MqlRates rates[];
CopyRates(_Symbol, _Period, 0, 10000, rates);
SORT_STRUCT(MqlRates, rates, high);
```

Aquí el array *rates* de tipo *MqlRates* está ordenado por el precio *high*.

int ArrayBsearch(const type &array[], type value)

La función busca un valor dado en un array numérico. Se admiten arrays de todos los tipos numéricos integrados. El array debe estar ordenado de forma ascendente por la primera dimensión, de lo contrario el resultado será incorrecto.

La función devuelve el índice del elemento coincidente (si hay varios, entonces el índice del primero de ellos) o el índice del elemento más cercano en valor (si no hay coincidencia exacta), es decir, puede ser un elemento con un valor mayor o menor que el buscado. Si el valor deseado es menor que el primero (mínimo), se devuelve 0. Si el valor buscado es mayor que el último (máximo), se devuelve su índice.

El índice depende del sentido de la numeración de los elementos del array: directo (del principio al final) o inverso (del final al principio). Este puede identificarse y modificarse mediante las funciones descritas en la sección Dirección de indexación de arrays como en las series temporales.

Si se produce un error, se devuelve -1.

Para arrays multidimensionales, la búsqueda se limita a la primera dimensión.

En el script *ArraySearch.mq5* se pueden encontrar ejemplos de uso de la función *ArrayBsearch*.

```
void OnStart()
{
   int array[] = {1, 5, 11, 17, 23, 23, 37};
     // indexes  0  1   2   3   4   5   6
   int data[][2] = {{1, 3}, {3, 2}, {5, 10}, {14, 10}, {21, 8}};
     // indexes     0       1       2         3         4
   int empty[];
   ...
```

Para tres arrays predefinidas (una de ellas vacía), se ejecutan las siguientes sentencias:

```
PRTS(ArrayBsearch(array, -1)); // 0
PRTS(ArrayBsearch(array, 11)); // 2
PRTS(ArrayBsearch(array, 12)); // 2
PRTS(ArrayBsearch(array, 15)); // 3
PRTS(ArrayBsearch(array, 23)); // 4
PRTS(ArrayBsearch(array, 50)); // 6

PRTS(ArrayBsearch(data, 7));   // 2
PRTS(ArrayBsearch(data, 9));   // 2
PRTS(ArrayBsearch(data, 10));  // 3
PRTS(ArrayBsearch(data, 11));  // 3
PRTS(ArrayBsearch(data, 14));  // 3

PRTS(ArrayBsearch(empty, 0));  // -1, 5053, ERR_ZEROSIZE_ARRAY
...
```

Además, en la función de ayuda *populateSortedArray*, el array *numbers* se rellena con valores aleatorios, y el array se mantiene constantemente en un estado ordenado utilizando *ArrayBsearch*.

```
void populateSortedArray(const int limit)
{
   double numbers[];  // array to fill
 doubleelement[1];// new value to insert

   ArrayResize(numbers, 0, limit); // allocate memory beforehand

   for(int i = 0; i < limit; ++i)
   {
      // generate a random number
      element[0] = NormalizeDouble(rand() * 1.0 / 32767, 3);
      // find where its place in the array
      int cursor = ArrayBsearch(numbers, element[0]);
      if(cursor == -1)
      {
         if(_LastError == 5053) // empty array
         {
            ArrayInsert(numbers, element, 0);
         }
         else break; // error
      }
      else
      if(numbers[cursor] > element[0]) // insert at 'cursor' position
      {
         ArrayInsert(numbers, element, cursor);
      }
      else // (numbers[cursor] <= value) // insert after 'cursor'
      {
         ArrayInsert(numbers, element, cursor + 1);
      }
   }
   ArrayPrint(numbers, 3);
}
```

Cada nuevo valor va primero en un array de un elemento *element*, porque así es más fácil insertarlo en el array resultante *numbers* utilizando la función *ArrayInsert*.

ArrayBsearch le permite determinar dónde debe insertarse el nuevo valor.

El resultado de la función se muestra en el registro:

```
void OnStart()
{
   ...
   populateSortedArray(80);
   /*
    example (will be different on each run due to randomization)
   [ 0] 0.050 0.065 0.071 0.106 0.119 0.131 0.145 0.148 0.154 0.159
        0.184 0.185 0.200 0.204 0.213 0.216 0.220 0.224 0.236 0.238
   [20] 0.244 0.259 0.267 0.274 0.282 0.293 0.313 0.334 0.346 0.366
        0.386 0.431 0.449 0.461 0.465 0.468 0.520 0.533 0.536 0.541
   [40] 0.597 0.600 0.607 0.612 0.613 0.617 0.621 0.623 0.631 0.634
        0.646 0.658 0.662 0.664 0.670 0.670 0.675 0.686 0.693 0.694
   [60] 0.725 0.739 0.759 0.762 0.768 0.783 0.791 0.791 0.791 0.799
        0.838 0.850 0.854 0.874 0.897 0.912 0.920 0.934 0.944 0.992
   */
```

int ArrayMaximum(const type &array[], int start = 0, int count = WHOLE_ARRAY)
int ArrayMinimum(const type &array[], int start = 0, int count = WHOLE_ARRAY)

Las funciones *ArrayMaximum* y *ArrayMinimum* buscan en un array numérico los elementos con los valores máximo y mínimo, respectivamente. El rango de índices para la búsqueda se establece mediante los parámetros *start* y *count*: con los valores predeterminados, se busca en todo el array.

La función devuelve la posición del elemento encontrado.

Si se establece la propiedad «serial» («timeseries») para un array, la indexación de los elementos en ella se realiza en orden inverso, y esto afecta al resultado de esta función (véase el ejemplo). Las funciones integradas para trabajar con la propiedad «serial» se abordan en la sección siguiente. Para más información sobre los arrays «en serie», consulte los capítulos sobre series temporales como indicadores.

En los arrays multidimensionales, la búsqueda se realiza en la primera dimensión.

Si hay varios elementos idénticos en el array con un valor máximo o mínimo, la función devolverá el índice del primero de ellos.

En el archivo *ArrayMaxMin.mq5* se ofrece un ejemplo de utilización de funciones.

```
#define LIMIT 10

void OnStart()
{
   // generating random data
   int array[];
   ArrayResize(array, LIMIT);
   for(int i = 0; i < LIMIT; ++i)
   {
      array[i] = rand();
   }

   ArrayPrint(array);
   // by default, the new array is not a timeseries
   PRTS(ArrayMaximum(array));
   PRTS(ArrayMinimum(array));
   // turn on the "serial" property
   PRTS(ArraySetAsSeries(array, true));
   PRTS(ArrayMaximum(array));
   PRTS(ArrayMinimum(array));
}
```

El script registrará algo parecido al siguiente conjunto de cadenas (debido a la generación aleatoria de datos, cada ejecución será diferente):

```
22242 5909 21570 5850 18026 24740 10852 2631 24549 14635
ArrayMaximum(array)=5 / status:0
ArrayMinimum(array)=7 / status:0
ArraySetAsSeries(array,true)=true / status:0
ArrayMaximum(array)=4 / status:0
ArrayMinimum(array)=2 / status:0
```

4.3.8 Dirección de indexación de series temporales en arrays

Debido a las especificidades de trading aplicadas, MQL5 aporta características adicionales al trabajo con arrays. Una de ellas es que los elementos del array pueden contener datos correspondientes a puntos temporales. Por ejemplo, arrays con cotizaciones de instrumentos financieros, ticks de precios y lecturas de indicadores técnicos. El orden cronológico de los datos significa que constantemente se añaden nuevos elementos al final del array y sus índices aumentan.

No obstante, desde el punto de vista del trading, es más conveniente contar desde el presente hacia el pasado. Así, el elemento 0 siempre contiene el valor más reciente y actualizado, el elemento 1 siempre contiene el valor anterior, y así sucesivamente.

MQL5 le permite seleccionar y cambiar la dirección de indexación del array z sobre la marcha. Un array numerado del presente al pasado se denomina serie temporal. Si el aumento de la indexación se produce del pasado al presente, se trata de un array regular. En las series temporales, el tiempo disminuye con el crecimiento de los índices. En los arrays ordinarios, el tiempo aumenta, como en la vida real.

Es importante señalar que no es necesario que un array contenga valores relacionados con el tiempo para poder cambiar el orden de direccionamiento del mismo; es solo que esta función es la más demandada y, de hecho, parecía funcionar con datos históricos.

Este atributo de array no afecta a la disposición de los datos en memoria. Sólo cambia el orden de numeración. En concreto, podríamos implementar su análogo en MQL5 nosotros mismos recorriendo el array en un bucle «de atrás hacia adelante». Sin embargo, MQL5 proporciona funciones preparadas para ocultar toda esta rutina a los programadores de aplicaciones.

Series temporales pueden ser cualquier array dinámico unidimensional descrito en un programa MQL, así como arrays externos pasados al programa MQL desde el núcleo de MetaTrader 5, tales como parámetros de funciones de utilidad. Por ejemplo, un tipo especial de programas MQL, los indicadores, reciben arrays con los datos de precios del gráfico actual en el controlador de eventos *OnCalculate*. Estudiaremos todas las características del uso aplicado de las series temporales más adelante, en la quinta Parte del libro.

Los arrays definidos en un programa MQL no son series temporales por defecto.

Veamos un conjunto de funciones para determinar y modificar el atributo «serie» de u array, así como su «pertenencia» al terminal. El script general de *ArrayAsSeries.mq5* con ejemplos se dará después de la descripción.

bool ArrayIsSeries(const void &array[])

La función devuelve un signo de si el array especificado es una serie temporal «real», es decir, controlada y proporcionada por el propio terminal. No se puede modificar esta característica de un array. Tales arrays están disponibles para el programa MQL en el modo «sólo lectura».

> En la documentación de MQL5, los términos «timeseries» y «series» se utilizan para describir tanto la indexación inversa de un array como el hecho de que el array puede «pertenecer» al terminal (el terminal le asigna memoria y lo llena de datos). En el libro intentaremos evitar esta ambigüedad y nos referiremos a los arrays con indexación inversa como «timeseries» (series temporales). Y los arrays del terminal serán simplemente los arrays *own* del terminal.

Puede cambiar la indexación de cualquier array personalizado del terminal a su discreción cambiándola al modo de series temporales o volviendo al estándar. Para ello se utiliza la función *ArraySetAsSeries*, que es aplicable no sólo a los propios, sino también a los arrays dinámicos personalizados (véase más adelante).

bool ArrayGetAsSeries(const void &array[])

La función devuelve un signo de si el modo de indexación de series temporales está activado para el array especificado, es decir, la indexación aumenta en la dirección del presente al pasado. Puede cambiar la dirección de indexación utilizando la función *ArraySetAsSeries*.

La dirección de indexación afecta a los valores devueltos por las funciones *ArrayBsearch*, *ArrayMaximum* y *ArrayMinimum* (véase la sección Comparar, ordenar y buscar en arrays).

bool ArraySetAsSeries(const void &array[], bool as_series)

La función establece la dirección de indexación en el array según el parámetro *as_series*: el valor *true* significa el orden inverso de indexación, mientras que *false* significa el orden normal de los elementos.

La función devuelve *true* en caso de ajuste correcto del atributo, o *false* en caso de error.

Se admiten arrays de cualquier tipo, pero está prohibido cambiar la dirección de indexación en arrays multidimensionales y de tamaño fijo.

El script *ArrayAsSeries.mq5* describe varios arrays pequeños para experimentos relacionados con las funciones anteriores.

```
#define LIMIT 10

template<typename T>
void indexArray(T &array[])
{
   for(int i = 0; i < ArraySize(array); ++i)
   {
      array[i] = (T)(i + 1);
   }
}

class Dummy
{
   int data[];
};

void OnStart()
{
   double array2D[][2];
   double fixed[LIMIT];
   double dynamic[];
   MqlRates rates[];
   Dummy dummies[];

   ArrayResize(dynamic, LIMIT); // allocating memory
   // fill in a couple of arrays with numbers: 1, 2, 3,...
   indexArray(fixed);
   indexArray(dynamic);
   ...
```

Tenemos un array bidimensional *array2D*, un array fijo y uno dinámico, todos ellos de tipo *double*, así como arrays de estructuras y objetos de clase. Los arrays *fixed* y *dynamic* se rellenan con números enteros consecutivos (utilizando la función auxiliar *indexArray*) a efectos de demostración. Para otros tipos de arrays sólo comprobaremos la aplicabilidad del modo «serie», ya que la idea del efecto de indexación inversa quedará clara en el ejemplo de los arrays rellenos.

En primer lugar, asegúrese de que ninguno de los arrays sea el propio array del terminal:

```
   PRTS(ArrayIsSeries(array2D)); // false
   PRTS(ArrayIsSeries(fixed));   // false
   PRTS(ArrayIsSeries(dynamic)); // false
   PRTS(ArrayIsSeries(rates));   // false
```

Todas las llamadas a *ArrayIsSeries* devuelven *false*, ya que hemos definido todos los arrays en el programa MQL. Veremos el valor *true* para los arrays de parámetros de la función *OnCalculate* en indicadores (en la quinta Parte).

A continuación, vamos a comprobar la dirección inicial de indexación del array:

```
PRTS(ArrayGetAsSeries(array2D)); // false, cannot be true
PRTS(ArrayGetAsSeries(fixed));   // false
PRTS(ArrayGetAsSeries(dynamic)); // false
PRTS(ArrayGetAsSeries(rates));   // false
PRTS(ArrayGetAsSeries(dummies)); // false
```

Y de nuevo nos encontraremos con *false* en todas partes.

Vamos a enviar los arrays *fixed* y *dynamic* al diario para ver el orden original de los elementos.

```
ArrayPrint(fixed, 1);
ArrayPrint(dynamic, 1);
/*
    1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0
    1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0
*/
```

Ahora intentamos cambiar el orden de indexación:

```
// error: parameter conversion not allowed
// PRTS(ArraySetAsSeries(array2D, true));

// warning: cannot be used for static allocated array
PRTS(ArraySetAsSeries(fixed, true));   // false

// after this everything is standard
PRTS(ArraySetAsSeries(dynamic, true)); // true
PRTS(ArraySetAsSeries(rates, true));   // true
PRTS(ArraySetAsSeries(dummies, true)); // true
```

Una sentencia para el array *array2D* provoca un error de compilación y, por lo tanto, se comenta.

Una sentencia para el array *fixed* emite una advertencia del compilador de que no se puede aplicar a un array de tamaño constante. En tiempo de ejecución, las tres últimas sentencias se realizan con éxito (*true*). Veamos cómo han cambiado los atributos de los arrays:

```
// attribute checks:
// first, whether they are native to the terminal
PRTS(ArrayIsSeries(fixed));            // false
PRTS(ArrayIsSeries(dynamic));          // false
PRTS(ArrayIsSeries(rates));            // false
PRTS(ArrayIsSeries(dummies));          // false

// second, indexing direction
PRTS(ArrayGetAsSeries(fixed));         // false
PRTS(ArrayGetAsSeries(dynamic));       // true
PRTS(ArrayGetAsSeries(rates));         // true
PRTS(ArrayGetAsSeries(dummies));       // true
```

Como era de esperar, los arrays no se han convertido en arrays propios del terminal. Sin embargo, tres de cada cuatro arrays han cambiado su indexación al modo de series temporales, incluyendo un array de estructuras y objetos. Para demostrar el resultado, los arrays *fixed* y *dynamic* aparecen de nuevo en el registro.

```
ArrayPrint(fixed, 1);    // without changes
ArrayPrint(dynamic, 1);  // reverse order
/*
    1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0
   10.0  9.0  8.0  7.0  6.0  5.0  4.0  3.0  2.0  1.0
*/
```

Como el modo no se aplicó al array de tamaño constante, éste permaneció inalterado. El array *dynamic* se muestra ahora en orden inverso.

Si pone el array en modo de indexación inversa, lo redimensiona y luego vuelve a la indexación anterior, los elementos añadidos se insertarán al principio del array.

4.3.9 Puesta a cero de objetos y arrays

Normalmente, la inicialización o llenado de variables y arrays no causa problemas. Así, para variables simples, podemos utilizar simplemente el operador '=' en la sentencia de definición junto con inicialización, o asignar el valor deseado en cualquier momento posterior.

La inicialización de vistas agregadas está disponible para estructuras (véase la sección Definición de estructuras):

```
Struct struct = {value1, value2, ...};
```

No obstante, ello sólo es posible si no hay cadenas y arrays dinámicos en la estructura. Además, la sintaxis de inicialización agregada no puede utilizarse para volver a limpiar una estructura. En lugar de ello, debe asignar valores a cada campo individualmente o reservar una instancia de la estructura vacía en el programa y copiarla en instancias que se puedan borrar.

Si al mismo tiempo estamos hablando de un array de estructuras, entonces el código fuente crecerá rápidamente debido a las instrucciones auxiliares pero necesarias.

Para los arrays existen las funciones *ArrayInitialize* y *ArrayFill*, pero sólo admiten tipos numéricos: un array de cadenas o estructuras no puede rellenarse con ellos.

En estos casos puede resultar útil la función *ZeroMemory*: no es la panacea, ya que tiene importantes limitaciones en su alcance, pero es bueno conocerla.

void ZeroMemory(void &entity)

La función puede aplicarse a una amplia gama de entidades diferentes: variables de tipo simple u objeto, así como sus arrays (fijos, dinámicos o multidimensionales).

Las variables obtienen el valor 0 (para números) o su equivalente (NULL para cadenas y punteros).

En el caso de un array, todos sus elementos se ponen a cero. No olvide que los elementos pueden ser objetos y, a su vez, contener objetos. En otras palabras: la función *ZeroMemory* realiza una limpieza profunda de la memoria en una sola llamada.

No obstante, existen restricciones en cuanto a los objetos válidos. Sólo puede rellenar con ceros los objetos de estructuras y clases, que:

- contengan sólo campos públicos (es decir, no contienen datos con tipo de acceso *private* o *protected*);
- no contengan campos con el modificador *const*;

- no contengan punteros.

Las dos primeras restricciones están integradas en el compilador: un intento de anular objetos con campos que no cumplan los requisitos especificados provocará errores (véase más adelante).

La tercera limitación es una recomendación: la puesta a cero externa de un puntero dificultará la comprobación de la integridad de los datos, lo que probablemente provocará la pérdida del objeto asociado y una fuga de memoria.

En sentido estricto, el requisito de publicidad de los campos en los objetos anulables viola el principio de encapsulación inherente a los objetos de clase, por lo que *ZeroMemory* se utiliza principalmente con objetos de estructuras simples y sus arrays.

En el script *ZeroMemory.mq5* se dan ejemplos de cómo trabajar con *ZeroMemory*.

Los problemas con la lista de inicialización agregada se demuestran utilizando la estructura *Simple*:

```
#define LIMIT 5

struct Simple
{
   MqlDateTime data[]; // dynamic array disables initialization list,
   // string s; // and a string field would also forbid,
   // ClassType *ptr; // and a pointer too
   Simple()
   {
      // allocating memory, it will contain arbitrary data
      ArrayResize(data, LIMIT);
   }
};
```

En la función *OnStart* o en el contexto global no podemos definir y anular inmediatamente un objeto de tal estructura:

```
void OnStart()
{
   Simple simple = {}; // error: cannot be initialized with initializer list
   ...
```

El compilador dará el error «no se puede utilizar la lista de inicialización», que es específico de campos como los arrays dinámicos, las variables de cadena y los punteros. En concreto, si el array *data* tuviera un tamaño fijo, no se produciría ningún error.

Por lo tanto, en lugar de una lista de inicialización, utilizamos *ZeroMemory*:

```
void OnStart()
{
   Simple simple;
   ZeroMemory(simple);
   ...
```

El relleno inicial con ceros también podría hacerse en el constructor de la estructura, pero es más conveniente hacer las limpiezas posteriores fuera (o proporcionar un método para ello con la misma función *ZeroMemory*).

La siguiente clase se define en *Base*.

```
class Base
{
public: // public is required for ZeroMemory
   // const for any field will cause a compilation error when calling ZeroMemory:
   // "not allowed for objects with protected members or inheritance"
   /* const */ int x;
   Simple t;   // using a nested structure: it will also be nulled
   Base()
   {
      x = rand();
   }
   virtual void print() const
   {
      PrintFormat("%d %d", &this, x);
      ArrayPrint(t.data);
   }
};
```

Dado que la clase se utiliza además en arrays de objetos anulables con *ZeroMemory*, nos vemos obligados a escribir una sección de acceso *public* para sus campos (lo cual, en principio, no es típico de las clases y se hace para ilustrar los requisitos impuestos por *ZeroMemory*). Además, tenga en cuenta que los campos no pueden tener el modificador *const*. De lo contrario, obtendremos un error de compilación con un texto que, por desgracia, no se ajusta realmente al problema: «prohibido para objetos con miembros protegidos o herencia».

El constructor de la clase rellena el campo *x* con un número aleatorio para que después se pueda ver claramente su limpieza por parte de la función *ZeroMemory*. El método *print* muestra el contenido de todos los campos para su análisis, incluido el número único de objeto (descriptor) *&this*.

MQL5 no impide la aplicación de *ZeroMemory* a una variable de puntero:

```
Base *base = new Base();
ZeroMemory(base); // will set the pointer to NULL but leave the object
```

No obstante, esto no debe hacerse, ya que la función restablece sólo la variable *base* en sí, y, si se refería a un objeto, éste quedará «colgado» en la memoria, inaccesible desde el programa debido a la pérdida del puntero.

Puede anular un puntero sólo después de que la instancia del mismo haya sido liberada utilizando el operador *delete*. Además, es más fácil restablecer un puntero independiente del ejemplo anterior, como cualquier otra variable simple (no compuesta), utilizando un operador de asignación. Tiene sentido utilizar *ZeroMemory* para objetos compuestos y arrays.

La función permite trabajar con objetos de la jerarquía de clases. Por ejemplo, podemos describir la derivada de la clase *Dummy* derivada de *Base*:

```
class Dummy : public Base
{
public:
   double data[]; // could also be multidimensional: ZeroMemory will work
   string s;
   Base *pointer; // public pointer (dangerous)

public:
   Dummy()
   {
      ArrayResize(data, LIMIT);

      // due to subsequent application of ZeroMemory to the object
      // we'll lose the 'pointer'
      // and get warnings when the script ends
      // about undeleted objects of type Base
      pointer = new Base();
   }

   ~Dummy()
   {
      // due to the use of ZeroMemory, this pointer will be lost
      // and will not be freed
      if(CheckPointer(pointer) != POINTER_INVALID) delete pointer;
   }

   virtual void print() const override
   {
      Base::print();
      ArrayPrint(data);
      Print(pointer);
      if(CheckPointer(pointer) != POINTER_INVALID) pointer.print();
   }
};
```

Incluye campos con un array dinámico de tipo *double*, cadena y puntero de tipo *Base* (este es el mismo tipo del que deriva la clase, pero se utiliza aquí sólo para demostrar los problemas de punteros, a fin de no describir otra clase ficticia). Cuando la función *ZeroMemory* anula el objeto *Dummy*, se pierde un objeto en *pointer* y no se puede liberar en el destructor. Como resultado, esto conduce a advertencias sobre fugas de memoria en los objetos restantes después de que el script termina.

ZeroMemory se utiliza en *OnStart* para borrar el array de objetos *Dummy*:

```
void OnStart()
{
   ...
   Print("Initial state");
   Dummy array[];
   ArrayResize(array, LIMIT);
   for(int i = 0; i < LIMIT; ++i)
   {
      array[i].print();
   }
   ZeroMemory(array);
   Print("ZeroMemory done");
   for(int i = 0; i < LIMIT; ++i)
   {
      array[i].print();
   }
```

El registro mostrará algo como lo siguiente (el estado inicial será diferente porque imprime el contenido de la memoria «sucia», recién asignada; aquí hay una pequeña parte de código):

```
Initial state
1048576 31539
     [year]     [mon]    [day] [hour] [min] [sec] [day_of_week] [day_of_year]
[0]       0     65665       32      0     0     0             0             0
[1]       0         0        0      0     0     0         65624             8
[2]       0         0        0      0     0     0             0             0
[3]       0         0        0      0     0     0             0             0
[4] 5242880 531430129 51557552      0     0 65665            32             0
0.0 0.0 0.0 0.0 0.0
...
ZeroMemory done
1048576 0
    [year] [mon] [day] [hour] [min] [sec] [day_of_week] [day_of_year]
[0]      0     0     0      0     0     0             0             0
[1]      0     0     0      0     0     0             0             0
[2]      0     0     0      0     0     0             0             0
[3]      0     0     0      0     0     0             0             0
[4]      0     0     0      0     0     0             0             0
0.0 0.0 0.0 0.0 0.0
...
5 undeleted objects left
5 objects of type Base left
3200 bytes of leaked memory
```

Para comparar el estado de los objetos antes y después de la limpieza, utilice descriptores.

Así, una sola llamada a *ZeroMemory* es capaz de restablecer el estado de una estructura de datos ramificada arbitraria (arrays, estructuras, arrays de estructuras con campos de estructura anidados y arrays).

Por último, veamos cómo *ZeroMemory* puede resolver el problema de la inicialización de arrays de cadenas. Las funciones *ArrayInitialize* y *ArrayFill* no funcionan con cadenas.

```
string text[LIMIT] = {};
// an algorithm populates and uses 'text'
// ...
// then you need to re-use the array
// calling functions gives errors:
// ArrayInitialize(text, NULL);
//      `-> no one of the overloads can be applied to the function call
// ArrayFill(text, 0, 10, NULL);
//      `->  'string' type cannot be used in ArrayFill function
ZeroMemory(text);               // ok
```

En las instrucciones comentadas, el compilador generaría errores, indicando que el tipo *string* no se admite en estas funciones.

La solución a este problema es la función *ZeroMemory*.

4.4 Funciones matemáticas

Las funciones matemáticas más populares suelen estar disponibles en todos los lenguajes de programación modernos, y MQL5 no es una excepción. En este capítulo examinaremos varios grupos de funciones listas para usar. Entre ellas se incluyen funciones de redondeo, trigonométricas, hiperbólicas, exponenciales, logarítmicas y de potencia, así como algunas especiales, como la generación de números aleatorios y la comprobación de la normalidad de los números reales.

La mayoría de las funciones tienen dos nombres: completo (con el prefijo «Math» y mayúsculas) y abreviado (sin prefijo, en minúsculas). Proporcionaremos ambas opciones: funcionan de la misma manera. La elección puede basarse en el estilo de formato de los códigos fuente.

Dado que las funciones matemáticas realizan algunos cálculos y devuelven un resultado como un número real, los posibles errores pueden llevar a una situación en la que el resultado sea indefinido. Por ejemplo, no se puede sacar la raíz cuadrada de un número negativo ni sacar el logaritmo de cero. En tales casos, las funciones devuelven valores especiales que no son números (NaN, «Not A Number»). Ya nos hemos enfrentado a ellos en las secciones Números reales, Operaciones aritméticas y De números a cadenas y viceversa. La corrección numérica y la ausencia de errores pueden analizarse mediante las funciones *MathIsValidNumber* y *MathClassify* (véase la sección Comprobación de la normalidad de los números reales).

La presencia de al menos un operando con valor NaN provocará que cualquier cálculo posterior que implique este operando, incluidas las llamadas a funciones, también dé como resultado NaN.

> Para autoaprendizaje y material visual, puede utilizar como anexo el script *MathPlot.mq5*, que permite visualizar gráficas de funciones matemáticas con un argumento de los descritos. El script utiliza la biblioteca de dibujo estándar *Graphic.mqh* proporcionada en MetaTrader 5 (fuera del alcance de este libro). La de abajo es una muestra del aspecto que podría tener una curva sinusoidal hiperbólica en la ventana de MetaTrader 5.

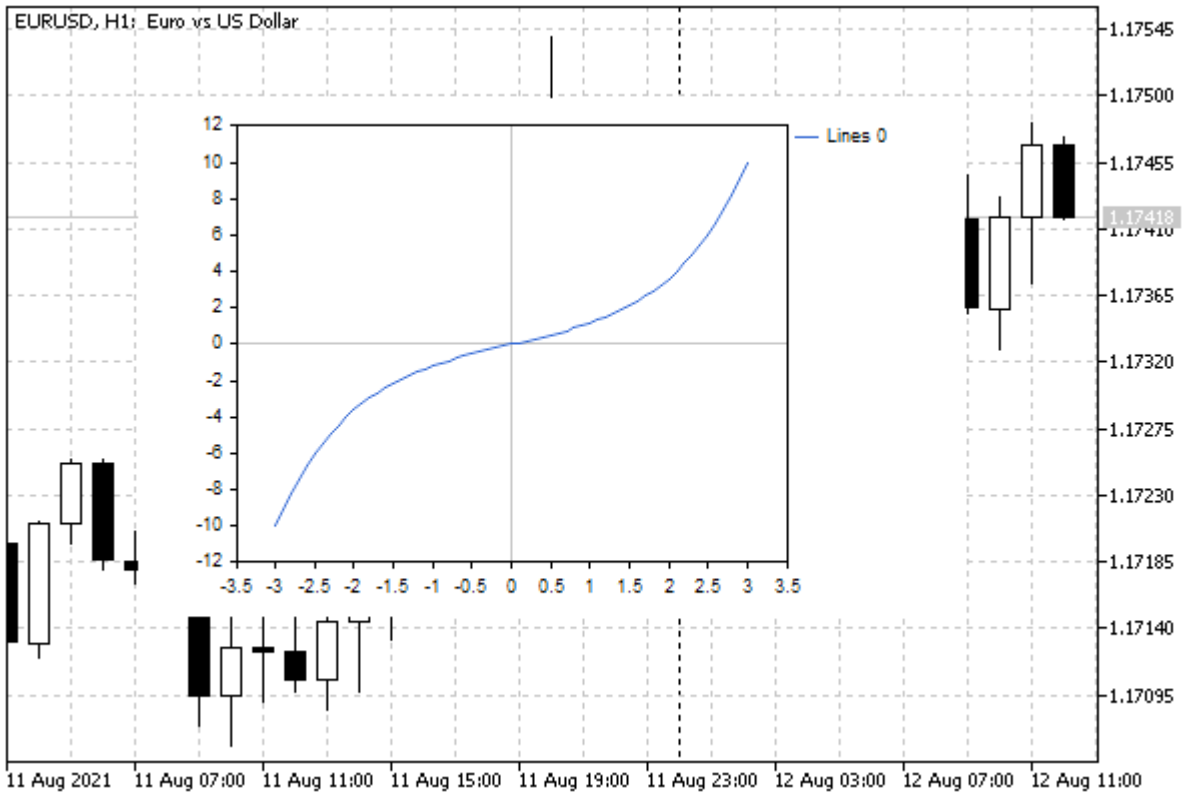


Gráfico del seno hiperbólico en la ventana de MetaTrader 5

4.4.1 El valor absoluto de un número

La API de MQL5 proporciona la función *MathAbs*, que puede eliminar el signo menos del número si existe. Por lo tanto, no es necesario codificar manualmente equivalentes más largos como éste:

```
if(x < 0) x = -x;
```

numeric MathAbs(numeric value) ≡ numeric fabs(numeric value)

La función devuelve el valor absoluto del número que se le pasa, es decir, su módulo. El argumento puede ser un número de cualquier tipo. En otras palabras: la función está sobrecargada para *char/uchar*, *short/ushort*, *int/uint*, *long/ulong*, *float* y *double*, aunque para los tipos sin signo los valores son siempre no negativos.

Al pasar una cadena, se convertirá implícitamente en un número *double*, y el compilador generará una advertencia pertinente.

El tipo del valor de retorno es siempre el mismo que el tipo del argumento, por lo que el compilador puede necesitar convertir el valor al tipo de la variable receptora si los tipos son diferentes.

Encontrará ejemplos de uso de las funciones en el archivo *MathAbs.mq5*.

```
void OnStart()
{
   double x = 123.45;
   double y = -123.45;
   int i = -1;

   PRT(MathAbs(x)); // 123.45, number left "as is"
   PRT(MathAbs(y)); // 123.45, minus sign gone
   PRT(MathAbs(i)); // 1, int is handled naturally

   int k = MathAbs(i);  // no warning: type int for parameter and result

   // situations with warnings:
   // double to long conversion required
   long j = MathAbs(x); // possible loss of data due to type conversion

   // need to be converted from large type (4 bytes) to small type (2 bytes)
   short c = MathAbs(i); // possible loss of data due to type conversion
   ...
```

Es importante tener en cuenta que convertir un entero con signo en un entero sin signo no equivale a tomar el módulo de un número:

```
   uint u_cast = i;
   uint u_abs = MathAbs(i);
   PRT(u_cast);             // 4294967295, 0xFFFFFFFF
   PRT(u_abs);              // 1
```

Tenga en cuenta también que el número 0 puede tener signo:

```
   ...
   double n = 0;
   double z = i * n;
   PRT(z);               // -0.0
   PRT(MathAbs(z));      //  0.0
   PRT(z == MathAbs(z)); // true
}
```

Uno de los mejores ejemplos de cómo utilizar *MathAbs* es comprobar la igualdad de dos números reales. Como es sabido, los números reales tienen una precisión limitada para representar valores, que puede degradarse aún más en el curso de cálculos largos (por ejemplo, la suma de diez valores 0.1 no da exactamente 1.0). La condición estricta *value1 == value2* puede dar *false* en la mayoría de los casos, cuando la igualdad puramente especulativa debería mantenerse.

Por lo tanto, para comparar valores reales, se suele utilizar la siguiente notación:

```
MathAbs(value1 - value2) < EPS
```

donde EPS es un pequeño valor positivo que indica una precisión (véase un ejemplo en la sección operaciones de comparación).

4.4.2 Máximo y mínimo de dos números

Para encontrar el número mayor o menor de dos, MQL5 ofrece las funciones *MathMax* y *MathMin*. Sus alias abreviados son *fmax* y *fmin*, respectivamente.

numeric MathMax(numeric value1, numeric value2) ≡ numeric fmax(numeric value1, numeric value2)
numeric MathMin(numeric value1, numeric value2) ≡ numeric fmin(numeric value1, numeric value2)

Las funciones devuelven el máximo o el mínimo de los dos valores pasados. Las funciones están sobrecargadas para todos los tipos integrados.

Si se pasan a la función parámetros de tipos diferentes, el parámetro del tipo «inferior» se convierte automáticamente en el tipo «superior»; por ejemplo, en un par de tipos *int* y *double*, *int* se llevará a *double*. Para obtener más información sobre la conversión implícita de tipos, consulte la sección Conversiones aritméticas de tipo. El tipo de retorno corresponde al tipo «más alto».

Cuando haya un parámetro de tipo *string*, será «senior», es decir, todo se reduce a una cadena. Las cadenas se compararán lexicográficamente, como en la función *StringCompare*.

El script *MathMaxMin.mq5* muestra las funciones en acción.

```
void OnStart()
{
   int i = 10, j = 11;
   double x = 5.5, y = -5.5;
   string s = "abc";

   // numbers
   PRT(MathMax(i, j)); // 11
   PRT(MathMax(i, x)); // 10
   PRT(MathMax(x, y)); // 5.5
   PRT(MathMax(i, s)); // abc

   // type conversions
   PRT(typename(MathMax(i, j))); // int, as is
   PRT(typename(MathMax(i, x))); // double
   PRT(typename(MathMax(i, s))); // string
}
```

4.4.3 Funciones de redondeo

La API de MQL5 incluye varias funciones para redondear números al entero más próximo (en una u otra dirección). A pesar de la operación de redondeo, todas las funciones devuelven un número del tipo *double* (con la parte fraccionaria vacía).

Desde un punto de vista técnico, aceptan argumentos de cualquier tipo numérico, pero sólo se redondean los números reales, y los enteros sólo se convierten a *double*.

Si desea redondear a un signo específico, utilice *NormalizeDouble* (véase la sección Normalización de doubles).

En el archivo *MathRound.mq5* se dan ejemplos de cómo trabajar con funciones.

double MathRound(numeric value) ≡ double round(numeric value)

La función redondea un número hacia arriba o hacia abajo al entero más próximo.

```
PRT((MathRound(5.5)));  // 6.0
PRT((MathRound(-5.5))); // -6.0
PRT((MathRound(11)));   // 11.0
PRT((MathRound(-11)));  // -11.0
```

Si el valor de la parte fraccionaria es mayor o igual que 0.5, la mantisa se incrementa en uno (independientemente del signo del número).

double MathCeil(numeric value) ≡ double ceil(numeric value)
double MathFloor(numeric value) ≡ double floor(numeric value)

Las funciones devuelven el valor entero mayor más cercano (para *ceil*) o el valor entero menor más cercano (para *floor*) al *value* transferido. Si *value* ya es igual a un entero (tiene una parte fraccionaria cero), se devuelve este entero.

```
PRT((MathCeil(5.5)));   // 6.0
PRT((MathCeil(-5.5)));  // -5.0
PRT((MathFloor(5.5)));  // 5.0
PRT((MathFloor(-5.5))); // -6.0
PRT((MathCeil(11)));    // 11.0
PRT((MathCeil(-11)));   // -11.0
PRT((MathFloor(11)));   // 11.0
PRT((MathFloor(-11)));  // -11.0
```

4.4.4 Resto tras la división (operación módulo)

Para dividir enteros con resto, MQL5 lleva integrado el operador módulo '%', que se describe en la sección Operaciones aritméticas. Sin embargo, este operador no es aplicable a los números reales. En el caso de que el divisor, el dividendo o ambos operandos sean reales, deberá utilizar la función *MathMod* (o la forma abreviada *fmod*).

double MathMod(double dividend, double divider) ≡ double fmod(double dividend, double divider)

La función devuelve el resto real después de dividir el primer número pasado (*dividend*) por el segundo (*divider*).

Si alguno de los argumentos es negativo, el signo del resultado se determina mediante las reglas descritas en la sección anterior.

Encontrará ejemplos de cómo funciona la función en el script *MathMod.mq5*.

```
PRT(MathMod(10.0, 3));     // 1.0
PRT(MathMod(10.0, 3.5));   // 3.0
PRT(MathMod(10.0, 3.49));  // 3.02
PRT(MathMod(10.0, M_PI));  // 0.5752220392306207
PRT(MathMod(10.0, -1.5));  // 1.0, the sign is gone
PRT(MathMod(-10.0, -1.5)); // -1.0
```

4.4.5 Potencias y raíces

La API de MQL5 proporciona una función genérica *MathPow* para elevar un número a una potencia arbitraria, así como una función para un caso especial con una potencia de 0.5, más familiar para nosotros como la operación de extraer una raíz cuadrada *MathSqrt*.

Para probar las funciones, utilice el script *MathPowSqrt.mq5*.

double MathPow(double base, double exponent) ≡ double pow(double base, double exponent)

La función eleva *base* a la potencia especificada *exponent*.

```
PRT(MathPow(2.0, 1.5));  // 2.82842712474619
PRT(MathPow(2.0, -1.5)); // 0.3535533905932738
PRT(MathPow(2.0, 0.5));  // 1.414213562373095
```

double MathSqrt(double value) ≡ double sqrt(double value)

La función devuelve la raíz cuadrada de un número.

```
PRT(MathSqrt(2.0));      // 1.414213562373095
PRT(MathSqrt(-2.0));     // -nan(ind)
```

MQL5 define varias constantes que contienen valores de cálculo ya preparados que implican *sqrt*.

Constante	Descripción	Valor
M_SQRT2	sqrt(2.0)	1.4142135623730950 4880
M_SQRT1_2	1 / sqrt(2.0)	0.7071067811865475 24401
M_2_SQRTPI	2.0 / sqrt(M_PI)	1.1283791670955125 7390

Aquí, M_PI es el número Pi (π=3.14159265358979323846, véase más adelante la sección Funciones trigonométricas).

Todas las constantes integradas se describen en la documentación.

4.4.6 Funciones exponenciales y logarítmicas

El cálculo de funciones exponenciales y logarítmicas está disponible en MQL5 utilizando la sección correspondiente de la API.

La ausencia del logaritmo binario en la API, que suele ser necesario en informática y combinatoria, no supone ningún problema, ya que es fácil de calcular, previa solicitud, mediante las funciones de logaritmo natural o decimal disponibles.

```
log2(x) = log(x) / log(2) = log(x) / M_LN2
log2(x) = log10(x) / log10(2)
```

Aquí, *log* y *log10* son funciones logarítmicas disponibles (basadas en *e* y 10, respectivamente), M_LN2 es una constante integrada igual a *log(2)*.

La siguiente tabla enumera todas las constantes que pueden ser útiles en los cálculos logarítmicos.

Constante	Descripción	Valor
M_E	e	2.7182818284590452353 6
M_LOG2E	log2(e)	1.4426950408889634073 6
M_LOG10E	log10(e)	0.4342944819032518276 51
M_LN2	ln(2)	0.6931471805599453094 17
M_LN10	ln(10)	2.3025850929940456840 2

En el archivo *MathExp.mq5* se recogen ejemplos de las funciones que se describen a continuación.

double MathExp(double value) ≡ double exp(double value)

La función devuelve el exponente, es decir, el número *e* (disponible como constante predefinida M_E) elevado a la potencia especificada *value*. En caso de desbordamiento, la función devuelve *inf* (una especie de NaN para el infinito).

```
PRT(MathExp(0.5));       // 1.648721270700128
PRT(MathPow(M_E, 0.5)); // 1.648721270700128
PRT(MathExp(10000.0));   // inf, NaN
```

double MathLog(double value) ≡ double log(double value)

La función devuelve el logaritmo natural del número introducido. Si *value* es negativo, la función devuelve *-nan(ind)* (NaN «valor indefinido»). Si *value* es 0, la función devuelve *inf* (NaN «infinito»).

```
PRT(MathLog(M_E));      // 1.0
PRT(MathLog(10000.0)); // 9.210340371976184
PRT(MathLog(0.5));      // -0.6931471805599453
PRT(MathLog(0.0));      // -inf, NaN
PRT(MathLog(-0.5));     // -nan(ind)
PRT(Log2(128));         // 7
```

La última línea utiliza la implementación del logaritmo binario a través de *MathLog*:

```
double Log2(double value)
{
   return MathLog(value) / M_LN2;
}
```

double MathLog10(double value) ≡ double log10(double value)

La función devuelve el logaritmo decimal de un número.

```
PRT(MathLog10(10.0)); // 1.0
PRT(MathLog10(10000.0)); // 4.0
```

double MathExpm1(double value) ≡ double expm1(double value)

La función devuelve el valor de la expresión *(MathExp(value) - 1)*. En cálculos económicos, la función se utiliza para calcular el interés efectivo (ingreso o pago) por unidad de tiempo en un esquema de interés compuesto cuando el número de periodos tiende a infinito.

```
PRT(MathExpm1(0.1)); // 0.1051709180756476
```

double MathLog1p(double value) ≡ double log1p(double value)

La función devuelve el valor de la expresión *MathLog(1 + value)*, es decir, realiza la acción contraria a la función *MathExpm1*.

```
PRT(MathLog1p(0.1)); // 0.09531017980432487
```

4.4.7 Funciones trigonométricas

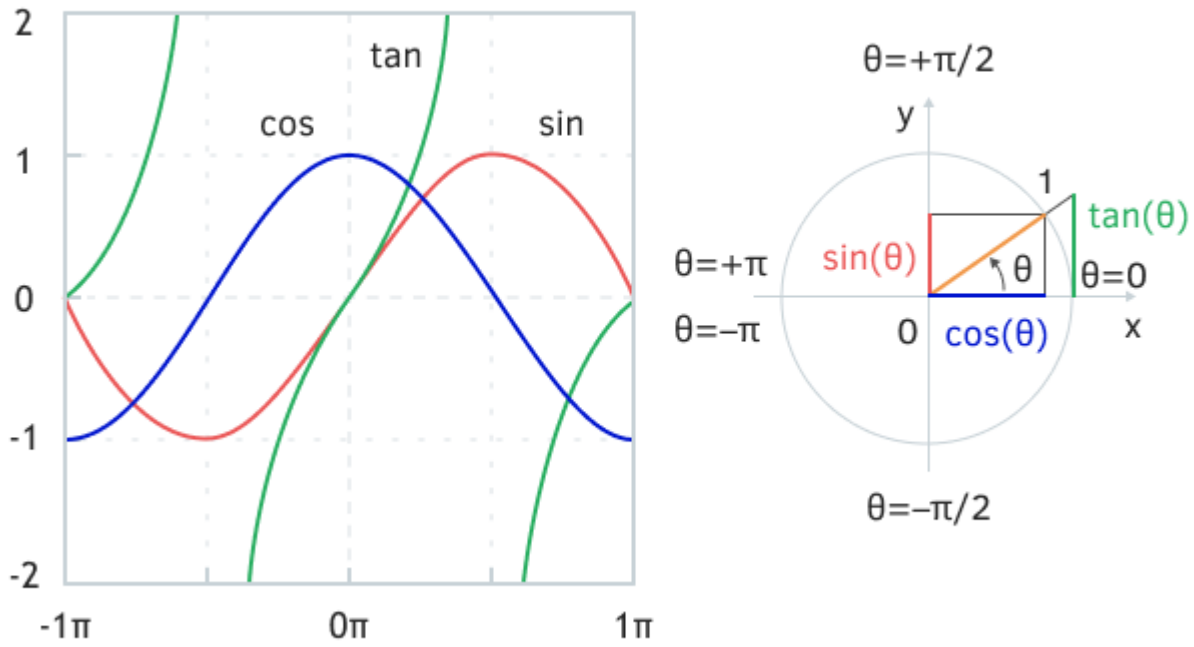
MQL5 proporciona las tres funciones trigonométricas principales (*MathCos, MathSin, MathTan*) y sus inversas (*MathArccos, MathArcsin, MathArctan*). Todos ellos trabajan con ángulos en radianes. Para ángulos en grados, utilice la fórmula:

```
radians = degrees * M_PI / 180
```

Aquí, M_PI es una de varias constantes con cantidades trigonométricas (pi y sus derivadas) integradas en el lenguaje.

Constante	Descripción	Valor
M_PI	π	3.1415926535897932384 6
M_PI_2	π/2	1.5707963267948966192 3
M_PI_4	π/4	0.7853981633974483096 16
M_1_PI	1/π	0.3183098861837906715 38
M_2_PI	2/π	0.6366197723675813430 76

La tangente del arco también puede calcularse para una cantidad representada por el cociente de dos coordenadas *y* y *x*: esta versión ampliada se denomina *MathArctan2*; es capaz de restituir ángulos en todo el rango del círculo de -M_PI a +M_PI, a diferencia de *MathArctan*, que se limita a -M_PI_2 a +M_PI_2.



Funciones trigonométricas y cuadrantes del círculo unitario

Encontrará ejemplos de cálculos en el script *MathTrig.mq5* (véase después de las descripciones).

double MathCos(double value) ≡ double cos(double value)
double MathSin(double value) ≡ double sin(double value)

Las funciones devuelven, respectivamente, el coseno y el seno del número pasado (el ángulo está en radianes).

double MathTan(double value) ≡ double tan(double value)

La función devuelve la tangente del número pasado (el ángulo está en radianes).

double MathArccos(double value) ≡ double acos(double value)
double MathArcsin(double value) ≡ double asin(double value)

Las funciones devuelven el valor, respectivamente, del arcocoseno y arcoseno del número pasado, es decir, el ángulo en radianes. Si *x = MathCos(t)*, entonces *t = MathArccos(x)*. El seno y el arcoseno tienen un esquema similar. Si *y = MathSin(t)*, entonces *t = MathArcsin(y)*.

El parámetro debe estar comprendido entre -1 y +1. En caso contrario, la función devolverá NaN.

El resultado del arcocoseno está en el rango de 0 a M_PI, y el resultado del arcoseno está entre -M_PI_2 y +M_PI_2. Los rangos indicados se denominan rangos principales, ya que las funciones son multivaloradas, es decir, sus valores se repiten periódicamente. Los semiperíodos seleccionados cubren completamente la zona de definición de -1 a +1.

El ángulo resultante para el coseno se encuentra en el semicírculo superior, y la solución simétrica del semicírculo inferior se puede obtener añadiendo un signo, es decir, *t=-t*. Para el seno, el ángulo resultante está en el semicírculo de la derecha, y la segunda solución en el semicírculo de la izquierda es *M_PI-t* (si para *t* negativo también se requiere obtener un ángulo adicional negativo, entonces *-M_PI-t*).

double MathArctan(double value) ≡ double atan(double value)

La función devuelve el valor de la tangente del arco para el número pasado, es decir, el ángulo en radianes, en el rango de -M_PI_2 a +M_PI_2.

La función es inversa a *MathTan*, pero con una salvedad.

Obsérvese que el período de la tangente es 2 veces menor que el período completo (circunferencia) debido a que la razón de seno y coseno se repite en cuadrantes opuestos (cuartos de circunferencia) por superposición de signos. Como resultado, el valor de la tangente por sí solo no es suficiente para determinar de forma unívoca el ángulo original en todo el rango de -M_PI a +M_PI. Para ello se puede utilizar la función *MathArctan2*, en la que la tangente se representa mediante dos componentes separados.

double MathArctan2(double y, double x) ≡ double atan2(double y, double x)

La función devuelve en radianes el valor del ángulo cuya tangente es igual al cociente de dos números especificados: coordenadas a lo largo del eje *y* y a lo largo del eje *x*.

El resultado (denominémoslo *r*) se encuentra en el intervalo de -M_PI a +M_PI, y para él se cumple la condición *MathTan(r) = y / x*.

La función tiene en cuenta el signo de ambos argumentos para determinar el cuadrante correcto (sujeto a condiciones límite, cuando *x* o *y* son iguales a 0, es decir, están en el límite de los cuadrantes).

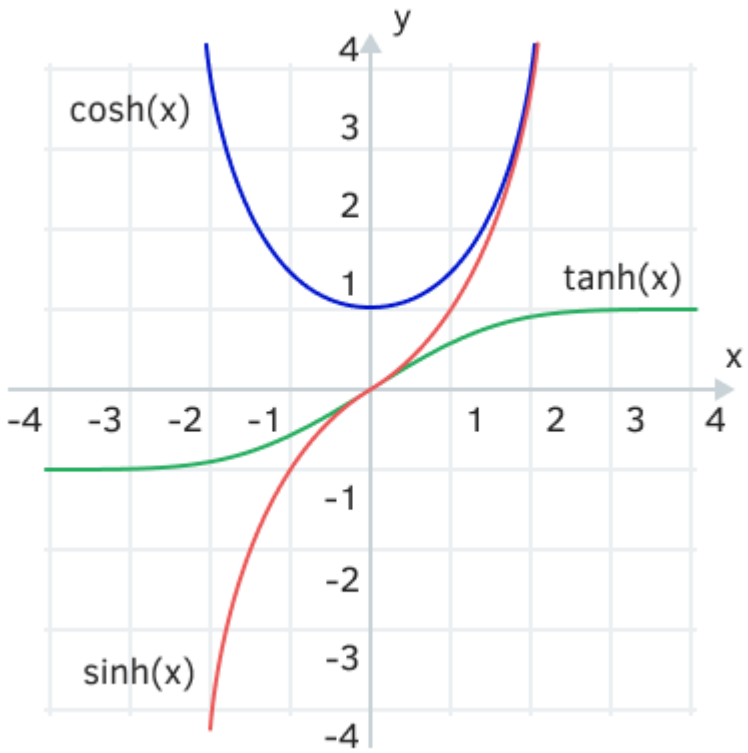
- 1 — x >= 0, y >= 0, 0 <= r <= M_PI_2
- 2 — x < 0, y >= 0, M_PI_2 < r <= M_PI
- 3 — x < 0, y < 0, -M_PI < r < -M_PI_2
- 4 — x >= 0, y < 0, -M_PI_2 <= r < 0

A continuación se muestran los resultados de llamar a funciones trigonométricas en el script *MathTrig.mq5*.

```
void OnStart()
{
   PRT(MathCos(1.0));      // 0.5403023058681397
   PRT(MathSin(1.0));      // 0.8414709848078965
   PRT(MathTan(1.0));      // 1.557407724654902
   PRT(MathTan(45 * M_PI / 180.0)); // 0.9999999999999999

   PRT(MathArccos(1.0));          // 0.0
   PRT(MathArcsin(1.0));          // 1.570796326794897 == M_PI_2
   PRT(MathArctan(0.5));          // 0.4636476090008061, Q1
   PRT(MathArctan2(1.0, 2.0));    // 0.4636476090008061, Q1
   PRT(MathArctan2(-1.0, -2.0)); // -2.677945044588987, Q3
}
```

4.4.8 Funciones hiperbólicas

La API de MQL5 incluye un conjunto de funciones hiperbólicas directas e inversas.



Funciones hiperbólicas

double MathCosh(double value) ≡ double cosh(double value)
double MathSinh(double value) ≡ double sinh(double value)
double MathTanh(double value) ≡ double tanh(double value)

Las tres funciones básicas calculan el coseno, el seno y la tangente hiperbólicos.

double MathArccosh(double value) ≡ double acosh(double value)
double MathArcsinh(double value) ≡ double asinh(double value)
double MathArctanh(double value) ≡ double atanh(double value)

Las tres funciones inversas calculan el coseno inverso hiperbólico, el seno inverso y la tangente de arco.

Para el arcocoseno, el argumento debe ser mayor o igual a +1. En caso contrario, la función devolverá NaN.

El arco tangente se define de -1 a +1. Si el argumento supera estos límites, la función devolverá NaN.

En el script *MathHyper.mq5* se muestran ejemplos de funciones hiperbólicas.

```
void OnStart()
{
   PRT(MathCosh(1.0));     // 1.543080634815244
   PRT(MathSinh(1.0));     // 1.175201193643801
   PRT(MathTanh(1.0));     // 0.7615941559557649

   PRT(MathArccosh(0.5)); // nan
   PRT(MathArcsinh(0.5)); // 0.4812118250596035
   PRT(MathArctanh(0.5)); // 0.5493061443340549

   PRT(MathArccosh(1.5)); // 0.9624236501192069
   PRT(MathArcsinh(1.5)); // 1.194763217287109
   PRT(MathArctanh(1.5)); // nan
}
```

4.4.9 Prueba de normalidad para números reales

Dado que los cálculos con números reales pueden tener situaciones anómalas, como salirse del ámbito de una función, obtener infinito matemático o pérdida de orden, entre otras, el resultado puede no contener ningún número. En lugar de ello, puede contener un valor especial que describa realmente la naturaleza del problema. Todos estos valores especiales tienen un nombre genérico «no es un número» (Not A Number, NaN).

Ya nos hemos enfrentado a ellos en las secciones anteriores del libro. En concreto, cuando se envía a un diario (véase la sección De números a cadenas y viceversa) se muestran como etiquetas de texto (por ejemplo, *nan(ind)*, *+inf* y otras). Otra característica es que un solo valor NaN entre los operandos de cualquier expresión es suficiente para que toda la expresión deje de evaluarse correctamente y comience a dar el resultado NaN. Las únicas excepciones son los «no números» que representan el más/menos del infinito: si divides algo por ellos, obtienes cero. Sin embargo, aquí hay una excepción esperada: si dividimos infinito por infinito, volvemos a obtener NaN.

Por lo tanto, es importante que los programas determinen el momento en que aparece NaN en los cálculos y manejen la situación de una manera especial: señalen un error, sustituyan algún valor por defecto aceptable o repitan el cálculo con otros parámetros (por ejemplo, reduzcan la precisión/el paso del algoritmo iterativo).

Hay dos funciones en MQL5 que le permiten analizar un número real para la normalidad:*MathIsValidNumber* da una respuesta simple: sí (*true*) o no (*false*), y *MathClassify* produce una categorización más detallada.

A nivel físico, todos los valores especiales se codifican en un número con una combinación especial de bits que no se utiliza para representar números ordinarios. Para los tipos *double* y *float* estas codificaciones son, por supuesto, diferentes. Echemos un vistazo a *double* entre bastidores (ya que tiene más demanda que *float*).

En el capítulo Plantillas anidadas creamos una clase *Converter* para cambiar de vista combinando dos tipos diferentes en una unión. Vamos a utilizar esta clase para estudiar el dispositivo de bits NaN.

Por comodidad, trasladaremos la clase a un archivo de encabezado independiente *ConverterT.mqh*. Vamos a conectar este archivo mqh en el script de prueba *MathInvalid.mq5* y a crear una instancia de un conversor para un grupo de tipos *double/ulong* (el orden no es importante ya que el conversor es capaz de trabajar en ambas direcciones).

```
static Converter<ulong, double> NaNs;
```

La combinación de bits en NaN está estandarizada, así que vamos a tomar algunos valores de uso común representados por las constantes *ulong*, y vamos a ver cómo reaccionan ante ellos las funciones integradas.

```
// basic NaNs
#define NAN_INF_PLUS  0x7FF0000000000000
#define NAN_INF_MINUS 0xFFF0000000000000
#define NAN_QUIET     0x7FF8000000000000
#define NAN_IND_MINUS 0xFFF8000000000000

// custom NaN examples
#define NAN_QUIET_1   0x7FF8000000000001
#define NAN_QUIET_2   0x7FF8000000000002

static double pinf = NaNs[NAN_INF_PLUS];  // +infinity
static double ninf = NaNs[NAN_INF_MINUS]; // -infinity
static double qnan = NaNs[NAN_QUIET];     // quiet NaN
static double nind = NaNs[NAN_IND_MINUS]; // -nan(ind)

void OnStart()
{
   PRT(MathIsValidNumber(pinf));               // false
   PRT(EnumToString(MathClassify(pinf)));      // FP_INFINITE
   PRT(MathIsValidNumber(nind));               // false
   PRT(EnumToString(MathClassify(nind)));      // FP_NAN
   ...
}
```

Como era de esperar, los resultados fueron los mismos.

Veamos la descripción formal de las funciones *MathIsValidNumber* y *MathClassify*, y continuemos con las pruebas.

bool MathIsValidNumber(double value)

La función comprueba la corrección de un número real. El parámetro puede ser del tipo *double* o *float*. El resultado *true* significa el número correcto, y *false* significa «no es un número» (una de las variedades de NaN).

ENUM_FP_CLASS MathClassify(double value)

La función devuelve la categoría de un número real (de tipo *double* o *float*) que es uno de los valores del enum ENUM_FP_CLASS:

- FP_NORMAL es un número normal.
- FP_SUBNORMAL es un número inferior al número mínimo representable de forma normalizada (por ejemplo, para el tipo *double*, se trata de valores inferiores a DBL_MIN, 2.2250738585072014e-308); pérdida de orden (precisión).
- FP_ZERO es cero (positivo o negativo).
- FP_INFINITE es infinito (positivo o negativo).

- FP_NAN significa todos los demás tipos de «no números» (subdivididos en familias de NaN «silenciosos» y «de señal»).

MQL5 no proporciona NaN de alerta que se utilizan en el mecanismo de excepciones y permite la interceptación y respuesta a errores críticos dentro del programa. No existe tal mecanismo en MQL5, así que, por ejemplo, en caso de una división cero, el programa MQL simplemente termina su trabajo (se descarga del gráfico).

Puede haber muchos NaN «silenciosos», y puede construirlos utilizando un conversor para diferenciar y manejar estados no estándar en sus algoritmos computacionales.

Realicemos algunos cálculos en *MathInvalid.mq5* para visualizar cómo se obtienen los números de las distintas categorías.

```
// calculations with double
  PRT(MathIsValidNumber(0));                        // true
  PRT(EnumToString(MathClassify(0)));               // FP_ZERO
  PRT(MathIsValidNumber(M_PI));                     // true
  PRT(EnumToString(MathClassify(M_PI)));            // FP_NORMAL
  PRT(DBL_MIN / 10);                                // 2.225073858507203e-309
  PRT(MathIsValidNumber(DBL_MIN / 10));             // true
  PRT(EnumToString(MathClassify(DBL_MIN / 10)));    // FP_SUBNORMAL
  PRT(MathSqrt(-1.0));                              // -nan(ind)
  PRT(MathIsValidNumber(MathSqrt(-1.0)));           // false
  PRT(EnumToString(MathClassify(MathSqrt(-1.0))));// FP_NAN
  PRT(MathLog(0));                                  // -inf
  PRT(MathIsValidNumber(MathLog(0)));               // false
  PRT(EnumToString(MathClassify(MathLog(0))));      // FP_INFINITE

// calculations with float
  PRT(1.0f / FLT_MIN / FLT_MIN);                              // inf
  PRT(MathIsValidNumber(1.0f / FLT_MIN / FLT_MIN));           // false
  PRT(EnumToString(MathClassify(1.0f / FLT_MIN / FLT_MIN))); // FP_INFINITE
```

Podemos utilizar el convertidor en sentido contrario: para obtener su representación en bits mediante el valor *double*, y detectar así los «no números»:

```
  PrintFormat("%I64X", NaNs[MathSqrt(-1.0)]);      // FFF8000000000000
  PRT(NaNs[MathSqrt(-1.0)] == NAN_IND_MINUS);      // true, nind
```

La función *PrintFormat* es similar a *StringFormat*; la única diferencia es que el resultado se imprime inmediatamente en el registro y no en una cadena.

Por último, asegurémonos de que los «no números» siempre sean no iguales:

```
  // NaN != NaN always true
  PRT(MathSqrt(-1.0) != MathSqrt(-1.0)); // true
  PRT(MathSqrt(-1.0) == MathSqrt(-1.0)); // false
```

Para obtener NaN o infinito en MQL5 existe un método basado en la conversión de las cadenas «nan» e «inf» a *double*.

```
double nan = (double)"nan";
double infinity = (double)"inf";
```

4.4.10 Generación de números aleatorios

Muchos algoritmos de trading requieren la generación de números aleatorios. MQL5 proporciona dos funciones que inicializan y luego sondean el generador de enteros pseudoaleatorios.

Para obtener una mayor «aleatoriedad», puede utilizar la biblioteca Alglib disponible en MetaTrader 5 (véase *MQL5/Include/Math/Alglib/alglib.mqh*).

void MathSrand(int seed) ≡ void srand(int seed)

La función establece un estado inicial del generador de enteros pseudoaleatorios. Debe llamarse una vez antes de iniciar el algoritmo. Los valores aleatorios propiamente dichos deben obtenerse mediante la llamada secuencial de la función *MathRand*.

Inicializando un generador con el mismo valor *seed* se pueden obtener secuencias de números reproducibles. El valor *seed* no es el primer número aleatorio obtenido de *MathRand*. El generador mantiene un cierto estado interno, que en cada momento del tiempo (entre llamadas a él para un nuevo número aleatorio) se caracteriza por un valor entero que está disponible desde el programa como la variable *uint* integrada *_RandomSeed*. Es este valor de estado inicial el que establece la llamada a *MathSrand*.

El funcionamiento del generador en cada llamada a *MathRand* se describe mediante dos fórmulas:

```
Xn = Tf(Xp)
R = Gf(Xn)
```

La función *Tf* se denomina transición. Calcula el nuevo estado interno del generador *Xn* a partir del estado anterior *Xp*.

La función *Gf* genera otro valor «aleatorio» que devolverá la función *MathRand*, utilizando para ello un nuevo estado interno.

En MQL5, estas fórmulas se implementan de la siguiente manera (pseudocódigo):

```
Tf: _RandomSeed = _RandomSeed * 214013 + 2531011
Gf: MathRand = (_RandomSeed >> 16) & 0x7FFF
```

Se recomienda pasar la función *GetTickCount* o *TimeLocal* como el valor *seed*.

int MathRand() ≡ int rand()

La función devuelve un entero pseudoaleatorio en el rango de 0 a 32767. La secuencia de números generados varía en función de la inicialización de apertura realizada al llamar a *MathSrand*.

En el archivo *MathRand.mq5* se ofrece un ejemplo de trabajo con el generador. Calcula estadísticas sobre la distribución de los números generados en un número determinado de subrangos (cestas). Lo ideal sería obtener una distribución uniforme.

```
#define LIMIT 1000 // number of attempts (generated numbers)
#define STATS 10   // number of baskets

int stats[STATS] = {}; // calculation of statistics of hits in baskets

void OnStart()
{
   const int bucket = 32767 / STATS;
   // generator reset
   MathSrand((int)TimeLocal());
   // repeat the experiment in a loop
   for(int i = 0; i < LIMIT; ++i)
   {
      // getting a new random number and updating statistics
      stats[MathRand() / bucket]++;
   }
   ArrayPrint(stats);
}
```

Ejemplo de resultados para tres ejecuciones (cada vez obtendremos una nueva secuencia):

```
 96  93 117  76  98  88 104 124 113  91
110  81 106  88 103  90 105 102 106 109
 89  98  98 107 114  90 101 106  93 104
```

4.4.11 Control de la codificación endian de números enteros

Varios sistemas de información, a nivel de hardware, utilizan diferentes órdenes de bytes cuando representan números en la memoria. Por lo tanto, al integrar programas MQL con el «mundo exterior», en concreto, al implementar protocolos de comunicación de red o leer o escribir archivos de formatos comunes, puede ser necesario cambiar el orden de los bytes.

Los ordenadores con Windows aplican el sistema «little-endian» (empezando por el byte menos significativo), es decir, el byte más bajo va primero en la celda de memoria asignada a la variable, luego le sigue el byte con bits más altos, y así sucesivamente. La alternativa «big-endian» (empezando por el dígito más alto, el byte más significativo) se utiliza mucho en Internet. En este caso, el primer byte de la celda de memoria es el byte con los bits altos, y el último byte es el bit bajo. Este orden es similar a la forma en que escribimos los números «de izquierda a derecha» en la vida real. Por ejemplo, el valor 1234 empieza por 1, que significa miles, seguido de 2 para las centenas, 3 para las decenas y, por último, 4 (orden bajo).

Veamos el orden de bytes por defecto en MQL5. Para ello, utilizaremos el script *MathSwap.mq5*.

Describe un patrón de concatenación que permite convertir un entero en un array de bytes:

```
template<typename T>
union ByteOverlay
{
   T value;
   uchar bytes[sizeof(T)];
   ByteOverlay(const T v) : value(v) { }
   void operator=(const T v) { value = v; }
};
```

Este código permite dividir visualmente el número en bytes y enumerarlos con índices del array.

En *OnStart* describimos la variable *uint* con el valor 0x12345678 (obsérvese que los dígitos son hexadecimales; en tal notación corresponden exactamente a límites de bytes: cada 2 dígitos es un byte separado). Convirtamos el número en un array y enviémoslo al log.

```
void OnStart()
{
   const uint ui = 0x12345678;
   ByteOverlay<uint> bo(ui);
   ArrayPrint(bo.bytes); // 120  86  52  18 <==> 0x78 0x56 0x34 0x12
   ...
```

La función *ArrayPrint* no puede imprimir números en hexadecimal, por lo que vemos su representación decimal, pero es fácil convertirlos a base 16 y asegurarse de que coinciden con los bytes originales. Visualmente, van en orden inverso: es decir, bajo el índice 0 del array está 0x78, y luego 0x56, 0x34 y 0x12. Obviamente, este orden empieza por el byte menos significativo (de hecho, estamos en el entorno Windows).

Ahora vamos a familiarizarnos con la función *MathSwap*, que MQL5 proporciona para cambiar el orden de los bytes.

integer MathSwap(integer value)

La función devuelve un entero en el que se invierte el orden de los bytes del argumento pasado. La función toma parámetros del tipo *ushort/uint/ulong* (es decir, de 2, 4, 8 bytes de tamaño).

Probemos la función en acción:

```
const uint ui = 0x12345678;
PrintFormat("%I32X -> %I32X", ui, MathSwap(ui));
const ulong ul = 0x0123456789ABCDEF;
PrintFormat("%I64X -> %I64X", ul, MathSwap(ul));
```

He aquí el resultado:

```
12345678 -> 78563412
123456789ABCDEF -> EFCDAB8967452301
```

Intentemos registrar un array de bytes después de convertir el valor 0x12345678 con *MathSwap*:

```
bo = MathSwap(ui);    // put the result of MathSwap into ByteOverlay
ArrayPrint(bo.bytes); //  18  52  86 120 <==> 0x12 0x34 0x56 0x78
```

En un byte con índice 0, donde antes había 0x78, ahora hay 0x12, y en elementos con otros números, los valores también se intercambian.

4.5 Trabajar con archivos

Es difícil encontrar un programa que no utilice entrada-salida de datos. Ya sabemos que los programas MQL pueden recibir ajustes a través de variables de entrada y enviar información al registro, ya que utilizamos este último en casi todos los scripts de prueba. Pero en la mayoría de los casos, esto no es suficiente.

Por ejemplo, una parte bastante significativa de la personalización de programas incluye cantidades de datos que no pueden acomodarse en los parámetros de entrada. Puede ser necesario integrar un programa con algunas herramientas analíticas externas, es decir, cargar información de mercado en un formato estándar o especializado, procesarla y luego cargarla en el terminal de una forma nueva, en concreto, como señales de trading, un conjunto de pesos de redes neuronales o coeficientes de árboles de decisión. Además, puede ser conveniente mantener un registro separado para un programa MQL.

El subsistema de archivos ofrece las posibilidades más universales para este tipo de tareas. La API de MQL5 proporciona una amplia gama de funciones para trabajar con archivos, incluidas funciones para crear, eliminar, buscar, escribir y leer los archivos. Abordaremos todo ello en este capítulo.

Todas las operaciones de archivo en MQL5 se limitan a un área especial en el disco, que se denomina «sandbox». Esto se hace por razones de seguridad, a fin de que ningún programa MQL pueda ser utilizado con fines maliciosos y dañar su ordenador o sistema operativo.

> Los usuarios avanzados pueden evitar esta limitación utilizando medidas especiales, de las que hablaremos más adelante, pero ello sólo debe hacerse en casos excepcionales, adoptando precauciones y asumiendo plena responsabilidad.

Para cada instancia del terminal instalada en el ordenador, el directorio raíz de la «sandbox» se encuentra en *<terminal_data_folder>/MQL5/Files/*. Desde el MetaEditor puede abrir la carpeta de datos utilizando el comando *Archivo -> Abrir carpeta de datos*. Si tiene suficientes derechos de acceso en el ordenador, este directorio suele ser el lugar en el que está instalado el terminal. Si no dispone de los permisos necesarios, la ruta tendrá el siguiente aspecto:

```
X:/Users/<user_name>/AppData/Roaming/MetaQuotes/Terminal/<instance_id>/MQL5/Files/
```

Aquí, *X* es la letra de una unidad en la que está instalado el sistema, *<user_name>* es el nombre de usuario de Windows, e *<instance_id>* es un identificador único de la instancia del terminal. La carpeta *Users* también tiene un alias: «*Documents and Settings*».

> Tenga en cuenta que, en el caso de una conexión remota a un ordenador a través de RDP (Remote Desktop Protocol), el terminal siempre utilizará el directorio *Roaming* y sus subdirectorios, incluso si tiene derechos de administrador.

Recordemos que la carpeta MQL5 del directorio de datos es el lugar en el que se almacenan todos los programas MQL, tanto sus códigos fuente como los archivos ex5 compilados. Cada tipo de programa MQL, incluidos indicadores, Asesores Expertos, scripts y demás, tiene una subcarpeta dedicada en la carpeta MQL5. Así que la carpeta *Files* para los archivos de trabajo está junto a ellos.

Además de esta «sandbox» individual de cada copia del terminal en el ordenador, existe una «sandbox» común y compartida para todos los terminales: pueden comunicarse a través de él. La ruta a la misma pasa por la carpeta de inicio del usuario de Windows y puede variar según la versión del sistema operativo. Por ejemplo, en instalaciones estándar de Windows 7, 8 y 10, tiene este aspecto:

```
X:/Users/<user_name>/AppData/Roaming/MetaQuotes/Terminal/Common/Files/
```

De nuevo, se puede acceder fácilmente a la carpeta a través de MetaTrader: ejecute el comando *Archivo -> Abrir carpeta de datos compartida* y se encontrará dentro de la carpeta Common.

Algunos tipos de programas MQL (Asesores Expertos e indicadores) pueden ejecutarse no sólo en el terminal, sino también en el probador. Cuando se ejecuta en él, la «sandbox» compartida sigue siendo accesible, y en lugar de una «sandbox» de instancia única, se utiliza una carpeta dentro del agente de pruebas. Por regla general, su aspecto es el siguiente:

```
X:/<terminal_path>/Tester/Agent-IP-port/MQL5/Files/
```

Esto puede no ser visible en el propio programa MQL, es decir, todas las funciones de archivo funcionan exactamente de la misma manera. Sin embargo, desde el punto de vista del usuario, puede parecer que hay algún tipo de problema. Por ejemplo, si el programa guarda los resultados de su trabajo en un archivo, éste se borrará en la carpeta del agente del probador una vez finalizada la ejecución (como si el archivo nunca se hubiera creado). Este enfoque rutinario está diseñado para evitar que los datos potencialmente valiosos de un programa se filtren a otro programa que pueda probarse en el mismo agente algún tiempo después (sobre todo porque los agentes pueden compartirse). Se proporcionan otras tecnologías para transferir archivos a los agentes y devolver resultados de los agentes al terminal, que trataremos en la quinta parte del libro.

Para evitar la limitación de la «sandbox» puede utilizar la capacidad de Windows para asignar enlaces simbólicos a objetos del sistema de archivos. En nuestro caso, las conexiones (junction) son las más adecuadas para redirigir el acceso a las carpetas del ordenador local. Se crean utilizando el siguiente comando (es decir, la línea de comandos de Windows):

```
mklink /J new_name existing_target
```

El parámetro *new_name* es el nombre de la nueva carpeta virtual que apuntará a la carpeta real *existing_target*.

Para crear conexiones a carpetas externas fuera de la «sandbox» se recomienda crear una carpeta dedicada dentro de MQL5/Archivos, por ejemplo, *Links*. A continuación, una vez introducido, puede ejecutar el comando anterior seleccionando *new_name* y sustituyendo la ruta real fuera de la «sandbox» por *existing_target*. Por ejemplo, el siguiente comando creará en la carpeta *Links* un nuevo enlace llamado *Settings*, que dará acceso a la carpeta MQL5/Presets:

```
mklink /J Settings "..\..\Presets\"
```

La ruta relativa «..\..\» asume que el comando se ejecuta en la carpeta MQL5/Files/Links especificada. Una combinación de dos puntos «..» indica el paso de la carpeta actual a la carpeta padre. Si se especifica dos veces, esta combinación indica que se debe subir dos veces en la jerarquía de la ruta. Como resultado, la carpeta de destino (*existing_target*) se generará como MQL5/Presets, pero en el parámetro *existing_target* también puede especificar una ruta absoluta.

Puede eliminar enlaces simbólicos como si fueran archivos normales (pero, por supuesto, primero debe asegurarse de que es la carpeta con el icono de la flecha en su esquina inferior izquierda la que se está eliminando, es decir, el enlace, y no la carpeta original). Se recomienda hacer esto inmediatamente, tan pronto como ya no necesite ir más allá de la «sandbox». El hecho es que las carpetas virtuales creadas están disponibles para todos los programas MQL, no sólo para el suyo, y no se sabe cómo los programas de otras personas pueden utilizar la libertad adicional.

Muchas secciones del capítulo tratan de los nombres de los archivos. Actúan como identificadores de elementos del sistema de archivos y tienen reglas similares, incluidas algunas restricciones.

Tenga en cuenta que el nombre del archivo no puede contener algunos caracteres que desempeñan funciones especiales en el sistema de archivos ('<', '>', '/', '\\', '"', ':', '|', '* ', '?'), así como cualquier carácter con códigos del 0 al 31, ambos incluidos.

Los siguientes nombres de archivo también están reservados para un uso especial en el sistema operativo y no se pueden utilizar: CON, PRN, AUX, NUL, COM1, COM2, COM3, COM4, COM5, COM6, COM7, COM8, COM9, LPT1, LPT2, LPT3, LPT4, LPT5, LPT6, LPT7, LPT8, LPT9.

Debe tenerse en cuenta que el sistema de archivos de Windows no ve la diferencia fundamental entre letras en distintos casos, por lo que nombres como «Nombre», «NOMBRE» y «nombre» se refieren al mismo elemento.

Windows permite utilizar tanto barras invertidas '\\' como barras inclinadas '/' como carácter separador entre los componentes de la ruta (subcarpetas y archivos). No obstante, la barra diagonal inversa debe protegerse (es decir, en realidad hay que escribirla dos veces) en las cadenas MQL5, porque el carácter '\' en sí es especial: se utiliza para construir secuencias de caracteres de control, como '\r', '\n', '\t' y otros (véase la sección Tipos de caracteres). Por ejemplo, las siguientes rutas son equivalentes: «MQL5Book/file.txt» y «MQL5Book\file.txt».

El carácter de punto '.' sirve de separador entre el nombre y la extensión. Si un elemento del sistema de archivos tiene varios puntos en su identificador, la extensión es el fragmento situado a la derecha del punto situado más a la derecha, y todo lo situado a la izquierda es el nombre. El título (antes del punto) o la extensión (después del punto) pueden estar vacíos. Por ejemplo, el nombre de archivo sin extensión es «text», y el archivo sin nombre (sólo con la extensión) es «.txt».

> La longitud total de la ruta y el nombre del archivo en Windows tiene limitaciones. Al mismo tiempo, para administrar archivos en MQL5 hay que tener en cuenta que a su ruta y nombre se añadirá la ruta a la «sandbox», es decir, se asignará aún menos espacio para los nombres de los objetos de archivo en las llamadas a funciones MQL. Por defecto, el límite de longitud total es la constante del sistema MAX_PATH, que es igual a 260. A partir de Windows 10 (versión 1607), puede aumentar este límite a 32767. Para ello, debe guardar el siguiente texto en un archivo .reg y ejecutarlo añadiéndolo al Registro de Windows.
>
> ```
> [HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem]
> "LongPathsEnabled"=dword:00000001
> ```
>
> Para otras versiones de Windows puede utilizar soluciones alternativas desde la línea de comandos. En concreto, puede acortar la ruta utilizando las conexiones comentadas anteriormente (creando una carpeta virtual con una ruta corta). También puede utilizar el comando shell *-subst*; por ejemplo, *subst z: c:\very\long\path* (consulte la Ayuda de Windows para obtener más detalles).

4.5.1 Métodos de almacenamiento de la información: texto y binario

Ya hemos visto en muchas secciones anteriores que la misma información puede representarse en forma textual y binaria. Por ejemplo, los números de los formatos *int*, *long* y *double*, la fecha y la hora (*datetime*) y los colores (*color*) se almacenan en la memoria como una secuencia de bytes de una longitud determinada. Este método es compacto y es mejor para la interpretación informática, pero es más cómodo para la persona analizar la información en forma de texto, aunque lleva más tiempo. Por ello, prestamos mucha atención a convertir números en cadenas y viceversa, y a funciones para trabajar con cadenas.

A nivel de archivo también se conserva la división entre la representación binaria y textual de los datos. Un archivo binario está diseñado para almacenar datos en la misma representación interna que se utiliza en la memoria. El archivo de texto contiene una representación de cadena.

Los archivos de texto se utilizan habitualmente para formatos estándar como CSV (Comma Separated Values), JSON (JavaScript Object Notation), XML (Extensible Markup Language) y HTML (HyperText Markup Language).

Los archivos binarios, por supuesto, también tienen formatos estándar para muchas aplicaciones, en particular para imágenes (PNG, GIF, JPG, BMP), sonidos (WAV, MP3) o archivos comprimidos (ZIP). Sin embargo, el formato binario supone inicialmente una mayor protección y un trabajo de bajo nivel con los datos, por lo que se utiliza más a menudo para resolver problemas internos, cuando sólo son importantes la eficiencia del almacenamiento y la disponibilidad de los datos para un programa específico. En otras palabras: los objetos de cualquier estructura y clase aplicada pueden guardar y restaurar fácilmente su estado en un archivo binario, realizando de hecho una impresión de memoria y sin preocuparse por la compatibilidad con ningún estándar.

En teoría, podríamos convertir manualmente los datos en cadenas al escribir en un archivo binario y luego volver a convertirlos de cadenas a números (o estructuras, o arrays) al leer el archivo. Esto sería similar a lo que el modo de archivo de texto proporciona automáticamente, pero requeriría un esfuerzo adicional. El modo de archivo de texto nos ahorra dicha rutina. Además, el subsistema de archivos MQL5 realiza implícitamente varias operaciones opcionales pero importantes que son necesarias cuando se trabaja con texto.

En primer lugar, el concepto de texto se basa en algunas reglas generales de uso de caracteres delimitadores. En concreto, se supone que todos los textos están formados por cadenas. Así es más cómodo leerlos y analizarlos de forma algorítmica. Por lo tanto, hay caracteres especiales que separan una cadena de otra.

Aquí nos encontramos con las primeras dificultades asociadas al hecho de que los distintos sistemas operativos aceptan diferentes combinaciones de estos caracteres. En Windows, el separador de líneas es la secuencia de dos caracteres '\r\n' (ya sea como códigos hexadecimales, 0xD 0xA, o como el nombre CRLF, que significa Carriage Return and Line Feed). En Unix y Linux, el carácter único '\n' es el estándar, pero algunas versiones y programas bajo MacOS pueden utilizar el carácter único '\r'.

Aunque MetaTrader 5 funciona bajo Windows, no tenemos ninguna garantía de que cualquier archivo de texto resultante no se guardará con separadores inusuales. Si tuviéramos que leerlo en modo binario y comprobar nosotros mismos los delimitadores para formar cadenas, estas discrepancias requerirían un tratamiento específico. Aquí viene al rescate el modo de texto de operación de archivos en MQL5: normaliza automáticamente los saltos de línea al leer y escribir.

> Es posible que MQL5 no corrija los saltos de línea en todos los casos. En particular, un único carácter '\r' no se interpretará como '\r\n' al leer un archivo de texto, mientras que un único '\n' se interpreta correctamente como '\r\n'.

En segundo lugar, las cadenas pueden almacenarse en memoria en múltiples representaciones. Por defecto, la cadena (tipo *string*) en MQL5 consta de caracteres de dos bytes. Esto ofrece compatibilidad para la codificación universal Unicode, lo que es bueno porque incluye scripts de todos los países. Sin embargo, en muchos casos no se requiere tal universalidad (por ejemplo, al almacenar números o mensajes en inglés), en cuyo caso es más eficiente utilizar cadenas de caracteres de un solo byte en la codificación ANSI. Las funciones de la API de MQL5 permiten elegir la forma preferida de escribir cadenas en modo texto en los archivos. Pero si controlamos la escritura en nuestro programa MQL, podemos garantizar la validez y fiabilidad del cambio de Unicode a caracteres de un solo byte. En este

caso, cuando se integra con algún servicio web o software externo, la página de código ANSI en sus archivos puede ser cualquiera. A este respecto, se plantea la siguiente cuestión.

En tercer lugar, debido a la presencia de muchas lenguas diferentes, hay que estar preparado para textos en diversas codificaciones ANSI. Sin la interpretación correcta de la codificación, el texto se puede escribir o leer con distorsiones, o incluso volverse ilegible. Lo vimos en la sección Trabajar con símbolos y páginas de códigos. Por ello, las funciones de archivo ya incluyen medios para el correcto tratamiento de los caracteres: basta con especificar en los parámetros la codificación deseada o esperada. La elección de la codificación se describe con más detalle en una sección por separado.

Por último, el modo de texto es compatible con el conocido formato CSV. Dado que el trading suele requerir datos tabulares, CSV es muy adecuado para ello. En un archivo de texto en modo CSV, las funciones de la API de MQL5 procesan no sólo delimitadores para envolver las líneas de texto, sino también un delimitador adicional para el borde de las columnas (campos de cada fila de la tabla). Esto suele ser un tabulador '\t', una coma ',' o un punto y coma ';'. Por ejemplo, este es el aspecto de un archivo CSV con noticias de Forex (se muestra un fragmento separado por comas):

```
Title,Country,Date,Time,Impact,Forecast,Previous
Bank Holiday,JPY,08-09-2021,12:00am,Holiday,,
CPI y/y,CNY,08-09-2021,1:30am,Low,0.8%,1.1%
PPI y/y,CNY,08-09-2021,1:30am,Low,8.6%,8.8%
Unemployment Rate,CHF,08-09-2021,5:45am,Low,3.0%,3.1%
German Trade Balance,EUR,08-09-2021,6:00am,Low,13.9B,12.6B
Sentix Investor Confidence,EUR,08-09-2021,8:30am,Low,29.2,29.8
JOLTS Job Openings,USD,08-09-2021,2:00pm,Medium,9.27M,9.21M
FOMC Member Bostic Speaks,USD,08-09-2021,2:00pm,Medium,,
FOMC Member Barkin Speaks,USD,08-09-2021,4:00pm,Medium,,
BRC Retail Sales Monitor y/y,GBP,08-09-2021,11:01pm,Low,4.9%,6.7%
Current Account,JPY,08-09-2021,11:50pm,Low,1.71T,1.87T
```

Y aquí está, para mayor claridad, en forma de tabla:

Título	País	Fecha	Hora	Impacto	Previsión	Anterior
Día festivo	JPY	08-09-202	12:00 a. m.	Vacaciones		
IPC interanual	CNY	08-09-202	1:30 a. m.	Bajo	0.8 %	1.1 %
IPP interanual	CNY	08-09-202	1:30 a. m.	Bajo	8.6 %	8.8 %
Tasa de desempleo	CHF	08-09-202	5:45 a. m.	Bajo	3.0 %	3.1 %
Balanza comercial alemana	EUR	08-09-202	6:00 a. m.	Bajo	13.9B	12.6B
Índice de confianza del inverso	EUR	08-09-202	8:30 a. m.	Bajo	29.2	29.8
Ofertas de empleo JOLTS	USD	08-09-202	2:00 p. m.	Medio	9.27M	9.21M
Declaraciones de Bostic, miemb	USD	08-09-202	2:00 p. m.	Medio		
Declaraciones de Barkin, miemb	USD	08-09-202	4:00 p. m.	Medio		
BRC Retail Sales Monitor interar	GBP	08-09-202	11:01 p. m.	Bajo	4.9 %	6.7 %
Cuenta corriente	JPY	08-09-202	11:50 p. m.	Bajo	1.71T	1.87T

4.5.2 Escritura y lectura de archivos en modo simplificado

Entre las funciones de archivo de MQL5 destinadas a la escritura y lectura de datos existe una división en dos grupos desiguales. El primero de ellos incluye dos funciones, *FileSave* y *FileLoad*, que permiten escribir o leer datos en modo binario con una sola llamada a una función. Por un lado, este planteamiento tiene una ventaja innegable, la sencillez, pero por otro tiene algunas limitaciones (más adelante hablaremos de ellas). En el segundo gran grupo, todas las funciones de archivo se utilizan de forma diferente: es necesario llamar a varias de ellas de forma secuencial para realizar una operación de lectura o escritura completa desde el punto de vista lógico. Esto parece más complejo, pero proporciona flexibilidad y control sobre el proceso. Las funciones del segundo grupo operan con enteros especiales, es decir, descriptores de archivo, que deben obtenerse utilizando la función *FileOpen* (véase la sección siguiente).

Vamos a ver la descripción formal de estas dos funciones y, a continuación, consideraremos su ejemplo (*FileSaveLoad.mq5*).

bool FileSave(const string filename, const void &data[], const int flag = 0)

La función escribe todos los elementos del array *data* pasado en un archivo binario llamado *filename*. El parámetro *filename* puede contener no sólo el nombre del archivo, sino también los nombres de carpetas de varios niveles de anidamiento: la función creará las carpetas especificadas si aún no existen. Si el archivo existe, se sobrescribirá (a menos que esté ocupado por otro programa).

Como parámetro *data*, se puede pasar un array de cualquier tipo integrado, excepto cadenas. También puede ser un array de estructuras simples que contenga campos de tipos integrados con la excepción de cadenas, arrays dinámicos y punteros. Tampoco se admiten clases.

El parámetro *flag* puede, si es necesario, contener la constante predefinida FILE_COMMON, que significa crear y escribir un archivo en el directorio de datos común de todos los terminales (*Common/Files/*). Si no se especifica la bandera (que corresponde al valor predeterminado de 0), el

archivo se escribe en el directorio de datos habitual (si el programa MQL se ejecuta en el terminal) o en el directorio del agente de pruebas (si sucede en el probador). En los dos últimos casos, se utiliza la «sandbox» *MQL5/Files/* dentro del directorio, tal y como se describe al principio del capítulo.

La función devuelve una indicación de éxito de la operación (*true*) o de error (*false*).

long FileLoad(const string filename, void &data[], const int flag = 0)

La función lee todo el contenido de un archivo binario *filename* en el array *data* especificado. El nombre del archivo puede incluir una jerarquía de carpetas dentro de la «sandbox» *MQL5/Files* o *Common/Files*.

El array *data* debe ser de cualquier tipo integrado excepto *string*, o de un tipo de estructura simple (véase más arriba).

El parámetro *flag* controla la selección del directorio donde se busca y abre el archivo: por defecto (con un valor de 0) es la «sandbox» estándar, pero si se establece el valor FILE_COMMON, entonces es la «sandbox» compartida por todos los terminales.

La función devuelve el número de elementos leídos, o -1 en caso de error.

Tenga en cuenta que los datos del archivo se leen en bloques de un elemento de array. Si el tamaño del archivo no es múltiplo del tamaño del elemento, los datos restantes se omiten (no se leen). Por ejemplo, si el tamaño del archivo es de 10 bytes, al leerlo en un array de tipo *double* (*sizeof(double)=8*) sólo se cargarán realmente 8 bytes, es decir, 1 elemento (y la función devolverá 1). Los 2 bytes restantes al final del archivo serán ignorados.

En el script *FileSaveLoad.mq5* definimos dos estructuras para las pruebas.

```
struct Pair
{
   short x, y;
};

struct Simple
{
   double d;
   int i;
   datetime t;
   color c;
   uchar a[10]; // fixed size array allowed
   bool b;
   Pair p;      // compound fields (nested simple structures) are also allowed

   // strings and dynamic arrays will cause a compilation error when used
   // FileSave/FileLoad: structures or classes containing objects are not allowed
   // string s;
   // uchar a[];

   // pointers are also not supported
   // void *ptr;
};
```

La estructura *Simple* contiene campos de la mayoría de los tipos permitidos, así como un campo compuesto con el tipo de estructura *Pair*. En la función *OnStart* rellenamos un pequeño array del tipo *Simple*.

```
void OnStart()
{
   Simple write[] =
   {
      {+1.0, -1, D'2021.01.01', clrBlue, {'a'}, true, {1000, 16000}},
      {-1.0, -2, D'2021.01.01', clrRed,  {'b'}, true, {1000, 16000}},
   };
   ...
```

Seleccionaremos el archivo para escribir los datos junto con la subcarpeta MQL5Book para que nuestros experimentos no se mezclen con sus archivos de trabajo:

```
   const string filename = "MQL5Book/rawdata";
```

Escribamos un array en un archivo, leámoslo en otro array y comparémoslos.

```
   PRT(FileSave(filename, write/*, FILE_COMMON*/)); // true

   Simple read[];
   PRT(FileLoad(filename, read/*, FILE_COMMON*/)); // 2

   PRT(ArrayCompare(write, read)); // 0
```

FileLoad ha devuelto 2, es decir, se han leído 2 elementos (2 estructuras). Si el resultado de la comparación es 0, significa que los datos coinciden. Puede abrir la carpeta en su administrador de archivos favorito *MQL5/Files/MQL5Book* y asegurarse de que está el archivo 'rawdata' (no se

recomienda ver su contenido usando un editor de texto; sugerimos usar un visor que admita modo binario).

Más adelante en el script convertimos el array de estructuras leída en bytes y los enviamos al registro en forma de códigos hexadecimales. Se trata de una especie de volcado de memoria que permite entender qué son los archivos binarios.

```
uchar bytes[];
for(int i = 0; i < ArraySize(read); ++i)
{
   uchar temp[];
   PRT(StructToCharArray(read[i], temp));
   ArrayCopy(bytes, temp, ArraySize(bytes));
}
ByteArrayPrint(bytes);
```

Resultado:

```
[00] 00 | 00 | 00 | 00 | 00 | 00 | F0 | 3F | FF | FF | FF | FF | 00 | 66 | EE | 5F |
[16] 00 | 00 | 00 | 00 | 00 | 00 | FF | 00 | 61 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
[32] 00 | 00 | 01 | E8 | 03 | 80 | 3E | 00 | 00 | 00 | 00 | 00 | 00 | F0 | BF | FE |
[48] FF | FF | FF | 00 | 66 | EE | 5F | 00 | 00 | 00 | 00 | FF | 00 | 00 | 00 | 62 |
[64] 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 01 | E8 | 03 | 80 | 3E |
```

Debido a que la función integrada *ArrayPrint* no puede imprimir en formato hexadecimal, hemos tenido que desarrollar nuestra propia función *ByteArrayPrint* (aquí no daremos su código fuente, vea el archivo adjunto).

A continuación, recordemos que *FileLoad* es capaz de cargar datos en un array de cualquier tipo, por lo que leeremos el mismo archivo usándolo directamente en un array de bytes.

```
uchar bytes2[];
PRT(FileLoad(filename, bytes2/*, FILE_COMMON*/)); // 78,  39 * 2
PRT(ArrayCompare(bytes, bytes2)); // 0, equality
```

Una comparación satisfactoria de arrays de dos bytes muestra que *FileLoad* puede operar con datos brutos del archivo de forma arbitraria, en la que se le indique (no hay información en el archivo de que almacena un array de estructuras *Simple*).

Es importante señalar aquí que, dado que el tipo byte tiene un tamaño mínimo (1), es múltiplo de cualquier tamaño de archivo. Por lo tanto, cualquier archivo se lee siempre en un array de bytes sin resto. Aquí, la función *FileLoad* ha devuelto el número 78 (el número de elementos es igual al número de bytes). Este es el tamaño del archivo (dos estructuras de 39 bytes cada una).

Básicamente, la capacidad de *FileLoad* para interpretar datos de cualquier tipo requiere atención y comprobaciones por parte del programador. En concreto, más adelante en el script, leemos el mismo archivo en un array de estructuras *MqlDateTime*. Esto, por supuesto, es incorrecto, pero funciona sin errores.

```
   MqlDateTime mdt[];
   PRT(sizeof(MqlDateTime)); // 32
   PRT(FileLoad(filename, mdt)); // 2
// attention: 14 bytes left unread
   ArrayPrint(mdt);
```

El resultado contiene un conjunto de números sin sentido:

```
         [year]      [mon] [day]     [hour]    [min]    [sec] [day_of_week] [day_of_ye
[0]           0 1072693248    -1 1609459200        0 16711680            97
[1] -402587648    4096003     0  -20975616 16777215  6286950     -16777216    1644167
```

Dado que el tamaño de *MqlDateTime* es 32, sólo caben dos estructuras de este tipo en un archivo de 78 bytes, y 14 bytes más resultan superfluos. La presencia de un resto indica que hay un problema. Pero incluso si no lo hay, ello no garantiza el sentido de la operación realizada, ya que dos tamaños diferentes pueden, por pura casualidad, encajar un número entero (pero diferente) de veces a lo largo del archivo. Además, dos estructuras de significado diferente pueden tener el mismo tamaño, pero esto no significa que deban escribirse y leerse de una a otra.

No es sorprendente que el registro del array de estructuras *MqlDateTime* muestre valores extraños, ya que se trataba, de hecho, de un tipo de datos completamente diferente.

Para que la lectura sea algo más cuidadosa, el script implementa un análogo de la función *FileLoad*, *MyFileLoad*. Analizaremos esta función en detalle, así como su par *MyFileSave*, en las secciones siguientes, cuando descubramos nuevas funciones de archivo y las utilicemos para modelar la estructura interna *FileSave/FileLoad*. Mientras tanto, sólo hay que tener en cuenta que en nuestra versión podemos comprobar la presencia de un resto no leído en el archivo y mostrar una advertencia.

Para terminar, veamos un par de errores potenciales más demostrados en el script.

```
   /*
// compilation error, string type not supported here
   string texts[];
   FileSave("any", texts); // parameter conversion not allowed
   */

   double data[];
   PRT(FileLoad("any", data)); // -1
   PRT(_LastError); // 5004, ERR_CANNOT_OPEN_FILE
```

El primero se produce en tiempo de compilación (por eso se comenta el bloque de código) porque no se permiten los arrays de cadenas.

El segundo es leer un archivo inexistente, razón por la cual *FileLoad* devuelve -1. Se puede obtener fácilmente un código de error explicativo utilizando *GetLastError* (o *_LastError*).

4.5.3 Abrir y cerrar archivos

Para escribir y leer datos de un archivo, la mayoría de las funciones de MQL5 requieren que primero se abra el archivo. Para ello existe la función *FileOpen*. Una vez realizadas las operaciones necesarias, el archivo abierto debe cerrarse mediante la función *FileClose*. El hecho es que un archivo abierto puede, dependiendo de las opciones aplicadas, estar bloqueado para el acceso desde otros programas. Además, las operaciones de archivo se almacenan en memoria (caché) por razones de rendimiento, y si no se cierra el archivo, es posible que no se carguen físicamente nuevos datos en él durante algún

tiempo. Esto es especialmente crítico si los datos que se escriben están a la espera de un programa externo (por ejemplo, al integrar un programa MQL con otros sistemas). Descubriremos una forma alternativa de vaciar el búfer en el disco en la descripción de la función *FileFlush*.

Un número entero especial denominado descriptor se asocia con un archivo abierto en un programa MQL. Lo devuelve la función *FileOpen*. Todas las operaciones relacionadas con el acceso o la modificación del contenido interno de un archivo requieren que se especifique este identificador en las funciones de la API correspondientes. Las funciones que operan sobre el archivo completo (copiar, borrar, mover, comprobar existencia) no necesitan descriptor. No es necesario abrir el archivo para realizar estos pasos.

```
int FileOpen(const string filename, int flags, const short delimiter = '\t', uint codepage = CP_ACP)
int FileOpen(const string filename, int flags, const string delimiter, uint codepage = CP_ACP)
```

La función abre un archivo con el nombre especificado, en el modo especificado por el parámetro *flags*. El parámetro *filename* puede contener subcarpetas delante del nombre real del archivo. En este caso, si el archivo se abre para escritura y la jerarquía de carpetas requerida aún no existe, se creará.

El parámetro *flags* debe contener una combinación de constantes que describan el modo requerido de trabajar con el archivo. La combinación se realiza mediante las operaciones de OR a nivel de bits. A continuación se muestra una tabla con las constantes disponibles.

Identificador	Valor	Descripción
FILE_READ	1	El archivo se abre para lectura
FILE_WRITE	2	El archivo se abre para escritura
FILE_BIN	4	Modo binario de lectura-escritura, sin conversión de datos de cadena a cadena
FILE_CSV	8	Archivo de tipo CSV; los datos que se escriben se convierten en texto del tipo apropiado (Unicode o ANSI, véase más abajo) y, al leerlos, se realiza la conversión inversa del texto al tipo requerido (especificado en la función de lectura); un registro CSV es una sola línea de texto, delimitada por caracteres de nueva línea (normalmente CRLF); dentro del registro CSV, los elementos están separados por un carácter delimitador (parámetro *delimiter*);
FILE_TXT	16	Archivo de texto sin formato, similar al modo CSV, pero no se utiliza carácter delimitador (se ignora el valor del parámetro *delimiter*).
FILE_ANSI	32	Cadenas de tipo ANSI (caracteres de un byte)
FILE_UNICODE	64	Cadenas de tipo Unicode (caracteres de doble byte)
FILE_SHARE_READ	128	Acceso de lectura compartido desde varios programas
FILE_SHARE_WRITE	256	Acceso de escritura compartido por varios programas
FILE_REWRITE	512	Permiso para sobrescribir un archivo (si ya existe) en funciones FileCopy y FileMove
FILE_COMMON	4096	Ubicación del archivo en la carpeta compartida de todos los terminales de cliente /Terminal/Common/Files (la bandera se utiliza al abrir archivos (FileOpen), copiar archivos (FileCopy, FileMove) y comprobar la existencia de archivos (FileIsExist))

Al abrir un archivo, debe especificarse una de las banderas FILE_WRITE, FILE_READ o su combinación.

Las banderas FILE_SHARE_READ y FILE_SHARE_WRITE no sustituyen ni anulan la necesidad de especificar las banderas FILE_READ y FILE_WRITE.

> El entorno de ejecución del programa MQL siempre almacena en búfer los archivos para su lectura, lo que equivale a añadir implícitamente la bandera FILE_READ. Debido a esto, FILE_SHARE_READ debe utilizarse siempre para trabajar correctamente con archivos compartidos (incluso si se sabe que otro proceso tiene abierto un archivo de sólo escritura).

Si no se especifica ninguna de las banderas FILE_CSV, FILE_BIN, FILE_TXT, se asume FILE_CSV como la prioridad más alta. Si se especifica más de una de estas tres banderas, se aplica la de mayor prioridad pasada (se enumeran más arriba en orden descendente de prioridad).

Para los archivos de texto, el modo por defecto es FILE_UNICODE.

El parámetro *delimiter* que sólo afecta a CSV puede ser del tipo *ushort* o *string*. En el segundo caso, si la longitud de la cadena es superior a 1, sólo se utilizará su primer carácter.

El parámetro *codepage* sólo afecta a los archivos abiertos en modo texto (FILE_TXT o FILE_CSV), y sólo si se selecciona el modo FILE_ANSI para las cadenas. Si las cadenas se almacenan en Unicode (FILE_UNICODE), la página de códigos no es importante.

Si tiene éxito, la función devuelve un descriptor de archivo, un número entero positivo. Es único sólo dentro de un programa MQL en particular; no tiene sentido compartirlo con otros programas. Para seguir trabajando con el archivo, el descriptor se pasa a las llamadas a otras funciones.

En caso de error, el resultado es INVALID_HANDLE (-1). La esencia del error debe aclararse a partir del código devuelto por la función *GetLastError*.

Todos los ajustes del modo de funcionamiento realizados en el momento de abrir el archivo permanecen inalterados mientras el archivo esté abierto. Si es necesario cambiar el modo, hay que cerrar el archivo y volver a abrirlo con los nuevos parámetros.

Para cada archivo abierto, el entorno de ejecución del programa MQL mantiene un puntero interno, es decir, la posición actual dentro del archivo. Inmediatamente después de abrir el archivo, el puntero se sitúa al principio (posición 0). En el proceso de escritura o lectura, la posición se desplaza adecuadamente, según la cantidad de datos transmitidos o recibidos de varias funciones de archivo. También es posible influir directamente en la posición (retroceder o avanzar). Todas estas oportunidades se analizarán en las secciones siguientes.

FILE_READ y FILE_WRITE en varias combinaciones le permiten lograr varios escenarios:

- FILE_READ - abrir un archivo sólo si existe; en caso contrario, la función devuelve un error y no se crea ningún archivo nuevo.
- FILE_WRITE - crear un nuevo archivo si no existe ya, o abrir un archivo existente; su contenido se borra y el tamaño se pone a cero.
- FILE_READ|FILE_WRITE - abrir un archivo existente con todo su contenido o crear un nuevo archivo si aún no existe.

Como puede ver, algunos escenarios son inaccesibles únicamente debido a las banderas. En concreto, no se puede abrir un archivo para escribir sólo si ya existe. Para ello se pueden utilizar funciones adicionales, como por ejemplo *FileIsExist*. Además, no será posible restablecer «automáticamente» un archivo abierto para una combinación de lectura y escritura: en este caso, MQL5 siempre deja el contenido.

Para añadir datos a un archivo, no sólo hay que abrir el archivo en modo FILE_READ|FILE_WRITE, sino también desplazar la posición actual dentro del archivo hasta el final llamando a *FileSeek*.

La descripción correcta del acceso compartido al archivo es un requisito previo para la correcta ejecución de *File Open*. Este aspecto se gestiona del siguiente modo:

- Si no se especifica ninguna de las banderas FILE_SHARE_READ y FILE_SHARE_WRITE, el programa actual obtiene acceso exclusivo al archivo si lo abre primero. Si el mismo archivo ya ha sido abierto antes por alguien (por otro programa o por el mismo programa), la llamada a la función fallará.
- Cuando la bandera FILE_SHARE_READ está activada, el programa permite peticiones posteriores para abrir el mismo archivo para lectura. Si en el momento de la llamada a la función el archivo ya está abierto para su lectura por otro programa o por el mismo, y esta bandera no está activada, la función fallará.

- Cuando la bandera FILE_SHARE_WRITE está activada, el programa permite peticiones posteriores para abrir el mismo archivo para escritura. Si en el momento de la llamada a la función el archivo ya ha sido abierto para escritura por otro programa o por el mismo y esta bandera no está activada, la función fallará.

El acceso compartido se comprueba no sólo en relación con otros programas MQL o procesos externos a MetaTrader 5, sino también en relación con el mismo programa MQL si vuelve a abrir el archivo.

Por lo tanto, el modo menos conflictivo implica que se especifiquen ambas banderas, pero sigue sin garantizar que el archivo se abra si a alguien ya se le ha emitido un descriptor a él sin compartir. No obstante, deben seguirse normas más estrictas en función de las lecturas o escrituras previstas.

Por ejemplo, al abrir un archivo para su lectura, tiene sentido dejar la posibilidad de que otros lo lean. Además, probablemente pueda permitir que otros escriban en él, si se trata de un archivo que se está reabasteciendo (por ejemplo, un diario). Sin embargo, cuando se abra un archivo para escritura, no merece la pena dejar el acceso de escritura a otros: esto llevaría a una superposición de datos impredecible.

void FileClose(int handle)

La función cierra un archivo previamente abierto por su manejador.

Una vez cerrado el archivo, su manejador en el programa deja de ser válido: si se intenta llamar a cualquier función de archivo sobre él, se producirá un error. No obstante, puede utilizar la misma variable para almacenar un manejador diferente si vuelve a abrir el mismo archivo u otro diferente.

Cuando el programa termina, los archivos abiertos se cierran de manera forzosa y el búfer de escritura, si no está vacío, se escribe en el disco. No obstante, se recomienda cerrar los archivos explícitamente.

Cerrar un archivo cuando haya terminado de trabajar con él es una regla importante que se debe seguir. Esto se debe no sólo al almacenamiento en caché de la información que se escribe, que puede permanecer en la RAM durante algún tiempo y no guardarse en el disco (como ya se ha mencionado anteriormente) si no se cierra el archivo. Además, un archivo abierto consume recursos internos del sistema operativo, y no hablamos de espacio en disco. El número de archivos abiertos simultáneamente es limitado (quizá varios cientos o miles, dependiendo de la configuración de Windows). Si muchos programas mantienen un gran número de archivos abiertos, dicho límite puede alcanzarse y los intentos de abrir nuevos archivos fallarán.

A este respecto es deseable protegerse de la posible pérdida de descriptores utilizando una clase envoltorio que abriría un archivo y recibiría un descriptor al crear un objeto, y el descriptor se liberaría y el archivo se cerraría automáticamente en el destructor.

Crearemos una clase envoltorio después de probar las funciones puras *FileOpen* y *FileClose*.

Pero antes de profundizar en los detalles del archivo, preparemos una nueva versión de la macro para ilustrar una salida de nuestras funciones al registro de llamadas. La nueva versión era necesaria porque, hasta ahora, macros como PRT y PRTS (utilizadas en secciones anteriores) «absorbían» los valores de retorno de las funciones durante la impresión. Por ejemplo, escribimos:

```
PRT(FileLoad(filename, read));
```

En este caso, el resultado de la llamada a *FileLoad* se envía al registro, pero no es posible obtenerlo en la cadena de código de llamada. A decir verdad, no lo necesitábamos. Pero ahora la función *FileOpen* devolverá un descriptor de archivo y deberá ser almacenado en una variable para posteriores manipulaciones del mismo.

Hay dos problemas con las macros antiguas: en primer lugar, se basan en la función *Print*, que consume los datos pasados (enviándolos al registro) pero no devuelve nada por sí misma. En segundo lugar, cualquier valor de una variable con resultado sólo puede obtenerse de una expresión, y una llamada a *Print* no puede formar parte de una expresión debido a que tiene el tipo *void*.

Para resolver estos problemas, necesitamos una función auxiliar de impresión que devuelva un valor imprimible. Y empaquetaremos su llamada en una nueva macro PRTF:

```
#include <MQL5Book/MqlError.mqh>

#define PRTF(A) ResultPrint(#A, (A))

template<typename T>
T ResultPrint(const string s, const T retval = 0)
{
   const string err = E2S(_LastError) + "(" + (string)_LastError + ")";
   Print(s, "=", retval, " / ", (_LastError == 0 ? "ok" : err));
 ResetLastError();// clear the error flag for the next call
   return retval;
}
```

Utilizando el operador mágico de conversión de cadenas '#', obtenemos un descriptor detallado del fragmento de código (expresión A) que se pasa como primer argumento a *ResultPrint*. La expresión en sí (el argumento de la macro) se evalúa (si existe una función, se llama a ella) y su resultado se pasa como segundo argumento a *ResultPrint*. A continuación entra en juego la función Print habitual y, por último, se devuelve el mismo resultado al código de llamada.

Para no buscar en la Ayuda la decodificación de los códigos de error, se ha preparado una macro E2S que utiliza la enumeración MQL_ERROR con todos los errores MQL5. Se encuentra en el archivo de encabezado *MQL5/Include/MQL5Book/MqlError.mqh*. La nueva macro y la función *ResultPrint* se definen en el archivo PRTF.mqh, junto a los scripts de prueba.

En el script *FileOpenClose.mq5* vamos a intentar abrir diferentes archivos y, en particular, el mismo archivo se abrirá varias veces en paralelo. Esto suele evitarse en los programas reales. Para la mayoría de las tareas basta con un único manejador de un archivo concreto en una instancia del programa.

Uno de los archivos, *MQL5Book/rawdata*, debe existir ya puesto que fue creado por un script de la sección Escritura y lectura de archivos en modo simplificado. Se creará otro archivo durante la prueba.

Elegiremos el tipo de archivo FILE_BIN. Trabajar con FILE_TXT o FILE_CSV sería similar en esta fase.

Reservemos un array para los descriptores de archivo de forma que al final del script cerremos todos los archivos a la vez.

En primer lugar, abramos *MQL5Book/rawdata* en modo lectura sin acceso compartido. Suponiendo que el archivo no lo está usando ninguna aplicación de terceros, podemos esperar que el manejador se reciba correctamente.

```
void OnStart()
{
   int ha[4] = {}; // array for test file handles

   // this file must exist after running FileSaveLoad.mq5
   const string rawdata = "MQL5Book/rawdata";
   ha[0] = PRTF(FileOpen(rawdata, FILE_BIN | FILE_READ)); // 1 / ok
```

Si intentamos abrir de nuevo el mismo archivo, nos encontraremos con un error porque ni la primera ni la segunda llamada permiten compartir.

```
   ha[1] = PRTF(FileOpen(rawdata, FILE_BIN | FILE_READ)); // -1 / CANNOT_OPEN_FILE(5004
```

Vamos a cerrar el primer manejador, a abrir de nuevo el archivo, pero con permisos de lectura compartida, y nos aseguraremos de que la reapertura ahora funciona (aunque también necesita permitir la lectura compartida):

```
   FileClose(ha[0]);
   ha[0] = PRTF(FileOpen(rawdata, FILE_BIN | FILE_READ | FILE_SHARE_READ)); // 1 / ok
   ha[1] = PRTF(FileOpen(rawdata, FILE_BIN | FILE_READ | FILE_SHARE_READ)); // 2 / ok
```

Abrir un archivo para escribir (FILE_WRITE) no funcionará, porque las dos llamadas anteriores de *FileOpen* sólo permiten FILE_SHARE_READ.

```
   ha[2] = PRTF(FileOpen(rawdata, FILE_BIN | FILE_READ | FILE_WRITE | FILE_SHARE_READ
   // -1 / CANNOT_OPEN_FILE(5004)
```

Ahora vamos a intentar crear un nuevo archivo *MQL5Book/newdata*. Si lo abre como sólo lectura, no se creará el archivo.

```
   const string newdata = "MQL5Book/newdata";
   ha[3] = PRTF(FileOpen(newdata, FILE_BIN | FILE_READ));
   // -1 / CANNOT_OPEN_FILE(5004)
```

Para crear un archivo, debe especificar el modo FILE_WRITE (la presencia de FILE_READ no es crítica aquí, pero hace que la llamada sea más universal: como recordamos, en esta combinación, la instrucción garantiza que o bien se abrirá el archivo antiguo, si existe, o bien se creará uno nuevo).

```
   ha[3] = PRTF(FileOpen(newdata, FILE_BIN | FILE_READ | FILE_WRITE)); // 3 / ok
```

Vamos a intentar escribir algo en un nuevo archivo utilizando la función *FileSave* ya conocido. Actúa como un «reproductor externo», ya que trabaja con el archivo saltándose nuestro descriptor, de forma muy similar a como podría hacerlo otro programa MQL o una aplicación de terceros.

```
   long x[1] = {0x123456789ABCDEF0};
   PRTF(FileSave(newdata, x)); // false
```

Esta llamada falla porque el manejador se abrió sin permisos de compartición. Cierre y vuelva a abrir el archivo con el máximo de «permisos».

```
   FileClose(ha[3]);
   ha[3] = PRTF(FileOpen(newdata,
      FILE_BIN | FILE_READ | FILE_WRITE | FILE_SHARE_READ | FILE_SHARE_WRITE)); // 3
```

Esta vez *FileSave* funciona conforme a lo esperado.

```
   PRTF(FileSave(newdata, x)); // true
```

Puede buscar en la carpeta *MQL5/Files/MQL5Book/* y encontrar allí el archivo *newdata*, de 8 bytes de longitud.

Tenga en cuenta que después de cerrar el archivo, su descriptor se devuelve a la reserva de descriptores libres, y la próxima vez que se abra un archivo (tal vez otro archivo), el mismo número vuelve a entrar en juego.

Para un cierre ordenado, cerraremos explícitamente todos los archivos abiertos.

```
   for(int i = 0; i < ArraySize(ha); ++i)
   {
      if(ha[i] != INVALID_HANDLE)
      {
        FileClose(ha[i]);
      }
   }
}
```

4.5.4 Gestión de descriptores de archivo

Dado que necesitamos recordar constantemente los archivos abiertos y liberar los descriptores locales al salir de las funciones, sería eficiente confiar toda la rutina a objetos especiales.

Este enfoque es bien conocido en programación y se denomina RAII (Resource Acquisition Is Initialization, por sus siglas en inglés). El uso de RAII facilita el control de los recursos y garantiza que se encuentran en el estado correcto. En concreto, esto es especialmente efectivo si la función que abre el archivo (y crea un objeto propietario para él) sale desde varios lugares diferentes.

El ámbito de aplicación de RAII no se limita a los archivos. En la sección Plantillas de tipos de objeto hemos creado la clase *AutoPtr*, que administra un puntero a un objeto. Este era otro ejemplo de este concepto, ya que un puntero es también un recurso (memoria) y es muy fácil perderlo, además de que consume muchos recursos liberarlo en varias ramas diferentes del algoritmo.

Una clase envolvente de archivos también puede ser útil de otra manera. La API de archivos no proporciona una función que permita obtener el nombre de un archivo mediante un descriptor (a pesar de que esa relación existe sin duda internamente). Al mismo tiempo, dentro del objeto, podemos almacenar este nombre e implementar nuestro propio enlace al descriptor.

En el caso más simple necesitamos alguna clase que almacene un descriptor de archivo y lo cierre automáticamente en el destructor. En el archivo *FileHandle.mqh* se muestra un ejemplo de aplicación.

```
class FileHandle
{
   int handle;
public:
   FileHandle(const int h = INVALID_HANDLE) : handle(h)
   {
   }

   FileHandle(int &holder, const int h) : handle(h)
   {
      holder = h;
   }

   int operator=(const int h)
   {
      handle = h;
      return h;
   }
   ...
```

Dos constructores, así como un operador de asignación sobrecargado, garantizan que un objeto esté vinculado a un archivo (descriptor). El segundo constructor permite pasar una referencia a una variable local (del código de llamada), que además obtendrá un nuevo descriptor. Será una especie de alias externo para el mismo descriptor, que podrá utilizarse de la forma habitual en otras llamadas a funciones.

Pero también se puede prescindir de un alias. Para estos casos, la clase define el operador '~', que devuelve el valor de la variable interna *handle*.

```
   int operator~() const
   {
      return handle;
   }
```

Por último, lo más importante para lo que se implementó la clase es el destructor inteligente:

```
~FileHandle()
{
   if(handle != INVALID_HANDLE)
   {
      ResetLastError();
      // will set internal error code if handle is invalid
      FileGetInteger(handle, FILE_SIZE);
      if(_LastError == 0)
      {
         #ifdef FILE_DEBUG_PRINT
            Print(__FUNCTION__, ": Automatic close for handle: ", handle);
         #endif
         FileClose(handle);
      }
      else
      {
         PrintFormat("%s: handle %d is incorrect, %s(%d)",
            __FUNCTION__, handle, E2S(_LastError), _LastError);
      }
   }
}
```

En él, tras varias comprobaciones, se llama a *FileClose* para la variable controlada *handle*. La cuestión es que el archivo puede cerrarse explícitamente en otra parte del programa, aunque esto ya no es necesario con esta clase. Como resultado, el descriptor puede dejar de ser válido en el momento en que se llama al destructor cuando la ejecución del algoritmo abandona el bloque en el que está definido el objeto *FileHandle*. Para averiguarlo, se realiza una llamada ficticia a la función *FileGetInteger*. Es ficticia porque no hace nada útil. Si el código de error interno sigue siendo 0 después de la llamada, el descriptor es válido.

Podemos omitir todas estas comprobaciones y escribir simplemente lo siguiente:

```
~FileHandle()
{
   if(handle != INVALID_HANDLE)
   {
      FileClose(handle);
   }
}
```

Si el descriptor está dañado, *FileClose* no devolverá ninguna advertencia. No obstante, hemos añadido comprobaciones para poder emitir información de diagnóstico.

Vamos a probar la clase *FileHandle* en acción. El script de prueba se llama *FileHandle.mq5*.

```
const string dummy = "MQL5Book/dummy";

void OnStart()
{
   // creating a new file or open an existing one and reset it
   FileHandle fh1(PRTF(FileOpen(dummy,
      FILE_TXT | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ))); // 1
   // another way to connect the descriptor via '='
   int h = PRTF(FileOpen(dummy,
      FILE_TXT | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ)); // 2
   FileHandle fh2 = h;
   // and another supported syntax:
   // int f;
   // FileHandle ff(f, FileOpen(dummy,
   //    FILE_TXT | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ));

   // data is supposed to be written here
   // ...

   // close the file manually (this is not necessary; only done to demonstrate
   // that the FileHandle will detect this and won't try to close it again)
   FileClose(~fh1); // operator '~' applied to an object returns a handle

   // descriptor handle in variable 'h' bound to object 'fh2' is not manually closed
   // and will be automatically closed in the destructor
}
```

Según la salida en el registro, todo funciona según lo previsto:

```
FileHandle::~FileHandle: Automatic close for handle: 2
FileHandle::~FileHandle: handle 1 is incorrect, INVALID_FILEHANDLE(5007)
```

Sin embargo, si hay muchos archivos, crear una copia del objeto de seguimiento para cada uno de ellos puede convertirse en un inconveniente. Para estas situaciones tiene sentido diseñar un único objeto que recoja todos los descriptores en un contexto determinado (por ejemplo, dentro de una función).

Dicha clase se implementa en el archivo *FileHolder.mqh* y se muestra en el script *FileHolder.mq5*. Una copia del propio *FileHolder* crea a petición objetos observadores auxiliares de la clase *FileOpener*, que comparte características comunes con *FileHandle*, especialmente el destructor, así como el campo *handle*.

Para abrir un archivo a través de *FileHolder* debe utilizar su método *FileOpen* (su firma repite la firma de la función estándar *FileOpen*).

```
class FileHolder
{
   static FileOpener *files[];
   int expand()
   {
      return ArrayResize(files, ArraySize(files) + 1) - 1;
   }
public:
   int FileOpen(const string filename, const int flags,
                const ushort delimiter = '\t', const uint codepage = CP_ACP)
   {
      const int n = expand();
      if(n > -1)
      {
         files[n] = new FileOpener(filename, flags, delimiter, codepage);
         return files[n].handle;
      }
      return INVALID_HANDLE;
   }
```

Todos los objetos de *FileOpener* se suman en el array *files* para hacer un seguimiento de su vida útil. En el mismo lugar, los elementos cero marcan los momentos de registro de los contextos locales (bloques de código) en los que se crean los objetos *FileHolder*. El constructor *FileHolder* se encarga de ello.

```
   FileHolder()
   {
      const int n = expand();
      if(n > -1)
      {
         files[n] = NULL;
      }
   }
```

Como sabemos, durante la ejecución de un programa, éste entra en bloques de código anidados (llama a funciones). Si requieren la administración de descriptores de archivo locales, los objetos *FileHolder* (uno por bloque o menos) deben describirse allí. Según las reglas de la pila (primero en entrar, último en salir), todas esas descripciones se suman en *files* y luego se liberan en orden inverso a medida que el programa abandona los contextos. El destructor es llamado en cada uno de estos momentos.

```
~FileHolder()
{
   for(int i = ArraySize(files) - 1; i >= 0; --i)
   {
      if(files[i] == NULL)
      {
         // decrement array and exit
         ArrayResize(files, i);
         return;
      }

      delete files[i];
   }
}
```

Su tarea consiste en eliminar los últimos objetos *FileOpener* del array hasta el primer elemento cero encontrado, lo cual indica el límite del contexto (más adelante en el array hay descriptores de otro contexto externo).

Puede estudiar toda la clase por su cuenta.

Veamos su uso en el script de prueba *FileHolder.mq5*. Además de la función *OnStart*, dispone de *SubFunc*. Las operaciones con archivos se realizan en ambos contextos.

```
const string dummy = "MQL5Book/dummy";

void SubFunc()
{
   Print(__FUNCTION__, " enter");
   FileHolder holder;
   int h = PRTF(holder.FileOpen(dummy,
      FILE_BIN | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ));
   int f = PRTF(holder.FileOpen(dummy,
      FILE_BIN | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ));
   // use h and f
   // ...
   // no need to manually close files and track early function exits
   Print(__FUNCTION__, " exit");
}

void OnStart()
{
   Print(__FUNCTION__, " enter");

   FileHolder holder;
   int h = PRTF(holder.FileOpen(dummy,
      FILE_BIN | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ));
   // writing data and other actions on the file by descriptor
   // ...
   /*
   int a[] = {1, 2, 3};
   FileWriteArray(h, a);
   */

   SubFunc();
   SubFunc();

 if(rand() >32000) // simulate branching by conditions
   {
      // thanks to the holder we don't need an explicit call
      // FileClose(h);
      Print(__FUNCTION__, " return");
      return; // there can be many exits from the function
   }

   /*
     ... more code
   */

   // thanks to the holder we don't need an explicit call
   // FileClose(h);
   Print(__FUNCTION__, " exit");
}
```

No hemos cerrado ningún manejador manualmente, las instancias de *FileHolder* lo harán automáticamente en los destructores.

He aquí un ejemplo de salida de registro:

```
OnStart enter
holder.FileOpen(dummy,FILE_BIN|FILE_WRITE|FILE_SHARE_WRITE|FILE_SHARE_READ)=1 / ok
SubFunc enter
holder.FileOpen(dummy,FILE_BIN|FILE_WRITE|FILE_SHARE_WRITE|FILE_SHARE_READ)=2 / ok
holder.FileOpen(dummy,FILE_BIN|FILE_WRITE|FILE_SHARE_WRITE|FILE_SHARE_READ)=3 / ok
SubFunc exit
FileOpener::~FileOpener: Automatic close for handle: 3
FileOpener::~FileOpener: Automatic close for handle: 2
SubFunc enter
holder.FileOpen(dummy,FILE_BIN|FILE_WRITE|FILE_SHARE_WRITE|FILE_SHARE_READ)=2 / ok
holder.FileOpen(dummy,FILE_BIN|FILE_WRITE|FILE_SHARE_WRITE|FILE_SHARE_READ)=3 / ok
SubFunc exit
FileOpener::~FileOpener: Automatic close for handle: 3
FileOpener::~FileOpener: Automatic close for handle: 2
OnStart exit
FileOpener::~FileOpener: Automatic close for handle: 1
```

4.5.5 Seleccionar una codificación para el modo texto

En el caso de los archivos de texto escrito, la codificación debe elegirse en función de las características del texto o ajustarse a los requisitos de los programas externos a los que van destinados los archivos generados. Si no existen requisitos externos, puede seguir la regla de utilizar siempre ANSI para textos sin formato con números, letras inglesas y signos de puntuación (se ofrece una tabla de esos 128 caracteres internacionales en la sección Comparación de cadenas). Cuando trabaje con varios idiomas o caracteres especiales, utilice UTF-8 o Unicode, respectivamente:

```
int u8 = FileOpen("utf8.txt", FILE_WRITE | FILE_TXT | FILE_ANSI, 0, CP_UTF8);
int u0 = FileOpen("unicode.txt", FILE_WRITE | FILE_TXT | FILE_UNICODE);
```

Por ejemplo, estos ajustes son útiles para guardar los nombres de los instrumentos financieros en un archivo, ya que a veces utilizan caracteres especiales que denotan divisas o modos de trading.

La lectura de sus propios archivos no debería ser un problema, ya que basta con especificar al leer la misma configuración de codificación que al escribir. No obstante, los archivos de texto pueden proceder de distintas fuentes. Su codificación puede ser desconocida o estar sujeta a cambios sin previo aviso. Por tanto, se plantea la cuestión de qué hacer si algunos de los archivos pueden suministrarse como cadenas de un solo byte (ANSI), otros como cadenas de dos bytes (Unicode) y otros como codificación UTF-8.

La codificación puede seleccionarse mediante los parámetros de entrada del programa. Sin embargo, esto sólo es efectivo para un archivo, y si tiene que abrir muchos archivos diferentes, es posible que sus codificaciones no coincidan. Por lo tanto, es conveniente dar instrucciones al sistema para que elija el modelo correcto sobre la marcha (de archivo a archivo).

MQL5 no permite la detección 100 % automática y la aplicación de codificaciones correctas; sin embargo, hay un modo más universal para leer una variedad de archivos de texto. Para ello, necesita configurar los siguientes parámetros de entrada de la función *FileOpen*:

```
int h = FileOpen(filename, FILE_READ | FILE_TXT | FILE_ANSI, 0, CP_UTF8);
```

Hay varios factores en juego.

En primer lugar, la codificación UTF-8 omite de forma transparente los mencionados 128 caracteres en cualquier codificación ANSI (es decir, se transmiten «uno a uno»).

En segundo lugar, es el más popular para protocolos de Internet.

En tercer lugar, MQL5 tiene un análisis adicional integrado para el formato de texto en Unicode de dos bytes, que le permite cambiar automáticamente el modo de operación de archivo a FILE_UNICODE, si es necesario, con independencia de los parámetros especificados. Y es que los archivos en formato Unicode suelen ir precedidos de un par especial de identificadores: 0xFFFE, o viceversa, 0xFEFF. Esta secuencia se denomina Byte Order Mark (BOM). Se necesita porque, como sabemos, los bytes pueden almacenarse dentro de números en un orden diferente en distintas plataformas (esto se abordó en la sección Control de la codificación endian de números enteros).

El formato FILE_UNICODE utiliza un entero de 2 bytes (código) por carácter, por lo que el orden de los bytes adquiere importancia, a diferencia de otras codificaciones. El orden de bytes BOM de Windows es 0xFFFE. Si el núcleo MQL5 encuentra esta etiqueta al principio de un archivo de texto, su lectura cambiará automáticamente al modo Unicode.

Veamos cómo funcionan las distintas configuraciones de modo con archivos de texto de distintas codificaciones. Para ello, utilizaremos el script *FileText.mq5* y varios archivos de texto con el mismo contenido, pero en diferentes codificaciones (entre paréntesis se indica el tamaño en bytes):

- ansi1252.txt (50): codificación europea 1252 (se mostrará completa sin distorsión en Windows con el idioma europeo)
- unicode1.txt (102): Unicode de dos bytes, al principio está la lista de materiales (BOM) inherente a Windows 0xFFFE
- unicode2.txt (100): Unicode de dos bytes sin lista de materiales BOM (en general, la lista de materiales es opcional)
- unicode3.txt (102): Unicode de dos bytes, al principio hay BOM inherente a Unix, 0xFEFF
- utf8.txt (54): codificación UTF-8

En la función *OnStart* leeremos estos archivos en bucles con diferentes ajustes de *FileOpen*. Tenga en cuenta que al utilizar *FileHandle* (revisado en la sección anterior) no tenemos que preocuparnos de cerrar archivos: todo ocurre automáticamente dentro de cada iteración.

```
void OnStart()
{
   Print("=====> UTF-8");
   for(int i = 0; i < ArraySize(texts); ++i)
   {
      FileHandle fh(FileOpen(texts[i], FILE_READ | FILE_TXT | FILE_ANSI, 0, CP_UTF8))
      Print(texts[i], " -> ", FileReadString(~fh));
   }

   Print("=====> Unicode");
   for(int i = 0; i < ArraySize(texts); ++i)
   {
      FileHandle fh(FileOpen(texts[i], FILE_READ | FILE_TXT | FILE_UNICODE));
      Print(texts[i], " -> ", FileReadString(~fh));
   }

   Print("=====> ANSI/1252");
   for(int i = 0; i < ArraySize(texts); ++i)
   {
      FileHandle fh(FileOpen(texts[i], FILE_READ | FILE_TXT | FILE_ANSI, 0, 1252));
      Print(texts[i], " -> ", FileReadString(~fh));
   }
}
```

La función *FileReadString* lee una cadena de un archivo. Lo abordaremos en la sección sobre escritura y lectura de variables.

A continuación se muestra un registro de ejemplo con los resultados de la ejecución del script:

```
=====> UTF-8
MQL5Book/ansi1252.txt -> This is a text with special characters: ?? / ? / ?
MQL5Book/unicode1.txt -> This is a text with special characters: ±Σ / £ / ¥
MQL5Book/unicode2.txt -> T
MQL5Book/unicode3.txt -> ??
MQL5Book/utf8.txt -> This is a text with special characters: ±Σ / £ / ¥
=====> Unicode
MQL5Book/ansi1252.txt -> 桔獵槧嘔整瑱眠瑩□夊捥慬档牡捡整獲›癈士⇟士艹
MQL5Book/unicode1.txt -> This is a text with special characters: ±Σ / £ / ¥
MQL5Book/unicode2.txt -> This is a text with special characters: ±Σ / £ / ¥
MQL5Book/unicode3.txt -> 吁栀椀狒 椀狒 ? 玲攀砀玲 明椀玲栀 狒濩攀捩椀？甦 捩栀？
MQL5Book/utf8.txt -> 桔 ? 槧 整 瑱 眠 瑩 □ 夊 捥 慬 档 牡 捡 整 獲 ?㱡⓪ ± 㑊 ? ⟴
=====> ANSI/1252
MQL5Book/ansi1252.txt -> This is a text with special characters: ±? / £ / ¥
MQL5Book/unicode1.txt -> This is a text with special characters: ±Σ / £ / ¥
MQL5Book/unicode2.txt -> T
MQL5Book/unicode3.txt -> þÿ
MQL5Book/utf8.txt -> This is a text with special characters: Â±Î£ / Â£ / Â¥
```

El archivo unicode1.txt siempre se lee correctamente porque tiene BOM 0xFFFE, y el sistema ignora la configuración del código fuente. No obstante, si falta la etiqueta o es big-endian, esta autodetección no funciona. Además, al establecer FILE_UNICODE, perdemos la capacidad de leer textos de un solo byte y UTF-8.

Como resultado, la combinación mencionada de FILE_ANSI y CP_UTF8 debería considerarse más resistente a las variaciones en el formato. Sólo se recomienda seleccionar una página de código nacional específica cuando se requiera explícitamente.

A pesar de la importante ayuda que proporciona la API al programador cuando trabaja con archivos en modo texto, podemos, si es necesario, evitar el modo FILE_TXT o FILE_CSV y abrir un archivo de texto en modo binario FILE_BINARY. Esto hará que toda la complejidad de analizar el texto y determinar la codificación recaiga sobre los hombros del programador, pero le permitirá admitir otros formatos no estándar. No obstante, lo más importante es que se puede leer y escribir texto en un archivo abierto en modo binario. Sin embargo, lo contrario, en el caso general, es imposible. Un archivo binario con datos arbitrarios (es decir, que no contenga exclusivamente cadenas) abierto en modo texto se interpretará muy probablemente como un «galimatías» de texto. Si necesita escribir datos binarios en un archivo de texto, utilice primero la función *CryptEncode* y la codificación CRYPT_BASE64.

4.5.6 Escritura y lectura de arrays

Dos funciones MQL5 sirven para escribir y leer arrays: *FileWriteArray* y *FileReadArray*. Con los archivos binarios, permiten manejar arrays de cualquier tipo integrado que no sean cadenas, así como arrays de estructuras simples que no contengan campos de cadena, objetos, punteros y arrays dinámicos. Estas limitaciones están relacionadas con la optimización de los procesos de escritura y lectura, lo cual es posible gracias a la exclusión de tipos con longitudes variables. Las cadenas, los objetos y los arrays dinámicos son exactamente así.

Al mismo tiempo, cuando se trabaja con archivos de texto, estas funciones son capaces de operar sobre arrays del tipo *string* (otros tipos de arrays en archivos con modo FILE_TXT/FILE_CSV no están permitidos por estas funciones). Estos arrays se almacenan en un archivo con el siguiente formato: un elemento por línea.

Si necesita almacenar estructuras o clases sin restricciones de tipo en un archivo, utilice funciones específicas de tipo que procesen un valor por llamada. Se describen en dos secciones sobre escritura y lectura de variables de tipos integrados: para archivos binarios y de texto.

Además, el soporte para estructuras con cadenas puede organizarse mediante la optimización interna del almacenamiento de información. Por ejemplo, en lugar de campos de cadena, puede utilizar campos de número entero, que contendrán los índices de las cadenas correspondientes en un array independiente con cadenas. Dada la posibilidad de redefinir muchas operaciones (en particular, la asignación) utilizando herramientas de programación orientada a objetos y de obtener un elemento estructural de un array por número, el aspecto del algoritmo prácticamente no cambiará. Pero al escribir, puede abrir primero un archivo en modo binario y llamar a *FileWriteArray* para obtener un array con un tipo de estructura simplificada y, a continuación, volver a abrir el archivo en modo texto y añadirle un array con todas las cadenas usando la segunda llamada a *FileWriteArray*. Para leer un archivo de este tipo, debe proporcionar un encabezado al principio del mismo que contenga el número de elementos de los arrays para poder pasarlo como el parámetro *count* a *FileReadArray* (véase más adelante).

Si necesita guardar o leer, no un array de estructuras, sino una única estructura, utilice las funciones *FileWriteStruct* y *FileReadStruct*, que se describen en la sección siguiente.

Vamos a estudiar las firmas de las funciones y, a continuación, consideraremos un ejemplo general (*FileArray.mq5*).

uint FileWriteArray(int handle, const void &array[], int start = 0, int count = WHOLE_ARRAY)

La función escribe *array array* en un archivo con el descriptor *handle*. El array puede ser multidimensional. Los parámetros *start* y *count* permiten establecer el rango de elementos; por defecto, es igual a todo el array. En el caso de arrays multidimensionales, el índice *start* y el número de elementos *count* se refieren a la numeración continua en todas las dimensiones, no a la primera dimensión del array. Por ejemplo, si el array tiene la configuración [][5], entonces el valor *start* igual a 7 apuntará al elemento con índices [1][2], y *count* = *2* le añadirá el elemento [1][3].

La función devuelve el número de elementos escritos. En caso de error, será 0.

Si *handle* se recibe en modo binario, los arrays pueden ser de cualquier tipo integrado excepto cadenas, o tipos de estructura simple. Si *handle* se abre en cualquiera de los modos de texto, el array debe ser del tipo *string*.

uint FileReadArray(int handle, const void &array[], int start = 0, int count = WHOLE_ARRAY)

La función lee datos de un archivo con el descriptor *handle* en un array. El array puede ser multidimensional y dinámico. Para arrays multidimensionales, los parámetros *start* y *count* funcionan sobre la base de la numeración continua de elementos en todas las dimensiones, descrita anteriormente. Un array dinámico, si es necesario, aumenta automáticamente de tamaño para ajustarse a los datos que se leen. Si *start* es mayor que la longitud original del array, estos elementos intermedios contendrán datos aleatorios tras la asignación de memoria (véase el ejemplo).

> Tenga en cuenta que la función no puede controlar si la configuración del array utilizado al escribir el archivo coincide con la configuración del array receptor al leer. Básicamente, no hay ninguna garantía de que el archivo que se está leyendo se haya escrito con *FileWriteArray*.
>
> Para comprobar la validez de la estructura de datos se suelen utilizar algunos formatos predefinidos de encabezados iniciales u otros descriptores dentro de los archivos. Las funciones en sí leerán cualquier contenido del archivo dentro de su tamaño y lo colocarán en el array especificado.

Si *handle* se recibe en modo binario, los arrays pueden ser cualquiera de los tipos integrados que no sean cadenas o tipos de estructura simple. Si *handle* se abre en modo texto, el array debe ser del tipo *string*.

Vamos a comprobar el trabajo tanto en modo binario como de texto utilizando el script *FileArray.mq5*. Para ello, reservaremos dos nombres de archivo.

```
const string raw = "MQL5Book/array.raw";
const string txt = "MQL5Book/array.txt";
```

En la función *OnStart* se describen tres arrays de tipo *long* y dos arrays de tipo *string*. Sólo se rellena con datos el primer array de cada tipo, y todos los demás se comprobarán para su lectura una vez que se hayan escrito los archivos.

```
void OnStart()
{
   long numbers1[][2] = {{1, 4}, {2, 5}, {3, 6}};
   long numbers2[][2];
   long numbers3[][2];

   string text1[][2] = {{"1.0", "abc"}, {"2.0", "def"}, {"3.0", "ghi"}};
   string text2[][2];
   ...
```

Además, para probar las operaciones con estructuras, se definen los tres tipos siguientes:

```
struct TT
{
   string s1;
   string s2;
};

struct B
{
private:
   int b;
public:
   void setB(const int v) { b = v; }
};

struct XYZ : public B
{
   color x, y, z;
};
```

No podremos utilizar una estructura del tipo *TT* en las funciones descritas porque contiene campos de cadena. Esto es necesario para demostrar un posible error de compilación en una sentencia comentada (véase más adelante). La herencia entre las estructuras *B* y *XYZ*, así como la presencia de un campo cerrado, no son un obstáculo para las funciones *FileWriteArray* y *FileReadArray*.

Las estructuras se utilizan para declarar un par de arrays:

```
TTtt[]; // empty, because data is not important
  XYZ xyz[1];
  xyz[0].setB(-1);
  xyz[0].x = xyz[0].y = xyz[0].z = clrRed;
```

Empecemos por el modo binario. Vamos a crear un nuevo archivo o a abrir uno ya existente, volcando su contenido. A continuación, en tres llamadas a *FileWriteArray*, intentaremos escribir tres arrays: *numbers1*, *text1* y *xyz*.

```
int writer = PRTF(FileOpen(raw, FILE_BIN | FILE_WRITE)); // 1 / ok
PRTF(FileWriteArray(writer, numbers1)); // 6 / ok
PRTF(FileWriteArray(writer, text1)); // 0 / FILE_NOTTXT(5012)
PRTF(FileWriteArray(writer, xyz)); // 1 / ok
FileClose(writer);
ArrayPrint(numbers1);
```

Los arrays *numbers1* y *xyz* se escriben correctamente, como indica el número de elementos escritos. El array *text1* falla con un error FILE_NOTTXT(5012) porque los arrays de cadenas requieren que el archivo se abra en modo texto. Por lo tanto, el contenido *xyz* se ubicará en el archivo inmediatamente después de todos los elementos de *numbers1*.

> Tenga en cuenta que cada función de escritura (o lectura) comienza escribiendo (o leyendo) datos en la posición actual dentro del archivo, y la desplaza en el tamaño de los datos escritos o leídos. Si este puntero se encuentra al final del archivo antes de la operación de escritura, se incrementa el tamaño del archivo. Si se alcanza el final del archivo durante la lectura, el puntero deja de moverse y el sistema emite un código de error interno especial 5027 (FILE_ENDOFFILE). En un nuevo archivo de tamaño cero, el principio y el final son iguales.

Desde un array *text1* se escribieron 0 elementos, por lo que nada en el archivo le recuerda que entre dos llamadas *FileWriteArray* correctas se produjo un fallo.

En el script de prueba mostramos simplemente el resultado de la función y el estado (código de error) en el registro, pero en un programa real, debemos analizar los problemas sobre la marcha y tomar algunas medidas: arreglar algo en los parámetros, en la configuración del archivo, o interrumpir el proceso con un mensaje al usuario.

Vamos a leer un archivo en el array *numbers2*.

```
int reader = PRTF(FileOpen(raw, FILE_BIN | FILE_READ)); // 1 / ok
PRTF(FileReadArray(reader, numbers2)); // 8 / ok
ArrayPrint(numbers2);
```

Dado que se han escrito dos arrays diferentes en el archivo (no sólo *numbers1*, sino también *xyz*), se han leído 8 elementos en el array receptor (es decir, todo el archivo hasta el final, porque no se especificó lo contrario mediante parámetros).

De hecho, el tamaño de la estructura *XYZ* es de 16 bytes (4 campos de 4 bytes: un *int* y tres *color*), que corresponde a una fila del array *numbers2* (2 elementos de tipo *long*). En este caso se trata de una coincidencia. Como ya se ha indicado, las funciones no tienen ni idea de la configuración y el tamaño de los datos en bruto y pueden leer cualquier cosa en cualquier array: el programador debe controlar la validez de la operación.

Vamos a comprobar los estados inicial y recibido. Array fuente *numbers1*:

```
     [,0][,1]
[0,]    1    4
[1,]    2    5
[2,]    3    6
```

Array resultante *numbers2*:

```
               [,0]          [,1]
[0,]              1             4
[1,]              2             5
[2,]              3             6
[3,] 1099511627775 1095216660735
```

El comienzo del array *numbers2* coincide completamente con el array original *numbers1*, es decir, la escritura y la lectura a través del archivo funcionan correctamente.

La última fila está ocupada en su totalidad por una única estructura *XYZ* (con valores correctos, pero representación incorrecta como dos números del tipo *long*).

Ahora llegamos al principio del archivo (utilizando la función *FileSeek*, de la que hablaremos más adelante en la sección Control de posición dentro de un archivo) y llamamos a *FileReadArray* indicando el número y la cantidad de elementos, es decir, realizamos una lectura parcial.

```
PRTF(FileSeek(reader, 0, SEEK_SET)); // true
PRTF(FileReadArray(reader, numbers3, 10, 3));
FileClose(reader);
ArrayPrint(numbers3);
```

Se leen tres elementos del archivo y se colocan, empezando por el índice 10, en el array receptor *numbers3*. Como el archivo se lee desde el principio, estos elementos son los valores 1, 4, 2. Y como un array bidimensional tiene la configuración [][2], el índice pasante 10 apunta al elemento [5,0]. Este es el aspecto que tiene en la memoria:

```
     [,0][,1]
[0,]    1   4
[1,]    1   4
[2,]    2   6
[3,]    0   0
[4,]    0   0
[5,]    1   4
[6,]    2   0
```

Los elementos marcados en amarillo son aleatorios (pueden cambiar en diferentes ejecuciones de script). Es posible que todos sean cero, pero no está garantizado. El array *numbers3* estaba inicialmente vacío y la llamada a *FileReadArray* inició una asignación de memoria necesaria para recibir 3 elementos en el offset 10 (13 en total). El bloque seleccionado no se rellena con nada, y sólo se leen 3 números del archivo. Por lo tanto, los elementos con índices de paso de 0 a 9 (es decir, las 5 primeras filas), así como el último, con índice 13, contienen basura.

Los arrays multidimensionales se escalan a lo largo de la primera dimensión y, por lo tanto, un aumento de 1 número significa añadir toda la configuración a lo largo de dimensiones superiores. En este caso, la distribución se refiere a una serie de dos números ([][2]). En otras palabras: el tamaño solicitado, 13, se redondea a un múltiplo de dos, es decir, 14.

Por último, vamos a probar cómo funcionan las funciones con arrays de cadenas. Vamos a crear un nuevo archivo o a abrir uno ya existente, volcando su contenido. A continuación, en dos llamadas a *FileWriteArray*, escribiremos los arrays *text1* y *numbers1*.

```
writer = PRTF(FileOpen(txt, FILE_TXT | FILE_ANSI | FILE_WRITE)); // 1 / ok
PRTF(FileWriteArray(writer, text1)); // 6 / ok
PRTF(FileWriteArray(writer, numbers1)); // 0 / FILE_NOTBIN(5011)
FileClose(writer);
```

El array de cadenas se guarda correctamente. El array numérico se ignora con un error FILE_NOTBIN(5011) porque debe abrir el archivo en modo binario.

Al intentar escribir un array de estructuras *tt*, obtenemos un error de compilación con un largo mensaje de «no se permiten estructuras o clases con objetos». Lo que el compilador quiere decir en realidad es que no le gustan los campos como *string* (se supone que las cadenas y los arrays dinámicos tienen una representación interna de algunos objetos de servicio). Así, a pesar de que el archivo se abre en modo texto y sólo hay campos de texto en la estructura, esta combinación no es compatible con MQL5.

```
// COMPILATION ERROR: structures or classes containing objects are not allowed
FileWriteArray(writer, tt);
```

La presencia de campos de cadena hace que la estructura sea «complicada» e inadecuada para trabajar con funciones *FileWriteArray/FileReadArray* en cualquier modo.

Después de ejecutar el script, puede cambiar al directorio *MQL5/Files/MQL5Book* y examinar el contenido de los archivos generados.

Anteriormente, en la sección Escritura y lectura de archivos en modo simplificado, hemos hablado de las funciones *FileSave* y *FileLoad*. En el script de prueba (*FileSaveLoad.mq5*) hemos implementado las versiones equivalentes de estas funciones utilizando *FileWriteArray* y *FileReadArray*. Pero no las hemos visto en detalle. Como ya estamos familiarizados con estas nuevas funciones, podemos examinar el código fuente:

```
template<typename T>
bool MyFileSave(const string name, const T &array[], const int flags = 0)
{
   const int h = FileOpen(name, FILE_BIN | FILE_WRITE | flags);
   if(h == INVALID_HANDLE) return false;
   FileWriteArray(h, array);
   FileClose(h);
   return true;
}

template<typename T>
long MyFileLoad(const string name, T &array[], const int flags = 0)
{
   const int h = FileOpen(name, FILE_BIN | FILE_READ | flags);
   if(h == INVALID_HANDLE) return -1;
   const uint n = FileReadArray(h, array, 0, (int)(FileSize(h) / sizeof(T)));
   // this version has the following check added compared to the standard FileLoad:
   // if the file size is not a multiple of the structure size, print a warning
   const ulong leftover = FileSize(h) - FileTell(h);
   if(leftover != 0)
   {
      PrintFormat("Warning from %s: Some data left unread: %d bytes",
         __FUNCTION__, leftover);
      SetUserError((ushort)leftover);
   }
   FileClose(h);
   return n;
}
```

MyFileSave se construye sobre una única llamada *FileWriteArray*, y *MyFileLoad* sobre una llamada *FileReadArray*, entre un par de llamadas *FileOpen/FileClose*. En ambos casos, se escriben y se leen todos los datos disponibles. Gracias a las plantillas, nuestras funciones también pueden aceptar arrays de tipos arbitrarios. Pero si algún tipo no soportado (por ejemplo, una clase) se deduce como metaparámetro T, se producirá un error de compilación, como ocurre con el acceso incorrecto a funciones integradas.

4.5.7 Escritura y lectura de estructuras (archivos binarios)

En la sección anterior aprendimos a realizar operaciones de E/S en arrays de estructuras. Cuando la lectura o escritura está relacionada con una estructura independiente, es más conveniente utilizar el par de funciones *FileWriteStruct* y *FileReadStruct*.

uint FileWriteStruct(int handle, const void &data, int size = -1)

La función escribe el contenido de una estructura simple *data* en un archivo binario con el descriptor *handle*. Como sabemos, estas estructuras sólo pueden contener campos de tipos de no cadena integrados y estructuras simples anidadas.

La característica principal de la función es el parámetro *size*, que ayuda a establecer el número de bytes que se van a escribir, lo que nos permite descartar alguna parte de la estructura (su final). Por defecto, el parámetro es -1, lo que significa que se guarda toda la estructura. Si *size* es mayor que el tamaño de la estructura, se ignora el exceso, es decir, sólo se escribe la estructura, *sizeof(data)* bytes.

En caso de éxito, la función devuelve el número de bytes escritos; en caso de error devuelve 0.

uint FileReadStruct(int handle, void &data, int size = -1)

La función lee el contenido de un archivo binario con el descriptor *handle* en la estructura *data*. El parámetro *size* especifica el número de bytes que se van a leer. Si no se especifica el tamaño de la estructura o éste se supera, se utiliza el tamaño exacto de la estructura especificada.

En caso de éxito, la función devuelve el número de bytes leídos; en caso de error devuelve 0.

La opción de cortar el final de la estructura sólo está presente en las funciones *FileWriteStruct* y *FileReadStruct*. Por lo tanto, su uso en un bucle se convierte en la alternativa más adecuada para guardar y leer un array de estructuras recortadas: las funciones *FileWriteArray* y *FileReadArray* no tienen esta capacidad, y la escritura y lectura por campos individuales puede consumir más recursos (veremos las funciones correspondientes en los siguientes apartados).

Hay que tener en cuenta que, para poder utilizar esta función, debe diseñar sus estructuras de forma que todos los campos de cálculo temporales e intermedios que no deban guardarse se encuentren al final de la estructura.

Veamos ejemplos de uso de estas dos funciones en el script *FileStruct.mq5*.

Supongamos que queremos archivar las últimas cotizaciones de vez en cuando, para poder comprobar su invariabilidad en el futuro o comparar con periodos similares de otros proveedores. Básicamente, esto se puede hacer manualmente a través del cuadro de diálogo Símbolos (en la pestaña Barras) en MetaTrader 5. Pero esto requeriría un esfuerzo adicional y cumplir un calendario. Es mucho más fácil hacerlo automáticamente, desde el programa. Además, la exportación manual de las cotizaciones se realiza en formato de texto CSV, y es posible que tengamos que enviar los archivos a un servidor externo. Por lo tanto, es deseable guardarlos en una forma binaria compacta. Además, supongamos que no nos interesa la información sobre ticks, diferencial y volúmenes reales (que siempre están vacíos para los símbolos Forex).

En la sección Comparar, ordenar y buscar en arrays, consideramos la estructura *MqlRates* y la función *CopyRates*. Se describirán detalladamente más adelante, mientras que ahora las utilizaremos una vez más como campo de pruebas para las operaciones con archivos.

Utilizando el parámetro *size* en *FileWriteStruct* podemos guardar sólo una parte de la estructura *MqlRates*, sin los últimos campos.

Al principio del script definimos las macros y el nombre del archivo de prueba.

```
#define BARLIMIT 10 // number of bars to write
#define HEADSIZE 10 // size of the header of our format
const string filename = "MQL5Book/struct.raw";
```

De particular interés es la constante HEADSIZE. Como se ha mencionado anteriormente, las funciones de archivo como tales no son responsables de la coherencia de los datos en el archivo, y de los tipos de estructuras en las que se leen estos datos. El programador debe proporcionar dicho control en su código. Por lo tanto, al principio del archivo se suele escribir un determinado encabezado, con ayuda del cual puede, en primer lugar, asegurarse de que se trata de un archivo con el formato requerido y, en segundo lugar, guardar en él la metainformación necesaria para su correcta lectura.

En concreto, el título puede indicar el número de entradas. En sentido estricto, esto último no siempre es necesario, ya que podemos leer el archivo gradualmente hasta que termine. No obstante, es más

eficiente asignar memoria para todos los registros esperados a la vez, basándose en el contador del encabezado.

Para nuestros fines, hemos desarrollado una estructura sencilla *FileHeader*.

```
struct FileHeader
{
   uchar signature[HEADSIZE];
   int n;
   FileHeader(const int size = 0) : n(size)
   {
      static uchar s[HEADSIZE] = {'C','A','N','D','L','E','S','1','.','0'};
      ArrayCopy(signature, s);
   }
};
```

Comienza con la firma de texto «CANDLES» (en el campo *signature*), el número de versión «1.0» (en el mismo lugar) y el número de entradas (en el campo *n*). Como no podemos utilizar un campo de cadena para la firma (entonces la estructura dejaría de ser simple y cumpliría los requisitos de las funciones de archivo), el texto se empaqueta en realidad en el array *uchar* del tamaño fijo HEADSIZE. Su inicialización en la instancia la realiza el constructor basándose en la copia estática local.

En la función *OnStart* solicitamos el BARLIMIT de las últimas barras, abrimos el archivo en modo FILE_WRITE y escribimos en el archivo el encabezado seguido de las cotizaciones resultantes de forma truncada.

```
void OnStart()
{
   MqlRates rates[], candles[];
   int n = PRTF(CopyRates(_Symbol, _Period, 0, BARLIMIT, rates)); // 10 / ok
   if(n < 1) return;

   // create a new file or overwrite the old one from scratch
   int handle = PRTF(FileOpen(filename, FILE_BIN | FILE_WRITE)); // 1 / ok

  FileHeaderfh(n);// header with the actual number of entries

   // first write the header
   PRTF(FileWriteStruct(handle, fh)); // 14 / ok

   // then write the data
   for(int i = 0; i < n; ++i)
   {
      FileWriteStruct(handle, rates[i], offsetof(MqlRates, tick_volume));
   }
   FileClose(handle);
   ArrayPrint(rates);
   ...
```

Como valor del parámetro *size* en la función *FileWriteStruct* utilizamos una expresión con el ya conocido operador *offsetof*: *offsetof(MqlRates, tick_volume)* es decir, todos los campos que empiezan por *tick_volume* se descartan al escribir en el archivo.

Para probar la lectura de datos, vamos a abrir el mismo archivo en modo FILE_READ y a leer la estructura *FileHeader*.

```
   handle = PRTF(FileOpen(filename, FILE_BIN | FILE_READ)); // 1 / ok
   FileHeader reference, reader;
   PRTF(FileReadStruct(handle, reader)); // 14 / ok
   // if the headers don't match, it's not our data
   if(ArrayCompare(reader.signature, reference.signature))
   {
      Print("Wrong file format; 'CANDLES' header is missing");
      return;
   }
```

La estructura *reference* contiene el encabezado por defecto sin cambios (firma). La estructura *reader* obtuvo 14 bytes del archivo. Si las dos firmas coinciden, podemos seguir trabajando, ya que el formato del archivo ha resultado ser correcto y el campo *reader.n* contiene el número de entradas leídas del archivo. Asignamos y ponemos a cero el tamaño de memoria necesario para el array receptor *candles* y, a continuación, leemos en él todas las entradas.

```
   PrintFormat("Reading %d candles...", reader.n);
 ArrayResize(candles, reader.n);// allocate memory for the expected data in advance
   ZeroMemory(candles);

   for(int i = 0; i < reader.n; ++i)
   {
      FileReadStruct(handle, candles[i], offsetof(MqlRates, tick_volume));
   }
   FileClose(handle);
   ArrayPrint(candles);
}
```

La puesta a cero era necesaria porque las estructuras *MqlRates* se leen parcialmente, y los campos restantes contendrían basura sin la puesta a cero.

Este es el registro que muestra los datos iniciales (en conjunto) para XAUUSD,H1:

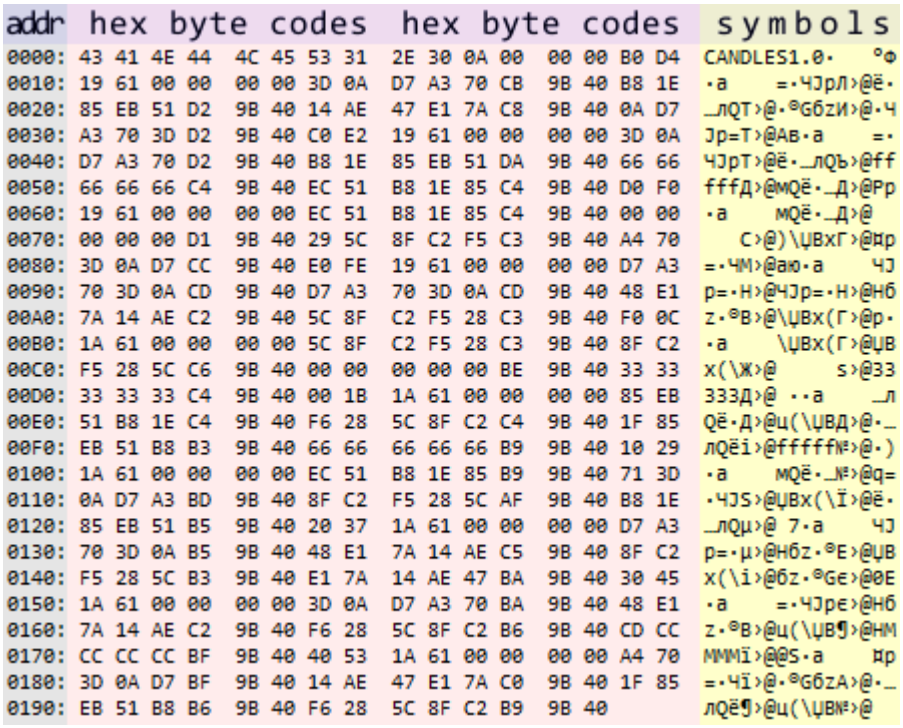
```
[hora] [apertura] [máximo] [mínimo] [cierre] [volumen_tic] [diferencial] [volumen_rea
[0] 2021.08.16 03:00:00 1778.86 1780.58 1778.12 1780.56 3049 5 0
[1] 2021.08.16 04:00:00 1780.61 1782.58 1777.10 1777.13 4633 5 0
[2] 2021.08.16 05:00:00 1777.13 1780.25 1776.99 1779.21 3592 5 0
[3] 2021.08.16 06:00:00 1779.26 1779.26 1776.67 1776.79 2535 5 0
[4] 2021.08.16 07:00:00 1776.79 1777.59 1775.50 1777.05 2052 6 0
[5] 2021.08.16 08:00:00 1777.03 1777.19 1772.93 1774.35 3213 5 0
[6] 2021.08.16 09:00:00 1774.38 1775.41 1771.84 1773.33 4527 5 0
[7] 2021.08.16 10:00:00 1773.26 1777.42 1772.84 1774.57 4514 5 0
[8] 2021.08.16 11:00:00 1774.61 1776.67 1773.69 1775.95 3500 5 0
[9] 2021.08.16 12:00:00 1775.96 1776.12 1773.68 1774.44 2425 5 0
```

Veamos ahora qué se ha leído.

```
[hora] [apertura] [máximo] [mínimo] [cierre] [volumen_tic] [diferencial] [volumen_rea
[0] 2021.08.16 03:00:00 1778.86 1780.58 1778.12 1780.56 0 0 0
[1] 2021.08.16 04:00:00 1780.61 1782.58 1777.10 1777.13 0 0 0
[2] 2021.08.16 05:00:00 1777.13 1780.25 1776.99 1779.21 0 0 0
[3] 2021.08.16 06:00:00 1779.26 1779.26 1776.67 1776.79 0 0 0
[4] 2021.08.16 07:00:00 1776.79 1777.59 1775.50 1777.05 0 0 0
[5] 2021.08.16 08:00:00 1777.03 1777.19 1772.93 1774.35 0 0 0
[6] 2021.08.16 09:00:00 1774.38 1775.41 1771.84 1773.33 0 0 0
[7] 2021.08.16 10:00:00 1773.26 1777.42 1772.84 1774.57 0 0 0
[8] 2021.08.16 11:00:00 1774.61 1776.67 1773.69 1775.95 0 0 0
[9] 2021.08.16 12:00:00 1775.96 1776.12 1773.68 1774.44 0 0 0
```

Las cotizaciones coinciden, pero los tres últimos campos de cada estructura están vacíos.

Puede abrir la carpeta *MQL5/Files/MQL5Book* y examinar la representación interna del archivo *struct.raw* (utilice un visor que admita el modo binario; a continuación se muestra un ejemplo).



```
addr  hex byte codes  hex byte codes  symbols
0000: 43 41 4E 44  4C 45 53 31  2E 30 0A 00  00 00 B0 D4  CANDLES1.0·   °Ф
0010: 19 61 00 00  00 00 3D 0A  D7 A3 70 CB  9B 40 B8 1E  ·a    =·ЧJрЛ›@ё·
0020: 85 EB 51 D2  9B 40 14 AE  47 E1 7A C8  9B 40 0A D7  …лQT›@·®GбzИ›@·Ч
0030: A3 70 3D D2  9B 40 C0 E2  19 61 00 00  00 00 3D 0A  Jp=T›@Ав·a    =·
0040: D7 A3 70 D2  9B 40 B8 1E  85 EB 51 DA  9B 40 66 66  ЧJpT›@ё·…лQЬ›@ff
0050: 66 66 66 C4  9B 40 EC 51  B8 1E 85 C4  9B 40 D0 F0  fffД›@мQё·…Д›@Рр
0060: 19 61 00 00  00 00 EC 51  B8 1E 85 C4  9B 40 00 00  ·a    мQё·…Д›@
0070: 00 00 00 D1  9B 40 29 5C  8F C2 F5 C3  9B 40 A4 70     С›@)\ЏВхГ›@¤р
0080: 3D 0A D7 CC  9B 40 E0 FE  19 61 00 00  00 00 D7 A3  =·ЧМ›@аю·a    ЧJ
0090: 70 3D 0A CD  9B 40 D7 A3  70 3D 0A CD  9B 40 48 E1  p=·Н›@ЧJp=·Н›@Нб
00A0: 7A 14 AE C2  9B 40 5C 8F  C2 F5 28 C3  9B 40 F0 0C  z·®В›@\ЏВх(Г›@р·
00B0: 1A 61 00 00  00 00 5C 8F  C2 F5 28 C3  9B 40 8F C2  ·a    \ЏВх(Г›@ЏВ
00C0: F5 28 5C C6  9B 40 00 00  00 00 00 BE  9B 40 33 33  х(\Ж›@     s›@33
00D0: 33 33 33 C4  9B 40 00 1B  1A 61 00 00  00 00 85 EB  333Д›@ ··a    …л
00E0: 51 B8 1E C4  9B 40 F6 28  5C 8F C2 C4  9B 40 1F 85  Qё·Д›@ц(\ЏВД›@·…
00F0: EB 51 B8 B3  9B 40 66 66  66 66 66 B9  9B 40 10 29  лQёі›@fffff№›@·)
0100: 1A 61 00 00  00 00 EC 51  B8 1E 85 B9  9B 40 71 3D  ·a    мQё·…№›@q=
0110: 0A D7 A3 BD  9B 40 8F C2  F5 28 5C AF  9B 40 B8 1E  ·ЧJS›@ЏВх(\Ї›@ё·
0120: 85 EB 51 B5  9B 40 20 37  1A 61 00 00  00 00 D7 A3  …лQµ›@ 7·a    ЧJ
0130: 70 3D 0A B5  9B 40 48 E1  7A 14 AE C5  9B 40 8F C2  p=·µ›@Нбz·®Е›@ЏВ
0140: F5 28 5C B3  9B 40 E1 7A  14 AE 47 BA  9B 40 30 45  х(\і›@бz·®Gє›@0Е
0150: 1A 61 00 00  00 00 3D 0A  D7 A3 70 BA  9B 40 48 E1  ·a    =·ЧJpє›@Нб
0160: 7A 14 AE C2  9B 40 F6 28  5C 8F C2 B6  9B 40 CD CC  z·®В›@ц(\ЏВ¶›@НМ
0170: CC CC CC BF  9B 40 40 53  1A 61 00 00  00 00 A4 70  МММї›@@S·a    ¤р
0180: 3D 0A D7 BF  9B 40 14 AE  47 E1 7A C0  9B 40 1F 85  =·Чї›@·®GбzА›@·…
0190: EB 51 B8 B6  9B 40 F6 28  5C 8F C2 B9  9B 40        лQё¶›@ц(\ЏВ№›@
```

Opciones para presentar un archivo binario con cotizaciones en un visor externo

Esta es una forma típica de visualizar archivos binarios: la columna de la izquierda muestra las direcciones (desplazamientos desde el principio del archivo), los códigos de bytes están en la columna central y las representaciones simbólicas de los bytes correspondientes se muestran en la columna de la derecha. La primera y la segunda columna utilizan la notación hexadecimal para los números. Los caracteres de la columna derecha pueden variar en función de la página de códigos ANSI seleccionada. Tiene sentido prestarles atención sólo en aquellos fragmentos en los que se conoce la presencia de texto. En nuestro caso, la firma «CANDLES1.0» se «manifiesta» claramente al principio. Los números deben ser analizados por la columna central. En esta columna, por ejemplo, después de la firma, se puede ver el valor de 4 bytes 0x0A000000, es decir, 0x0000000A de forma invertida (recuerde la sección Control de la codificación endian de números enteros): es 10, el número de estructuras escritas.

4.5.8 Escritura y lectura de variables (archivos binarios)

Si una estructura contiene campos de tipos prohibidos para las estructuras simples (cadenas, arrays dinámicos, punteros), no será posible escribirla en un archivo ni leerla de un archivo mediante las funciones consideradas anteriormente. Lo mismo ocurre con los objetos de clase. Sin embargo, estas entidades suelen contener la mayor parte de los datos de los programas y también requieren guardar y restaurar su estado.

Con el ejemplo de la estructura de encabezado de la sección anterior se ha demostrado claramente que las cadenas (y otros tipos de longitud variable) pueden evitarse, pero en este caso hay que inventar implementaciones alternativas y más engorrosas de los algoritmos (por ejemplo, sustituir una cadena por un array de caracteres).

Para escribir y leer datos de complejidad arbitraria, MQL5 proporciona conjuntos de funciones de nivel inferior que operan sobre un único valor de un tipo determinado: *double*, *float*, *int/uint*, *long/ulong* o *string*. Todos los demás tipos integrados de MQL5 son equivalentes a enteros de diferentes tamaños: *char/uchar* es de 1 byte, *short/ushort* es de 2 bytes, *color* es de 4 bytes, las enumeraciones son de 4 bytes y *datetime* es de 8 bytes. Estas funciones pueden denominarse atómicas (es decir, indivisibles), porque las funciones de lectura y escritura en archivos a nivel de bits ya no existen.

Por supuesto, la escritura o lectura elemento por elemento también elimina la restricción de las operaciones de archivo con arrays dinámicos.

En cuanto a los punteros a objetos, siguiendo el espíritu del paradigma de la programación orientada a objetos podemos permitir que guarden y restauren objetos: basta con implementar en cada clase una interfaz (un conjunto de métodos) que se encargue de transferir contenidos importantes a archivos y viceversa, y utilizar funciones de bajo nivel. Entonces, si nos encontramos con un campo puntero a otro objeto como parte del objeto, simplemente delegamos en él el guardado o la lectura, y a su vez, él se ocupará de sus campos, entre los que puede haber otros punteros, y la delegación seguirá profundizando hasta cubrir todos los elementos.

> Tenga en cuenta que en esta sección examinaremos las funciones atómicas para archivos binarios. Sus equivalentes para los archivos de texto se presentarán en la sección siguiente. Todas las funciones de esta sección devuelven el número de bytes escritos, o 0 en caso de error.

uint FileWriteDouble(int handle, double value)
uint FileWriteFloat(int handle, float value)
uint FileWriteLong(int handle, long value)

Las funciones escriben el valor del tipo correspondiente pasado en el parámetro *value* (*double*, *float*, *long*) en un archivo binario con el descriptor *handle*.

uint FileWriteInteger(int handle, int value, int size = INT_VALUE)

La función escribe el entero *value* en un archivo binario con el descriptor *handle*. El tamaño del valor en bytes se establece mediante el parámetro *size* y puede ser una de las constantes predefinidas: CHAR_VALUE (1), SHORT_VALUE (2), INT_VALUE (4, por defecto), que corresponden a los tipos *char*, *short* y *int* (con y sin signo).

La función admite un modo de escritura no documentado de un entero de 3 bytes. No se recomienda su uso.

El puntero del archivo se mueve por el número de bytes escritos (no por el tamaño de *int*).

uint FileWriteString(int handle, const string value, int length = -1)

La función escribe una cadena del parámetro *value* en un archivo binario con el descriptor *handle*. Puede especificar el número de caracteres para escribir el parámetro *length*. Si es inferior a la longitud de la cadena, sólo se incluirá en el archivo la parte especificada de la cadena. Si *length* es -1 o no se especifica, la cadena completa se transfiere al archivo sin el nulo terminal. Si *length* es mayor que la longitud de la cadena, los caracteres sobrantes se rellenan con ceros.

Tenga en cuenta que al escribir en un archivo abierto con la bandera FILE_UNICODE (o sin la bandera FILE_ANSI), la cadena se guarda en formato Unicode (cada carácter ocupa 2 bytes). Cuando se escribe en un archivo abierto con la bandera FILE_ANSI, cada carácter ocupa 1 byte (los caracteres de idiomas extranjeros pueden aparecer distorsionados).

> La función *FileWriteString* también puede trabajar con archivos de texto. Este aspecto de su aplicación se describe en la siguiente sección.

double FileReadDouble(int handle)
float FileReadFloat(int handle)
long FileReadLong(int handle)

Las funciones leen un número del tipo apropiado, *double*, *float* o *long*, de un archivo binario con el descriptor especificado. Si es necesario, convierta el resultado a *ulong* (si se espera un unsigned long en el archivo en esa posición).

int FileReadInteger(int handle, int size = INT_VALUE)

La función lee un valor entero de un archivo binario con el descriptor *handle*. El tamaño del valor en bytes se especifica en el parámetro *size*.

Dado que el resultado de la función es del tipo *int*, debe convertirse explícitamente al tipo de destino requerido si es diferente de *int* (es decir, a *uint*, o *short/ushort*, o *char/uchar*). De lo contrario, obtendrá como mínimo una advertencia del compilador y como máximo, una pérdida de signo.

El hecho es que al leer CHAR_VALUE o SHORT_VALUE, el resultado por defecto es siempre positivo (es decir, corresponde a *uchar* y *ushort*, que «se ajustan» totalmente en *int*). En estos casos, si los números son realmente de los tipos *uchar* y *ushort*, las advertencias del compilador son puramente nominales, puesto que ya estamos seguros de que dentro del valor del tipo *int* sólo se rellenan 1 o 2 bytes bajos, y son sin signo. Esto ocurre sin distorsión.

No obstante, cuando se almacenan valores con signo (tipos *char* y *short*) en el archivo, la conversión se hace necesaria porque, sin ella, los valores negativos se convertirán en positivos inversos con la misma representación de bits (véase el apartado «Enteros con y sin signo» en la sección Conversiones aritméticas de tipo).

En cualquier caso, es mejor evitar las advertencias mediante la conversión explícita de tipos.

La función admite el modo de lectura de enteros de 3 bytes. No se recomienda su uso.

El puntero del archivo se mueve por el número de bytes leídos (no por el tamaño *int*).

string FileReadString(int handle, int size = -1)

La función lee una cadena del tamaño especificado en caracteres de un archivo con el descriptor *handle*. El parámetro *size* debe establecerse cuando se trabaja con un archivo binario (el valor por defecto sólo es adecuado para archivos de texto que utilizan caracteres separadores). En caso

contrario, la cadena no se lee (la función devuelve una cadena vacía) y el código de error interno *_LastError* es 5016 (FILE_BINSTRINGSIZE).

Por lo tanto, incluso en la fase de escritura de una cadena en un archivo binario, es necesario pensar en cómo se leerá la cadena. Hay tres opciones principales:

- Escribir cadenas con un carácter terminal nulo al final. En este caso, habrá que analizarlas carácter por carácter en un bucle y combinar los caracteres en una cadena hasta encontrar el 0.
- Escribir siempre una cadena de longitud fija (predefinida). La longitud debe elegirse con un margen para la mayoría de los escenarios, o de acuerdo con la especificación (términos de referencia, protocolo, etc.), pero esto no es económico y no garantiza al 100 % que alguna cadena rara no se acorte al escribir en un archivo.
- Escribir la longitud como un entero antes de la cadena.

La función *FileReadString* también puede trabajar con archivos de texto. Este aspecto de su aplicación se describe en la siguiente sección.

Observe también que si el parámetro *size* es 0 (lo que puede ocurrir durante algunos cálculos), la función no lee: el puntero del archivo permanece en el mismo lugar y la función devuelve una cadena vacía.

Como ejemplo para esta sección, mejoraremos el script *FileStruct.mq5* de la sección anterior. El nuevo nombre del programa es *FileAtomic.mq5*.

La tarea sigue siendo la misma: guardar un número determinado de estructuras *MqlRates* truncadas con cotizaciones a un archivo binario. Pero ahora la estructura *FileHeader* se convertirá en una clase (y la firma de formato se almacenará en una cadena, no en un array de caracteres). Un encabezado de este tipo y un array de cotizaciones formarán parte de otra clase de control *Candles*, y ambas clases se heredarán de la interfaz *Persistent* para escribir objetos arbitrarios en un archivo y leer de un archivo.

Esta es la interfaz:

```
interface Persistent
{
   bool write(int handle);
   bool read(int handle);
};
```

En la clase *FileHeader* implementaremos el guardado y la comprobación de la firma de formato (vamos a cambiarla a «CANDLES/1.1») y los nombres del símbolo actual y el marco temporal del gráfico (más información sobre *_Symbol* y *_Period*).

La escritura se realiza en la implementación del método *write* heredado de la interfaz.

```
class FileHeader : public Persistent
{
   const string signature;
public:
   FileHeader() : signature("CANDLES/1.1") { }
   bool write(int handle) override
   {
      PRTF(FileWriteString(handle, signature, StringLen(signature)));
      PRTF(FileWriteInteger(handle, StringLen(_Symbol), CHAR_VALUE));
      PRTF(FileWriteString(handle, _Symbol));
      PRTF(FileWriteString(handle, PeriodToString(), 3));
      return true;
   }
```

La firma se escribe exactamente según su longitud, ya que la muestra se almacena en el objeto y la misma longitud se establecerá al leer.

Para el instrumento del gráfico actual, primero guardamos la longitud de su nombre en el archivo (1 byte es suficiente para longitudes de hasta 255), y sólo entonces guardamos la cadena en sí.

El nombre del marco temporal nunca supera los tres símbolos, si se excluye de él el prefijo constante «PERIOD_», por lo que se elige una longitud fija para esta cadena. El nombre del marco temporal sin prefijo se obtiene en la función auxiliar *PeriodToString*: se encuentra en un archivo de encabezado independiente *Periods.mqh* (ello se abordará con más detalle en la sección Símbolos y marcos temporales).

La lectura se realiza en el método *read* en orden inverso (por supuesto, se supone que la lectura se llevará a cabo en un objeto nuevo y diferente).

```
bool read(int handle) override
{
   const string sig = PRTF(FileReadString(handle, StringLen(signature)));
   if(sig != signature)
   {
      PrintFormat("Wrong file format, header is missing: want=%s vs got %s",
         signature, sig);
      return false;
   }
   const int len = PRTF(FileReadInteger(handle, CHAR_VALUE));
   const string sym = PRTF(FileReadString(handle, len));
   if(_Symbol != sym)
   {
      PrintFormat("Wrong symbol: file=%s vs chart=%s", sym, _Symbol);
      return false;
   }
   const string stf = PRTF(FileReadString(handle, 3));
   if(_Period != StringToPeriod(stf))
   {
      PrintFormat("Wrong timeframe: file=%s(%s) vs chart=%s",
         stf, EnumToString(StringToPeriod(stf)), EnumToString(_Period));
      return false;
   }
   return true;
}
```

Si alguna de las propiedades (firma, símbolo, marco temporal) no coincide en el archivo y en el gráfico actual, la función devuelve *false* para indicar un error.

La transformación inversa del nombre del marco temporal en la enumeración ENUM_TIMEFRAMES la realiza la función *StringToPeriod*, también desde el archivo *Periods.mqh*.

La clase *Candles* principal para solicitar, guardar y leer el archivo de cotizaciones es la siguiente:

```
class Candles : public Persistent
{
   FileHeader header;
   int limit;
   MqlRates rates[];
public:
   Candles(const int size = 0) : limit(size)
   {
      if(size == 0) return;
      int n = PRTF(CopyRates(_Symbol, _Period, 0, limit, rates));
      if(n < 1)
      {
 limit =0; // initialization failed
      }
 limit =n; // may be less than requested
   }
```

Los campos son el encabezado del tipo *FileHeader*, el número de barras solicitado *limit* y un array que recibe las estructuras *MqlRates* de MetaTrader 5. El array se rellena en el constructor. En caso de error, el campo *limit* se pone a cero.

Al derivar de la interfaz *Persistent*, la clase *Candles* requiere la implementación de los métodos *write* y *read*. En el método *write*, primero ordenamos al objeto de encabezado que se guarde a sí mismo y, a continuación, añadimos al archivo el número de cotizaciones, el intervalo de fechas (como referencia) y el propio array.

```
   bool write(int handle) override
   {
      if(!limit) return false; // no data
      if(!header.write(handle)) return false;
      PRTF(FileWriteInteger(handle, limit));
      PRTF(FileWriteLong(handle, rates[0].time));
      PRTF(FileWriteLong(handle, rates[limit - 1].time));
      for(int i = 0; i < limit; ++i)
      {
         FileWriteStruct(handle, rates[i], offsetof(MqlRates, tick_volume));
      }
      return true;
   }
```

La lectura se realiza en orden inverso:

```
bool read(int handle) override
{
   if(!header.read(handle))
   {
      return false;
   }
   limit = PRTF(FileReadInteger(handle));
   ArrayResize(rates, limit);
   ZeroMemory(rates);
   // dates need to be read: they are not used, but this shifts the position in th
   // it was possible to explicitly change the position, but this function has not
   datetime dt0 = (datetime)PRTF(FileReadLong(handle));
   datetime dt1 = (datetime)PRTF(FileReadLong(handle));
   for(int i = 0; i < limit; ++i)
   {
      FileReadStruct(handle, rates[i], offsetof(MqlRates, tick_volume));
   }
   return true;
}
```

En un programa real para archivar cotizaciones, la presencia de un rango de fechas permitiría construir su secuencia correcta a lo largo de una larga historia mediante los encabezados de los archivos y, en cierta medida, protegería contra el cambio arbitrario de nombre de los archivos.

Existe un sencillo método *print* para controlar el proceso:

```
void print() const
{
   ArrayPrint(rates);
}
```

En la función principal del script, creamos dos objetos *Candles*, y utilizando uno de ellos, primero guardamos el archivo de cotizaciones y luego lo restauramos con la ayuda del otro. Los archivos se administran mediante el envoltorio *FileHandle* que ya conocemos (véase la sección Gestión de descriptores de archivos).

```
const string filename = "MQL5Book/atomic.raw";

void OnStart()
{
   // create a new file and reset the old one
   FileHandle handle(PRTF(FileOpen(filename,
      FILE_BIN | FILE_WRITE | FILE_ANSI | FILE_SHARE_READ)));
   // form data
   Candles output(BARLIMIT);
   // write them to a file
   if(!output.write(~handle))
   {
      Print("Can't write file");
      return;
   }
   output.print();

   // open the newly created file for checking
   handle = PRTF(FileOpen(filename,
      FILE_BIN | FILE_READ | FILE_ANSI | FILE_SHARE_READ | FILE_SHARE_WRITE));
   // create an empty object to receive quotes
   Candles inputs;
   // read data from the file into it
   if(!inputs.read(~handle))
   {
      Print("Can't read file");
   }
   else
   {
      inputs.print();
   }
```

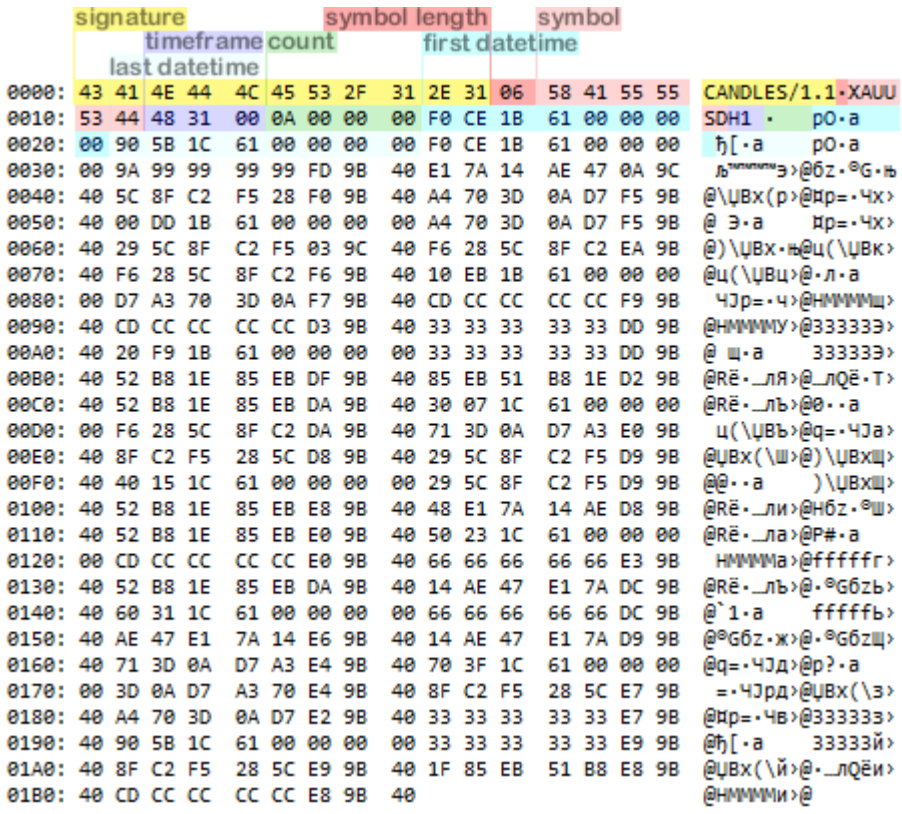
Aquí hay un ejemplo de registros de datos iniciales para XAUUSD,H1:

```
FileOpen(filename,FILE_BIN|FILE_WRITE|FILE_ANSI|FILE_SHARE_READ)=1 / ok
CopyRates(_Symbol,_Period,0,limit,rates)=10 / ok
FileWriteString(handle,signature,StringLen(signature))=11 / ok
FileWriteInteger(handle,StringLen(_Symbol),CHAR_VALUE)=1 / ok
FileWriteString(handle,_Symbol)=6 / ok
FileWriteString(handle,PeriodToString(),3)=3 / ok
FileWriteInteger(handle,limit)=4 / ok
FileWriteLong(handle,rates[0].time)=8 / ok
FileWriteLong(handle,rates[limit-1].time)=8 / ok
[hora] [apertura] [máximo] [mínimo] [cierre] [volumen_tic] [diferencial] [volumen_rea
[0] 2021.08.17 15:00:00 1791.40 1794.57 1788.04 1789.46 8157 5 0
[1] 2021.08.17 16:00:00 1789.46 1792.99 1786.69 1789.69 9285 5 0
[2] 2021.08.17 17:00:00 1789.76 1790.45 1780.95 1783.30 8165 5 0
[3] 2021.08.17 18:00:00 1783.30 1783.98 1780.53 1782.73 5114 5 0
[4] 2021.08.17 19:00:00 1782.69 1784.16 1782.09 1782.49 3586 6 0
[5] 2021.08.17 20:00:00 1782.49 1786.23 1782.17 1784.23 3515 5 0
[6] 2021.08.17 21:00:00 1784.20 1784.85 1782.73 1783.12 2627 6 0
[7] 2021.08.17 22:00:00 1783.10 1785.52 1782.37 1785.16 2114 5 0
[8] 2021.08.17 23:00:00 1785.11 1785.84 1784.71 1785.80 922 5 0
[9] 2021.08.18 01:00:00 1786.30 1786.34 1786.18 1786.20 13 5 0
```

Y aquí tiene un ejemplo de los datos recuperados (recuerde que las estructuras se guardan truncadas según nuestra tarea técnica hipotética):

```
FileOpen(filename,FILE_BIN|FILE_READ|FILE_ANSI|FILE_SHARE_READ|FILE_SHARE_WRITE)=2 /
FileReadString(handle,StringLen(signature))=CANDLES/1.1 / ok
FileReadInteger(handle,CHAR_VALUE)=6 / ok
FileReadString(handle,len)=XAUUSD / ok
FileReadString(handle,3)=H1 / ok
FileReadInteger(handle)=10 / ok
FileReadLong(handle)=1629212400 / ok
FileReadLong(handle)=1629248400 / ok
[hora] [apertura] [máximo] [mínimo] [cierre] [volumen_tic] [diferencial] [volumen_rea
[0] 2021.08.17 15:00:00 1791.40 1794.57 1788.04 1789.46 0 0 0
[1] 2021.08.17 16:00:00 1789.46 1792.99 1786.69 1789.69 0 0 0
[2] 2021.08.17 17:00:00 1789.76 1790.45 1780.95 1783.30 0 0 0
[3] 2021.08.17 18:00:00 1783.30 1783.98 1780.53 1782.73 0 0 0
[4] 2021.08.17 19:00:00 1782.69 1784.16 1782.09 1782.49 0 0 0
[5] 2021.08.17 20:00:00 1782.49 1786.23 1782.17 1784.23 0 0 0
[6] 2021.08.17 21:00:00 1784.20 1784.85 1782.73 1783.12 0 0 0
[7] 2021.08.17 22:00:00 1783.10 1785.52 1782.37 1785.16 0 0 0
[8] 2021.08.17 23:00:00 1785.11 1785.84 1784.71 1785.80 0 0 0
[9] 2021.08.18 01:00:00 1786.30 1786.34 1786.18 1786.20 0 0 0
```

Es fácil asegurarse de que los datos se almacenen y lean correctamente. Y ahora veamos cómo se ven dentro del archivo:



Visualización de la estructura interna de un archivo binario con un archivo de cotizaciones en un programa externo

Aquí se resaltan con color varios campos de nuestro encabezado: firma, longitud del nombre del símbolo, nombre del símbolo, nombre del marco temporal, etc.

4.5.9 Escritura y lectura de variables (archivos de texto)

Los archivos de texto tienen su propio conjunto de funciones para el almacenamiento atómico (elemento por elemento) y para la lectura de datos. Es ligeramente diferente de los archivos binarios establecidos en la sección anterior. También hay que tener en cuenta que no existen funciones analógicas para escribir/leer una estructura o un array de estructuras en un archivo de texto. Si intenta utilizar cualquiera de estas funciones con un archivo de texto, no tendrán ningún efecto, sino que emitirán un código de error interno 5011 (FILE_NOTBIN).

Como ya sabemos, los archivos de texto en MQL5 tienen dos formas: texto plano y texto en formato CSV. El modo correspondiente, FILE_TXT o FILE_CSV, se establece cuando se abre el archivo y no se puede cambiar sin cerrar y volver a adquirir el manejador. La diferencia entre ellos sólo aparece al leer archivos. Ambos modos se graban de la misma manera.

En el modo TXT, cada llamada a la función de lectura (cualquiera de las funciones que veremos en esta sección) encuentra la siguiente nueva línea en el archivo (un carácter '\n' o un par de '\r\n') y procesa todo hasta ella. El objetivo del tratamiento es convertir el texto del archivo en un valor de un tipo específico correspondiente a la función llamada. En el caso más sencillo, si se llama a la función *FileReadString* no se realiza ningún procesamiento (la cadena se devuelve «tal cual»).

En el modo CSV, cada vez que se llama a la función de lectura, el texto del archivo se divide de manera lógica, no sólo por nuevas líneas, sino también por un delimitador adicional especificado al abrir el archivo. El resto del procesamiento del fragmento desde la posición actual del archivo hasta el delimitador más cercano es similar.

En otras palabras: la lectura del texto y la transferencia de la posición interna dentro del archivo se realiza en fragmentos de delimitador a delimitador, donde delimitador significa no sólo el carácter *delimiter* de la lista de parámetros *FileOpen*, sino también una nueva línea ('\n', '\r\n'), así como el principio y el final del archivo.

El delimitador adicional tiene el mismo efecto al escribir texto en archivos FILE_TXT y FILE_CSV, pero sólo cuando se utiliza la función *FileWrite*: inserta automáticamente este carácter entre los elementos grabados. El separador de funciones *FileWriteString* se ignora.

Veamos las descripciones formales de las funciones y, a continuación, un ejemplo en *FileTxtCsv.mq5*.

uint FileWrite(int handle, ...)

La función pertenece a la categoría de funciones que toman un número variable de parámetros. Estos parámetros se indican en el prototipo de la función con una elipsis. Sólo se admiten los tipos de datos integrados. Para escribir estructuras u objetos de clase, debe «desreferenciar» sus elementos y pasarlos individualmente.

La función escribe todos los argumentos pasados después del primero en un archivo de texto con el descriptor *handle*. Los argumentos se separan por comas, como en una lista de argumentos normal. El número de argumentos que se envían al archivo no puede ser superior a 63.

En la salida, los datos numéricos se convierten a formato de texto según las reglas de la conversión estándar a *(string)*. Los valores o el tipo *double* tienen salida a 16 dígitos significativos, ya sea en formato tradicional o en formato de exponente científico (se elige la opción más compacta). Los datos del tipo *float* se muestran con una precisión de 7 dígitos significativos. Para mostrar números reales con una precisión diferente o en un formato especificado explícitamente, utilice la función *DoubleToString* (véase De números a cadenas y viceversa).

Los valores del tipo *datetime* se imprimen en el formato «AAAA.MM.DD hh:mm:ss» (véase Fecha y hora).

Un color estándar (de la lista de colores web) se muestra como un nombre, un color no estándar se muestra como un triple de valores de componentes RGB (véase Color), separados por comas (nota: la coma es el carácter separador más común en CSV).

En el caso de las enumeraciones se muestra un número entero que denota el elemento en lugar de su identificador (nombre). Por ejemplo, al escribir VIERNES (de ENUM_DAY_OF_WEEK, véase Enumeraciones) obtenemos el número 5 en el archivo.

Los valores del tipo *bool* se muestran como cadenas «true» o «false».

Si al abrir el archivo se ha especificado un carácter delimitador distinto de 0, éste se insertará entre dos líneas adyacentes resultantes de la conversión de los argumentos correspondientes.

Una vez que todos los argumentos se escriben en el archivo, se añade un terminador de línea '\r\n'.

La función devuelve el número de bytes escritos, o 0 en caso de error.

uint FileWriteString(int handle, const string text, int length = -1)

La función escribe el parámetro de cadena *text* en un archivo de texto con el descriptor *handle*. El parámetro *length* sólo es aplicable a archivos binarios y se ignora en este contexto (la línea se escribe completa).

> La función *FileWriteString* también puede trabajar con archivos binarios. Esta aplicación de la función se describe en el apartado anterior.

Los separadores (entre elementos de una línea) y las nuevas líneas deben ser insertados o añadidos por el programador.

La función devuelve el número de bytes escritos (en modo FILE_UNICODE será 2 veces la longitud de la cadena en caracteres) o 0 en caso de error.

string FileReadString(int handle, int length = -1)

La función lee una cadena hasta el siguiente delimitador de un archivo con el descriptor *handle* (carácter delimitador en un archivo CSV, carácter de salto de línea en cualquier archivo, o hasta el final del archivo). El parámetro *length* sólo se aplica a los archivos binarios y se ignora en este contexto.

La cadena resultante se puede convertir en un valor del tipo requerido utilizando normas de reducción estándar o mediante funciones de conversión. Como alternativa, se pueden utilizar funciones de lectura especializadas, como *FileReadBool*, *FileReadDatetime*, *FileReadNumber*, que se describen a continuación.

En caso de error, se devolverá una cadena vacía. El código de error puede encontrarse por medio de la variable *_LastError* o de la función *GetLastError*. En concreto, cuando se alcance el final del archivo, el código de error será 5027 (FILE_ENDOFFILE).

bool FileReadBool(int handle)

La función lee un fragmento de un archivo CSV hasta el siguiente delimitador, o hasta el final de la línea y lo convierte en un valor del tipo *bool*. Si el fragmento contiene el texto «true» (ya sea en minúsculas, mayúsculas o una combinación de ambas, como en «True»), o un número distinto de cero, obtenemos *true*. En otros casos, obtenemos *false*.

La palabra «true» debe ocupar todo el elemento de lectura. Aun cuando la cadena empiece por «true» y tenga una continuación (por ejemplo, «True Volume»), obtendremos *false*.

datetime FileReadDatetime(int handle)

La función lee en un archivo CSV una cadena de uno de los siguientes formatos: «AAAA.MM.DD hh:mm:ss», «AAAA.MM.DD» o «hh:mm:ss», y lo convierte en un valor del tipo *datetime*. Si el fragmento no contiene una representación textual válida de la fecha y/o la hora, la función devolverá cero o una hora «rara», dependiendo de los caracteres que pueda interpretar como fragmentos de fecha y hora. Para cadenas vacías o no numéricas, obtenemos la fecha actual con hora cero.

Se puede conseguir una lectura más flexible de la fecha y la hora (con más formatos admitidos) combinando dos funciones: *StringToTime(FileReadString(handle))*. Para obtener más información sobre *StringToTime*, consulte Fecha y hora.

double FileReadNumber(int handle)

La función lee un fragmento del archivo CSV hasta el siguiente delimitador o hasta el final de la línea, y lo convierte en un valor de tipo *double* según las normas estándar de conversión de tipos.

Tenga en cuenta que *double* puede perder la precisión de valores muy grandes, lo que puede afectar a la lectura de números grandes de los tipos *long/ulong* (el valor a partir del cual se distorsionan los enteros dentro de *double* es 9007199254740992; se ofrece un ejemplo de este fenómeno en la sección Uniones).

> Las funciones analizadas en la sección anterior, incluidas *FileReadDouble*, *FileReadFloat*, *FileReadInteger*, *FileReadLong* y *FileReadStruct*, no pueden aplicarse a archivos de texto.

El script *FileTxtCsv.mq5* demuestra cómo trabajar con archivos de texto. La última vez cargamos las cotizaciones en un archivo binario. Ahora vamos a hacerlo en formatos TXT y CSV.

Básicamente, MetaTrader 5 permite exportar e importar cotizaciones en formato CSV desde el cuadro de diálogo «Símbolos». Sin embargo, a efectos educativos, reproduciremos este proceso. Además, la aplicación informática le permite desviarse del formato exacto que se genera por defecto. A continuación se muestra un fragmento del historial XAUUSD H1 exportado de la forma estándar.

```
<DATE> » <TIME> » <OPEN> » <HIGH> » <LOW> » <CLOSE> » <TICKVOL> » <VOL> » <SPREAD>
2021.01.04 » 01:00:00 » 1909.07 » 1914.93 » 1907.72 » 1913.10 » 4230 » 0 » 5
2021.01.04 » 02:00:00 » 1913.04 » 1913.64 » 1909.90 » 1913.41 » 2694 » 0 » 5
2021.01.04 » 03:00:00 » 1913.41 » 1918.71 » 1912.16 » 1916.61 » 6520 » 0 » 5
2021.01.04 » 04:00:00 » 1916.60 » 1921.89 » 1915.49 » 1921.79 » 3944 » 0 » 5
2021.01.04 » 05:00:00 » 1921.79 » 1925.26 » 1920.82 » 1923.19 » 3293 » 0 » 5
2021.01.04 » 06:00:00 » 1923.20 » 1923.71 » 1920.24 » 1922.67 » 2146 » 0 » 5
2021.01.04 » 07:00:00 » 1922.66 » 1922.99 » 1918.93 » 1921.66 » 3141 » 0 » 5
2021.01.04 » 08:00:00 » 1921.66 » 1925.60 » 1921.47 » 1922.99 » 3752 » 0 » 5
2021.01.04 » 09:00:00 » 1922.99 » 1925.54 » 1922.47 » 1924.80 » 2895 » 0 » 5
2021.01.04 » 10:00:00 » 1924.85 » 1935.16 » 1924.59 » 1932.07 » 6132 » 0 » 5
```

Aquí, en concreto, puede que no estemos satisfechos con el carácter separador por defecto (tabulador, denotado como «»), el orden de las columnas o el hecho de que la fecha y la hora estén divididas en dos campos.

En nuestro script, elegiremos la coma como separador y generaremos las columnas en el orden de los campos de la estructura *MqlRates*. La descarga y posterior lectura de prueba se realizará en los modos FILE_TXT y FILE_CSV.

```
const string txtfile = "MQL5Book/atomic.txt";
const string csvfile = "MQL5Book/atomic.csv";
const short delimiter = ',';
```

Las cotizaciones se solicitarán al principio de la función *OnStart* de la forma habitual:

```
void OnStart()
{
   MqlRates rates[];
   int n = PRTF(CopyRates(_Symbol, _Period, 0, 10, rates)); // 10
```

Especificaremos los nombres de las columnas del array por separado, y también los combinaremos utilizando la función de ayuda *StringCombine*. Los títulos separados son necesarios porque los combinamos en un título común utilizando un carácter delimitador seleccionable (una solución alternativa podría basarse en *StringReplace*). Le animamos a que trabaje con el código fuente *StringCombine* de forma independiente: realiza la operación inversa con respecto al *StringSplit* integrado.

```
const string columns[] = {"DateTime", "Open", "High", "Low", "Close",
                          "Ticks", "Spread", "True"};
const string caption = StringCombine(columns, delimiter) + "\r\n";
```

La última columna debería haberse llamado «Volume», pero utilizaremos su ejemplo para comprobar el rendimiento de la función *FileReadBool*. Puede suponer que el nombre actual implica «True Volume» (pero tal cadena no se interpretaría como *true*).

A continuación vamos a abrir dos archivos en los modos FILE_TXT y FILE_CSV, y a escribir en ellos el encabezado preparado.

```
int fh1 = PRTF(FileOpen(txtfile, FILE_TXT | FILE_ANSI | FILE_WRITE, delimiter));//
int fh2 = PRTF(FileOpen(csvfile, FILE_CSV | FILE_ANSI | FILE_WRITE, delimiter));//

PRTF(FileWriteString(fh1, caption)); // 48
PRTF(FileWriteString(fh2, caption)); // 48
```

Como la función *FileWriteString* no añade automáticamente una nueva línea, hemos añadido «\r\n» a la variable *caption*.

```
for(int i = 0; i < n; ++i)
{
   FileWrite(fh1, rates[i].time,
      rates[i].open, rates[i].high, rates[i].low, rates[i].close,
      rates[i].tick_volume, rates[i].spread, rates[i].real_volume);
   FileWrite(fh2, rates[i].time,
      rates[i].open, rates[i].high, rates[i].low, rates[i].close,
      rates[i].tick_volume, rates[i].spread, rates[i].real_volume);
}

FileClose(fh1);
FileClose(fh2);
```

La escritura de campos de estructura desde el array *rates* se realiza de la misma manera, llamando a *FileWrite* en un bucle para cada uno de los dos archivos. Recuerde que la función *FileWrite* inserta automáticamente un carácter delimitador entre los argumentos y añade «\r\n» al final de la cadena. Por supuesto, era posible convertir independientemente todos los valores de salida en cadenas y enviarlos a un archivo utilizando *FileWriteString*, pero entonces tendríamos que ocuparnos nosotros mismos de los separadores y las nuevas líneas. En algunos casos no son necesarios, por ejemplo, si está escribiendo en formato JSON de forma compacta (esencialmente en una línea gigante).

Así, en la fase de grabación, ambos archivos se gestionaron de la misma manera y resultaron ser iguales. He aquí un ejemplo de su contenido para XAUUSD,H1 (sus resultados pueden variar):

```
DateTime,Open,High,Low,Close,Ticks,Spread,True
2021.08.19 12:00:00,1785.3,1789.76,1784.75,1789.06,4831,5,0
2021.08.19 13:00:00,1789.06,1790.02,1787.61,1789.06,3393,5,0
2021.08.19 14:00:00,1789.08,1789.95,1786.78,1786.89,3536,5,0
2021.08.19 15:00:00,1786.78,1789.86,1783.73,1788.82,6840,5,0
2021.08.19 16:00:00,1788.82,1792.44,1782.04,1784.02,9514,5,0
2021.08.19 17:00:00,1784.04,1784.27,1777.14,1780.57,8526,5,0
2021.08.19 18:00:00,1780.55,1784.02,1780.05,1783.07,5271,6,0
2021.08.19 19:00:00,1783.06,1783.15,1780.73,1782.59,3571,7,0
2021.08.19 20:00:00,1782.61,1782.96,1780.16,1780.78,3236,10,0
2021.08.19 21:00:00,1780.79,1780.9,1778.54,1778.65,1017,13,0
```

Las diferencias a la hora de trabajar con estos archivos empezarán a aparecer en la fase de lectura.

Vamos a abrir un archivo de texto para leerlo y «escaneémoslo» utilizando la función *FileReadString* en un bucle, hasta que devuelva una cadena vacía (es decir, hasta el final del archivo).

```
string read;
fh1 = PRTF(FileOpen(txtfile, FILE_TXT | FILE_ANSI | FILE_READ, delimiter)); // 1
Print("===== Reading TXT");
do
{
   read = PRTF(FileReadString(fh1));
}
while(StringLen(read) > 0);
```

El registro mostrará algo como esto:

```
===== Reading TXT
FileReadString(fh1)=DateTime,Open,High,Low,Close,Ticks,Spread,True / ok
FileReadString(fh1)=2021.08.19 12:00:00,1785.3,1789.76,1784.75,1789.06,4831,5,0 / ok
FileReadString(fh1)=2021.08.19 13:00:00,1789.06,1790.02,1787.61,1789.06,3393,5,0 / ok
FileReadString(fh1)=2021.08.19 14:00:00,1789.08,1789.95,1786.78,1786.89,3536,5,0 / ok
FileReadString(fh1)=2021.08.19 15:00:00,1786.78,1789.86,1783.73,1788.82,6840,5,0 / ok
FileReadString(fh1)=2021.08.19 16:00:00,1788.82,1792.44,1782.04,1784.02,9514,5,0 / ok
FileReadString(fh1)=2021.08.19 17:00:00,1784.04,1784.27,1777.14,1780.57,8526,5,0 / ok
FileReadString(fh1)=2021.08.19 18:00:00,1780.55,1784.02,1780.05,1783.07,5271,6,0 / ok
FileReadString(fh1)=2021.08.19 19:00:00,1783.06,1783.15,1780.73,1782.59,3571,7,0 / ok
FileReadString(fh1)=2021.08.19 20:00:00,1782.61,1782.96,1780.16,1780.78,3236,10,0 / o
FileReadString(fh1)=2021.08.19 21:00:00,1780.79,1780.9,1778.54,1778.65,1017,13,0 / ok
FileReadString(fh1)= / FILE_ENDOFFILE(5027)
```

Cada llamada de *FileReadString* lee la línea completa (hasta '\r\n') en el modo FILE_TXT. Para separarlo en elementos, debemos aplicar un tratamiento adicional. Opcionalmente, podemos utilizar el modo FILE_CSV.

Hagamos lo mismo con el archivo CSV.

```
   fh2 = PRTF(FileOpen(csvfile, FILE_CSV | FILE_ANSI | FILE_READ, delimiter)); // 2
   Print("===== Reading CSV");
   do
   {
      read = PRTF(FileReadString(fh2));
   }
   while(StringLen(read) > 0);
```

Esta vez habrá muchas más entradas en el registro:

```
===== Reading CSV
FileReadString(fh2)=DateTime / ok
FileReadString(fh2)=Open / ok
FileReadString(fh2)=High / ok
FileReadString(fh2)=Low / ok
FileReadString(fh2)=Close / ok
FileReadString(fh2)=Ticks / ok
FileReadString(fh2)=Spread / ok
FileReadString(fh2)=True / ok
FileReadString(fh2)=2021.08.19 12:00:00 / ok
FileReadString(fh2)=1785.3 / ok
FileReadString(fh2)=1789.76 / ok
FileReadString(fh2)=1784.75 / ok
FileReadString(fh2)=1789.06 / ok
FileReadString(fh2)=4831 / ok
FileReadString(fh2)=5 / ok
FileReadString(fh2)=0 / ok
...
FileReadString(fh2)=2021.08.19 21:00:00 / ok
FileReadString(fh2)=1780.79 / ok
FileReadString(fh2)=1780.9 / ok
FileReadString(fh2)=1778.54 / ok
FileReadString(fh2)=1778.65 / ok
FileReadString(fh2)=1017 / ok
FileReadString(fh2)=13 / ok
FileReadString(fh2)=0 / ok
FileReadString(fh2)= / FILE_ENDOFFILE(5027)
```

La cuestión es que la función *FileReadString* en el modo FILE_CSV tiene en cuenta el carácter delimitador y divide las cadenas en elementos. Cada llamada a *FileReadString* devuelve un único valor (celda) de una tabla CSV. Obviamente, las cadenas resultantes deben convertirse posteriormente a los tipos adecuados.

Este problema puede resolverse de forma generalizada utilizando las funciones especializadas *FileReadDatetime*, *FileReadNumber*, *FileReadBool*. Sin embargo, en cualquier caso, el desarrollador debe llevar un registro del número de la columna legible actual y determinar su significado práctico. En el tercer paso de la prueba se ofrece un ejemplo de un algoritmo de este tipo. Utiliza el mismo archivo CSV (para simplificar, lo cerramos al final de cada paso y lo abrimos al principio del siguiente).

Para simplificar la asignación del siguiente campo en la estructura *MqlRates* por el número de columna, hemos creado una estructura hija *MqlRates* que contiene un método de plantilla *set*:

```
struct MqlRatesM : public MqlRates
{
   template<typename T>
   void set(int field, T v)
   {
      switch(field)
      {
         case 0: this.time = (datetime)v; break;
         case 1: this.open = (double)v; break;
         case 2: this.high = (double)v; break;
         case 3: this.low = (double)v; break;
         case 4: this.close = (double)v; break;
         case 5: this.tick_volume = (long)v; break;
         case 6: this.spread = (int)v; break;
         case 7: this.real_volume = (long)v; break;
      }
   }
};
```

En la función *OnStart* hemos descrito un array de una estructura de este tipo, donde añadiremos los valores entrantes. El array se requiere para simplificar el registro con *ArrayPrint* (no hay ninguna función ya preparada en MQL5 para imprimir una estructura por sí misma).

```
Print("===== Reading CSV (alternative)");
MqlRatesM r[1];
int count = 0;
int column = 0;
const int maxColumn = ArraySize(columns);
```

La variable *count* que cuenta los registros era necesaria no sólo para las estadísticas, sino también como medio para omitir la primera línea, que contiene encabezados y no datos. El número de columna actual se registra en la variable *column*. Su valor máximo no debe superar el número de columnas *maxColumn*.

Ahora sólo tenemos que abrir el archivo y leer elementos del mismo en un bucle utilizando varias funciones hasta que se produzca un error; en concreto, un error esperado como 5027 (FILE_ENDOFFILE), es decir, que se llegue al final del archivo.

Cuando el número de columna es 0, aplicamos la función *FileReadDatetime*. Para otras columnas, utilice *FileReadNumber*. La excepción es el caso de la primera línea con encabezados: para ello llamamos a la función *FileReadBool* a fin de demostrar cómo reaccionaría ante el encabezado «True» añadido deliberadamente a la última columna.

```
fh2 = PRTF(FileOpen(csvfile, FILE_CSV | FILE_ANSI | FILE_READ, delimiter)); // 1
do
{
   if(column)
   {
      if(count == 1) // demo for FileReadBool on the 1st record with headers
      {
         r[0].set(column, PRTF(FileReadBool(fh2)));
      }
      else
      {
         r[0].set(column, FileReadNumber(fh2));
      }
   }
   else // 0th column is the date and time
   {
      ++count;
      if(count >1) // the structure from the previous line is ready
      {
         ArrayPrint(r, _Digits, NULL, 0, 1, 0);
      }
      r[0].time = FileReadDatetime(fh2);
   }
   column = (column + 1) % maxColumn;
}
while(_LastError == 0); // exit when end of file 5027 is reached (FILE_ENDOFFILE)

// printing the last structure
if(column == maxColumn - 1)
{
   ArrayPrint(r, _Digits, NULL, 0, 1, 0);
}
```

Esto es lo que se registra:

```
===== Reading CSV (alternative)
FileOpen(csvfile,FILE_CSV|FILE_ANSI|FILE_READ,delimiter)=1 / ok
FileReadBool(fh2)=false / ok
FileReadBool(fh2)=false / ok
FileReadBool(fh2)=false / ok
FileReadBool(fh2)=false / ok
FileReadBool(fh2)=false / ok
FileReadBool(fh2)=false / ok
FileReadBool(fh2)=true / ok
2021.08.19 00:00:00   0.00   0.00  0.00    0.00          0     0       1
2021.08.19 12:00:00 1785.30 1789.76 1784.75 1789.06       4831     5       0
2021.08.19 13:00:00 1789.06 1790.02 1787.61 1789.06       3393     5       0
2021.08.19 14:00:00 1789.08 1789.95 1786.78 1786.89       3536     5       0
2021.08.19 15:00:00 1786.78 1789.86 1783.73 1788.82       6840     5       0
2021.08.19 16:00:00 1788.82 1792.44 1782.04 1784.02       9514     5       0
2021.08.19 17:00:00 1784.04 1784.27 1777.14 1780.57       8526     5       0
2021.08.19 18:00:00 1780.55 1784.02 1780.05 1783.07       5271     6       0
2021.08.19 19:00:00 1783.06 1783.15 1780.73 1782.59       3571     7       0
2021.08.19 20:00:00 1782.61 1782.96 1780.16 1780.78       3236    10       0
2021.08.19 21:00:00 1780.79 1780.90 1778.54 1778.65       1017    13       0
```

Como ve, de todos los encabezados, sólo el último se convierte al valor *true*, y todos los anteriores son *false*.

El contenido de las estructuras leídas es el mismo que el de los datos originales.

4.5.10 Gestión de la posición en un archivo

Como ya sabemos, el sistema asocia un determinado puntero a cada archivo abierto que determina el lugar del archivo (offset desde su inicio) en el que se escribirán o leerán los datos la próxima vez que se llame a cualquier función de E/S. Una vez ejecutada la función, el puntero se desplaza según el tamaño de los datos escritos o leídos.

En algunos casos se desea cambiar la posición del puntero sin operaciones de E/S. En concreto, cuando necesitamos añadir datos al final de un archivo, lo abrimos en modo «mixto» FILE_READ | FILE_WRITE, y entonces debemos terminar de alguna manera al final del archivo (de lo contrario, empezaremos a sobrescribir los datos desde el principio). Podríamos llamar a las funciones de lectura mientras hay algo que leer (desplazando así el puntero), pero esto no es eficiente. Es mejor utilizar la función especial FileSeek. Y la función *FileTell* permite obtener el valor real del puntero (posición en el archivo).

En esta sección exploraremos éstas y un par de funciones más relacionadas con la posición actual en un archivo. Algunas de ellas funcionan de la misma manera para archivos en modo texto y binario, mientras que otras son diferentes.

bool FileSeek(int handle, long offset, ENUM_FILE_POSITION origin)

La función desplaza el puntero del archivo el número *offset* de bytes utilizando como referencia *origin*, que es una de las posiciones predefinidas descritas en la enumeración ENUM_FILE_POSITION. *offset* puede ser positivo (desplazamiento hasta el final del archivo y más allá) o negativo (desplazamiento hasta el principio). ENUM_FILE_POSITION tiene los siguientes miembros:

- SEEK_SET para el inicio del archivo
- SEEK_CUR para la posición actual

- SEEK_END para el final del archivo

Si el cálculo de la nueva posición relativa al punto de anclaje ha dado un valor negativo (es decir, se solicita un desplazamiento a la izquierda del principio del archivo), entonces el puntero del archivo se fijará al principio del archivo.

Si establece la posición más allá del final del archivo (el valor es mayor que el tamaño del archivo), la escritura posterior en el archivo no se realizará desde el final del archivo, sino desde la posición establecida. En este caso, se escribirán valores indefinidos entre el final anterior del archivo y la posición dada (véase más abajo).

La función devuelve *true* en caso de éxito y *false* en caso de error.

ulong FileTell(int handle)

Para un archivo abierto con el descriptor *handle*, la función devuelve la posición actual del puntero interno (un desplazamiento relativo al principio del archivo). En caso de error, se devolverá ULONG_MAX ((ulong)-1). El código de error está disponible en la variable *_LastError*, o a través de la función *GetLastError*.

bool FileIsEnding(int handle)

La función devuelve una indicación de si el puntero se encuentra al final del archivo *handle*. Si es así, el resultado es *true*.

bool FileIsLineEnding(int handle)

Para un archivo de texto con el descriptor *handle*, la función devuelve un signo de si el puntero del archivo está al final de la línea (inmediatamente después de los caracteres de nueva línea '\n' o '\r\n'). En otras palabras: el valor de retorno *true* significa que la posición actual está al principio de la siguiente línea (o al final del archivo). Para archivos binarios, el resultado es siempre *false*.

El script de prueba de las funciones mencionadas se llama *FileCursor.mq5*. Funciona con tres archivos: dos binarios y uno de texto.

```
const string fileraw = "MQL5Book/cursor.raw";
const string filetxt = "MQL5Book/cursor.csv";
const string file100 = "MQL5Book/k100.raw";
```

Para simplificar el registro de la posición actual, además de los signos de fin de archivo (End-Of-File, EOF) y fin de línea (End-Of-Line, EOL), hemos creado una función de ayuda *FileState*.

```
string FileState(int handle)
{
   return StringFormat("P:%I64d, F:%s, L:%s",
      FileTell(handle),
      (string)FileIsEnding(handle),
      (string)FileIsLineEnding(handle));
}
```

El escenario para probar las funciones en un archivo binario incluye los pasos que figuran a continuación.

Cree un archivo nuevo o abra un archivo *fileraw* existente («MQL5Book/cursor.raw») en modo lectura/escritura. Inmediatamente después de la apertura, y después de cada operación, mostramos el estado actual del archivo llamando a *FileState*.

```
void OnStart()
{
   int handle;
   Print("\n * Phase I. Binary file");
   handle = PRTF(FileOpen(fileraw, FILE_BIN | FILE_WRITE | FILE_READ));
   Print(FileState(handle));
   ...
```

Mueva el puntero al final del archivo, lo que nos permitirá añadir datos a este archivo cada vez que se ejecute el script (y no sobrescribirlo desde el principio). La forma más obvia de referirse al final del archivo es *offset* nulo relativo a *origin=SEEK_END*.

```
PRTF(FileSeek(handle, 0, SEEK_END));
Print(FileState(handle));
```

Si el archivo ya no está vacío (no es nuevo), podemos leer los datos existentes en su posición arbitraria (relativa o absoluta). En concreto, si el parámetro *origin* de la función *FileSeek* es igual a SEEK_CUR, significa que con un *offset* negativo la posición actual se moverá el número correspondiente de bytes hacia atrás (a la izquierda), y con positivo se moverá hacia delante (a la derecha).

En este ejemplo estamos intentando retroceder el tamaño de un valor de tipo *int*. Un poco más adelante veremos que en este lugar debe haber un campo *day_of_year* (último campo) de la estructura *MqlDateTime*, ya que los escribimos en un archivo en instrucciones posteriores, y estos datos están disponibles en el archivo en la siguiente ejecución. El valor leído se registra para compararlo con el que se guardó anteriormente.

```
if(PRTF(FileSeek(handle, -1 * sizeof(int), SEEK_CUR)))
{
   Print(FileState(handle));
   PRTF(FileReadInteger(handle));
}
```

En un nuevo archivo vacío, la llamada a *FileSeek* terminará con el error 4003 (INVALID_PARAMETER), y el bloque de sentencia *if* no se ejecutará.

A continuación, se rellena el archivo con datos. En primer lugar, se escribe la hora local actual del ordenador (8 bytes de *datetime*) con *FileWriteLong*.

```
datetime now = TimeLocal();
PRTF(FileWriteLong(handle, now));
Print(FileState(handle));
```

A continuación, intentamos retroceder desde la ubicación actual 4 bytes (-4) y leemos *long*.

```
PRTF(FileSeek(handle, -4, SEEK_CUR));
long x = PRTF(FileReadLong(handle));
Print(FileState(handle));
```

Este intento terminará con el error 5015 (FILE_READERROR), porque estábamos al final del archivo y después de desplazar 4 bytes a la izquierda no podemos leer 8 bytes de la derecha (tamaño *long*). No obstante, como veremos en el registro, como resultado de este intento fallido el puntero se moverá de nuevo al final del archivo.

Si retrocede 8 bytes (-8), la lectura posterior del valor *long* se realizará correctamente, y ambos valores de tiempo, incluido el original y el recibido del archivo, deben coincidir.

```
PRTF(FileSeek(handle, -8, SEEK_CUR));
Print(FileState(handle));
x = PRTF(FileReadLong(handle));
PRTF((now == x));
```

Por último, escriba en el archivo la estructura *MqlDateTime* rellenada con la misma hora. La posición en el archivo aumentará en 32 (el tamaño de la estructura en bytes).

```
MqlDateTime mdt;
TimeToStruct(now, mdt);
StructPrint(mdt); // display the date/time in the log visually
PRTF(FileWriteStruct(handle, mdt)); // 32 = sizeof(MqlDateTime)
Print(FileState(handle));
FileClose(handle);
```

Después de la primera ejecución del script para el escenario con el archivo *fileraw* (MQL5Book/cursor.raw) obtenemos algo como lo siguiente (el tiempo será diferente):

```
first run
 * Phase I. Binary file
FileOpen(fileraw,FILE_BIN|FILE_WRITE|FILE_READ)=1 / ok
P:0, F:true, L:false
FileSeek(handle,0,SEEK_END)=true / ok
P:0, F:true, L:false
FileSeek(handle,-1*sizeof(int),SEEK_CUR)=false / INVALID_PARAMETER(4003)
FileWriteLong(handle,now)=8 / ok
P:8, F:true, L:false
FileSeek(handle,-4,SEEK_CUR)=true / ok
FileReadLong(handle)=0 / FILE_READERROR(5015)
P:8, F:true, L:false
FileSeek(handle,-8,SEEK_CUR)=true / ok
P:0, F:false, L:false
FileReadLong(handle)=1629683392 / ok
(now==x)=true / ok
  2021     8    23      1    49    52             1           234
FileWriteStruct(handle,mdt)=32 / ok
P:40, F:true, L:false
```

Según el estado, el tamaño del archivo es inicialmente cero porque la posición es «P:0» tras el desplazamiento al final del archivo («F:true»). Después de cada grabación (utilizando *FileWriteLong* y *FileWriteStruct*), la posición P se incrementa en el tamaño de los datos escritos.

Tras la segunda ejecución del script podrá observar algunos cambios en el registro:

```
second run
 * Phase I. Binary file
FileOpen(fileraw,FILE_BIN|FILE_WRITE|FILE_READ)=1 / ok
P:0, F:false, L:false
FileSeek(handle,0,SEEK_END)=true / ok
P:40, F:true, L:false
FileSeek(handle,-1*sizeof(int),SEEK_CUR)=true / ok
P:36, F:false, L:false
FileReadInteger(handle)=234 / ok
FileWriteLong(handle,now)=8 / ok
P:48, F:true, L:false
FileSeek(handle,-4,SEEK_CUR)=true / ok
FileReadLong(handle)=0 / FILE_READERROR(5015)
P:48, F:true, L:false
FileSeek(handle,-8,SEEK_CUR)=true / ok
P:40, F:false, L:false
FileReadLong(handle)=1629683397 / ok
(now==x)=true / ok
  2021     8    23      1    49    57             1           234
FileWriteStruct(handle,mdt)=32 / ok
P:80, F:true, L:false
```

En primer lugar, el tamaño del archivo tras la apertura es 40 (según la posición «P:40» tras el desplazamiento al final del archivo). Cada vez que se ejecute el script, el archivo crecerá en 40 bytes.

En segundo lugar, como el archivo no está vacío, es posible navegar por él y leer los datos «antiguos». En particular, después de retroceder a *-1*sizeof(int)* desde la posición actual (que es también el final del archivo), leemos con éxito el valor 234, que es el último campo de la estructura *MqlDateTime* (es el número del día de un año y lo más probable es que sea diferente para usted).

El segundo escenario de prueba funciona con el archivo de texto csv *filetxt* (MQL5Book/cursor.csv). También lo abriremos en el modo combinado de lectura y escritura, pero no moveremos el puntero al final del archivo. Por eso, cada ejecución del script sobrescribirá los datos, comenzando desde el principio del archivo. Para facilitar la detección de las diferencias, los números de la primera columna del CSV se generan aleatoriamente. En la segunda columna siempre se sustituyen las mismas cadenas de la plantilla en la función *StringFormat*.

```
Print(" * Phase II. Text file");
srand(GetTickCount());
// create a new file or open an existing file for writing/overwriting
// from the very beginning and subsequent reading; inside CSV data (Unicode)
handle = PRTF(FileOpen(filetxt, FILE_CSV | FILE_WRITE | FILE_READ, ','));
// three rows of data (number,string pair in each), separated by '\n'
// note that the last element does not end with a newline '\n'
// this is optional, but allowed
string content = StringFormat(
   "%02d,abc\n%02d,def\n%02d,ghi",
   rand() % 100, rand() % 100, rand() % 100);
// '\n' will be replaced with '\r\n' automatically, thanks to FileWriteString
PRTF(FileWriteString(handle, content));
```

He aquí un ejemplo de los datos generados:

```
34,abc
20,def
02,ghi
```

A continuación, volvemos al principio del archivo y lo leemos en un bucle con *FileReadString*, registrando constantemente el estado.

```
PRTF(FileSeek(handle, 0, SEEK_SET));
Print(FileState(handle));
// count the lines in the file using the FileIsLineEnding feature
int lineCount = 0;
while(!FileIsEnding(handle))
{
   PRTF(FileReadString(handle));
   Print(FileState(handle));
   // FileIsLineEnding also equals true when FileIsEnding equals true,
   // even if there is no trailing '\n' character
   if(FileIsLineEnding(handle)) lineCount++;
}
FileClose(handle);
PRTF(lineCount);
```

A continuación se muestran los registros del archivo *filetxt* después de la primera y segunda ejecución del script. En primer lugar la primera:

```
first run
 * Phase II. Text file
FileOpen(filetxt,FILE_CSV|FILE_WRITE|FILE_READ,',')=1 / ok
FileWriteString(handle,content)=44 / ok
FileSeek(handle,0,SEEK_SET)=true / ok
P:0, F:false, L:false
FileReadString(handle)=08 / ok
P:8, F:false, L:false
FileReadString(handle)=abc / ok
P:18, F:false, L:true
FileReadString(handle)=37 / ok
P:24, F:false, L:false
FileReadString(handle)=def / ok
P:34, F:false, L:true
FileReadString(handle)=96 / ok
P:40, F:false, L:false
FileReadString(handle)=ghi / ok
P:46, F:true, L:true
lineCount=3 / ok
```

Y aquí está la segunda:

```
second run
 * Phase II. Text file
FileOpen(filetxt,FILE_CSV|FILE_WRITE|FILE_READ,',')=1 / ok
FileWriteString(handle,content)=44 / ok
FileSeek(handle,0,SEEK_SET)=true / ok
P:0, F:false, L:false
FileReadString(handle)=34 / ok
P:8, F:false, L:false
FileReadString(handle)=abc / ok
P:18, F:false, L:true
FileReadString(handle)=20 / ok
P:24, F:false, L:false
FileReadString(handle)=def / ok
P:34, F:false, L:true
FileReadString(handle)=02 / ok
P:40, F:false, L:false
FileReadString(handle)=ghi / ok
P:46, F:true, L:true
lineCount=3 / ok
```

Como puede ver, el archivo no cambia de tamaño, pero se escriben números diferentes en los mismos desplazamientos. Como este archivo CSV tiene dos columnas, después de cada segundo valor que leemos vemos una bandera EOL («L:true») inclinada.

El número de líneas detectadas es 3, a pesar de que sólo hay 2 caracteres de nueva línea en el archivo: la última (tercera) línea termina con el archivo.

Para terminar, el último escenario de prueba utiliza el archivo *file100* (MQL5Book/k100.raw) para mover el puntero más allá del final del archivo (hasta la marca de 1000000 bytes), y así aumentar su tamaño (reserva espacio en disco para posibles futuras operaciones de escritura).

```
Print(" * Phase III. Allocate large file");
handle = PRTF(FileOpen(file100, FILE_BIN | FILE_WRITE));
PRTF(FileSeek(handle, 1000000, SEEK_END));
// to change the size, you need to write at least something
PRTF(FileWriteInteger(handle, 0xFF, 1));
PRTF(FileTell(handle));
FileClose(handle);
```

La salida de registro para este script no cambia de una ejecución a otra; sin embargo, los datos aleatorios que terminan en el espacio asignado para el archivo pueden diferir (su contenido no se muestra aquí: utilice un visor binario externo).

```
 * Phase III. Allocate large file
FileOpen(file100,FILE_BIN|FILE_WRITE)=1 / ok
FileSeek(handle,1000000,SEEK_END)=true / ok
FileWriteInteger(handle,0xFF,1)=1 / ok
FileTell(handle)=1000001 / ok
```

4.5.11 Obtención de las propiedades de un archivo

En el proceso de trabajo con archivos, además de escribir y leer datos directamente, a menudo es necesario analizar sus propiedades. Una de las principales propiedades, el tamaño del archivo, puede

obtenerse utilizando la función *FileSize*. Pero hay algunas características más que pueden solicitarse utilizando *FileGetInteger*.

Tenga en cuenta que la función *FileSize* requiere un gestor de archivo abierto. *FileGetInteger* tiene algunas propiedades, incluido el tamaño, que pueden reconocerse por el nombre del archivo, y no es necesario abrirlo primero.

ulong FileSize(int handle)

La función devuelve el tamaño de un archivo abierto por su descriptor. En caso de error, el resultado es igual a 0, que es un tamaño válido para la ejecución normal de la función, por lo que siempre debe analizar los posibles errores utilizando *_LastError* (o *GetLastError*).

El tamaño del archivo también puede obtenerse desplazando el puntero al final del archivo *FileSeek(handle, 0, SEEK_END)* y llamando a *FileTell(handle)*. Estas dos funciones se describen en la sección anterior.

long FileGetInteger(int handle, ENUM_FILE_PROPERTY_INTEGER property)
long FileGetInteger(const string filename, ENUM_FILE_PROPERTY_INTEGER property, bool common = false)

La función tiene dos opciones: trabajar a través de un descriptor de archivo abierto, y por el nombre del archivo (incluyendo uno cerrado).

La función devuelve una de las propiedades de archivo especificadas en el parámetro *property*. La lista de propiedades válidas es diferente para cada una de las opciones (véase más abajo). Aunque el tipo de valor es *long*, dependiendo de la propiedad solicitada, puede contener no sólo un número entero sino también *datetime* o *bool*: realice la conversión de tipos requerida explícitamente.

Cuando solicite una propiedad por el nombre del archivo puede utilizar de forma adicional el parámetro *common* para especificar en qué carpeta debe buscarse el archivo: la carpeta actual del terminal *MQL5/Files* (*false*, por defecto) o la carpeta común *Users/<user_name>...MetaQuotes/Terminal/Common/Files* (*true*). Si el programa MQL se está ejecutando en el probador, el directorio de trabajo se encuentra dentro de la carpeta del agente de pruebas (*Tester/<agent>/MQL5/Files*), véase la introducción del capítulo Trabajar con archivos.

La siguiente tabla enumera todos los miembros de ENUM_FILE_PROPERTY_INTEGER.

Propiedad	Descripción
FILE_EXISTS *	Comprobación de existencia (similar a FileIsExist)
FILE_CREATE_DATE *	Fecha de creación
FILE_MODIFY_DATE *	Fecha de la última modificación
FILE_ACCESS_DATE *	Fecha de último acceso
FILE_SIZE *	Tamaño del archivo en bytes (similar a FileSize)
FILE_POSITION	Posición del puntero en el archivo (similar a FileTell)
FILE_END	Posición al final del archivo (similar a FileIsEnding)
FILE_LINE_END	Posición al final de una cadena (similar a FileIsLineEnding)
FILE_IS_COMMON	Archivo abierto en la carpeta compartida de terminales (FILE_COMMON)
FILE_IS_TEXT	Archivo abierto como texto (FILE_TXT)
FILE_IS_BINARY	Archivo abierto como binario (FILE_BIN)
FILE_IS_CSV	Archivo abierto como CSV (FILE_CSV)
FILE_IS_ANSI	Archivo abierto como ANSI (FILE_ANSI)
FILE_IS_READABLE	Archivo abierto para lectura (FILE_READ)
FILE_IS_WRITABLE	Archivo abierto para escritura (FILE_WRITE)

Las propiedades cuyo uso está permitido por nombre de archivo están marcadas con un asterisco. Si intenta obtener otras propiedades, la segunda versión de la función devolverá un error 4003 (INVALID_PARAMETER).

Algunas propiedades pueden cambiar mientras se trabaja con un archivo abierto: FILE_MODIFY_DATE, FILE_ACCESS_DATE, FILE_SIZE, FILE_POSITION, FILE_END, FILE_LINE_END (sólo para archivos de texto).

En caso de error, el resultado de la llamada es -1.

La segunda versión de la función permite comprobar si el nombre especificado es el nombre de un archivo o directorio. Si se especifica un directorio al obtener propiedades por nombre, la función establecerá un código de error interno especial 5018 (ERR_MQL_FILE_IS_DIRECTORY), mientras que el valor devuelto será correcto.

Probaremos las funciones de esta sección utilizando el script *FileProperties.mq5*. Funcionará en un archivo con un nombre predefinido.

```
const string fileprop = "MQL5Book/fileprop";
```

Al principio de *OnStart*, vamos a intentar solicitar el tamaño mediante un descriptor erróneo (no se recibió a través de la llamada a *File Open*). Después de *FileSize* se requiere la comprobación de la variable *_LastError*, y *FileGetInteger* devuelve inmediatamente un valor especial, un indicador de error (-1).

```
void OnStart()
{
   int handle = 0;
   ulong size = FileSize(handle);
   if(_LastError)
   {
      Print("FileSize error=", E2S(_LastError) + "(" + (string)_LastError + ")");
      // We will get: FileSize 0, error=WRONG_FILEHANDLE(5008)
   }

   PRTF(FileGetInteger(handle, FILE_SIZE)); // -1 / WRONG_FILEHANDLE(5008)
```

A continuación, creamos un nuevo archivo o abrimos un archivo existente y lo reiniciamos, y luego escribimos el texto de prueba.

```
   handle = PRTF(FileOpen(fileprop, FILE_TXT | FILE_WRITE | FILE_ANSI)); // 1
   PRTF(FileWriteString(handle, "Test Text\n")); // 11
```

Solicitamos selectivamente algunas de las propiedades.

```
   PRTF(FileGetInteger(fileprop, FILE_SIZE)); // 0, not written to the disk yet
   PRTF(FileGetInteger(handle, FILE_SIZE)); // 11
   PRTF(FileSize(handle)); // 11
   PRTF(FileGetInteger(handle, FILE_MODIFY_DATE)); //1629730884, number of seconds si
   PRTF(FileGetInteger(handle, FILE_IS_TEXT)); // 1, bool true
   PRTF(FileGetInteger(handle, FILE_IS_BINARY)); // 0, bool false
```

La información sobre la longitud del archivo por su descriptor tiene en cuenta el búfer de caché actual, y por el nombre del archivo, la longitud real estará disponible sólo después de que se cierre el archivo, o si se llama a la función *FileFlush* (véase la sección Forzar escritura de caché en disco).

La función devuelve fechas y horas como el número de segundos de la época estándar desde el 1 de enero de 1970, que corresponde al tipo *datetime* y puede ser llevada a él.

La solicitud de banderas de apertura de archivos (su modo) tiene éxito para la versión de la función con un descriptor; en concreto, recibimos una respuesta que el archivo es de texto y no binario. Sin embargo, la siguiente petición similar de un nombre de archivo fallará porque la propiedad sólo se admite cuando se pasa un manejador válido. Esto ocurre aunque el nombre apunte al mismo archivo que hemos abierto.

```
   PRTF(FileGetInteger(fileprop, FILE_IS_TEXT)); // -1 / INVALID_PARAMETER(4003)
```

Esperemos un segundo, cerremos el archivo y volvamos a comprobar la fecha de modificación (esta vez por el nombre, ya que el descriptor ya no es válido).

```
   Sleep(1000);
   FileClose(handle);
   PRTF(FileGetInteger(fileprop, FILE_MODIFY_DATE)); // 1629730885 / ok
```

Aquí puedes ver claramente que el tiempo ha aumentado en 1.

Por último, asegúrese de que las propiedades están disponibles para los directorios (carpetas).

```
PRTF((datetime)FileGetInteger("MQL5Book", FILE_CREATE_DATE));
// We will get: 2021.08.09 22:38:00 / FILE_IS_DIRECTORY(5018)
```

Dado que todos los ejemplos del libro se encuentran en la carpeta «MQL5Book», esta ya debe existir. Sin embargo, su tiempo de creación real será diferente. El código de error FILE_IS_DIRECTORY en este caso nos lo muestra la macro PRTF. En el programa de trabajo, la llamada a la función debe realizarse sin macro y, a continuación, debe leerse el código en *_LastError*.

4.5.12 Forzar escritura de caché en disco

La escritura y lectura de archivos en MQL5 se realiza en la caché, lo que significa que se mantiene un cierto búfer en memoria para los datos y ello aumenta la eficacia del trabajo. Así, los datos transferidos mediante llamadas a funciones durante la escritura llegan al búfer de salida, y sólo después de que esté lleno, tiene lugar la escritura física en el disco. Al leer, por el contrario, se leen más datos del disco en el búfer que los que el programa solicitó mediante funciones (si no es el final del archivo), y las operaciones de lectura posteriores (que son muy probables) son más rápidas.

El almacenamiento en caché es una tecnología estándar utilizada en la mayoría de las aplicaciones y al nivel del propio sistema operativo. No obstante, además de sus pros, el almacenamiento en caché también tiene sus contras.

En concreto, si los archivos se utilizan como medio de intercambio de datos entre programas, el retraso en la escritura puede ralentizar considerablemente la comunicación y hacerla menos predecible, ya que el tamaño del búfer puede ser bastante grande y la frecuencia de su «volcado» al disco puede ajustarse según algunos algoritmos.

Por ejemplo, en MetaTrader 5 existe toda una categoría de programas MQL para copiar señales de trading de una instancia del terminal a otra. Suelen utilizar archivos para transferir información, y para ellos es muy importante que el almacenamiento en caché no ralentice las cosas. Para este caso, MQL5 proporciona la función *FileFlush*.

void FileFlush(int handle)

La función realiza un vaciado forzado a disco de todos los datos que quedan en el búfer de E/S del archivo con el descriptor *handle*.

Si no utiliza esta función, entonces parte de los datos «enviados» desde el programa pueden, en el peor de los casos, llegar al disco sólo cuando se cierra el archivo.

Esta función ofrece mayores garantías de seguridad de los datos valiosos en caso de imprevistos (como un bloqueo del sistema operativo o del programa). No obstante, por otro lado, no se recomiendan las llamadas frecuentes a *FileFlush* durante la grabación masiva, ya que pueden afectar negativamente al rendimiento.

Si el archivo se abre en modo mixto, simultáneamente para escritura y lectura, se debe llamar a la función *FileFlush* entre lecturas y escrituras en el archivo.

Como ejemplo, analicemos el script *FileFlush.mq5*, en el que implementamos dos modos que simulan el funcionamiento de la copiadora de transacciones. Tendremos que ejecutar dos instancias del script en gráficos diferentes, con una de ellas como emisora de datos y la otra como receptora.

El script tiene dos parámetros de entrada: *EnableFlashing* permite comparar las acciones de los programas utilizando la función *FileFlush* y sin ella, y *UseCommonFolder* indica la necesidad de crear un archivo que actúe como medio de transferencia de datos, a elegir entre hacerlo en la carpeta de la

instancia actual del terminal o en una carpeta compartida (en este último caso, se puede probar la transferencia de datos entre diferentes terminales).

```
#property script_show_inputs
input bool EnableFlashing = false;
input bool UseCommonFolder = false;
```

Recuerde que, para que aparezca un diálogo con variables de entrada cuando se lanza el script, debe establecer adicionalmente la propiedad *script_show_inputs*.

El nombre del archivo de tránsito se especifica en la variable *dataport*. La opción *UseCommonFolder* controla la bandera FILE_COMMON añadida al conjunto de conmutadores de modo para archivos abiertos en la función *File Open*.

```
const string dataport = "MQL5Book/dataport";
const int flag = UseCommonFolder ? FILE_COMMON : 0;
```

La función principal *OnStart* consta en realidad de dos partes: la configuración del archivo abierto y un bucle que envía o recibe datos periódicamente.

Necesitaremos ejecutar dos instancias del script, y cada una tendrá su propio descriptor de archivo apuntando al mismo archivo en disco pero abierto en diferentes modos.

```
void OnStart()
{
   bool modeWriter = true; // by default the script should write data
   int count = 0;          // number of writes/reads made
   // create a new or reset the old file in read mode, as a "sender"
   int handle = PRTF(FileOpen(dataport,
      FILE_BIN | FILE_WRITE | FILE_SHARE_READ | flag));
   // if writing is not possible, most likely another instance of the script is alrea
   // so we try to open it for reading
   if(handle == INVALID_HANDLE)
   {
      // if it is possible to open the file for reading, we will continue to work as
      handle = PRTF(FileOpen(dataport,
         FILE_BIN | FILE_READ | FILE_SHARE_WRITE | FILE_SHARE_READ | flag));
      if(handle == INVALID_HANDLE)
      {
         Print("Can't open file"); // something is wrong
         return;
      }
      modeWriter = false; // switch model/role
   }
```

Al principio, estamos intentando abrir el archivo en modo FILE_WRITE, sin compartir el permiso de escritura (FILE_SHARE_WRITE), por lo que la primera instancia del script en ejecución capturará el archivo e impedirá que la segunda trabaje en modo escritura. La segunda instancia obtendrá un error e INVALID_HANDLE después de la primera llamada a *FileOpen* e intentará abrir el archivo en modo lectura (FILE_READ) con la segunda llamada a *FileOpen* utilizando la bandera de escritura paralela FILE_SHARE_WRITE lo que, idealmente, debería funcionar. A continuación, la variable *modeWriter* se establecerá en *false* para indicar la función real del script.

El bucle operativo principal tiene la siguiente estructura:

```
while(!IsStopped())
{
   if(modeWriter)
   {
      // ...write test data
   }
   else
   {
      // ...read test data
   }
   Sleep(5000);
}
```

El bucle se ejecutará hasta que el usuario elimine manualmente el script del gráfico: esto se indicará mediante la función *IsStopped*. Dentro del bucle, la acción se activa cada 5 segundos llamando a la función *Sleep*, que «congela» el programa durante el número de milisegundos especificado (5000 en este caso). Esto se hace para facilitar el análisis de los cambios en curso y evitar registros de estado demasiado frecuentes. En un programa real sin registros detallados, puede enviar datos cada 100 milisegundos o incluso con mayor frecuencia.

Los datos transmitidos incluirán la hora actual (un valor *datetime*, 8 bytes). En la primera rama de la instrucción *if(modeWriter)*, donde se escribe el archivo, llamamos a *FileWriteLong* con la última cuenta (obtenida de la función *TimeLocal*), aumentamos el contador de operaciones en 1 (*count*++) y enviamos el estado actual al registro.

```
long temp = TimeLocal(); // get the current local time datetime
FileWriteLong(handle, temp); // append it to the file (every 5 seconds)
count++;
if(EnableFlashing)
{
   FileFlush(handle);
}
Print(StringFormat("Written[%d]: %I64d", count, temp));
```

Es importante señalar que la llamada a la función *FileFlush* después de cada entrada sólo se realiza si el parámetro de entrada *EnableFlashing* se establece en *true*.

En la segunda rama del operador *if*, en la que leemos los datos, primero reiniciamos la bandera de error interno llamando a *ResetLastError*. Esto es necesario porque vamos a leer los datos del archivo mientras haya datos. Cuando no haya más datos para leer, el programa obtendrá un código de error específico 5015 (ERR_FILE_READERROR).

Dado que los temporizadores integrados en MQL5, incluida la función *Sleep*, tienen una precisión limitada (aproximadamente 10 ms), no podemos excluir que se produjeran dos escrituras consecutivas entre dos intentos consecutivos de leer un archivo. Por ejemplo, una lectura se produjo a las 10:00:00'200 y la segunda, a las 10:00:05'210 (en la notación *"hours:minutes:seconds'milliseconds"*). En este caso se produjeron dos grabaciones en paralelo: una a las 10:00:00'205 y la segunda a las 10:00:05'205, y ambas cayeron en el periodo mencionado. Esta situación es poco probable pero posible. Incluso con intervalos de tiempo absolutamente precisos, el sistema en tiempo ejecución de MQL5 puede verse obligado a elegir entre dos scripts en ejecución (cuál invocar antes que el otro) si el número total de programas es grande y no hay suficientes núcleos de procesador para todos ellos.

MQL5 proporciona temporizadores de alta precisión (hasta microsegundos), pero esto no es crítico para la tarea actual.

El bucle anidado es necesario por otra razón más. Inmediatamente después de lanzar el script como «receptor» de datos, debe procesar todos los registros del archivo que se hayan acumulado desde el lanzamiento del «emisor» (es poco probable que ambos scripts puedan lanzarse simultáneamente). Probablemente alguien preferiría un algoritmo diferente: omitir todos los registros «antiguos» y registrar sólo los nuevos. Esto puede hacerse, pero aquí se aplica la opción «sin pérdidas».

```
ResetLastError();
while(true)// loop as long as there is data and no problems
{
   bool reportedEndBeforeRead = FileIsEnding(handle);
   ulong reportedTellBeforeRead = FileTell(handle);

   temp = FileReadLong(handle);
   // if there is no more data, we will get an error 5015 (ERR_FILE_READERRO
   if(_LastError)break; // exit the loop on any error

   // here the data is received without errors
   count++;
   Print(StringFormat("Read[%d]: %I64d\t"
      "(size=%I64d, before=%I64d(%s), after=%I64d)",
      count, temp,
      FileSize(handle), reportedTellBeforeRead,
      (string)reportedEndBeforeRead, FileTell(handle)));
}
```

Tenga en cuenta lo siguiente: los metadatos sobre el archivo abierto para lectura, como su tamaño, devueltos por la función *FileSize* (véase Obtención de las propiedades de un archivo) no cambian después de abrir el archivo. Si más tarde otro programa añade algo al archivo que abrimos para lectura, la longitud «detectable» del mismo no se actualizará aunque llamemos a *FileFlash* para el descriptor de lectura. Sería posible cerrar y volver a abrir el archivo (antes de cada lectura, pero esto no es eficaz), ya que entonces aparecería la nueva longitud para el nuevo descriptor. Pero prescindiremos de él, con la ayuda de otro truco.

El truco consiste en seguir leyendo datos utilizando funciones de lectura (en nuestro caso *FileReadLong*) mientras devuelvan datos sin errores. Es importante no utilizar otras funciones que operen sobre metadatos. En concreto, debido al hecho de que el fin de archivo de sólo lectura permanece constante, la comprobación con la función *FileIsEnding* (véase Control de posición dentro de un archivo) dará *true* en la posición antigua, a pesar de la posible reposición del archivo desde otro proceso. Además, un intento de mover el puntero del archivo interno hasta el final (*FileSeek(handle, 0, SEEK_END)*; véase para la función *FileSeek* la misma sección) no saltará al final real de los datos, sino a la posición obsoleta en la que se encontraba el final en el momento de la apertura.

La función nos indica la posición real dentro del archivo *FileTell* (véase la misma sección). A medida que se añada información al archivo desde otra instancia del script y se lea en este bucle, el puntero se moverá cada vez más a la derecha, superando, por extraño que sea, *FileSize*. Para una demostración visual de cómo el puntero se mueve más allá del tamaño del archivo, vamos a guardar sus valores antes y después de llamar a *FileReadLong*, y luego la salida de los valores junto con el tamaño del registro.

Cuando la lectura con *FileReadLong* genere algún error, el bucle interno se romperá. La salida del bucle normal implica el error 5015 (ERR_FILE_READERROR). En concreto, se produce cuando no hay datos disponibles para la lectura en la posición actual del archivo.

Los últimos datos leídos con éxito se envían al registro, y es fácil compararlos con lo que el script remitente produce allí.

Vamos a ejecutar un nuevo script dos veces. Para distinguir entre sus copias, lo haremos en los gráficos de diferentes instrumentos.

> Al ejecutar ambos scripts, es importante observar el mismo valor del parámetro *UseCommonFolder*. Dejémoslo en nuestras pruebas igual a *false*, ya que lo haremos todo en un solo terminal. Se sugiere la transferencia de datos entre terminales diferentes con *UseCommonFolder* configurado en *true* para realizar pruebas independientes.

En primer lugar, vamos a ejecutar la primera instancia en el gráfico EURUSD,H1, dejando todos los ajustes por defecto, incluido *EnableFlashing=false*. A continuación, ejecutaremos la segunda instancia en el gráfico XAUUSD,H1 (también con la configuración predeterminada). El registro será el siguiente (su tiempo será diferente):

```
(EURUSD,H1) *
(EURUSD,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=1 / ok
(EURUSD,H1) Written[1]: 1629652995
(XAUUSD,H1) *
(XAUUSD,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=-1 / CANNOT_O
(XAUUSD,H1) FileOpen(dataport,FILE_BIN|FILE_READ|FILE_SHARE_WRITE|FILE_SHARE_READ|fla
(EURUSD,H1) Written[2]: 1629653000
(EURUSD,H1) Written[3]: 1629653005
(EURUSD,H1) Written[4]: 1629653010
(EURUSD,H1) Written[5]: 1629653015
```

El remitente abrió correctamente el archivo para escritura y comenzó a enviar datos cada 5 segundos, según las líneas con la palabra «Escrito» y los valores crecientes. Menos de 5 segundos después de que se iniciara el emisor, también se inició el receptor. Dio un mensaje de error porque no podía abrir el archivo para escribir. Pero luego abrió con éxito el archivo para su lectura. Sin embargo, no hay constancia de que haya podido encontrar en el archivo los datos transmitidos. Los datos se quedaban «colgados» en la caché del remitente.

Vamos a detener ambos scripts y a ejecutarlos de nuevo en la misma secuencia: primero, ejecutamos el emisor en EURUSD, y luego el receptor en XAUUSD. Pero esta vez especificaremos *EnableFlashing=true* para el remitente.

Esto es lo que ocurre en el registro:

```
(EURUSD,H1) *
(EURUSD,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=1 / ok
(EURUSD,H1) Written[1]: 1629653638
(XAUUSD,H1) *
(XAUUSD,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=-1 / CANNOT_O
(XAUUSD,H1) FileOpen(dataport,FILE_BIN|FILE_READ|FILE_SHARE_WRITE|FILE_SHARE_READ|fla
(XAUUSD,H1) Read[1]: 1629653638 (size=8, before=0(false), after=8)
(EURUSD,H1) Written[2]: 1629653643
(XAUUSD,H1) Read[2]: 1629653643 (size=8, before=8(true), after=16)
(EURUSD,H1) Written[3]: 1629653648
(XAUUSD,H1) Read[3]: 1629653648 (size=8, before=16(true), after=24)
(EURUSD,H1) Written[4]: 1629653653
(XAUUSD,H1) Read[4]: 1629653653 (size=8, before=24(true), after=32)
(EURUSD,H1) Written[5]: 1629653658
```

El mismo archivo se abre de nuevo con éxito en diferentes modos en ambos scripts, pero esta vez los valores escritos son leídos regularmente por el receptor.

Es interesante observar que antes de cada siguiente lectura de datos, excepto la primera, la función *FileIsEnding* devuelve *true* (que aparece en la misma cadena que los datos recibidos, entre paréntesis después de la cadena «antes»). Así, hay una señal de que estamos al final del archivo, pero entonces *FileReadLong* lee con éxito un valor supuestamente fuera del límite del archivo y desplaza la posición hacia la derecha. Por ejemplo, la entrada «size=8, before=8(true), after=16» significa que el tamaño del archivo se declara al programa MQL como 8, el puntero actual antes de la llamada a *FileReadLong* también es igual a 8 y el signo de fin de archivo se habilita. Después de llamar con éxito a *FileReadLong*, el puntero se mueve a 16. Sin embargo, en la siguiente iteración y en todas las demás, volvemos a ver «size=8», y el puntero se desplaza gradualmente cada vez más lejos del archivo.

Dado que la escritura en el emisor y la lectura en el receptor se producen una vez cada 5 segundos, en función de sus fases de desfase de bucle, podemos observar el efecto de un retraso diferente entre las dos operaciones, de hasta casi 5 segundos en el peor de los casos. Sin embargo, esto no significa que el vaciado de la caché sea tan lento. De hecho, es un proceso casi instantáneo. Para garantizar una detección de cambios más rápida, puede reducir el periodo de espera en los bucles (tenga en cuenta que esta prueba, si el retraso es demasiado corto, llenará rápidamente el registro: a diferencia de un programa real, aquí siempre se generan nuevos datos, ya que se trata de la hora actual del emisor al segundo más cercano).

Por cierto: puede ejecutar varios destinatarios, a diferencia del remitente, que debe ser sólo uno. El siguiente registro muestra el funcionamiento de un emisor en EURUSD y de dos receptores en los gráficos XAUUSD y USDRUB.

```
(EURUSD,H1) *
(EURUSD,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=1 / ok
(EURUSD,H1) Written[1]: 1629671658
(XAUUSD,H1) *
(XAUUSD,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=-1 / CANNOT_O
(XAUUSD,H1) FileOpen(dataport,FILE_BIN|FILE_READ|FILE_SHARE_WRITE|FILE_SHARE_READ|fla
(XAUUSD,H1) Read[1]: 1629671658 (size=8, before=0(false), after=8)
(EURUSD,H1) Written[2]: 1629671663
(USDRUB,H1) *
(USDRUB,H1) FileOpen(dataport,FILE_BIN|FILE_WRITE|FILE_SHARE_READ|flag)=-1 / CANNOT_O
(USDRUB,H1) FileOpen(dataport,FILE_BIN|FILE_READ|FILE_SHARE_WRITE|FILE_SHARE_READ|fla
(USDRUB,H1) Read[1]: 1629671658 (size=16, before=0(false), after=8)
(USDRUB,H1) Read[2]: 1629671663 (size=16, before=8(false), after=16)
(XAUUSD,H1) Read[2]: 1629671663 (size=8, before=8(true), after=16)
(EURUSD,H1) Written[3]: 1629671668
(USDRUB,H1) Read[3]: 1629671668 (size=16, before=16(true), after=24)
(XAUUSD,H1) Read[3]: 1629671668 (size=8, before=16(true), after=24)
(EURUSD,H1) Written[4]: 1629671673
(USDRUB,H1) Read[4]: 1629671673 (size=16, before=24(true), after=32)
(XAUUSD,H1) Read[4]: 1629671673 (size=8, before=24(true), after=32)
(EURUSD,H1) Written[5]: 1629671678
```

Cuando se lanzó el tercer script en USDRUB, ya había dos registros de 8 bytes en el archivo, por lo que el bucle interno realizó inmediatamente dos iteraciones desde *FileReadLong*, y el tamaño del archivo «parece» ser igual a 16.

4.5.13 Eliminación de un archivo y comprobación de su existencia

Comprobar si un archivo existe y eliminarlo son acciones críticas relacionadas con el sistema de archivos, es decir, con el entorno externo en el que «viven» los archivos. Hasta ahora, hemos visto funciones que manipulan el contenido interno de los archivos. A partir de esta sección, la atención se centrará en las funciones que gestionan los archivos como unidades indivisibles.

bool FileIsExist(const string filename, int flag = 0)

La función comprueba si existe un archivo con el nombre *filename* y devuelve *true* en caso afirmativo. El directorio de búsqueda se selecciona mediante el parámetro *flag*: si es 0 (valor por defecto), el archivo se busca en el directorio de la instancia del terminal actual (*MQL5/Files*); si *flag* es igual a FILE_COMMON, se comprueba el directorio común de todos los terminales *Users/<user>...MetaQuotes/Terminal/Common/Files*. Si el programa MQL se ejecuta en el probador, el directorio de trabajo se encuentra dentro de la carpeta del agente del probador (*Tester/<agent>/MQL5/Files*); véase una parte introductoria del capítulo Trabajar con archivos.

El nombre especificado puede no pertenecer a un archivo, sino a un directorio. En este caso, la función *FileIsExist* devolverá *false* y se registrará un pseudoerror 5018 (FILE_IS_DIRECTORY) en la variable *_LastError*.

bool FileDelete(const string filename, int flag = 0)

La función borra el archivo con el nombre especificado *filename*. El parámetro *flag* especifica la ubicación del archivo. Con el valor predeterminado, la eliminación se realiza en el directorio de trabajo de la instancia de terminal actual (*MQL5/Files*) o del agente del probador (*Tester/<agent>/MQL5/Files*) si el programa se está ejecutando en el probador. Si *flag* es igual a FILE_COMMON, el archivo debe estar ubicado en la carpeta común de todos los terminales (*/Terminal/Common/Files*).

La función devuelve un signo de éxito (*true*) o de error (*false*).

Esta función no permite borrar directorios. Para ello, utilice la función *FolderDelete* (véase Trabajar con carpetas).

Para ver cómo funcionan las funciones descritas, utilizaremos el script *FileExist.mq5*. Haremos varias manipulaciones con un archivo temporal.

```
const string filetemp = "MQL5Book/temp";
void OnStart()
{
   PRTF(FileIsExist(filetemp)); // false / FILE_NOT_EXIST(5019)
   PRTF(FileDelete(filetemp));  // false / FILE_NOT_EXIST(5019)

   int handle = PRTF(FileOpen(filetemp, FILE_TXT | FILE_WRITE | FILE_ANSI)); // 1

   PRTF(FileIsExist(filetemp)); // true
   PRTF(FileDelete(filetemp));  // false / CANNOT_DELETE_FILE(5006)

   FileClose(handle);

   PRTF(FileIsExist(filetemp)); // true
   PRTF(FileDelete(filetemp));  // true
   PRTF(FileIsExist(filetemp)); // false / FILE_NOT_EXIST(5019)

   PRTF(FileIsExist("MQL5Book")); // false / FILE_IS_DIRECTORY(5018)
   PRTF(FileDelete("MQL5Book"));  // false / FILE_IS_DIRECTORY(5018)
}
```

El archivo no existe inicialmente, por lo que ambas funciones, *FileIsExist* y *FileDelet*, devuelven *false*, y el código de error es 5019 (FILE_NOT_EXIST).

A continuación, creamos un archivo y la función *FileIsExist* informa de su presencia. No obstante, no se puede borrar porque está abierto y ocupado con nuestro proceso (código de error 5006, CANNOT_DELETE_FILE).

Una vez cerrado, el archivo puede borrarse.

Al final del script se comprueba el directorio «MQL5Book» y se intenta borrar. *FileIsExist* devuelve *false* porque no es un archivo; sin embargo, el código de error 5018 (FILE_IS_DIRECTORY) especifica que es un directorio.

4.5.14 Copia y desplazamiento de archivos

Las principales operaciones con archivos a nivel del sistema de archivos son copiar y mover. Para estos fines, MQL5 implementa dos funciones con prototipos idénticos.

bool FileCopy(const string source, int flag, const string destination, int mode)

La función copia el archivo *source* en el archivo *destination*. Ambos parámetros mencionados pueden contener sólo nombres de archivo, o nombres junto con rutas con prefijo (jerarquías de carpetas) en los entornos virtuales de MQL5. Los parámetros *flag* y *mode* determinan en qué carpeta de trabajo se busca el archivo de origen y cuál es la carpeta de trabajo de destino: 0 significa que se trata de una

carpeta de la instancia local actual del terminal (o del agente del probador, si el programa se está ejecutando en el probador), y el valor FILE_COMMON indica la carpeta común para todos los terminales.

Además, en el parámetro *mode* puede especificar opcionalmente la constante FILE_REWRITE (si necesita combinar FILE_REWRITE y FILE_COMMON; esto se hace utilizando el operador a nivel de bits OR (|)). En ausencia de FILE_REWRITE está prohibido copiar sobre un archivo existente. En otras palabras: si el archivo con la ruta y el nombre especificados en el parámetro *destination* ya existe, debe confirmar su intención de sobrescribirlo estableciendo FILE_REWRITE. Si no se hace así, la llamada a la función fallará.

La función devuelve *true* si se completa con éxito, o *false* en caso de error.

La copia puede fallar si el archivo de origen o destino está ocupado (abierto) por otro proceso.

> Al copiar archivos suelen guardarse sus metadatos (hora de creación, derechos de acceso, flujos de datos alternativos). Si necesita realizar una copia «pura» únicamente de los datos del propio archivo, puede utilizar las llamadas sucesivas *FileLoad* y *FileSave*, véase Escritura y lectura de archivos en modo simplificado.

bool FileMove(const string source, int flag, const string destination, int mode)

La función mueve o renombra un archivo. La ruta de origen y el nombre se especifican en el parámetro *source*, y la ruta de destino y el nombre se especifican en *destination*.

La lista de parámetros y sus principios de funcionamiento son los mismos que para la función *FileCopy*. A grandes rasgos, *FileMove* hace el mismo trabajo que *FileCopy*, pero además borra el archivo original después de que la copia se realiza correctamente.

Vamos a aprender a trabajar con funciones en la práctica utilizando el script *FileCopy.mq5*. Tiene dos variables con los nombres de los archivos. Ambos archivos no existen cuando se ejecuta el script.

```
const string source = "MQL5Book/source";
const string destination = "MQL5Book/destination";
```

En *OnStart* realizamos una secuencia de acciones según un escenario sencillo. En primer lugar, intentamos copiar el archivo *source* del directorio de trabajo local al archivo *destination* del directorio general. Como era de esperar, obtenemos *false*, y el código de error en *_LastError* será 5019 (FILE_NOT_EXIST).

```
void OnStart()
{
   PRTF(FileCopy(source, 0, destination, FILE_COMMON)); // false / FILE_NOT_EXIST(501
   ...
```

Por lo tanto, crearemos un archivo fuente de la forma habitual, escribiremos algunos datos y los descargaremos en el disco.

```
   int handle = PRTF(FileOpen(source, FILE_TXT | FILE_WRITE)); // 1
   PRTF(FileWriteString(handle, "Test Text\n")); // 22
   FileFlush(handle);
```

Como el archivo se dejó abierto y no se especificó el permiso FILE_SHARE_READ al abrirlo, el acceso al mismo de otras formas (saltándose el manejador) sigue bloqueado. Por lo tanto, el siguiente intento de copia volverá a fallar.

```
   PRTF(FileCopy(source, 0, destination, FILE_COMMON)); // false / CANNOT_OPEN_FILE(5
```

Vamos a cerrar el archivo e intentarlo de nuevo. Pero antes, obtengamos las propiedades del archivo resultante en el registro: cuándo se creó y cuándo se modificó. Ambas propiedades contendrán la marca de tiempo actual de su ordenador.

```
   FileClose(handle);
   PRTF(FileGetInteger(source, FILE_CREATE_DATE)); // 1629757115, example
   PRTF(FileGetInteger(source, FILE_MODIFY_DATE)); // 1629757115, example
```

Esperemos 3 segundos antes de llamar a *FileCopy*. Esto le permitirá ver la diferencia en las propiedades del archivo original y su copia. Esta pausa no tiene nada que ver con el bloqueo previo del archivo: podríamos copiar inmediatamente después de cerrar el archivo, o incluso mientras lo escribimos si la opción FILE_SHARE_READ estaba activada.

```
   Sleep(3000);
```

Copiemos el archivo. Esta vez la operación tiene éxito. Veamos las propiedades de la copia.

```
   PRTF(FileCopy(source, 0, destination, FILE_COMMON)); // true
   PRTF(FileGetInteger(destination, FILE_CREATE_DATE, true)); // 1629757118, +3 secon
   PRTF(FileGetInteger(destination, FILE_MODIFY_DATE, true)); // 1629757115, example
```

Cada archivo tiene su propio tiempo de creación (para una copia es tres segundos más tarde que para el original), pero el tiempo de modificación es el mismo (la copia ha heredado las propiedades del original).

Ahora vamos a intentar mover la copia de nuevo a la carpeta local. No se puede hacer sin la opción FILE_REWRITE porque no hay permiso para sobrescribir el archivo original.

```
   PRTF(FileMove(destination, FILE_COMMON, source, 0)); // false / FILE_CANNOT_REWRIT
```

Cambiando el valor del parámetro conseguiremos que la transferencia de archivos se realice correctamente.

```
   PRTF(FileMove(destination, FILE_COMMON, source, FILE_REWRITE)); // true
```

Por último, también se elimina el archivo original para dejar un entorno limpio para nuevos experimentos con este script.

```
   ...
   FileDelete(source);
}
```

4.5.15 Búsqueda de archivos y carpetas

MQL5 le permite buscar archivos y carpetas dentro de las «sandbox» de los terminales, los agentes probadores y la «sandbox» común para todos los terminales (para obtener más detalles sobre «sandboxes», consulte el capítulo de la introducción Trabajar con archivos). Si conoce exactamente el nombre y la ubicación del archivo/directorio requerido, utilice la función *FileIsExist*.

long FileFindFirst(const string filter, string &found, int flag = 0)

La función inicia la búsqueda de archivos y carpetas de acuerdo con el filtro pasado. El filtro puede contener una ruta formada por subcarpetas dentro del sandbox y debe contener el nombre exacto o el

patrón de nombres de los elementos del sistema de archivos que se buscan. El parámetro *filter* no puede estar vacío.

Una plantilla es una cadena que contiene uno o varios caracteres comodín. Hay dos tipos de caracteres de este tipo: el asterisco ('*') sustituye a cualquier número de caracteres (incluido el cero) y el signo de interrogación ('?') no sustituye más que un carácter. Por ejemplo, el filtro «*» encontrará todos los archivos y carpetas, y «???.*» encontrará sólo aquellos cuyo nombre no tenga más de 3 caracteres, y la extensión puede o no estar presente. Los archivos con la extensión «csv» se pueden encontrar mediante el filtro «*.csv» (pero tenga en cuenta que la carpeta también puede tener una extensión). El filtro «*.» encuentra elementos sin extensión, y «.*» encuentra elementos sin nombre. No obstante, conviene recordar lo que se indica a continuación.

> En muchas versiones de Windows se generan dos tipos de nombres para los elementos del sistema de archivos: largo (por defecto, hasta 260 caracteres) y corto (en el formato 8.3 heredado de MS-DOS). El segundo tipo se genera automáticamente a partir del nombre largo si éste supera los 8 caracteres o la extensión es superior a 3. Esta generación de nombres cortos puede desactivarse en el sistema si ningún software los utiliza, pero normalmente están activados.
>
> Los archivos se buscan con ambos tipos de nombres, por lo que la lista devuelta puede contener elementos inesperados a primera vista. En particular, un nombre corto, si es generado por el sistema a partir de un nombre largo, contiene siempre una parte inicial antes del punto, de hasta 8 caracteres. Puede encontrar accidentalmente una coincidencia con el patrón deseado.

Si necesita encontrar archivos con varias extensiones, o con diferentes fragmentos en el nombre que no se pueden generalizar con un patrón, tendrá que repetir el proceso de búsqueda varias veces con diferentes configuraciones.

La búsqueda se realiza sólo en una carpeta específica (ya sea en la carpeta raíz de la «sandbox» si no hay ruta en el filtro, o en la subcarpeta especificada si el filtro contiene una ruta) y no entra en los subdirectorios.

La búsqueda no distingue entre mayúsculas y minúsculas. Por ejemplo, una solicitud de archivos «*.txt» también devolverá archivos con la extensión «TXT», «Txt», etc.

Si se encuentra un archivo o carpeta con un nombre coincidente, dicho nombre se coloca en el parámetro de salida *found* (requiere una variable porque el resultado se pasa por referencia) y la función devuelve un manejador de búsqueda: éste deberá pasarse a la función *FileFindNext* para continuar iterando sobre los elementos coincidentes si hay muchos.

En el parámetro *found* sólo se devuelven el nombre y la extensión, sin la ruta (jerarquía de carpetas) que pudiera haberse especificado en el filtro.

Si el elemento encontrado es una carpeta, se añade un carácter '\' (barra invertida) a la derecha de su nombre.

El parámetro *flag* permite seleccionar la zona de búsqueda entre la carpeta de trabajo local de la copia actual del terminal (por valor 0) o la carpeta común de todos los terminales (por valor FILE_COMMON). Cuando un programa MQL se ejecuta en un probador, su «sandbox» local (0) se encuentra en el directorio del agente del probador.

Una vez finalizado el procedimiento de búsqueda, el manejador recibido debe liberarse llamando a *FileFindClose* (ver más adelante).

bool FileFindNext(long handle, string &found)

La función continúa buscando elementos adecuados del sistema de archivos, iniciada por la función *FileFindFirst*. El primer parámetro es el descriptor recibido de *FileFindFirst*, debido al cual se aplican todas las condiciones de búsqueda anteriores.

Si se encuentra el siguiente elemento, su nombre se pasa al código de llamada a través del argumento *found*, y la función devuelve *true*.

Si no hay más elementos, la función devuelve *false*.

void FileFindClose(long handle)

La función cierra el descriptor de búsqueda recibido como resultado de la llamada *FileFindFirst*.

La función debe llamarse una vez finalizado el procedimiento de búsqueda para liberar recursos del sistema.

Como ejemplo, consideremos el script *FileFind.mq5*. En las secciones anteriores probamos muchos otros scripts que creaban archivos en el directorio *MQL5/Files/MQL5Book*. Solicite una lista de todos los archivos de este tipo.

```
void OnStart()
{
   string found; // receiving variable
   // start searching and get descriptor
   long handle = PRTF(FileFindFirst("MQL5Book/*", found));
   if(handle != INVALID_HANDLE)
   {
      do
      {
         Print(found);
      }
      while(FileFindNext(handle, found));
      FileFindClose(handle);
   }
}
```

Aunque haya vaciado este directorio, puede copiar en él los archivos de muestra suministrados con el libro en diversas codificaciones. Por lo tanto, el script *FileFind.mq5* debería mostrar al menos la siguiente lista (el orden de enumeración puede cambiar):

```
ansi1252.txt
unicode1.txt
unicode2.txt
unicode3.txt
utf8.txt
```

Para simplificar el proceso de búsqueda, el script dispone de una función *DirList* auxiliar. Contiene todas las llamadas necesarias a las funciones integradas y un bucle para construir un array de cadenas con una lista de elementos que coinciden con el filtro.

```
bool DirList(const string filter, string &result[], bool common = false)
{
   string found[1];
   long handle = FileFindFirst(filter, found[0]);
   if(handle == INVALID_HANDLE) return false;
   do
   {
      if(ArrayCopy(result, found, ArraySize(result)) != 1) break;
   }
   while(FileFindNext(handle, found[0]));
   FileFindClose(handle);

   return true;
}
```

Con él, solicitaremos una lista de directorios en la «sandbox» local. Para ello, partimos de la base de que los directorios no suelen tener extensión (en teoría, no siempre es así y, por lo tanto, aquellos que lo deseen deberían implementar de forma diferente una solicitud más estricta de una lista de subcarpetas). El filtro para elementos sin extensión es «*» (puede comprobarlo con el comando *dir* en el shell «dir *.» de Windows). Sin embargo, esta plantilla provoca el error 5002 (WRONG_FILENAME) en las funciones de MQL5. Por lo tanto, especificaremos una plantilla «*.?» más «vaga»: significa elementos sin extensión o con una extensión de 1 carácter.

```
void OnStart()
{
   ...
   string list[];
   // try to request elements without extension
   // (works on the Windows command line)
   PRTF(DirList("*.", list)); // false / WRONG_FILENAME(5002)

   // expand the condition: the extension must be no more than 1 character
   if(DirList("*.?", list))
   {
      ArrayPrint(list);
      // example: "MQL5Book\" "Tester\"
   }
}
```

En mi instancia de MetaTrader 5, el script encuentra dos carpetas «MQL5Book\» y «Tester\». Debería tener también el primero si ha ejecutado los scripts de prueba anteriores.

4.5.16 Trabajar con carpetas

Es difícil imaginar un sistema de archivos sin la capacidad de estructurar la información almacenada mediante una jerarquía arbitraria de directorios: contenedores de conjuntos de archivos relacionados lógicamente. En el nivel MQL5, esta función también se admite. Si es necesario, podemos crear, limpiar y eliminar carpetas utilizando las funciones integradas *FolderCreate*, *FolderClean*, y *FolderDelete*.

Anteriormente vimos ya una forma de crear una carpeta, y, tal vez, no una, sino toda la jerarquía necesaria de subcarpetas a la vez. Para ello, al crear (abrir) un archivo mediante *FileOpen*, o al copiarlo

(*FileCopy*, *FileMove*), debe especificar no sólo un nombre, sino encabezarlo con la ruta requerida. Por ejemplo:

```
FileCopy("MQL5Book/unicode1.txt", 0, "ABC/DEF/code.txt", 0);
```

Esta sentencia creará la carpeta «ABC» en la «sandbox», la carpeta «DEF» en ella, y copiará allí el archivo con un nuevo nombre (el archivo fuente debe existir).

Si no desea crear un archivo fuente de antemano, puede crear un archivo ficticio sobre la marcha:

```
uchar dummy[];
FileSave("ABC/DEF/empty", dummy);
```

Aquí obtendremos la misma jerarquía de carpetas que en el ejemplo anterior pero con un archivo «vacío» de tamaño cero.

Con estos enfoques, la creación de carpetas se convierte en una especie de subproducto del trabajo con archivos. Sin embargo, a veces es necesario operar con carpetas como entidades independientes y sin efectos secundarios; en concreto, basta con crear una carpeta vacía. Esto lo ofrece la función *FolderCreate*.

bool FolderCreate(const string folder, int flag = 0)

La función crea una carpeta llamada *folder*, que puede incluir una ruta (varios nombres de carpetas de nivel superior). En ambos casos se crea una única carpeta o jerarquía de carpetas en la «sandbox» definida por el parámetro *flag*. De manera predeterminada, cuando *flag* es 0, se utiliza la carpeta de trabajo local *MQL5/Files* del terminal o del agente del probador (si el programa se está ejecutando en el probador). Si *flag* es igual a FILE_COMMON, se utiliza la carpeta compartida de todos los terminales.

La función devuelve *true* en caso de éxito, o si la carpeta ya existe. En caso de error, el resultado es *false*.

bool FolderClean(const string folder, int flag = 0)

La función elimina todos los archivos y carpetas de cualquier nivel de anidamiento (junto con todo el contenido) del directorio *folder* especificado. El parámetro *flag* especifica la «sandbox» (local o global) en la que tiene lugar la acción.

> Utilice esta función con precaución, ya que todos los archivos y subcarpetas (con sus archivos) se eliminan permanentemente.

bool FolderDelete(const string folder, int flag = 0)

La función borra la carpeta especificada (*folder*). Antes de llamar a la función, la carpeta debe estar vacía; de lo contrario no se podrá eliminar.

Las técnicas para trabajar con estas tres funciones se muestran en el script *FileFolder.mq5*. Puedes ejecutar este script en el modo de depuración paso a paso (sentencia por sentencia) y observar en el administrador de archivos cómo aparecen y desaparecen carpetas y archivos. Sin embargo, tenga en cuenta que antes de ejecutar la siguiente instrucción, debe usar el gestor de archivos para salir de las carpetas creadas hasta el nivel «MQL5Book», ya que de lo contrario las carpetas pueden estar ocupadas por el administrador de archivos, y esto interrumpirá el script.

Primero creamos varias subcarpetas como subproducto de escribir en ellas un archivo ficticio vacío.

```
void OnStart()
{
   const string filename = "MQL5Book/ABC/DEF/dummy";
   uchar dummy[];
   PRTF(FileSave(filename, dummy)); // true
```

A continuación, creamos otra carpeta en el nivel de anidamiento inferior con *FolderCreate*: Esta vez la carpeta aparece sola, sin el archivo de ayuda.

```
   PRTF(FolderCreate("MQL5Book/ABC/GHI")); // true
```

Si intenta eliminar la carpeta «DEF» fallará porque no está vacía (hay un archivo allí).

```
   PRTF(FolderDelete("MQL5Book/ABC/DEF")); // false / CANNOT_DELETE_DIRECTORY(5024)
```

Para eliminarlo, primero debe borrarlo, y la forma más fácil de hacerlo es con *FolderClean*. No obstante, intentaremos simular una situación común cuando algunos archivos de las carpetas que se están limpiando pueden estar bloqueados por otros programas MQL, aplicaciones externas o el propio terminal. Vamos a abrir el archivo para leerlo y a llamar a *FolderClean*.

```
   int handle = PRTF(FileOpen(filename, FILE_READ)); // 1
   PRTF(FolderClean("MQL5Book/ABC")); // false / CANNOT_CLEAN_DIRECTORY(5025)
```

La función devuelve *false* y expone el código de error 5025 (CANNOT_CLEAN_DIRECTORY). Después de cerrar el archivo, la limpieza y eliminación de toda la jerarquía de carpetas se realiza correctamente.

```
   FileClose(handle);
   PRTF(FolderClean("MQL5Book/ABC")); // true
   PRTF(FolderDelete("MQL5Book/ABC")); // true
}
```

Los posibles bloqueos son más probables cuando se utiliza un directorio de terminal compartido, donde el mismo archivo o carpeta puede ser «reclamado» por diferentes instancias del programa. Pero incluso en una «sandbox» local no hay que olvidarse de posibles conflictos (por ejemplo, si se abre un archivo csv en Excel). Implemente diagnósticos detallados y salida de errores para las partes de código que trabajan con carpetas, de forma que el usuario pueda darse cuenta y solucionar el problema.

4.5.17 Cuadro de diálogo de selección de archivos o carpetas

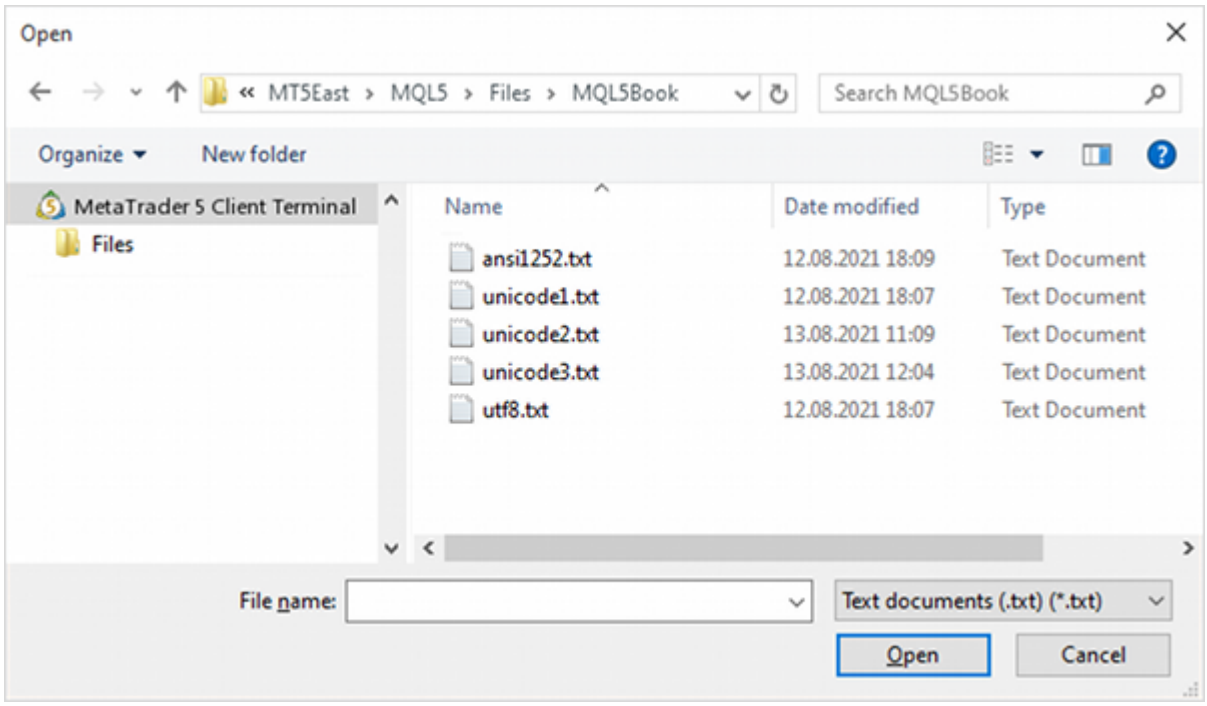
En el grupo de funciones para trabajar con archivos y carpetas, hay una que permite solicitar interactivamente al usuario el nombre de un archivo o carpeta, así como un grupo de archivos para pasar esta información a un programa MQL. La llamada a la función *FileSelectDialog* hace que aparezca en el terminal una ventana estándar de Windows para seleccionar archivos y carpetas.

> Dado que el diálogo interrumpe la ejecución del programa MQL hasta que se cierra, la llamada a la función sólo se permite en dos tipos de programas MQL que se ejecutan en hilos separados: EAs y scripts (véase Tipos de programas MQL). El uso de esta función está prohibido en indicadores y servicios: los primeros se ejecutan en el hilo de interfaz del terminal (y detenerlos congelaría la actualización de los gráficos de los instrumentos correspondientes), mientras que los segundos se ejecutan en segundo plano y no pueden acceder a la interfaz de usuario.

Todos los elementos del sistema de archivos con los que trabaja la función se encuentran dentro de la «sandbox», es decir, en el directorio de la copia actual del terminal o del agente de pruebas (si el programa se ejecuta en el probador), en la subcarpeta *MQL5/Files*.

Si la bandera FSD_COMMON_FOLDER está presente en el parámetro *flags* (ver más adelante), se utiliza una «sandbox» común de todos los terminales *Users/<user>...MetaQuotes/Terminal/Common/Files*.

El aspecto del cuadro de diálogo depende del sistema operativo Windows. A continuación se muestra una de las posibles opciones de interfaz.



Cuadro de diálogo de selección de archivos y carpetas de Windows

```
int FileSelectDialog(const string caption, const string initDir, const string filter,
 uint flags, string &filenames[], const string defaultName)
```

La función muestra un cuadro de diálogo estándar de Windows para abrir o crear un archivo o seleccionar una carpeta. El título se especifica en el parámetro *caption*. Si el valor es NULL, se utiliza el título estándar: «Abrir» para leer o «Guardar como» para escribir un archivo, o «Seleccionar carpeta», en función de las banderas del parámetro *flags*.

El parámetro *initDir* permite establecer la carpeta inicial para la que se abrirá el cuadro de diálogo. Si se establece en NULL, se mostrará el contenido de la carpeta *MQL5/Files*. La misma carpeta se utiliza si se especifica una ruta inexistente en *initDir*.

Mediante el parámetro *filter* puede limitar el conjunto de extensiones de archivo que se mostrarán en el cuadro de diálogo. Los archivos de otros formatos quedarán ocultos. NULL significa sin restricciones.

El formato de la cadena *filter* es el siguiente:

```
"<description 1>|<extension 1>|<description 2>|<extension 2>..."
```

Cualquier cadena se admite como *description*. Puede escribir cualquier filtro con los caracteres sustituidos '*' y '?' de los que hablamos en la sección Búsqueda de archivos y carpetas como *extensions*. El símbolo '|' es un delimitador.

Dado que la descripción y la extensión adyacentes forman un par relacionado lógicamente, el número total de elementos de la línea debe ser par, y el número de delimitadores, impar.

Cada combinación de descripción y extensión genera una selección independiente en la lista desplegable del cuadro de diálogo. La descripción se muestra al usuario y la extensión se utiliza para filtrar.

Por ejemplo, «Documentos de texto (*.txt)|*.txt|Todos los archivos (*.*)|*.*», mientras que la primera extensión «Documentos de texto (*.txt)|*.txt» se seleccionará como tipo de archivo por defecto.

En el parámetro *flags* puede indicar una máscara de bits que especifique los modos de funcionamiento mediante el operador '|'. Para ello se definen las siguientes constantes:

- FSD_WRITE_FILE - modo de escritura de archivos («Guardar como»). En ausencia de esta bandera, se utiliza por defecto el modo de lectura («Abrir»). Si esta bandera está presente, la entrada de un nuevo nombre arbitrario está siempre permitida, independientemente de la bandera FSD_FILE_MUST_EXIST.
- FSD_SELECT_FOLDER - modo de selección de carpetas (sólo una y sólo las existentes). Con esta bandera, todas las otras banderas excepto FSD_COMMON_FOLDER son ignoradas o causan un error. No se puede solicitar explícitamente la creación de una carpeta, pero es posible crear interactivamente una carpeta en el cuadro de diálogo y seleccionarla inmediatamente.
- FSD_ALLOW_MULTISELECT - permiso para seleccionar múltiples archivos en modo lectura. Esta bandera se ignora si se especifica FSD_WRITE_FILE o FSD_SELECT_FOLDER.
- FSD_FILE_MUST_EXIST - los archivos seleccionados deben existir. Si el usuario intenta especificar un nombre arbitrario, el cuadro de diálogo mostrará una advertencia y permanecerá abierto. Esta bandera se ignora si se especifica el modo FSD_WRITE_FILE.
- FSD_COMMON_FOLDER - el cuadro de diálogo se abre para una «sandbox» común de todos los terminales cliente.

La función rellenará un array de cadenas *filenames* con los nombres de los archivos o carpetas seleccionados. Si el array es dinámico, su tamaño cambia para ajustarse a la cantidad real de datos; en concreto, se expande o trunca hasta 0 si no se ha seleccionado nada. Si el array es fijo, debe ser lo suficientemente grande como para aceptar los datos esperados. De lo contrario, se producirá un error 4007 (ARRAY_RESIZE_ERROR).

El parámetro *defaultName* especifica el nombre de archivo/carpeta por defecto, que será sustituido en el campo de entrada correspondiente inmediatamente después de abrir el cuadro de diálogo. Si el parámetro es NULL, el campo estará inicialmente vacío.

> Si se establece el parámetro *defaultName*, entonces durante las pruebas no visuales del programa MQL, la llamada *FileSelectDialog* devolverá 1 y el propio valor *defaultName* se copiará en el array *filenames*.

La función devuelve el número de elementos seleccionados (0 si el usuario no ha seleccionado nada), o -1 si se ha producido un error.

Vea ejemplos de cómo funciona la función en el script *FileSelect.mq5*. En la función *OnStart*, llamaremos secuencialmente a *FileSelectDialog* con diferentes configuraciones. Mientras el usuario seleccione algo (no haga clic en el botón «Cancelar» del cuadro de diálogo), la prueba continúa hasta el último paso (incluso si la función se ejecuta con un código de error).

```
void OnStart()
{
 string filenames[]; // a dynamic array suitable for any call
 string fixed[1]; // too small array if there are more than 1 files
 const stringfilter = // filter example
      "Text documents (*.txt)|*.txt"
      "|Files with short names|????.*"
      "|All files (*.*)|*.*";
```

Primero, pediremos al usuario un archivo de la carpeta «MQL5Book». Puede seleccionar un archivo existente o introducir un nuevo nombre (porque no hay ninguna bandera FSD_FILE_MUST_EXIST).

```
   Print("Open a file");
   if(PRTF(FileSelectDialog(NULL, "MQL5book", filter,
      0, filenames, NULL)) == 0) return;             // 1
   ArrayPrint(filenames);                            // "MQL5Book\utf8.txt"
```

Suponiendo que la carpeta contenga al menos 5 archivos de la entrega del libro, aquí se selecciona uno de ellos.

Entonces haremos una petición similar en modo «para escribir» (con la bandera FSD_WRITE_FILE).

```
   Print("Save as a file");
   if(PRTF(FileSelectDialog(NULL, "MQL5book", NULL,
      FSD_WRITE_FILE, filenames, NULL)) == 0) return;// 1
   ArrayPrint(filenames);                            // "MQL5Book\newfile"
```

Aquí el usuario también podrá seleccionar tanto un archivo existente como introducir un nuevo nombre. El programador debe comprobar si el usuario va a sobrescribir un archivo existente (el cuadro de diálogo no genera advertencias).

Ahora vamos a comprobar la selección de múltiples archivos (FSD_ALLOW_MULTISELECT) en un array dinámico.

```
   if(PRTF(FileSelectDialog(NULL, "MQL5book", NULL,
     FSD_FILE_MUST_EXIST | FSD_ALLOW_MULTISELECT, filenames, NULL)) == 0) return; //
   ArrayPrint(filenames);
   // "MQL5Book\ansi1252.txt" "MQL5Book\unicode1.txt" "MQL5Book\unicode2.txt"
   // "MQL5Book\unicode3.txt" "MQL5Book\utf8.txt"
```

La presencia de la bandera FSD_FILE_MUST_EXIST significa que el cuadro de diálogo mostrará una advertencia y permanecerá abierto si intenta introducir un nuevo nombre.

Si intentamos seleccionar más de un archivo en un array de tamaño fijo de forma similar, obtendremos un error.

```
   Print("Open multiple files (fixed, choose more than 1 file for error)");
   if(PRTF(FileSelectDialog(NULL, "MQL5book", NULL,
      FSD_FILE_MUST_EXIST | FSD_ALLOW_MULTISELECT, fixed, NULL)) == 0) return;
   // -1 / ARRAY_RESIZE_ERROR(4007)
   ArrayPrint(fixed); // null
```

Por último, vamos a revisar las operaciones con carpetas (FSD_SELECT_FOLDER).

```
   Print("Select a folder");
   if(PRTF(FileSelectDialog(NULL, "MQL5book/nonexistent", NULL,
      FSD_SELECT_FOLDER, filenames, NULL)) == 0) return; // 1
   ArrayPrint(filenames); // "MQL5Book"
```

En este caso, se especifica la subcarpeta «nonexistent» como ruta de inicio, por lo que el cuadro de diálogo se abrirá en la raíz de la «sandbox» *MQL5/Files*. Allí elegimos «MQL5book».

Si realizamos una combinación no válida de banderas, obtendremos otro error.

```
   if(PRTF(FileSelectDialog(NULL, "MQL5book", NULL,
      FSD_SELECT_FOLDER | FSD_WRITE_FILE, filenames, NULL)) == 0) return;
   // -1 / INTERNAL_ERROR(4001)
   ArrayPrint(filenames); // "MQL5Book"
}
```

Debido a un error, la función no modificó el array pasado, y el antiguo elemento «MQL5Book» permaneció en él.

En esta prueba comprobamos deliberadamente los resultados sólo para 0 con el fin de demostrar todas las opciones, independientemente de la presencia de errores. En un programa real, compruebe el resultado de la función teniendo en cuenta los errores, es decir, con condiciones para tres resultados: elección realizada (>0), elección no realizada (==0) y error (<0).

4.6 Variables globales del terminal cliente

En el capítulo anterior estudiamos las funciones de MQL5 que trabajan con archivos. Ofrecen opciones amplias y flexibles para escribir y leer datos arbitrarios. Sin embargo, a veces un programa MQL necesita una forma más sencilla de guardar y restaurar el estado de un atributo entre ejecuciones.

Por ejemplo, queremos calcular ciertas estadísticas, como por ejemplo, cuántas veces se lanzó el programa, cuántas instancias del mismo se ejecutan en paralelo en diferentes gráficos, etc. Es imposible acumular esta información dentro del propio programa. Debe haber algún tipo de almacenamiento externo a largo plazo, pero sería costoso crear un archivo para ello, aunque también es factible.

Muchos programas están diseñados para interactuar entre sí, es decir, deben intercambiar información de alguna manera. Si hablamos de integración con un programa externo al terminal, o de transferir una gran cantidad de datos, entonces es realmente difícil hacerlo sin utilizar archivos. Sin embargo, cuando no hay suficientes datos para enviar, y todos los programas están escritos en MQL5 y se ejecutan dentro de MetaTrader 5, el uso de archivos parece redundante. El terminal ofrece una tecnología más sencilla para este caso: las variables globales.

Una variable global es una ubicación con nombre en la memoria compartida del terminal. Se puede crear, modificar o eliminar por cualquier programa MQL, pero no le pertenecerá en exclusiva, y está disponible para el resto de programas MQL. El nombre de una variable global es cualquier cadena única (entre todas las variables) de no más de 63 caracteres. Esta cadena no tiene que cumplir los requisitos para los identificadores de variables en MQL5, ya que las variables globales del terminal no son variables en el sentido habitual. El programador no las define en el código fuente según la sintaxis que aprendimos en Variables, no son parte integrante del programa MQL, y cualquier acción con ellos se realiza únicamente llamando a una de las funciones especiales que describiremos en este capítulo.

Las variables globales sólo permiten almacenar valores del tipo *double*. Si es necesario, puede empaquetar o convertir valores de otros tipos en *double* o utilizar parte del nombre de la variable (siguiendo un determinado prefijo, por ejemplo) para almacenar cadenas.

Mientras se ejecuta el terminal, las variables globales se almacenan en la RAM y están disponibles casi al instante: la única sobrecarga está asociada a las llamadas a funciones. Esto da, definitivamente, una ventaja a las variables globales frente al uso de archivos, ya que cuando se trata de estos últimos, la obtención de un manejador es un proceso relativamente lento, y el propio manejador consume algunos recursos adicionales.

Al final de la sesión del terminal, las variables globales se descargan en un archivo especial (*gvariables.dat*) y se restauran desde él la próxima vez que se ejecute el terminal.

Una determinada variable global es destruida automáticamente por el terminal si no ha sido reclamada en un plazo de 4 semanas. Este comportamiento se basa en realizar el seguimiento y almacenar la hora del último uso de una variable, donde uso se refiere a establecer un nuevo valor o leer uno antiguo (pero no comprobar la existencia u obtener la hora del último uso).

> Tenga en cuenta que las variables globales no están vinculadas a una cuenta, perfil o cualquier otra característica del entorno de trading. Por lo tanto, si se supone que deben almacenar algo relacionado con el entorno (por ejemplo, algunos límites generales para una cuenta concreta), los nombres de las variables deben construirse teniendo en cuenta todos los factores que afectan al algoritmo y a la toma de decisiones. Para distinguir entre las variables globales de varias instancias del mismo Asesor Experto (EA), es posible que tenga que añadir al nombre un símbolo de trabajo, un marco temporal o un «número mágico» de la configuración del EA.

Además de los programas MQL, las variables globales también puede crearlas manualmente el usuario. La lista de variables globales existentes, así como los medios para su gestión interactiva, se encuentran en el cuadro de diálogo abierto en el terminal mediante el comando *Herramientas -> Variables globales* (F3).

Utilizando los botones correspondientes podrá *Añadir* y *Eliminar* variables globales, y haciendo doble clic en las columnas *Variable* o *Significado* podrá editar el nombre o el valor de una variable concreta. Las siguientes teclas de acceso rápido funcionan desde el teclado: F2 para editar el nombre, F3 para editar el valor, Ins para añadir una nueva variable, Supr para borrar la variable seleccionada.

Un poco más adelante estudiaremos dos tipos principales de programas MQL: Asesores Expertos e Indicadores. Su característica especial es la posibilidad de ejecutarse en el comprobador, donde también funcionan las funciones para variables globales. Sin embargo, las variables globales son creadas, almacenadas y gestionadas por el agente comprobador en el probador. En otras palabras, las listas de variables globales terminales no están disponibles en el comprobador, y aquellas variables que son creadas por el programa bajo prueba pertenecen a un agente específico, y su tiempo de vida está limitado a una pasada de prueba. Es decir, las variables globales del agente no son visibles desde otros agentes y serán eliminadas al final de la ejecución de la prueba. En concreto, si el EA se optimiza en varios agentes, puede manipular variables globales para «comunicarse» con los indicadores que utiliza en el contexto del mismo agente ya que se ejecutan allí conjuntamente, pero en agentes paralelos, otras copias del EA formarán sus propias listas de variables.

> El intercambio de datos entre programas MQL utilizando variables globales no es la única forma disponible, y no siempre es la más adecuada. En concreto, los EA y los indicadores son tipos interactivos de programas MQL que pueden generar y aceptar eventos en gráficos. Puede pasar varios tipos de información en los parámetros de los eventos. Además, se pueden preparar arrays de datos calculados y proporcionarlas a otros programas MQL en forma de búferes indicadores. Los

programas MQL ubicados en gráficos pueden utilizar objetos gráficos de la IU para transferir y almacenar información.

Desde el punto de vista técnico, el número máximo de variables globales sólo está limitado por los recursos del sistema operativo. Sin embargo, para un gran número de elementos, se recomienda utilizar medios más adecuados: archivos o bases de datos.

4.6.1 Escritura y lectura de variables globales

La API de MQL5 proporciona dos funciones para escribir y leer variables globales: *GlobalVariableSet* y *GlobalVariableGet* (en dos versiones).

datetime GlobalVariableSet(const string name, double value)

La función establece un nuevo *value* a la variable global «name». Si la variable no existía antes de llamar a la función, se creará. Si la variable ya existe, el valor anterior será sustituido por *value*.

Si tiene éxito, la función devuelve la hora de modificación de la variable (la hora local actual del ordenador). En caso de error, obtenemos 0.

double GlobalVariableGet(const string name)
bool GlobalVariableGet(const string name, double &value)

La función devuelve el valor de la variable global 'nombre'. El resultado de llamar a la primera versión de la función contiene sólo el valor de la variable (en caso de éxito) o 0 (en caso de error). Dado que la variable puede contener el valor 0 (que equivale a una indicación de error), esta opción requiere analizar el código de error interno *_LastError* si se recibe cero, para distinguir la versión estándar de la no estándar. En concreto, si se intenta leer una variable que no existe, se genera un error interno 4501 (GLOBALVARIABLE_NOT_FOUND).

Esta versión de la función es conveniente utilizarla en algoritmos en los que obtener cero es un análogo adecuado de la inicialización por defecto para una variable previamente inexistente (véase el ejemplo siguiente). Si la ausencia de una variable debe ser tratada de una manera especial (en concreto, para calcular algún otro valor inicial), primero debe comprobar la existencia de la variable utilizando la función *GlobalVariableCheck* y, en función de su resultado, ejecutar diferentes ramas de código. Opcionalmente, puede utilizar la segunda versión.

La segunda versión de la función devuelve *true* o *false* dependiendo del éxito de la ejecución. Si tiene éxito, el valor de la variable global del terminal se coloca en la variable *value* receptora, pasada por referencia como segundo parámetro. Si no hay variable, obtenemos *false*.

En el script de prueba *GlobalsRunCount.mq5* utilizamos una variable global para contar el número de veces que se ha ejecutado. El nombre de la variable es el nombre del archivo fuente.

```
const string gv = __FILE__;
```

Recordemos que la macro integrada __FILE__ (véase Constantes predefinidas) es expandida por el compilador en el nombre del archivo compilado, es decir, en este caso, «GlobalsRunCount.mq5».

En la función *OnStart* intentaremos leer la variable global dada y guardar el resultado en la variable local *count*. Si todavía no había ninguna variable global, obtenemos 0, lo cual está bien para nosotros (empezamos a contar desde cero).

Antes de guardar el valor en *count*, es necesario realizar una conversión de tipo a (*int*), ya que la función *GlobalVariableGet* devuelve *double*, y sin la conversión, el compilador genera un aviso de posible pérdida de datos (no sabe que pensamos almacenar sólo enteros).

```
void OnStart()
{
   int count = (int)PRTF(GlobalVariableGet(gv));
   count++;
   PRTF(GlobalVariableSet(gv, count));
   Print("This script run count: ", count);
}
```

A continuación, incrementamos el contador en 1 y lo escribimos de nuevo en la variable global con *GlobalVariableSet*. Si ejecutamos el script varias veces, obtendremos algo parecido a las siguientes entradas de registro (sus marcas de tiempo serán diferentes):

```
GlobalVariableGet(gv)=0.0 / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalVariableSet(gv,count)=2021.08.29 16:04:40 / ok
This script run count: 1
GlobalVariableGet(gv)=1.0 / ok
GlobalVariableSet(gv,count)=2021.08.29 16:05:00 / ok
This script run count: 2
GlobalVariableGet(gv)=2.0 / ok
GlobalVariableSet(gv,count)=2021.08.29 16:05:21 / ok
This script run count: 3
```

Es importante señalar que en la primera ejecución recibimos un valor de 0, y se generó la bandera de error interno 4501. Todas las llamadas posteriores se ejecutan sin problemas, ya que la variable existe (puede verse en la ventana «Variables globales» del terminal). Quien lo desee puede cerrar el terminal, reiniciarlo y volver a ejecutar el script: el contador seguirá aumentando desde el valor anterior.

4.6.2 Comprobar la existencia y la hora de la última actividad

Como vimos en la sección anterior, se puede comprobar la existencia de una variable global intentando leer su valor: si ello no da lugar a un código de error en *_LastError*, entonces la variable global existe, y ya hemos obtenido su valor y podemos utilizarlo en el algoritmo. Sin embargo, si en algunas condiciones sólo necesita comprobar la existencia, pero no leer la variable global, es más conveniente utilizar otra función específicamente diseñada para ello: *GlobalVariableCheck*.

Existe otra forma de comprobarlo, a saber: con la función *GlobalVariableTime*. Como su nombre indica, esta función permite averiguar la última vez que se utilizó una variable, pero si la variable no existe, entonces el tiempo de su uso está ausente, es decir, es igual a 0.

bool GlobalVariableCheck(const string name)

La función comprueba la existencia de una variable global con el nombre especificado y devuelve el resultado *true* (la variable existe) o *false* (sin variable).

datetime GlobalVariableTime(const string name)

La función devuelve la hora en que se utilizó por última vez la variable global con el nombre especificado. El hecho de la utilización puede representarse mediante la modificación o la lectura del valor de la variable.

> Comprobar la existencia de la variable con *GlobalVariableCheck* u obtener su tiempo a través de *GlobalVariableTime* no cambian el tiempo de uso.

En el script *GlobalsRunCheck.mq5* completaremos ligeramente el código de *GlobalsRunCount.mq5* para que al principio de la función *OnStart* compruebe la presencia de una variable y el momento de su utilización.

```
void OnStart()
{
   PRTF(GlobalVariableCheck(gv));
   PRTF(GlobalVariableTime(gv));
   ...
```

El código siguiente no se modifica. Mientras tanto, tenga en cuenta que la variable *gv* definida a través de __FILE__ contendrá esta vez el nuevo nombre de script «GlobalsRunCheck.mq5» como nombre de la variable global (es decir, cada script tiene su propio contador global).

Todas las ejecuciones excepto la primera mostrarán *true* de la función *GlobalVariableCheck* (la variable existe) y el tiempo de la variable de la ejecución anterior. He aquí un registro de ejemplo:

```
GlobalVariableCheck(gv)=false / ok
GlobalVariableTime(gv)=1970.01.01 00:00:00 / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalVariableGet(gv)=0.0 / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalVariableSet(gv,count)=2021.08.29 16:59:35 / ok
This script run count: 1
GlobalVariableCheck(gv)=true / ok
GlobalVariableTime(gv)=2021.08.29 16:59:35 / ok
GlobalVariableGet(gv)=1.0 / ok
GlobalVariableSet(gv,count)=2021.08.29 16:59:45 / ok
This script run count: 2
GlobalVariableCheck(gv)=true / ok
GlobalVariableTime(gv)=2021.08.29 16:59:45 / ok
GlobalVariableGet(gv)=2.0 / ok
GlobalVariableSet(gv,count)=2021.08.29 16:59:56 / ok
This script run count: 3
```

4.6.3 Obtener una lista de variables globales

Muy a menudo, se necesita que un programa MQL mire en las variables globales existentes y seleccione aquellas que cumplan determinados criterios. Por ejemplo, si un programa utiliza parte del nombre de una variable para almacenar información textual, lo único que se conoce de antemano es el prefijo. El propósito de este prefijo es identificar «su propia» variable, y la «carga útil» adjunta al prefijo no permite buscar una variable por el nombre exacto.

La API de MQL5 tiene dos funciones que le permiten enumerar las variables globales.

int GlobalVariablesTotal()

La función devuelve el número total de variables globales.

string GlobalVariableName(int index)

La función devuelve el nombre de la variable global por su número de índice en la lista de variables globales. El parámetro *index* con el número de la variable solicitada debe estar en el rango de 0 a *GlobalVariablesTotal() - 1*.

En caso de error, la función devolverá NULL, y el código de error puede obtenerse a partir de la variable de servicio *_LastError* o de la función *GetLastError*.

Probemos este par de funciones con el script *GlobalsList.mq5*.

```
void OnStart()
{
   PRTF(GlobalVariableName(1000000));
   int n = PRTF(GlobalVariablesTotal());
   for(int i = 0; i < n; ++i)
   {
      const string name = GlobalVariableName(i);
      PrintFormat("%d %s=%f", i, name, GlobalVariableGet(name));
   }
}
```

La primera cadena pide deliberadamente el nombre de una variable con un número grande, que, muy probablemente, no existe, y ese hecho debería provocar un error. A continuación, se solicita el número real de variables y se realiza un bucle a través de todas ellas, con la salida del nombre y el valor. El registro siguiente incluye variables creadas por scripts de prueba anteriores y una variable de terceros.

```
GlobalVariableName(1000000)= / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalVariablesTotal()=3 / ok
0 GlobalsRunCheck.mq5=3.000000
1 GlobalsRunCount.mq5=4.000000
2 abracadabra=0.000000
```

El orden en que el terminal devuelve las variables por un índice no está definido.

4.6.4 Borrar variables globales

Si es necesario, un programa MQL puede eliminar una variable global o un grupo de ellas que se ha convertido en redundante. La lista de variables globales consume algunos recursos del ordenador, y un buen estilo de programación sugiere que se liberen recursos siempre que sea posible.

bool GlobalVariableDel(const string name)

La función elimina la variable global 'nombre'. En caso de éxito, la función devuelve *true*; en caso contrario devuelve *false*.

int GlobalVariablesDeleteAll(const string prefix = NULL, datetime limit = 0)

La función borra las variables globales con el prefijo especificado en el nombre y con un tiempo de uso anterior al valor del parámetro *limit*.

Si se especifica el prefijo NULL (por defecto) o una cadena vacía, entonces todas las variables globales que también coincidan con el criterio de borrado por fecha (si está establecido) entran en el criterio de borrado.

Si el parámetro *limit* es cero (por defecto), se eliminan las variables globales con cualquier fecha que tenga en cuenta el prefijo.

Si se especifican ambos parámetros, se eliminan las variables globales que coincidan tanto con el prefijo como con el criterio temporal.

Cuidado: llamar a *GlobalVariablesDeleteAll* sin parámetros eliminará todas las variables.

La función devuelve el número de variables eliminadas.

Consideremos el script *GlobalsDelete.mq5*, que explota dos nuevas características.

```
void OnStart()
{
   PRTF(GlobalVariableDel("#123%"));
   PRTF(GlobalVariablesDeleteAll("#123%"));
   ...
```

Al principio, se intenta eliminar las variables globales inexistentes por su nombre y prefijo exactos. Ninguno de los dos tiene efecto sobre las variables existentes.

Llamar a *GlobalVariablesDeleteAll* con un filtro de tiempo en el pasado (hace más de 4 semanas) también tiene un resultado nulo, porque el terminal borra dichas variables antiguas automáticamente (tales variables no pueden existir).

```
   PRTF(GlobalVariablesDeleteAll(NULL, D'2021.01.01'));
```

A continuación, creamos una variable con el nombre «abracadabra» (si no existía) y la borramos inmediatamente. Estas llamadas deberían tener éxito.

```
   PRTF(GlobalVariableSet(abracadabra, 0));
   PRTF(GlobalVariableDel(abracadabra));
```

Por último, eliminemos las variables que empiezan por el prefijo «GlobalsRun»: deberían haber sido creadas por los scripts de prueba de las dos secciones anteriores sobre nombres de archivos (respectivamente, «GlobalsRunCount.mq5» y «GlobalsRunCheck.mq5»).

```
   PRTF(GlobalVariablesDeleteAll("GlobalsRun"));
   PRTF(GlobalVariablesTotal());
}
```

El script debería mostrar en el registro algo parecido al siguiente conjunto de cadenas (algunos indicadores dependen de las condiciones externas y de la hora de inicio).

```
GlobalVariableDel(#123%)=false / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalVariablesDeleteAll(#123%)=0 / ok
GlobalVariablesDeleteAll(NULL,D'2021.01.01')=0 / ok
GlobalVariableSet(abracadabra,0)=2021.08.30 14:02:32 / ok
GlobalVariableDel(abracadabra)=true / ok
GlobalVariablesDeleteAll(GlobalsRun)=2 / ok
GlobalVariablesTotal()=0 / ok
```

Al final, imprimimos el número total de variables globales restantes (en este caso, obtenemos 0, es decir, no hay variables). Puede diferir para usted si las variables globales fueron creadas por otros programas MQL o por el usuario.

4.6.5 Variables globales temporales

En el subsistema de variables globales del terminal es posible hacer que algunas variables sean temporales: se almacenan sólo en la memoria y no se escriben en el disco cuando se cierra el terminal.

Debido a su naturaleza específica, las variables globales temporales se utilizan exclusivamente para el intercambio de datos entre programas MQL y no son adecuadas para guardar estados entre lanzamientos de MetaTrader 5. Uno de los usos más obvios de las variables temporales son diversas métricas de actividad operativa (por ejemplo, contadores de copias de programas en ejecución) que deben recalcularse de forma dinámica en cada inicio, en lugar de restaurarse desde el disco.

Una variable global debe declararse temporal de antemano, antes de asignarle ningún valor, mediante la función *GlobalVariableTemp*.

Por desgracia, es imposible averiguar por el nombre de una variable global si ésta es temporal: MQL5 no proporciona medios para ello.

Las variables temporales sólo pueden crearse mediante programas MQL. Las variables temporales se muestran en la ventana «Variables globales» junto con las variables globales ordinarias (persistentes), pero el usuario no tiene la posibilidad de añadir su propia variable temporal desde la GUI.

bool GlobalVariableTemp(const string name)

La función crea una nueva variable global con el nombre especificado, que existirá sólo hasta el final de la sesión de terminal actual.

Si ya existe una variable con el mismo nombre, no se convertirá en variable temporal.

No obstante, si una variable aún no existe, obtendrá el valor 0. Después, puede trabajar con ella como de costumbre; en concreto, puede asignarle otros valores utilizando la función *GlobalVariableSet*.

Mostraremos un ejemplo de esta función junto con las funciones de la siguiente sección.

4.6.6 Sincronización de programas mediante variables globales

Dado que las variables globales existen fuera de los programas MQL, son útiles para organizar las banderas externas que controlan varias copias del mismo programa o pasan señales entre diferentes programas. El ejemplo más sencillo es limitar el número de copias de un programa que se pueden ejecutar. Esto puede ser necesario para evitar la duplicación accidental del Asesor Experto en diferentes gráficos (debido a lo cual las órdenes comerciales pueden duplicarse), o para implementar una versión demo.

A primera vista, esta comprobación podría hacerse en el código fuente de la siguiente manera:

```
void OnStart()
{
   const string gv = "AlreadyRunning";
   // if the variable exists, then one instance is already running
   if(GlobalVariableCheck(gv)) return;
   // create a variable as a flag signaling the presence of a working copy
   GlobalVariableSet(gv, 0);

   while(!IsStopped())
   {
       // work cycle
   }
   // delete variable before exit
   GlobalVariableDel(gv);
}
```

Aquí se muestra la versión más sencilla utilizando un script como ejemplo. Para otros tipos de programas MQL, el concepto general de comprobación será el mismo, aunque la ubicación de las instrucciones puede diferir: en lugar de un ciclo de trabajo interminable, los Asesores Expertos y los indicadores utilizan sus característicos manejadores de eventos llamados repetidamente por el terminal. Estudiaremos estos problemas más adelante.

El problema del código presentado es que no tiene en cuenta la ejecución en paralelo de los programas MQL.

Un programa MQL normalmente se ejecuta en su propio hilo. Para tres de los cuatro tipos de programas MQL, es decir, para Asesores Expertos, scripts y servicios, el sistema definitivamente asigna hilos separados. En cuanto a los indicadores, se asigna un hilo común a todas sus instancias, que trabajan en la misma combinación de marco temporal y símbolo de trabajo. Pero los indicadores de las distintas combinaciones siguen perteneciendo a hilos diferentes.

Casi siempre se ejecutan muchos hilos en el terminal, muchos más que el número de núcleos del procesador. Por eso, el sistema suspende cada hilo de vez en cuando para permitir que otros hilos trabajen. Como todos estos cambios entre hilos se producen muy rápidamente, nosotros, como usuarios, no nos damos cuenta de esta «organización interna». Sin embargo, cada suspensión puede afectar a la secuencia en la que los distintos hilos acceden a los recursos compartidos. Las variables globales son recursos de este tipo.

Desde el punto de vista del programa, puede producirse una pausa entre cualquier instrucción adyacente. Si, sabiendo esto, volvemos a fijarnos en nuestro ejemplo, no es difícil ver un lugar donde se puede romper la lógica de trabajar con una variable global.

De hecho, la primera copia (hilo) puede realizar una comprobación y no encontrar ninguna variable, pero suspenderse inmediatamente. Como resultado, antes de tener tiempo de crear la variable con su siguiente instrucción, el contexto de ejecución cambia a la segunda copia. Esa tampoco encontrará la variable y decidirá seguir trabajando, como la primera. Para mayor claridad, el código fuente idéntico de las dos copias se muestra a continuación como dos columnas de instrucciones en el orden de su ejecución intercalada.

Copia 1

```
void OnStart()
{
   const string gv = "AlreadyRunning";

   if(GlobalVariableCheck(gv)) return;
   // no variable

   GlobalVariableSet(gv, 0);
   // "I am the first and only"
   while(!IsStopped())

   {
      ;

   }
   GlobalVariableDel(gv);

}
```

Copia 2

```
void OnStart()
{

   const string gv = "AlreadyRunning";

   if(GlobalVariableCheck(gv)) return;
   // still no variable

   GlobalVariableSet(gv, 0);
   // "No, I'm the first and only one"
   while(!IsStopped())
   {

      ;
   }

   GlobalVariableDel(gv);
}
```

Por supuesto, un esquema de este tipo para cambiar entre hilos tiene bastante de convencional. Pero en este caso, la posibilidad misma de violar la lógica del programa es importante, incluso en una sola cadena. Cuando hay muchos programas (hilos), aumenta la probabilidad de que se produzcan acciones imprevistas con recursos comunes. Esto puede ser suficiente para llevar al EA a una pérdida en el momento más inesperado o para obtener estimaciones de análisis técnico distorsionadas.

Lo más frustrante de este tipo de errores es que son muy difíciles de detectar. El compilador no es capaz de detectarlos, y se manifiestan esporádicamente en tiempo de ejecución. Pero que el error no se revele durante mucho tiempo no significa que no haya error.

Para resolver estos problemas es necesario sincronizar de alguna manera el acceso de todas las copias de los programas a los recursos compartidos (en este caso, a las variables globales).

En informática, existe un concepto especial, mutex (exclusión mutua), que es un objeto para proporcionar acceso exclusivo a un recurso compartido desde programas paralelos. Un mutex evita que los datos se pierdan o se corrompan debido a cambios asíncronos. Normalmente, acceder a un mutex sincroniza diferentes programas debido a que sólo uno de ellos puede editar los datos protegidos capturando el mutex en un momento determinado, y el resto se ve obligado a esperar hasta que el mutex se libere.

No hay mutex ya preparados en MQL5 en su forma pura, pero para las variables globales se puede obtener un efecto similar mediante la función que vamos a analizar a continuación.

bool GlobalVariableSetOnCondition(const string name, double value, double precondition)

La función establece un nuevo *value* de la variable global existente *name* siempre que su valor actual sea igual a *precondition*.

Si tiene éxito, la función devuelve *true*. En caso contrario, devuelve *false*, y el código de error estará disponible en *_LastError*. En concreto, si la variable no existe, la función generará un error ERR_GLOBALVARIABLE_NOT_FOUND (4501).

La función proporciona acceso atómico a una variable global, es decir, realiza dos acciones de forma inseparable: comprueba su valor actual y, si coincide con la condición, asigna a la variable un nuevo *value*.

El código de función equivalente puede representarse aproximadamente del siguiente modo (más adelante explicaremos por qué es «aproximadamente»):

```
bool GlobalVariableZetOnCondition(const string name, double value, double preconditio
{
   bool result = false;
   { /* enable interrupt protection */ }
   if(GlobalVariableCheck(name) && (GlobalVariableGet(name) == precondition))
   {
      GlobalVariableSet(name, value);
      result = true;
   }
   { /* disable interrupt protection */ }
   return result;
}
```

Implementar un código así, que funcione como se pretende, es imposible por dos razones. En primer lugar, no hay nada para implementar bloques que activan y desactivan la protección de interrupción en MQL5 puro (dentro de la función integrada *GlobalVariableSetOnCondition* esto lo proporciona el propio núcleo). En segundo lugar, la llamada a la función *GlobalVariableGet* modifica la última vez que se utilizó la variable, mientras que la función *GlobalVariableSetOnCondition* no la modifica si no se cumplió la condición previa.

Para demostrar cómo utilizar *GlobalVariableSetOnCondition*, pasaremos a un nuevo tipo de programa MQL: los servicios. Las estudiaremos en detalle en una sección aparte. Por ahora, hay que señalar que su estructura es muy similar a la de los scripts: para ambos, sólo hay una función principal (punto de entrada), *OnStart*. La única diferencia significativa es que el script se ejecuta en el gráfico, mientras que el servicio se ejecuta por sí mismo (en segundo plano).

La necesidad de sustituir los scripts por servicios se explica por el hecho de que el significado aplicado de la tarea en la que utilizamos *GlobalVariableSetOnCondition* consiste en contar el número de instancias en ejecución del programa, con la posibilidad de establecer un límite. En este caso, las colisiones con modificación simultánea del contador compartido sólo pueden producirse en el momento de lanzar varios programas. Sin embargo, en el caso de los scripts, es bastante difícil ejecutar varias copias de los mismos en distintos gráficos en un periodo de tiempo relativamente corto. Para los servicios, por el contrario, la interfaz del terminal dispone de un cómodo mecanismo de lanzamiento por lotes (en grupo). Además, todos los servicios activados se iniciarán automáticamente en el siguiente arranque del terminal.

El mecanismo propuesto para contar el número de copias también lo demandarán programas MQL de otros tipos. Dado que los Asesores Expertos y los indicadores permanecen unidos a los gráficos incluso cuando se apaga el terminal, la siguiente vez que se enciende todos los programas leen sus configuraciones y recursos compartidos casi simultáneamente. Por lo tanto, si se incorpora un límite en el número de copias en algunos Asesores Expertos e indicadores, es fundamental sincronizar el recuento basado en variables globales.

En primer lugar, consideremos un servicio que implementa el control de copias de un modo ingenuo, sin utilizar *GlobalVariableSetOnCondition*, y asegurémonos de que el problema de los fallos del contador es

real. Los servicios se encuentran en un subdirectorio dedicado en el directorio general del código fuente, así la ruta ampliada es *MQL5/Services/MQL5Book/p4/GlobalsNoCondition.mq5*.

Al principio del archivo de servicio debe haber una directiva:

```
#property service
```

En el servicio, proporcionaremos dos variables de entrada para establecer un límite en el número de copias que se permite ejecutar en paralelo y un retraso para emular la interrupción de la ejecución debido a una carga masiva en el disco y la CPU del ordenador, lo que suele ocurrir cuando se lanza el terminal. Esto facilitará la reproducción del problema sin tener que reiniciar el terminal muchas veces esperando que pierda la sincronización. Así pues, vamos a detectar un fallo que sólo puede producirse esporádicamente, pero que al mismo tiempo, si ocurre, está cargado de graves consecuencias.

```
input int limit = 1;       // Limit
input int startPause = 100;// Delay(ms)
```

La emulación del retraso se basa en la función *Sleep*.

```
void Delay()
{
   if(startPause > 0)
   {
      Sleep(startPause);
   }
}
```

En primer lugar, se declara una variable global temporal dentro de la función *OnStart*. Dado que está diseñada para contar las copias en ejecución del programa, no tiene sentido hacerla constante: cada vez que se inicia el terminal hay que contar de nuevo.

```
void OnStart()
{
   PRTF(GlobalVariableTemp(__FILE__));
   ...
```

Para evitar el caso de que un usuario cree de antemano una variable con el mismo nombre y le asigne un valor negativo, introducimos una protección.

```
   int count = (int)GlobalVariableGet(__FILE__);
   if(count < 0)
   {
      Print("Negative count detected. Not allowed.");
      return;
   }
```

A continuación, comienza el fragmento con la funcionalidad principal. Si el contador ya es mayor o igual que la cantidad máxima permitida, interrumpimos el lanzamiento del programa.

```
   if(count >= limit)
   {
      PrintFormat("Can't start more than %d copy(s)", limit);
      return;
   }
```

En caso contrario, incrementamos el contador en 1 y lo escribimos en la variable global. Con antelación, emulamos el retraso para provocar una situación en la que otro programa pudiera intervenir entre la lectura de una variable y su escritura en nuestro programa.

```
   Delay();
   PRTF(GlobalVariableSet(__FILE__, count + 1));
```

Si esto ocurre realmente, nuestra copia del programa incrementará y asignará un valor ya obsoleto e incorrecto. Se producirá una situación en la que, en otra copia del programa que se ejecuta en paralelo con la nuestra, el mismo valor *count* ya se ha procesado o se procesará de nuevo.

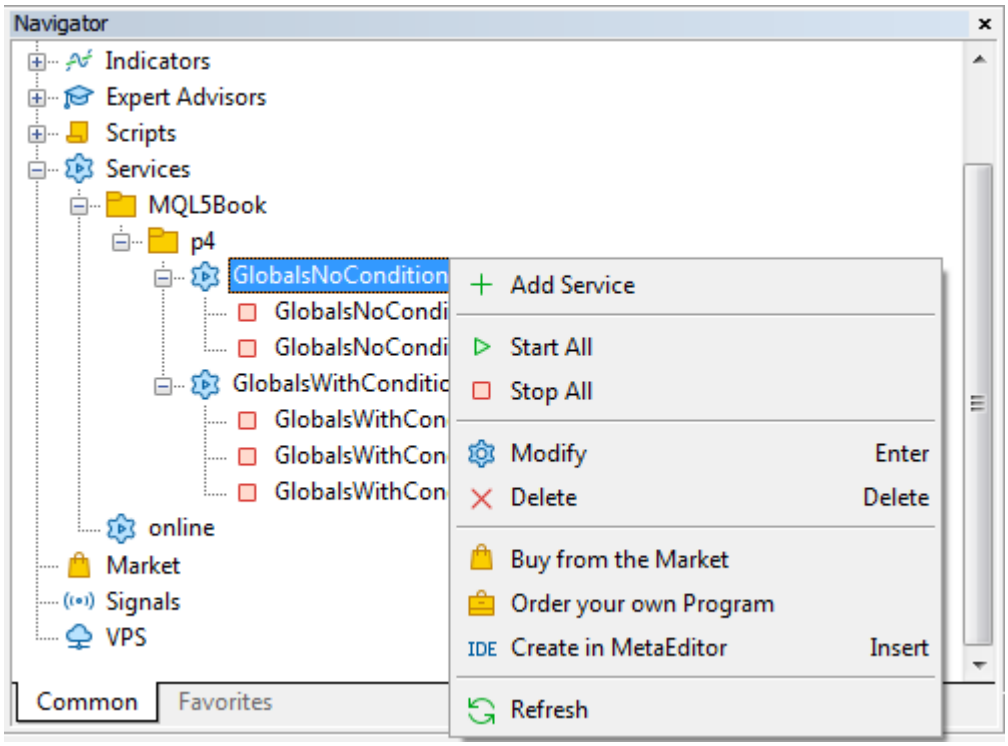
El trabajo útil del servicio está representado por el siguiente bucle:

```
   int loop = 0;
   while(!IsStopped())
   {
      PrintFormat("Copy %d is working [%d]...", count, loop++);
      // ...
      Sleep(3000);
   }
```

Después de que el usuario detenga el servicio (para ello, la interfaz dispone de un menú contextual, del que hablaremos más adelante), el ciclo terminará y tendremos que disminuir el contador.

```
   int last = (int)GlobalVariableGet(__FILE__);
   if(last > 0)
   {
      PrintFormat("Copy %d (out of %d) is stopping", count, last);
      Delay();
      PRTF(GlobalVariableSet(__FILE__, last - 1));
   }
   else
   {
      Print("Count underflow");
   }
}
```

Los servicios compilados entran en la rama correspondiente del «Navegador».



Servicios en el "Navegador" y menú contextual

Haciendo clic con el botón derecho, abriremos el menú contextual y crearemos dos instancias del servicio *GlobalsNoCondition.mq5* llamando dos veces al comando *Añadir servicio*. En este caso, cada vez se abrirá un cuadro de diálogo con la configuración del servicio, en el que deberá dejar los valores predeterminados para los parámetros.

> Es importante señalar que el comando *Añadir servicio* inicia inmediatamente el servicio creado. Pero no lo necesitamos. Por lo tanto, inmediatamente después de lanzar cada copia, tenemos que llamar de nuevo al menú contextual y ejecutar el comando *Detener* (si se selecciona una instancia específica), o *Detener todo* (si se selecciona el programa, es decir, todo el grupo de instancias generadas).

La primera instancia del servicio tendrá por defecto un nombre que coincide completamente con el archivo de servicio («GlobalsNoCondition»), y en todas las instancias posteriores se añadirá automáticamente un número creciente. En concreto, la segunda instancia aparece como «GlobalsNoCondition 1». El terminal le permite renombrar instancias en texto arbitrario usando el comando *Renombrar*, pero no lo haremos.

Ahora todo está listo para el experimento. Intentemos ejecutar dos instancias al mismo tiempo. Para ello, vamos a ejecutar el comando *Ejecutar todo* para la rama *GlobalsNoCondition* correspondiente.

Recordemos que en los parámetros se estableció un límite de 1 instancia. Sin embargo, según los registros, no funcionó.

```
GlobalsNoCondition GlobalVariableTemp(GlobalsNoCondition.mq5)=true / ok
GlobalsNoCondition 1 GlobalVariableTemp(GlobalsNoCondition.mq5)=false / GLOBALVARIABL
GlobalsNoCondition GlobalVariableSet(GlobalsNoCondition.mq5,count+1)=2021.08.31 17:47
GlobalsNoCondition Copy 0 is working [0]...
GlobalsNoCondition 1 GlobalVariableSet(GlobalsNoCondition.mq5,count+1)=2021.08.31 17:
GlobalsNoCondition 1 Copy 0 is working [0]...
GlobalsNoCondition Copy 0 is working [1]...
GlobalsNoCondition 1 Copy 0 is working [1]...
GlobalsNoCondition Copy 0 is working [2]...
GlobalsNoCondition 1 Copy 0 is working [2]...
GlobalsNoCondition Copy 0 is working [3]...
GlobalsNoCondition 1 Copy 0 is working [3]...
GlobalsNoCondition Copy 0 (out of 1) is stopping
GlobalsNoCondition GlobalVariableSet(GlobalsNoCondition.mq5,last-1)=2021.08.31 17:47:
GlobalsNoCondition 1 Count underflow
```

Ambas copias «piensan» que son el número 0 (salida «Copia 0» del bucle de trabajo) y su número total es erróneamente igual a 1 porque ese es el valor que ambas copias tienen almacenado en la variable contador.

Es por esto que cuando se detienen los servicios (el comando *Detener todo*), recibimos un mensaje sobre un estado incorrecto («Count underflow»): al fin y al cabo, cada una de las copias está tratando de disminuir el contador en 1, y como resultado, la que se ejecutó en segundo lugar recibió un valor negativo.

Para resolver el problema debe utilizar la función *GlobalVariableSetOnCondition*. A partir del código fuente del servicio anterior se ha preparado una versión mejorada *GlobalsWithCondition.mq5*. En general, reproduce la lógica de su predecesora, pero hay diferencias significativas.

En lugar de limitarse a llamar a *GlobalVariableSet* para aumentar el contador, hubo que escribir una estructura más compleja.

```
const int maxRetries = 5;
int retry = 0;

while(count < limit && retry < maxRetries)
{
   Delay();
   if(PRTF(GlobalVariableSetOnCondition(__FILE__, count + 1, count))) break;
   // condition is not met (count is obsolete), assignment failed,
   // let's try again with a new condition if the loop does not exceed the limit
   count = (int)GlobalVariableGet(__FILE__);
   PrintFormat("Counter is already altered by other instance: %d", count);
   retry++;
}

if(count == limit || retry == maxRetries)
{
   PrintFormat("Start failed: count: %d, retries: %d", count, retry);
   return;
}
...
```

Dado que la función *GlobalVariableSetOnCondition* no puede escribir un nuevo valor de contador, si el antiguo ya está obsoleto, volvemos a leer la variable global en el bucle y repetimos los intentos de incrementarla hasta que se supere el valor máximo permitido del contador. La condición de bucle también limita el número de intentos. Si el bucle termina con una violación de una de las condiciones, entonces la actualización del contador ha fallado y el programa no debe continuar ejecutándose.

Estrategias de sincronización

En teoría, existen varias estrategias estándar para implementar la captura de recursos compartidos.

La primera consiste en comprobar suavemente si el recurso está libre y bloquearlo sólo si lo está en ese momento. Si está ocupado, el algoritmo planifica el siguiente intento después de un cierto periodo, y en ese momento se dedica a otras tareas (por eso este enfoque es preferible para programas que tienen varias áreas de actividad o responsabilidad). Un análogo de este esquema de comportamiento en la transcripción para la función *GlobalVariableSetOnCondition* es una sola llamada, sin bucle, saliendo del bloque actual en caso de fallo. El cambio variable se pospone «en espera de tiempos mejores».

La segunda estrategia es más persistente, y se aplica en nuestro script. Se trata de un bucle que repite la solicitud de un recurso durante un número determinado de veces, o un tiempo predefinido (el periodo de tiempo de espera permitido para el recurso). Si el bucle expira y no se alcanza un resultado positivo (la llamada a la función *GlobalVariableSetOnCondition* nunca devolvió «true»), el programa también sale del bloque actual y probablemente planea volver a intentarlo más tarde.

Por último, la tercera estrategia, la más dura, consiste en solicitar un recurso «hasta las últimas consecuencias». Se puede ver como un bucle infinito con una llamada a una función. Este enfoque tiene sentido utilizarlo en programas que se centran en una tarea específica y no pueden seguir funcionando sin un recurso incautado. En MQL5, utilice el bucle *while(!IsStopped())* para esto y no se olvide de llamar a *Sleep* dentro.

Es importante señalar aquí el problema potencial de la apropiación «dura» de múltiples recursos. Imagine que un programa MQL modifica varias variables globales (lo cual es, en teoría, una situación habitual). Si una copia captura una variable, y la segunda copia captura otra, y ambas esperan la liberación, se producirá su bloqueo mutuo (deadlock).

Basándose en lo anterior, el uso compartido de variables globales y otros recursos (por ejemplo, archivos) debe diseñarse y analizarse cuidadosamente para detectar bloqueos y las denominadas «condiciones de carrera», cuando la ejecución paralela de programas conduce a un resultado indefinido (dependiendo del orden de su trabajo).

Tras la finalización del ciclo de trabajo en la nueva versión del servicio, el algoritmo de disminución del contador se ha modificado de forma similar.

```
retry = 0;
int last = (int)GlobalVariableGet(__FILE__);
while(last > 0 && retry < maxRetries)
{
   PrintFormat("Copy %d (out of %d) is stopping", count, last);
   Delay();
   if(PRTF(GlobalVariableSetOnCondition(__FILE__, last - 1, last))) break;
   last = (int)GlobalVariableGet(__FILE__);
   retry++;
}

if(last <= 0)
{
   PrintFormat("Unexpected exit: %d", last);
}
else
{
   PrintFormat("Stopped copy %d: count: %d, retries: %d", count, last, retry);
}
```

Como experimento, vamos a crear tres instancias para el nuevo servicio. En la configuración de cada uno de ellos, en el parámetro Limit, especificamos dos instancias (para realizar una prueba en condiciones modificadas). Recordemos que la creación de cada instancia la lanza inmediatamente, lo que no necesitamos, y por lo tanto cada instancia recién creada debe ser detenida.

Las instancias recibirán los nombres predeterminados «GlobalsWithCondition», «GlobalsWithCondition 1» y «GlobalsWithCondition 2».

Cuando todo esté listo, ejecutamos todas las instancias a la vez y obtenemos algo como esto en el registro.

```
GlobalsWithCondition 2 GlobalVariableTemp(GlobalsWithCondition.mq5)= »
 » false / GLOBALVARIABLE_EXISTS(4502)
GlobalsWithCondition 1 GlobalVariableTemp(GlobalsWithCondition.mq5)= »
 » false / GLOBALVARIABLE_EXISTS(4502)
GlobalsWithCondition GlobalVariableTemp(GlobalsWithCondition.mq5)=true / ok
GlobalsWithCondition GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,count+1,co
» true / ok
GlobalsWithCondition 1 GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,count+1,
» false / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalsWithCondition 2 GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,count+1,
» false / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalsWithCondition 1 Counter is already altered by other instance: 1
GlobalsWithCondition Copy 0 is working [0]...
GlobalsWithCondition 2 Counter is already altered by other instance: 1
GlobalsWithCondition 1 GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,count+1,
GlobalsWithCondition 1 Copy 1 is working [0]...
GlobalsWithCondition 2 GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,count+1,
» false / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalsWithCondition 2 Counter is already altered by other instance: 2
GlobalsWithCondition 2 Start failed: count: 2, retries: 2
GlobalsWithCondition Copy 0 is working [1]...
GlobalsWithCondition 1 Copy 1 is working [1]...
GlobalsWithCondition Copy 0 is working [2]...
GlobalsWithCondition 1 Copy 1 is working [2]...
GlobalsWithCondition Copy 0 is working [3]...
GlobalsWithCondition 1 Copy 1 is working [3]...
GlobalsWithCondition Copy 0 (out of 2) is stopping
GlobalsWithCondition GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,last-1,las
GlobalsWithCondition Stopped copy 0: count: 2, retries: 0
GlobalsWithCondition 1 Copy 1 (out of 1) is stopping
GlobalsWithCondition 1 GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,last-1,l
GlobalsWithCondition 1 Stopped copy 1: count: 1, retries: 0
```

En primer lugar, preste atención a la demostración aleatoria, pero al mismo tiempo visual, del efecto descrito del cambio de contexto para programas que se ejecutan en paralelo. La primera instancia en crear una variable temporal ha sido «GlobalsWithCondition» sin número: esto se puede ver en el resultado de la función *GlobalVariableTemp*, que es *true*. Sin embargo, en el registro, esta línea ocupa sólo la tercera posición, y las dos anteriores contienen los resultados de llamar a la misma función en copias bajo los nombres con números 1 y 2; en ellas, la función *GlobalVariableTemp* devolvía *false*. Esto significa que estas copias comprobaron la variable más tarde, aunque sus hilos superaron al hilo no numerado «GlobalsWithCondition» y terminaron antes en el registro.

Pero volvamos a nuestro algoritmo principal de recuento de programas. La instancia «GlobalsWithCondition» fue la primera en pasar la comprobación, y empezó a funcionar con el identificador interno «Copy 0» (no podemos averiguar en el código de servicio cómo nombró el usuario a la instancia: no existe tal función en la API de MQL5, al menos de momento).

Gracias a la función *GlobalVariableSetOnCondition*, en las instancias 1 y 2 («GlobalsWithCondition 1», «GlobalsWithCondition 2»), se detectó el hecho de modificar el contador: era 0 al principio, pero GlobalsWithCondition lo incrementó en 1. Ambas instancias tardías emiten el mensaje «El contador ya ha sido alterado por otra instancia: 1». Una de estas instancias («GlobalsWithCondition 1») por delante del número 2, consiguió obtener un nuevo valor de 1 de la variable y aumentarlo a 2. Esto se indica mediante una llamada exitosa *GlobalVariableSetOnCondition* (devolvió *true*). Y había un mensaje de que empezaba a funcionar, «Copia 1 está funcionando».

El hecho de que el valor del contador interno sea el mismo que el número de instancia externo es pura coincidencia. Bien podría ser que «GlobalsWithCondition 2» hubiera empezado antes que «GlobalsWithCondition 1» (o en alguna otra secuencia, dado que hay tres copias). Entonces, la numeración exterior e interior serían diferentes. Puede repetir el experimento arrancando y parando todos los servicios muchas veces, y la secuencia en la que las instancias incrementan la variable contador será muy probablemente diferente. Pero en cualquier caso, el límite del número total cortará una instancia extra.

Cuando la última instancia de «GlobalsWithCondition 2» obtiene acceso a una variable global, el valor 2 ya está almacenado allí. Puesto que éste es el límite que hemos fijado, el programa no se inicia.

```
GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,count+1,count)= »
» false / GLOBALVARIABLE_NOT_FOUND(4501)
Counter is already altered by other instance: 2
Start failed: count: 2, retries: 2
```

Más adelante, las copias de «GlobalsWithCondition» y «GlobalsWithCondition 1» «giran» en el ciclo de trabajo hasta que se detienen los servicios.

Puede intentar detener sólo una instancia. Entonces será posible lanzar otra que previamente haya recibido una prohibición de ejecución por superar la cuota.

Por supuesto, la versión propuesta de protección contra la modificación paralela sólo es eficaz para coordinar el comportamiento de sus propios programas, pero no para limitar una copia única de la versión de demostración, ya que el usuario puede simplemente borrar la variable global. Para ello, las variables globales se pueden utilizar de una manera diferente, en relación con el ID del gráfico: un programa MQL funciona sólo mientras su variable global creada contenga su ID artes gráficas. Otras formas de controlar los datos compartidos (contadores y demás información) son las proporcionadas por recursos y base de datos.

4.6.7 Descarga de variables globales en disco

Para optimizar el rendimiento, las variables globales residen en memoria mientras se ejecuta el terminal. Sin embargo, como sabemos, las variables se almacenan entre sesiones en un archivo especial. Esto se aplica a todas las variables globales excepto a las variables temporales. Normalmente, la escritura de variables en un archivo se produce cuando se cierra el terminal. Sin embargo, si su ordenador se bloquea de repente, sus datos pueden perderse. Por lo tanto, puede ser útil iniciar forzosamente la escritura para garantizar la seguridad de los datos en cualquier situación imprevista. Para ello, la API de MQL5 proporciona la función *GlobalVariablesFlush*.

void GlobalVariablesFlush()

La función obliga a escribir en disco el contenido de las variables globales. La función no tiene parámetros y no devuelve nada.

El ejemplo más sencillo se ofrece en el script *GlobalsFlush.mq5*.

```
void OnStart()
{
   GlobalVariablesFlush();
}
```

Con él, puede vaciar las variables en el disco en cualquier momento, si es necesario. Puede utilizar su administrador de archivos preferido y asegurarse de que la fecha y la hora del archivo *gvariables.dat*

cambian inmediatamente después de ejecutar el script. Sin embargo, tenga en cuenta que el archivo sólo se actualizará si las variables globales han sido editadas de alguna manera o simplemente leídas (esto cambia el tiempo de acceso) desde que se guardaron la vez anterior.

Este script es útil para aquellos que mantienen el terminal encendido durante mucho tiempo, y en él se ejecutan programas que modifican variables globales.

4.7 Funciones para trabajar con el tiempo

El tiempo es un factor fundamental en la mayoría de los procesos y desempeña un importante papel aplicado al trading.

Como sabemos, el principal sistema de coordenadas en trading se basa en dos dimensiones: precio y tiempo. Se muestran en el gráfico a lo largo de los ejes vertical y horizontal, respectivamente. Más adelante abordaremos otro eje importante, que puede representarse como perpendicular a los dos primeros y que se adentra en el gráfico, en el que se marcan los volúmenes de trading. Pero por ahora, centrémonos en el tiempo.

Esta medida es común a todos los gráficos, utiliza las mismas unidades de medida y, por extraño que parezca, se caracteriza por la constancia (el transcurso del tiempo es predecible).

El terminal ofrece una plétora de herramientas integradas relacionadas con el cálculo y el análisis del tiempo. Así pues, nos familiarizaremos con ellos gradualmente, a medida que avancemos por los capítulos del libro, de lo sencillo a lo complejo.

En este capítulo estudiaremos las funciones que permiten controlar el tiempo y pausar la actividad del programa durante un intervalo especificado.

En el capítulo Fecha y hora, en la sección sobre transformación de datos, hemos visto ya un par de funciones relacionadas con el tiempo: *TimeToStruct* y *StructToTime*. Estas funciones dividen un valor del tipo *datetime* en componentes o viceversa, construyen *datetime* a partir de campos individuales: recuerde que se resumen en la estructura *MqlDateTime*.

```
struct MqlDateTime
{
   int year;         // year (1970 – 3000)
   int mon;          // month (1 – 12)
   int day;          // day (1 – 31)
   int hour;         // hours (0 – 23)
   int min;          // minutes (0 – 59)
   int sec;          // seconds (0 – 59)
   int day_of_week; // day of the week, numbered from 0 (Sunday) to 6 (Saturday)
                     // according to enum ENUM_DAY_OF_WEEK
   int day_of_year; // ordinal number of the day in the year, starting from 0 (Januar
};
```

Pero, ¿de dónde puede obtener un programa MQL el valor *datetime*?

Por ejemplo, los precios y tiempos históricos se reflejan en cotizaciones, mientras que los datos actuales en directo llegan en forma de ticks. Ambos tienen marcas de tiempo, que aprenderemos a obtener en las secciones pertinentes sobre series temporales y eventos terminales. Sin embargo, un programa MQL puede consultar la hora actual por sí mismo (sin precios ni otra información de trading) utilizando varias funciones.

Se necesitaban varias funciones porque el sistema está distribuido: consta de un terminal de cliente y un servidor intermediario situados en partes arbitrarias del mundo que, muy probablemente, pertenezcan a husos horarios diferentes.

Cualquier huso horario se caracteriza por un desfase temporal con respecto al punto de referencia global del tiempo, la hora del meridiano de Greenwich (GMT). Por regla general, el desfase de una zona horaria es un número entero de horas N (aunque también hay zonas exóticas con un paso de media hora) y, por lo tanto, se indica como GMT + N o GMT - N, dependiendo de si la zona está al este o al oeste del meridiano. Por ejemplo, Europa continental, situada al este de Londres, utiliza la Hora Central Europea (CET), igual a GMT + 1, o la Hora de Europa Oriental (EET), igual a GMT + 2, mientras que en América existen zonas «negativas», como la Hora Estándar del Este (EST) o GMT-5.

> Hay que tener en cuenta que GMT corresponde al tiempo astronómico (solar), que es ligeramente no lineal, ya que la rotación de la Tierra se está ralentizando poco a poco. A este respecto, en las últimas décadas se ha producido una transición a un sistema de cronometraje más preciso (basado en relojes atómicos), en el que la hora mundial se denomina Hora Universal Coordinada (UTC). En muchas áreas de aplicación, incluido el trading, la diferencia entre GMT y UTC no es significativa, por lo que las designaciones de zonas horarias en el nuevo formato UTC±N y el antiguo GMT±N deben considerarse análogas. Por ejemplo, muchos brokers ya indican las horas de sesión en UTC en sus especificaciones, mientras que la API de MQL5 ha utilizado históricamente la notación GMT.

La API de MQL5 permite conocer la hora actual del terminal (de hecho, la hora local del ordenador) y la hora del servidor, que son devueltas por las funciones *TimeLocal* y *TimeCurrent*, respectivamente. Además, un programa MQL puede obtener la hora GMT actual (función *TimeGMT*) basándose en la configuración de la zona horaria de Windows. Así, un operador de trading y un programador obtienen una vinculación de la hora local con la global, y por la diferencia entre la hora local y la del servidor, se puede determinar la «zona horaria» del servidor y de las cotizaciones. Aquí hay no obstante un par de puntos interesantes.

En primer lugar, en muchos países existe la práctica de cambiar al horario de verano (DST). Normalmente, esto significa añadir 1 hora al horario estándar (invierno) desde marzo/abril hasta octubre/noviembre (en el hemisferio norte; en el sur es al revés). Al mismo tiempo, la hora GMT/UTC permanece siempre constante, es decir, no está sujeta a la corrección DST, por lo que son potencialmente posibles varias opciones de convergencia o discrepancia entre la hora del cliente y la del servidor:

- las fechas de transición pueden variar de un país a otro;
- algunos países no aplican el horario de verano.

Debido a esto, algunos programas MQL necesitan realizar un seguimiento de dichos cambios de zona horaria si los algoritmos se basan en la referencia a la hora intradía (por ejemplo, comunicados de prensa) y no a los movimientos de precios o concentraciones de volumen.

Y si la traducción de la hora en el ordenador del usuario es bastante fácil de determinar, gracias a la función *TimeDaylightSavings*, entonces no hay ningún análogo listo para la hora del servidor.

En segundo lugar, el probador regular de MetaTrader 5, en el que podemos depurar o evaluar programas MQL de tipos tales como Asesores Expertos e indicadores, no emula, por desgracia, el tiempo del servidor de trading. En lugar de ello, las tres funciones anteriores *TimeLocal*, *TimeGMT* y *TimeCurrent* devolverán la misma hora, es decir, la zona horaria será siempre virtualmente GMT.

> **Hora absoluta y relativa**
>
> La contabilización del tiempo en los algoritmos, como en la vida, puede realizarse en coordenadas absolutas o relativas. Cada momento del pasado, del presente y del futuro se describe mediante un

valor absoluto al que podemos referirnos para indicar el inicio de un periodo contable o el momento en que se publica una noticia económica. Es esta hora la que almacenamos en MQL5 utilizando el tipo *datetime*. Al mismo tiempo, a menudo es necesario mirar al futuro o retroceder al pasado durante un número determinado de unidades de tiempo desde el momento actual. En este caso, no nos interesa el valor absoluto, sino el intervalo de tiempo.

En concreto, los algoritmos tienen el concepto de timeout, que es un periodo de tiempo durante el cual se debe realizar una determinada acción, y si no se realiza por cualquier motivo, la cancelamos y dejamos de esperar el resultado (porque, al parecer, algo ha ido mal). Puede medir el intervalo en distintas unidades: horas, segundos, milisegundos o incluso microsegundos (al fin y al cabo, ahora los ordenadores son rápidos).

En MQL5, algunas funciones relacionadas con el tiempo trabajan con valores absolutos (por ejemplo, *TimeLocal*, *TimeCurrent*) y la parte, con intervalos (por ejemplo, *GetTickCount*, *GetMicrosecondCount*).

Sin embargo, la medición de intervalos o la activación del programa a intervalos especificados se puede implementar no sólo a través de las funciones de esta sección, sino también utilizando temporizadores integrados que funcionan según el conocido principio de un reloj despertador. Cuando están activados, utilizan eventos especiales para notificar a los programas MQL y a las funciones que implementamos que manejen estos eventos, *OnTimer* (son similares a *OnStart*). Trataremos este aspecto de la gestión del tiempo en una sección aparte, después de estudiar el concepto general de eventos en MQL5 (véase Visión general de las funciones de gestión de eventos).

4.7.1 Hora local y del servidor

Siempre hay dos tipos de tiempo en la plataforma MetaTrader 5: local (cliente) y servidor (broker).

La hora local corresponde a la hora del ordenador en el que se ejecuta el terminal, y aumenta continuamente, al mismo ritmo que en el mundo real.

El tiempo de los servidores fluye de forma diferente. La base para ello la establece la hora en el ordenador del broker; sin embargo, el cliente recibe información al respecto sólo junto con los siguientes cambios de precio, que se empaquetan en estructuras especiales llamadas ticks (véase la sección sobre *MqlTick*) y se pasan a los programas MQL mediante eventos.

Así, la hora actualizada del servidor sólo se conoce en el terminal como resultado de un cambio en el precio de al menos un instrumento financiero en el mercado, es decir, de entre los seleccionados en la ventana de Observación de Mercado. La última hora conocida del servidor se muestra en la barra de título de esta ventana. Si no hay ticks, la hora del servidor en el terminal permanece fija. Esto se nota especialmente los fines de semana y días festivos, cuando todas las bolsas y plataformas de Forex están cerradas.

En concreto, en un domingo, lo más probable es que la hora del servidor sea la del viernes por la tarde. Las únicas excepciones son las instancias de MetaTrader 5 que ofrecen instrumentos de negociación continua, como las criptodivisas. No obstante, incluso en este caso, durante los periodos de baja volatilidad la hora del servidor puede retrasarse notablemente con respecto a la hora local.

Todas las funciones de esta sección operan con la hora con una precisión de hasta un segundo (la precisión de la representación de la hora en el tipo *datetime*.

Para obtener la hora local y del servidor, la API de MQL5 proporciona tres funciones: *TimeLocal*, *TimeCurrent* y *TimeTradeServer*. Las tres funciones tienen dos versiones del prototipo: la primera devuelve el tiempo como un valor del tipo *datetime*, y la segunda acepta adicionalmente por referencia y rellena la estructura *MqlDateTime* con componentes de tiempo.

datetime TimeLocal()
datetime TimeLocal(MqlDateTime &dt)

La función devuelve la hora local del ordenador en el formato *datetime*.

Es importante tener en cuenta que la hora incluye el horario de verano si está activado. Es decir, *TimeLocal* es igual a la hora estándar de la zona horaria del ordenador, menos la corrección *TimeDaylightSavings*. Condicionalmente, la fórmula puede representarse del siguiente modo:

```
TimeLocal summer() = TimeLocal winter() - TimeDaylightSavings()
```

Aquí *TimeDaylightSavings* suele equivaler a -3600, es decir, adelantar el reloj 1 hora (se pierde 1 hora). Así pues, el valor estival de *TimeLocal* es mayor que el valor invernal (con igual hora astronómica del día) con respecto a UTC. Por ejemplo, si en invierno *TimeLocal* equivale a UTC+2, en verano es UTC+3. UTC puede obtenerse utilizando la función *TimeGMT*.

datetime TimeCurrent()
datetime TimeCurrent(MqlDateTime &dt)

La función devuelve la última hora conocida del servidor en el formato *datetime*. Esta es la hora de llegada de la última cotización de la lista de todos los instrumentos financieros de Observación de Mercado. La única excepción es el manejador de eventos *OnTick* en los Asesores Expertos, donde esta función devolverá la hora del tick procesado (incluso si ya han aparecido ticks con una hora más reciente en Observación de Mercado).

Además, tenga en cuenta que la hora en el eje horizontal de todos los gráficos en MetaTrader 5 corresponde a la hora del servidor (en la historia). La última barra (la actual, más a la derecha) contiene *TimeCurrent*. Consulte los detalles en la sección Gráficos.

datetime TimeTradeServer()
datetime TimeTradeServer(MqlDateTime &dt)

La función devuelve la hora actual estimada del servidor de trading. A diferencia de *TimeCurrent*, cuyos resultados pueden no cambiar si no hay nuevas cotizaciones, *TimeTradeServer* permite obtener una estimación del tiempo de servidor en continuo aumento. El cálculo se basa en la última diferencia conocida entre las zonas horarias del cliente y del servidor, que se añade a la hora local actual.

En el comprobador, el valor *TimeTradeServer* es siempre igual a *TimeCurrent*.

En el script *TimeCheck.mq5* se ofrece un ejemplo de cómo funcionan las funciones.

La función principal tiene un bucle infinito que registra todos los tipos de tiempo cada segundo hasta que el usuario detiene el script.

```
void OnStart()
{
   while(!IsStopped())
   {
      PRTF(TimeLocal());
      PRTF(TimeCurrent());
      PRTF(TimeTradeServer());
      PRTF(TimeTradeServerExact());
      Sleep(1000);
   }
}
```

Además de las funciones estándar, aquí se aplica una función personalizada *TimeTradeServerExact*.

```
datetime TimeTradeServerExact()
{
   enum LOCATION
   {
      LOCAL,
      SERVER,
   };
   static datetime now[2] = {}, then[2] = {};
   static int shiftInHours = 0;
   static long shiftInSeconds = 0;

   // constantly detect the last 2 timestamps here and there
   then[LOCAL] = now[LOCAL];
   then[SERVER] = now[SERVER];
   now[LOCAL] = TimeLocal();
   now[SERVER] = TimeCurrent();

   // at the first call we don't have 2 labels yet,
   // needed to calculate the stable difference
   if(then[LOCAL] == 0 && then[SERVER] == 0) return 0;

   // when the time course is the same on the client and on the server,
   // and the server is not "frozen" due to weekends/holidays,
   // updating difference
   if(now[LOCAL] - now[SERVER] == then[LOCAL] - then[SERVER]
   && now[SERVER] != then[SERVER])
   {
      shiftInSeconds = now[LOCAL] - now[SERVER];
      shiftInHours = (int)MathRound(shiftInSeconds / 3600.0);
      // debug print
      PrintFormat("Shift update: hours: %d; seconds: %lld", shiftInHours, shiftInSecc
   }

   // NB: The built-in function TimeTradeServer calculates like this:
   //                TimeLocal() - shiftInHours * 3600
   return (datetime)(TimeLocal() - shiftInSeconds);
}
```

Era necesario porque el algoritmo de la función integrada *TimeTradeServer* puede no convenirle a todo el mundo. La función integrada encuentra la diferencia entre la hora local y la del servidor en horas (es decir, la diferencia de zona horaria), y luego obtiene la hora del servidor como una corrección de la hora local para esta diferencia. Como resultado, si los minutos y segundos van en el cliente y el servidor de forma no sincrónica (lo que es muy probable), la aproximación estándar de la hora del servidor mostrará los minutos y segundos del cliente, no del servidor.

Lo ideal sería que los relojes locales de todos los ordenadores estuvieran sincronizados con la hora mundial, pero en la práctica se producen desviaciones. Así, si se produce algún desplazamiento, por pequeño que sea, en uno de los lados, *TimeTradeServer* ya no puede repetir la hora en el servidor con la máxima precisión.

En nuestra implementación de la misma función en MQL5 no redondeamos la diferencia entre la hora del cliente y la del servidor a husos horarios. En lugar de ello se utiliza en el cálculo la diferencia exacta

en segundos. Por eso *TimeTradeServerExact* devuelve la hora en la que los minutos y segundos van exactamente igual que en el servidor.

A continuación se muestra un ejemplo de registro generado por el script.

```
TimeLocal()=2021.09.02 16:03:34 / ok
TimeCurrent()=2021.09.02 15:59:39 / ok
TimeTradeServer()=2021.09.02 16:03:34 / ok
TimeTradeServerExact()=1970.01.01 00:00:00 / ok
```

Puede verse que las zonas horarias del cliente y del servidor son las mismas, pero hay una ausencia de sincronización de varios minutos (para mayor claridad). En la primera llamada, *TimeTradeServerExact* devolvió 0. Además, los datos para calcular la diferencia ya llegarán, y veremos los cuatro tipos de tiempo, «caminando» uniformemente con un intervalo de unos pocos segundos.

```
TimeLocal()=2021.09.02 16:03:35 / ok
TimeCurrent()=2021.09.02 15:59:40 / ok
TimeTradeServer()=2021.09.02 16:03:35 / ok
Shift update: hours: 0; seconds: 235
TimeTradeServerExact()=2021.09.02 15:59:40 / ok
TimeLocal()=2021.09.02 16:03:36 / ok
TimeCurrent()=2021.09.02 15:59:41 / ok
TimeTradeServer()=2021.09.02 16:03:36 / ok
Shift update: hours: 0; seconds: 235
TimeTradeServerExact()=2021.09.02 15:59:41 / ok
TimeLocal()=2021.09.02 16:03:37 / ok
TimeCurrent()=2021.09.02 15:59:41 / ok
TimeTradeServer()=2021.09.02 16:03:37 / ok
TimeTradeServerExact()=2021.09.02 15:59:42 / ok
TimeLocal()=2021.09.02 16:03:38 / ok
TimeCurrent()=2021.09.02 15:59:43 / ok
TimeTradeServer()=2021.09.02 16:03:38 / ok
TimeTradeServerExact()=2021.09.02 15:59:43 / ok
```

4.7.2 Horario de verano (local)

Para determinar si los relojes locales se cambian al horario de verano, MQL5 proporciona la función *TimeDaylightSavings*, que toma la configuración de su sistema operativo.

Determinar el horario de verano en un servidor no es tan fácil. Para ello, es necesario aplicar el análisis MQL5 de cotizaciones, eventos de calendario económico o una hora de rollover/swap en el historial de trading de la cuenta. En el siguiente ejemplo mostraremos una de las opciones.

int TimeDaylightSavings()

La función devuelve la corrección en segundos si se ha aplicado el horario de verano. El horario de invierno es estándar para cada zona horaria, por lo que la corrección para este periodo es cero. En forma condicional, la fórmula para obtener la corrección puede escribirse del siguiente modo:

```
TimeDaylightSavings() = TimeLocal winter() - TimeLocal summer()
```

Por ejemplo, si la zona horaria estándar (*winter*) es igual a UTC+3 (es decir, la hora de la zona está 3 horas por delante de UTC), durante la transición al horario de verano (*summer*) añadimos 1 hora y obtenemos UTC+4. Donde *TimeDaylightSavings* devolverá -3600.

Un ejemplo de uso de la función se ofrece en el script *TimeSummer.mq5*, que también sugiere una de las posibles formas empíricas de identificar el modo apropiado en el servidor.

```
void OnStart()
{
   PRTF(TimeLocal());           // local time of the terminal
   PRTF(TimeCurrent());         // last known server time
   PRTF(TimeTradeServer());     // estimated server time
   PRTF(TimeGMT());             // GMT time (calculation from local via time zone shif
   PRTF(TimeGMTOffset());       // time zone shift compare to GMT, in seconds
   PRTF(TimeDaylightSavings());// correction for summer time in seconds
   ...
```

En primer lugar, vamos a mostrar todos los tipos de tiempo y su corrección proporcionada por MQL5 (las funciones *TimeGMT* y *TimeGMTOffset* se presentarán en la siguiente sección sobre Hora universal, pero su significado ya debería estar claro en general por la descripción anterior).

Se supone que el script se ejecuta en días de trading. Las entradas del registro corresponderán a la configuración de su ordenador y del servidor del broker.

```
TimeLocal()=2021.09.09 22:06:17 / ok
TimeCurrent()=2021.09.09 22:06:10 / ok
TimeTradeServer()=2021.09.09 22:06:17 / ok
TimeGMT()=2021.09.09 19:06:17 / ok
TimeGMTOffset()=-10800 / ok
TimeDaylightSavings()=0 / ok
```

En este caso, la zona horaria del cliente está a 3 horas de GMT (UTC+3), no hay ajuste por horario de verano.

Ahora echemos un vistazo al servidor. Basándonos en el valor de la función *TimeCurrent* podemos determinar la hora actual del servidor, pero no su zona horaria estándar, ya que esta hora puede implicar la transición al horario de verano (MQL5 no proporciona información sobre si se utiliza en absoluto y si está actualmente activada).

Para determinar la zona horaria real del servidor y el horario de verano, utilizaremos el hecho de que la traducción de la hora del servidor afecta a las cotizaciones. Como la mayoría de los métodos empíricos para resolver problemas, éste puede no dar resultados completamente correctos en determinadas circunstancias. Si la comparación con otras fuentes muestra discrepancias, debe elegirse otro método.

El mercado Forex abre el domingo a las 22:00 UT (lo que corresponde al inicio de las operaciones matutinas en la región Asia-Pacífico) y cierra el viernes a las 22:00 (el cierre de las operaciones en América). Esto significa que en los servidores de la zona UTC+2 (Europa oriental), las primeras barras aparecerán exactamente a las 0 horas 0 minutos del lunes. Según la hora de Europa central, que corresponde a UTC+1, la semana de trading comienza a las 23:00 del domingo.

Una vez calculadas las estadísticas del desplazamiento intradiario de la primera barra H1 después de cada pausa de fin de semana, obtendremos una estimación de la zona horaria del servidor. Por supuesto, para ello es mejor utilizar el instrumento más líquido de Forex, que es EURUSD.

Si en las estadísticas de un periodo anual se encuentran dos desplazamientos intradiarios máximos y estos están situados uno al lado del otro, significará que el broker está cambiando al horario de verano y viceversa.

Tenga en cuenta que los periodos de verano e invierno no son iguales. Así, al cambiar al horario de verano a principios de marzo y volver al de invierno a principios de noviembre, tenemos unos 8 meses de horario de verano. Esto afectará a la proporción de máximos en las estadísticas.

Al disponer de dos husos horarios podemos determinar fácilmente cuál de ellos está activo en ese momento y, de ese modo, averiguar la presencia o ausencia de una corrección para el horario de verano.

Al cambiar los relojes al horario de verano, la zona horaria del broker cambiará de UTC+2 a UTC+3, lo que desplazará el comienzo de la semana de las 22:00 a las 21:00. Esto afectará a la estructura de las barras H1: en el gráfico veremos tres barras el domingo por la tarde en lugar de dos.

Cambio de horas del horario de invierno (UTC+2) al de verano (UTC+3) en el gráfico EURUSD H1

Para ello, disponemos de una función independiente, *ServerTimeZone*. La llamada a la función integrada *CopyTime* se encarga de obtener cotizaciones, o marcas de tiempo de barras, para ser más precisos (estudiaremos esta función en la sección sobre acceso a series temporales).

```
ServerTime ServerTimeZone(const string symbol = NULL)
{
  const int year = 365 * 24 * 60 * 60;
  datetime array[];
  if(PRTF(CopyTime(symbol, PERIOD_H1, TimeCurrent() - year, TimeCurrent(), array)) >
  {
     // here we get about 6000 bars in the array
     const int n = ArraySize(array);
     PrintFormat("Got %d H1 bars, ~%d days", n, n / 24);
     // (-V-) loop through H1 bars
     ...
  }
}
```

La función *CopyTime* recibe como parámetros el instrumento de trabajo, el marco temporal H1 y el intervalo de fechas del último año. El valor NULL en lugar del instrumento significa el símbolo del gráfico actual en el que se colocará el script, por lo que se recomienda seleccionar la ventana con EURUSD. La constante PERIOD_H1 corresponde a H1, como se puede adivinar. Ya estamos familiarizados con la función *TimeCurrent*: devolverá la hora actual, la última conocida, del servidor. Y si le restamos el número de segundos de un año, que se coloca en la variable *year*, obtendremos la fecha y la hora de hace exactamente un año. Los resultados entrarán en *array*.

Para calcular las estadísticas sobre cuántas veces se ha iniciado una semana por medio de una barra a una hora determinada, reservamos el array *hours[24]*. El cálculo se realizará en un bucle mediante el *array* resultante, es decir, por barras desde el pasado hasta el presente. En cada iteración, la hora de inicio de la semana que se está visualizando se almacenará en la variable *current*. Cuando finalice el bucle, la zona horaria actual del servidor permanecerá en *current*, ya que la semana actual se procesará en último lugar.

```
// (-v-) cycle through H1 bars
int hours[24] = {};
int current = 0;
for(int i = 0; i < n; ++i)
{
// (-V-) processing of the i-th bar H1
   ...
}

Print("Week opening hours stats:");
ArrayPrint(hours);
```

Dentro del bucle de días utilizaremos la clase *datetime* del archivo de encabezado *MQL5Book/DateTime.mqh* (véase Fecha y hora).

```
      // (-v-) processing the i-th bar H1
      // find the day of the week of the bar
      const ENUM_DAY_OF_WEEK weekday = TimeDayOfWeek(array[i]);
      // skip all days except Sunday and Monday
      if(weekday > MONDAY) continue;
      // analyze the first bar H1 of the next trading week
      // find the hour of the first bar after the weekend
      current = _TimeHour();
      // calculate open hours statistics
      hours[current]++;

      // skip next 2 days
      // (because the statistics for the beginning of this week have already been u
      i += 48;
```

El algoritmo propuesto no es óptimo, pero no requiere comprender los detalles técnicos de la organización de las series temporales, que aún desconocemos.

Algunas semanas no tienen formato (comienzan después de las vacaciones). Si esta situación se produce en la última semana, la variable *current* contendrá un desplazamiento inusual. Esto se puede comprobar por estadística: para la hora resultante habrá un número muy pequeño de «inicios» registrados de la semana. En el script de prueba, en este caso, simplemente se muestra un mensaje en el registro. En la práctica, debe aclarar el inicio estándar de una o dos semanas anteriores.

```
// (-V-) cycle through H1 bars
...
if(hours[current] <= 52 / 4)
{
   // TODO: check for previous weeks
   Print("Extraordinary week detected");
}
```

Si el broker no cambia al horario de verano, las estadísticas tendrán un máximo, que incluirá todas o casi todas las semanas. Si el broker practica un cambio de huso horario, habrá dos máximos en las estadísticas.

```
// find the most frequent time shift
int max = ArrayMaximum(hours);
// then check if there is another regular shift
hours[max] = 0;
int sub = ArrayMaximum(hours);
```

Tenemos que determinar hasta qué punto es significativo el segundo extremo (es decir, diferente de los días festivos aleatorios que podrían desplazar el inicio de la semana). Para ello, evaluamos las estadísticas de un trimestre del año (52 semanas / 4). Si se supera este límite, el broker admite el horario de verano.

```
      int DST = 0;
      if(hours[sub] > 52 / 4)
      {
         // basically, DST is supported
         if(current == max || current == sub)
         {
            if(current == MathMin(max, sub))
               DST =fabs(max -sub); // DST is enabled now
         }
      }
```

Si el desfase en el inicio de la semana actual (en la variable actual) coincide con uno de los dos extremos principales, entonces la semana actual se inició normalmente, y se puede utilizar para sacar una conclusión sobre la zona horaria (esta condición de protección es necesaria porque no tenemos ninguna corrección para las semanas no estándar y en su lugar sólo se emite una advertencia).

Ahora todo está listo para formar la respuesta de nuestra función: la zona horaria del servidor y el signo del horario de verano activado.

```
   current +=2 +DST;// +2 to get offset from UTC
       current %= 24;
   // timezones are always in the range [UTC-12,UTC+12]
       if(current > 12) current = current - 24;
```

Puesto que tenemos dos características para devolver desde una función (*current* y *DST*), y además de eso, podemos decirle al código llamado si el broker utiliza el horario de verano para empezar (incluso si ahora es invierno), tiene sentido declarar una estructura especial *ServerTime* con todos los campos requeridos.

```
  struct ServerTime
  {
   intoffsetGMT;      // timezone in seconds relative to UTC/GMT
   intoffsetDST;      // DST correction in seconds (included in offsetGMT)
   boolsupportDST;    // DST correction detected in quotes in principle
   stringdescription; // result description
  };
```

A continuación, en la función *ServerTimeZone*, podemos rellenar y devolver dicha estructura como resultado del trabajo.

```
      ServerTime st = {};
      st.description = StringFormat("Server time offset: UTC%+d, including DST%+d", cu
      st.offsetGMT = -current * 3600;
      st.offsetDST = -DST * 3600;
      return st;
```

Si por alguna razón la función no puede obtener cotizaciones, devolveremos una estructura vacía.

```
ServerTime ServerTimeZone(const string symbol = NULL)
{
  const int year = 365 * 24 * 60 * 60;
  datetime array[];
  if(PRTF(CopyTime(symbol, PERIOD_H1, TimeCurrent() - year, TimeCurrent(), array)) >
  {
     ...
     return st;
  }
  ServerTime empty = {-INT_MAX, -INT_MAX, false};
  return empty;
}
```

Comprobemos la nueva función en acción, para lo cual añadimos en *OnStart* las siguientes instrucciones:

```
   ...
   ServerTime st = ServerTimeZone();
   Print(st.description);
   Print("ServerGMTOffset: ", st.offsetGMT);
   Print("ServerTimeDaylightSavings: ", st.offsetDST);
}
```

Veamos los posibles resultados:

```
CopyTime(symbol,PERIOD_H1,TimeCurrent()-year,TimeCurrent(),array)=6207 / ok
Got 6207 H1 bars, ~258 days
Week opening hours stats:
52  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
Server time offset: UTC+2, including DST+0
ServerGMTOffset: -7200
ServerTimeDaylightSavings: 0
```

Según las estadísticas recopiladas de las barras H1, la semana para este broker se inicia estrictamente a las 00:00 del lunes. Así, la zona horaria real es igual a UTC+2, y no hay corrección para el horario de verano, es decir, la hora del servidor debe coincidir con EET (UTC+2). Sin embargo, en la práctica, como vimos en la primera parte del registro, la hora en el servidor difiere de GMT en 3 horas.

Aquí podemos suponer que nos encontramos con un servidor que funciona todo el año en verano. En ese caso, la función *ServerTimeZone* no podrá distinguir la corrección de la hora adicional en la «zona horaria»: como resultado, el modo DST será igual a cero, y la hora GMT calculada a partir de las cotizaciones del servidor se desplazará hacia la derecha una hora con respecto a la real. O bien, nuestra suposición inicial de que las cotizaciones empiezan a llegar a las 22:00 del domingo no se corresponde con el modo en que funciona este servidor. Estas cuestiones deben aclararse con el servicio de asistencia del broker.

4.7.3 Hora universal

En MQL5, puede averiguar la hora GMT global (UTC) basándote en la hora local del ordenador y su zona horaria.

datetime TimeGMT()
datetime TimeGMT(MqlDateTime &dt)

La función devuelve la hora GMT en el formato *datetime*, contándola a partir de la hora local del ordenador, y teniendo en cuenta la transición al horario de invierno o verano.

Fórmula de cálculo generalizada:

```
TimeGMT() = TimeLocal() + TimeGMTOffset()
```

Así, la precisión de la representación de la hora universal depende de la correcta configuración del reloj del ordenador local. Lo ideal es que el valor recuperado coincida con el valor conocido por el servidor.

Para las estrategias de trading basadas en noticias económicas externas, lo más fácil es utilizar calendarios en la zona horaria GMT: así se puede hacer un seguimiento de los eventos venideros *TimeGMT*. Para vincular un evento a la hora del servidor en el gráfico, debe corregir el evento para la diferencia entre la zona horaria del servidor y GMT (*TimeTradeServer()* - *TimeGMT()*). Pero recuerde que MQL5 tiene su propio calendariointegrado.

int TimeGMTOffset()

La función devuelve la diferencia actual entre GMT y la hora local del ordenador en segundos, basada en la configuración de zona horaria de Windows, teniendo en cuenta el horario de verano actual. En la mayoría de los casos, la zona horaria se indica como un número entero de horas respecto a GMT, por lo que *TimeGMTOffset* es igual a la zona horaria multiplicada por -3600 (convertido a segundos). Por ejemplo, en invierno la zona horaria puede ser igual a UTC + 2, lo que da un desfase de -7200, y en verano puede ser UTC + 3, lo que da -10800. El signo menos es necesario porque las zonas horarias positivas, al convertir su hora a GMT, requieren restar el número de segundos mencionado, y las negativas requieren sumarlos.

Un script que usa *TimeGMT* y *TimeGMTOffset* se mostró en la sección anterior.

4.7.4 Pausar un programa

Como vimos anteriormente en los ejemplos, los programas a veces necesitan repetir ciertas acciones periódicamente, ya sea en una programación simple o después de que los intentos anteriores hayan fallado. Cuando esto se hace en bucle, se recomienda pausar el programa regularmente para evitar peticiones demasiado frecuentes y una carga innecesaria de la CPU, así como para dar tiempo a que los «actores» externos hagan su trabajo (por ejemplo, si estamos esperando datos de otro programa, cargando el histórico de cotizaciones, etc.).

Para ello, MQL5 proporciona la función *Sleep*. En esta sección se ofrece su descripción formal, y en la siguiente se dará un ejemplo, junto con las funciones para las mediciones de intervalos de tiempo.

void Sleep(int milliseconds)

La función pausa la ejecución del programa MQL durante el número de milisegundos especificado. Tras su expiración, las instrucciones que siguen a la llamada *Sleep* seguirán ejecutándose.

Tiene sentido utilizar la función en primer lugar en scripts y servicios porque este tipo de programas no tienen otra forma de esperar.

En los Asesores Expertos e indicadores se recomienda utilizar temporizadores y el evento *OnTimer*. En este esquema, el programa MQL devuelve el control al terminal y será llamado después de un intervalo especificado.

Además, la función *Sleep* no puede llamarse desde los indicadores, ya que éstos se ejecutan en hilos de la interfaz del terminal, cuya suspensión afectaría a la representación de los gráficos.

Si el usuario interrumpe el programa MQL desde la interfaz del terminal mientras espera a que finalice la llamada *Sleep*, la salida de la función se produce inmediatamente (en 100 ms); es decir, la pausa finaliza antes de lo previsto. Esto activará la bandera de parada *_StopFlag* (también disponible a través de la función *IsStopped*), y el programa debe detener la ejecución lo más rápida y correctamente posible.

4.7.5 Contadores de intervalos de tiempo

Para detectar un intervalo de tiempo de hasta un segundo, basta con tomar la diferencia entre dos valores de *datetime* obtenidos mediante *TimeLocal*. Sin embargo, a veces necesitamos una precisión aún mayor. Para ello, MQL5 permite obtener contadores de milisegundos (*GetTickCount*, *GetTickCount64*) o microsegundos (*GetMicrosecondCount*) del sistema.

uint GetTickCount()
ulong GetTickCount64()

Las funciones devuelven el número de milisegundos que han pasado desde que se cargó el sistema operativo. La precisión de la temporización está limitada por el temporizador estándar del sistema (~10-15 milisegundos). Para una medición más precisa de los intervalos, utilice la función *GetMicrosecondCount*.

En el caso de la función *GetTickCount*, el tipo de retorno *uint* predetermina el periodo de tiempo tras el cual se desbordará el contador: aproximadamente 49,7 días. En otras palabras: la cuenta atrás volverá a empezar desde 0 si el ordenador no se ha apagado durante un tiempo tan prolongado.

En cambio, la función *GetTickCount64* devuelve valores de *ulong*, y este contador no se desbordará en un futuro previsible (584'942'417 años).

ulong GetMicrosecondCount()

La función devuelve el número de microsegundos que han pasado desde el inicio del programa MQL.

En el script *TimeCount.mq5* se resumen ejemplos de uso de las funciones de contador y *Sleep*.

```
void OnStart()
{
   const uint startMs = GetTickCount();
   const ulong startMcs =  GetMicrosecondCount();

   // loop for 5 seconds
   while(PRTF(GetTickCount()) < startMs + 5000)
   {
      PRTF(GetMicrosecondCount());
      Sleep(1000);
   }

   PRTF(GetTickCount() - startMs);
   PRTF(GetMicrosecondCount() - startMcs);
}
```

Este es el aspecto que podría tener la salida de registro del script.

```
GetTickCount()=12912811 / ok
GetMicrosecondCount()=278 / ok
GetTickCount()=12913903 / ok
GetMicrosecondCount()=1089845 / ok
GetTickCount()=12914995 / ok
GetMicrosecondCount()=2182216 / ok
GetTickCount()=12916087 / ok
GetMicrosecondCount()=3273823 / ok
GetTickCount()=12917179 / ok
GetMicrosecondCount()=4365889 / ok
GetTickCount()=12918271 / ok
GetTickCount()-startMs=5460 / ok
GetMicrosecondCount()-startMcs=5458271 / ok
```

4.8 Interacción con el usuario

La conexión del programa con el «mundo exterior» es siempre bidireccional, y los medios para organizarla pueden dividirse condicionalmente en categorías de entrada y salida de datos. En la versión clásica, el usuario proporciona al programa algunos parámetros y recibe de él un resultado. Si el programa se integra con alguna aplicación o servicio externo, la entrada y salida, por regla general, se realizan mediante protocolos especiales de intercambio (a través de archivos, red, memoria compartida, etc.), obviando la interfaz de usuario.

El entorno de ejecución de programas MQL permite organizar la interacción con el usuario de MetaTrader 5 de muchas maneras.

En este capítulo veremos las más sencillos, que permiten mostrar mensajes en un registro o gráfico, mostrar un cuadro de diálogo sencillo y emitir alertas sonoras.

Recordemos que el estándar para introducir datos en un programa MQL son las variables de entrada, pero éstas sólo pueden establecerse en la inicialización del programa. Cambiar las propiedades del programa a través del cuadro de diálogo de configuración significa «reiniciarlo» con nuevos valores (más adelante hablaremos de algunos casos especiales relacionados con un tipo de programa MQL debido a lo cual el *restart* está entre comillas).

Una relación interactiva más flexible implica la capacidad de controlar el comportamiento del programa sin detenerlo. En casos elementales, el cuadro de diálogo MessageBox (por ejemplo), que abordaremos más adelante, sería adecuado para ello, pero para la mayoría de las aplicaciones prácticas esto no es suficiente.

Por lo tanto, en las siguientes partes del libro ampliaremos significativamente la lista de herramientas para implementar la interfaz de usuario y aprenderemos a crear programas interactivos basados en la interfaz objetos, mostrar información gráfica en indicadores o recursos, enviar notificaciones push a los dispositivos móviles de los usuarios, y mucho más.

4.8.1 Mensajes de logging (registro)

El registro es la forma más común de comunicar al usuario información actual sobre el funcionamiento del programa. Dicha información puede consistir en el estado de una realización regular, de una indicación de progreso durante un cálculo largo o de datos de depuración para encontrar y reproducir errores.

Por desgracia, ningún programador es inmune a que haya errores en su código. Por ello, los desarrolladores suelen intentar dejar el denominado «rastro de migas de pan»: registrar las principales etapas de ejecución del programa (al menos, la secuencia de llamadas a funciones).

Ya conocemos dos funciones de registro: *Print* y *PrintFormat*. Las hemos utilizado en los ejemplos de las secciones anteriores. Tuvimos que emplearlas con antelación de un modo simplificado, ya que es casi imposible prescindir de ellas.

Una llamada a una función genera, por regla general, un registro. No obstante, si se encuentra un carácter de nueva línea ('\n') en la cadena de salida, dividirá la información en dos partes.

Tenga en cuenta que todas las llamadas a *Print* y *PrintFormat* se transforman en entradas de registro en la pestaña *Experts* de la ventana *Toolbox*. Aunque la pestaña se llama *Experts*, recopila los resultados de todas las instrucciones de impresión, con independencia del tipo de programa MQL.

Los registros se almacenan en archivos organizados según el principio «un día = un archivo»: tienen los nombres AAAAMMDD.log (A por año, M por mes y D por día). Los archivos se encuentran en *<data directory>/MQL5/Logs* (no los confunda con los registros del sistema del terminal en la carpeta *<data directory>/Logs*).

> Tenga en cuenta que, durante el registro masivo (si las llamadas a la función *Print* generan una gran cantidad de información en poco tiempo), el terminal sólo muestra algunas entradas en la ventana. Esto se hace para optimizar el rendimiento. Además, en cualquier caso, el usuario no puede ver todos los mensajes sobre la marcha. Para ver la versión completa del registro, debe ejecutar el comando *Ver* del menú contextual. Como resultado, se abrirá una ventana con un registro.
>
> También hay que tener en cuenta que la información del registro se almacena en caché cuando se escribe en el disco, es decir, se escribe en los archivos en grandes bloques de modo perezoso, razón por la cual en un momento dado el archivo de registro, por regla general, no contiene las entradas más recientes (aunque sean visibles en una ventana). Para iniciar un vaciado de caché en el disco puede ejecutar el comando *View* o *Open* en el menú contextual del registro.

Cada entrada de registro va precedida de una hora con una precisión de milisegundos, así como del nombre del programa (y sus gráficos) que ha generado o causado este mensaje.

void Print(argument, ...)

La función imprime uno o más valores en el registro de expertos, en una línea (si los datos de salida no contienen el carácter '\n').

Los argumentos pueden ser de cualquier tipo integrado. Se separan por comas. El número de parámetros no puede ser superior a 64. Su número variable se indica mediante una elipsis en el prototipo, pero MQL5 no permite describir funciones propias con una característica similar: sólo algunas funciones integradas de la API tienen un número variable de parámetros (en particular, *StringFormat*, *Print*, *PrintFormat* y *Comment*).

Para estructuras y clases, debe implementar un método de impresión integrado, o mostrar sus campos por separado.

Además, la función no es capaz de manejar arrays. Puede visualizarlos elemento por elemento, o utilizar la función *ArrayPrint*.

Los valores del tipo *double* son mostrados por la función con una precisión de hasta 16 dígitos significativos (juntos en la mantisa y la parte fraccionaria). Un número puede mostrarse en formato tradicional o científico (con un exponente), lo que resulte más compacto. Los valores del tipo *float* se muestran con una precisión de 7 decimales. Para mostrar números reales con una precisión diferente, o para especificar explícitamente el formato, debe utilizar la función *PrintFormat*.

Los valores de tipo *bool* se muestran como cadenas «true» o «false».

Las fechas se muestran con el día y la hora especificados con la máxima precisión (hasta un segundo), en el formato «AAAA.MM.DD hh:mm:ss». Para visualizar la fecha en un formato diferente, utilice la función *TimeToString* (véase el apartado Fecha y hora).

Los valores de enumeración se muestran como números enteros. Para mostrar los nombres de los elementos, utilice la función *EnumToString* (véase la sección Enumeraciones).

Los caracteres de uno y dos bytes también se muestran como números enteros. Para visualizar símbolos como caracteres o letras, utilice las funciones *CharToString* o *ShortToString*; véase la sección Trabajar con símbolos y páginas de códigos).

Los valores del tipo *color* se muestran como una cadena con un triple de números que indican la intensidad de cada componente de color («R, G, B») o como un nombre de color si dicho color está presente en el conjunto de colores.

Para obtener más información sobre la conversión de valores de distintos tipos a cadenas, véase el capítulo Conversión de datos de tipos integrados (especialmente en las secciones De números a cadenas y viceversa, Fecha y hora, Color).

Cuando se trabaja en el comprobador de estrategias en modo de una sola pasada (probar Asesor Experto o indicador), los resultados de la función *Print* se envían al registro del agente de pruebas.

Cuando se trabaja en el probador de estrategias en el modo de optimización, el registro se suprime por razones de rendimiento, por lo que la función *Print* no tiene ningún efecto visible. Sin embargo, se evalúan todas las expresiones dadas como argumentos.

Todos los argumentos, una vez convertidos a una representación de cadena, se concatenan en una cadena común sin caracteres delimitadores. Si es necesario, dichos caracteres deben escribirse explícitamente en la lista de argumentos. Por ejemplo:

```
int x;
bool y;
datetime z;
...
Print(x, ", ", y, ", ", z);
```

Aquí se registran 3 variables, separadas por comas. Si no fuera por los literales intermedios «,», los valores de las variables estarían pegados en la entrada del registro.

Se pueden encontrar muchos casos de aplicación de *Print* ya en las primeras secciones del libro (por ejemplo, Primer programa, Asignación e inicialización, expresiones y arrays, entre otros).

Como una nueva forma de trabajar con *Print* implementaremos una clase simple que le permitirá mostrar una secuencia de valores arbitrarios sin especificar un carácter separador entre cada valor vecino. Utilizamos el enfoque de sobrecarga del operador '<<', similar a lo que se utiliza en los flujos de E/S de C++ (std::cout).

La definición de la clase se colocará en un archivo de encabezado independiente *OutputStream.mqh*. A continuación se muestra una clase de forma simplificada.

```
class OutputStream
{
protected:
   ushort delimiter;
   string line;

   // add the next argument, separated by a separator (if any)
   void appendWithDelimiter(const string v)
   {
      line += v;
      if(delimiter != 0)
      {
         line += ShortToString(delimiter);
      }
   }

public:
   OutputStream(ushort d = 0): delimiter(d) { }

   template<typename T>
   OutputStream *operator<<(const T v)
   {
      appendWithDelimiter((string)v);
      return &this;
   }

   OutputStream *operator<<(OutputStream &self)
   {
      if(&this == &self)
      {
         print(line);// output of the composed string
         line = NULL;
      }
      return &this;
   }
};
```

Su objetivo es acumular en una variable de cadena *line* representaciones de cadena de cualquier argumento pasado utilizando el operador '<<'. Si se especifica un carácter separador en el constructor de la clase, se insertará automáticamente entre los argumentos. Dado que el operador sobrecargado devuelve un puntero a un objeto, podemos pasar en cadena una secuencia de argumentos:

```
OutputStream out(',');
out << x << y << z << out;
```

Como atributo del final de la recogida de datos, y para la salida real del contenido *line* en el registro, se utiliza una sobrecarga del mismo operador para el objeto en sí.

La clase real es algo más complicada. En concreto, permite establecer no sólo el carácter separador, sino también la precisión de la visualización de números reales, así como banderas para seleccionar

campos en valores de fecha y hora. Además, la clase admite la impresión de caracteres, *ushort*, en forma de caracteres (en lugar de códigos enteros), la salida simplificada de arrays (en una cadena separada), colores en formato hexadecimal como un único valor (y no un triple de números separados por comas, ya que la coma se utiliza a menudo como carácter separador, y entonces los componentes de color en el registro parecen 3 variables diferentes).

En el script *OutputStream.mq5* se ofrece una demostración del uso de la clase.

```
void OnStart()
{
   OutputStream os(5, ',');

   bool b = true;
   datetime dt = TimeCurrent();
   color clr = C'127, 128, 129';
   int array[] = {100, 0, -100};
   os << M_PI << "text" << clrBlue << b << array << dt << clr << '@' << os;

   /*
      output example

      3.14159,text,clrBlue,true
      [100,0,-100]
      2021.09.07 17:38,clr7F8081,@
   */
}
```

void PrintFormat(const string format, ...) ≡ void printf(const string format, ...)

La función registra un conjunto de argumentos basados en la cadena de formato especificada. El parámetro *format* no sólo proporciona una plantilla de cadena de salida de texto libre que se muestra «tal cual», sino que también puede contener secuencias de escape que describen cómo deben formatearse argumentos específicos.

El número total de parámetros, incluida la cadena de formato, no puede ser superior a 64. Las restricciones sobre los tipos de parámetros son similares a las de las funciones *print*.

PrintFormat Los principios de funcionamiento y de formato son idénticos a los descritos para la función *StringFormat* (véase la sección Salida universal de datos formateados a una cadena). La única diferencia es que *StringFormat* devuelve la cadena formada al código de llamada, y *print format* la envía al diario. Podemos decir que *PrintFormat* tiene el siguiente equivalente condicional:

```
Print(StringFormat(<list of arguments as is, including format>))
```

Además del nombre completo *PrintFormat* puede utilizar un alias más corto *printf*.

Al igual que la función *Print*, *PrintFormat* tiene algunas características específicas cuando se trabaja en el probador en el modo de optimización: su salida al registro se suprime para mejorar el rendimiento.

Ya hemos conocido en muchas secciones scripts que utilizan *PrintFormat*, como por ejemplo, Transición return, Color, Arrays dinámicos, Gestión de descriptores de archivos, Obtener una lista de variables globales.

4.8.2 Alertas

En esta sección, la señal significará la función *Alert* para emitir avisos al usuario del terminal.

El término «alerta» tiene múltiples significados en MetaTrader 5. Estos son los dos contextos en los que se utiliza:

- Alertas configurables (manualmente) por el usuario en la pestaña *Alerts* del panel *Toolbox*. Con ellas, puede hacer un seguimiento de la activación de condiciones simples para superar los valores establecidos por precio, volumen o tiempo, y emitir notificaciones de varias maneras.
- «Alertas» de programa generadas a partir del código MQL por la función *Alert*. No tienen nada que ver con las anteriores.

void Alert(argument, ...)

La función muestra un mensaje en un cuadro de diálogo no modal, acompañado de una señal sonora estándar (según la selección en el cuadro diálogo *Opciones*, en la pestaña *Events*, en el terminal). Si la ventana está oculta, se mostrará encima de la ventana principal del terminal (se puede entonces cerrar, minimizar o apartar sin dejar de trabajar con la ventana principal). El mensaje también se añade al registro de Expertos, marcado como «Alerta».

> No hay ningún comando en la interfaz de MetaTrader 5 para abrir manualmente la ventana de alerta si previamente se ha cerrado. Para volver a ver la lista de advertencias (en su forma pura, sin necesidad de filtrar el registro), tendrá que generar una nueva señal de alguna manera.

El paso de argumentos, la visualización de la información y los principios generales de la función son exactamente los mismos que los indicados para la función *Print*.

La demostración de la función *Alert* con una captura de pantalla se ha mostrado en el ejemplo de los saludos introductorios del primer capítulo, en la sección Salida de datos.

Utilice *Alert* en lugar de *Print* en los casos en que sea necesario llamar la atención del usuario sobre la información mostrada. No obstante, no se debe abusar, ya que la aparición frecuente de la ventana puede entorpecer el trabajo del usuario, obligarle a ignorar mensajes o detener el programa MQL. Proporcione un algoritmo en su programa para limitar la frecuencia de generación de posibles mensajes.

4.8.3 Visualización de mensajes en la ventana de gráficos

Como hemos visto en las secciones anteriores, MQL5 permite enviar mensajes al registro o a la ventana de alertas. El primer método es principalmente para información técnica y no puede garantizar que el usuario vea el mensaje (porque la ventana de registro puede estar oculta). Al mismo tiempo, el segundo método puede parecer demasiado intrusivo si se utiliza para mostrar estados de programas que cambian con frecuencia. Una opción intermedia ofrece la función *Comment*.

void Comment(argument, ...)

La función muestra un mensaje compuesto por todos los argumentos pasados en la esquina superior izquierda del gráfico. El mensaje permanece allí hasta que este u otro programa lo elimina o lo sustituye por otro.

La ventana sólo puede contener un comentario: en cada llamada de *Comment*, el contenido antiguo (si existe) se sustituye por el nuevo.

Para borrar un comentario, basta con llamar a la función con una cadena vacía: *Comment("")*.

El número de parámetros no debe ser superior a 64. Sólo se admiten argumentos de tipo integrado. Los conceptos de formación de la cadena resultante a partir de los valores pasados son similares a los descritos para la función *Print*.

La longitud total del mensaje mostrado está limitada a 2045 caracteres. Si se supera el límite, se cortará el final de la línea.

> El contenido actual de un comentario es una de las propiedades de cadena del gráfico, que puede encontrarse llamando a la función *ChartGetString(NULL, CHART_COMMENT)*. Hablaremos de ésta y otras propiedades de los gráficos (no sólo de los de cadenas) en un capítulo independiente.

Al igual que en las funciones *Print*, *PrintFormat* y *Alert*, los argumentos de cadena pueden contener un carácter de nueva línea ('\n' o '\r\n'), que hará que el mensaje se divida en el número apropiado de cadenas. Para *Comment* esta es la única manera de mostrar un mensaje multilínea. Si puede llamarlas varias veces para obtener el mismo efecto utilizando las funciones print y signal, con *Comment* no podrá hacerlo, ya que cada llamada sustituirá la cadena antigua por la nueva.

Un ejemplo de trabajo de la función *Comment* se muestra en la imagen de la ventana con el script de bienvenida del primer capítulo, en la sección Salida de datos.

Además, desarrollaremos una clase y funciones simplificadas para mostrar comentarios multilínea basados en un búfer circular de un tamaño determinado. El script de prueba (*OutputComment.mq5*) y el archivo de encabezado con el código de la clase (*Comments.mqh*) están incluidos en el libro.

```
class Comments
{
 const int capacity; // maximum number of strings
 const bool reverse; // display order (new ones on top if true)
 string lines[];     // text buffer
 int cursor;         // where to put the next string
 int size;           // actual number of strings saved

public:
   Comments(const int limit = N_LINES, const bool r = false):
      capacity(limit), reverse(r), cursor(0), size(0)
   {
      ArrayResize(lines, capacity);
   }

   void add(const string line);
   void clear();
};
```

El trabajo principal se realiza mediante el método *add*.

```
void Comments::add(const string line)
{
   ...
   // if the passed text contains multiple strings,
   // split it into elements by newline character
   string inputs[];
   const int n = StringSplit(line, '\n', inputs);

   // add all new elements to the ring buffer
   // overwriting the oldest entries at the cursor
   // cursor increases by capacity module (reset to 0 on overflow)
   for(int i = 0; i < n; ++i)
   {
      lines[cursor] = inputs[reverse ? n - i - 1 : i];
      cursor = (cursor + 1) % capacity;
      if(size < capacity) size++;
   }
   // concatenate all text entries in forward or reverse order
   // gluing with newline characters
   string result = "";
   for(int i = 0, k = size == capacity ? cursor % capacity : 0;
      i < size; ++i, k = ++k % capacity)
   {
      if(reverse)
      {
         result = lines[k] + "\n" + result;
      }
      else
      {
         result += lines[k] + "\n";
      }
   }

   // output the result
   Comment(result);
}
```

Si es necesario, el comentario y el búfer de texto pueden borrarse mediante el método *clear*, o llamando a *add(NULL)*.

```
void Comments::clear()
{
   Comment("");
   cursor = 0;
   size = 0;
}
```

Dada una clase de este tipo, se puede definir un objeto con la capacidad de búfer y la dirección de salida requeridas y, a continuación, utilizar sus métodos.

```
Comments c(30/*capacity*/, true/*order*/);

void function()
{
   ...
   c.add("123");
}
```

No obstante, para simplificar la generación de comentarios al estilo funcional habitual, por analogía con la función *Comment*, se implementan un par de funciones de ayuda.

```
void MultiComment(const string line = NULL)
{
   static Comments com(N_LINES, true);
   com.add(line);
}

void ChronoComment(const string line = NULL)
{
   static Comments com(N_LINES, false);
   com.add(line);
}
```

Sólo se diferencian en la dirección de la salida del búfer. *MultiComment* muestra las filas en orden cronológico inverso, es decir, las más recientes en la parte superior, como en un tablón de anuncios. Esta función se recomienda para una visualización episódica de información de duración indefinida con preservación de la historia. *ChronoComment* muestra las filas en orden de avance, es decir, las nuevas se añaden al final. Esta función se recomienda para la salida por lotes de mensajes multilínea.

Por defecto, el número de líneas del búfer es N_LINES (10). Si define esta macro con un valor diferente antes de incluir el archivo de encabezado, se redimensionará.

El script de prueba contiene un bucle en el que se generan mensajes periódicamente.

```
void OnStart()
{
   for(int i = 0; i < 50 && !IsStopped(); ++i)
   {
      if((i + 1) % 10 == 0) MultiComment();
      MultiComment("Line " + (string)i + ((i % 3 == 0) ? "\n  (details)" : ""));
      Sleep(1000);
   }
   MultiComment();
}
```

Cada diez iteraciones se borra el comentario. En cada tercera iteración se crea un mensaje a partir de dos líneas (para el resto, a partir de una). Un retardo de 1 segundo permite ver la dinámica en acción.

He aquí un ejemplo de la ventana mientras se ejecuta el script (en modo «nuevos mensajes en la parte superior»).

Comentarios de varias líneas en el gráfico

La visualización de información multilínea en un comentario tiene unas posibilidades bastante limitadas. Si necesita organizar la salida de datos por columnas, resaltar con colores o fuentes diferentes, reaccionar a los clics del ratón o ubicaciones arbitrarias en el gráfico, debe utilizar objetos gráficos.

4.8.4 Cuadro de diálogo de mensajes

La API de MQL5 proporciona la función *MessageBox* para pedir interactivamente al usuario que confirme acciones o seleccione una opción para manejar una situación particular.

int MessageBox(const string message, const string caption = NULL, int flags = 0)

La función abre un cuadro de diálogo sin modelo con el mensaje (*message*), el encabezado (*caption*) y la configuración (*flags*) dados. La ventana permanece visible encima de la ventana principal del terminal hasta que el usuario la cierra pulsando uno de los botones disponibles (véase más adelante).

El mensaje también se muestra en el registro de expertos con la marca «Mensaje».

Si el parámetro *caption* es NULL, se utiliza el nombre del programa MQL como título.

El parámetro *flags* debe contener una combinación de banderas de bits combinadas con una operación OR ('|'). El conjunto general de banderas admitidas se divide en 3 grupos que definen:

- un conjunto de botones en el cuadro de diálogo
- imagen del icono en el cuadro de diálogo
- selección del botón activo por defecto

En la siguiente tabla se enumeran las constantes y los valores de las banderas para definir los botones del cuadro de diálogo.

Constante	Valor	Descripción
MB_OK	0x0000	1 botón OK (por defecto)
MB_OKCANCEL	0x0001	2 botones: OK y Cancelar
MB_ABORTRETRYIGNORE	0x0002	3 botones: Abortar, Reintentar, Ignorar
MB_YESNOCANCEL	0x0003	3 botones: Sí, No, Cancelar
MB_YESNO	0x0004	2 botones: Sí y No
MB_RETRYCANCEL	0x0005	2 botones: Reintentar y Cancelar
MB_CANCELTRYCONTINUE	0x0006	3 botones: Cancelar, Intentar de nuevo, Continuar

En la siguiente tabla se enumeran las imágenes disponibles (que aparecen a la izquierda del mensaje).

Constante	Valor	Descripción
MB_ICONSTOP MB_ICONERROR MB_ICONHAND	0x0010	Señal de STOP 
MB_ICONQUESTION	0x0020	Signo de interrogación 
MB_ICONEXCLAMATION MB_ICONWARNING	0x0030	Signo de exclamación 
MB_ICONINFORMATION MB_ICONASTERISK	0x0040	Señal de información 

Todos los iconos dependen de la versión del sistema operativo. Los ejemplos mostrados pueden diferir en su ordenador.

Los siguientes valores están reservados para seleccionar el botón activo.

Constante	Valor	Descripción
MB_DEFBUTTON1	0x0000	El primer botón (por defecto) si no se selecciona ninguna de las otras constantes.
MB_DEFBUTTON2	0x0100	El segundo botón
MB_DEFBUTTON3	0x0200	El tercer botón
MB_DEFBUTTON4	0x0300	El cuarto botón

Puede surgir la pregunta de cuál es este cuarto botón si las constantes anteriores no permiten establecer más de tres. El hecho es que entre las banderas también hay MB_HELP (0x00004000), que ordena mostrar el botón Ayuda en el cuadro de diálogo. Entonces puede convertirse en el cuarto consecutivo si hay tres botones principales. No obstante, al hacer clic en el botón Ayuda no

se cierra el cuadro de diálogo, a diferencia de otros botones. Según el estándar de Windows, se puede asociar un archivo de ayuda al programa, que debe abrirse con la ayuda necesaria al pulsar el botón Ayuda. Sin embargo, los programas MQL no admiten actualmente esta tecnología.

La función devuelve uno de los valores predefinidos dependiendo de cómo se cerró el cuadro de diálogo (qué botón se pulsó).

Constante	Valor	Descripción
IDOK	1	Botón OK
IDCANCEL	2	Botón Cancelar
IDABORT	3	Botón Abortar
IDRETRY	4	Botón Reintentar
IDIGNORE	5	Botón Ignorar
IDYES	6	Botón Sí
IDNO	7	Botón No
IDTRYAGAIN	10	Botón Intentar de nuevo
IDCONTINUE	11	Botón Continuar

Si el buzón de mensajes tiene un botón Cancelar, la función devuelve IDCANCEL cuando se pulsa la tecla ESC (además del botón Cancelar). Si el cuadro de mensaje no tiene botón Cancelar, pulsar ESC no tiene ningún efecto.

Llamar a *MessageBox* suspende la ejecución del programa MQL actual hasta que el usuario cierre el cuadro de diálogo. Por esta razón, el uso de *MessageBox* está prohibido en indicadores, ya que los indicadores se ejecutan en el hilo de la interfaz del terminal, y esperar la respuesta del usuario ralentizaría la actualización de los gráficos.

Además, la función no puede utilizarse en servicios, ya que no tienen conexión con la interfaz de usuario, mientras que otros tipos de programas MQL se ejecutan en el contexto del gráfico.

Cuando se trabaja en el probador de estrategias, la función *MessageBox* no tiene ningún efecto y devuelve 0.

Después de obtener el resultado de la llamada a la función, puede procesarlo de la forma que desee, por ejemplo:

```
   int result = MessageBox("Continue?", NULL, MB_YESNOCANCEL);
 // use 'switch' or 'if' as needed
   switch(result)
   {
   case IDYES:
     // ...
     break;
   case IDNO:
     // ...
     break;
   case IDCANCEL:
     // ...
     break;
   }
```

La función *MessageBox* puede probarse utilizando el script *OutputMessage.mq5*, en el que el usuario puede seleccionar los parámetros del cuadro de diálogo utilizando variables de entrada y verlo en acción.

Los grupos de ajustes para botones, iconos y el botón seleccionado por defecto, así como los códigos de retorno, se describen en enumeraciones especiales: ENUM_MB_BUTTONS, ENUM_MB_ICONS, ENUM_MB_DEFAULT, ENUM_MB_RESULT. Esto proporciona una entrada visual a través de listas desplegables y simplifica su conversión a cadenas mediante *EnumToString*.

Por ejemplo, así se definen las dos primeras enumeraciones:

```
enum ENUM_MB_BUTTONS
{
   _OK = MB_OK,                                    // Ok
   _OK_CANCEL = MB_OKCANCEL,                       // Ok | Cancel
   _ABORT_RETRY_IGNORE = MB_ABORTRETRYIGNORE,      // Abort | Retry | Ignore
   _YES_NO_CANCEL = MB_YESNOCANCEL,                // Yes | No | Cancel
   _YES_NO = MB_YESNO,                             // Yes | No
   _RETRY_CANCEL = MB_RETRYCANCEL,                 // Retry | Cancel
   _CANCEL_TRYAGAIN_CONTINUE = MB_CANCELTRYCONTINUE, // Cancel | Try Again | Continue
};

enum ENUM_MB_ICONS
{
   _ICON_NONE = 0,                                 // None
   _ICON_QUESTION = MB_ICONQUESTION,               // Question
   _ICON_INFORMATION_ASTERISK = MB_ICONINFORMATION, // Information (Asterisk)
   _ICON_WARNING_EXCLAMATION = MB_ICONWARNING,     // Warning (Exclamation)
   _ICON_ERROR_STOP_HAND = MB_ICONERROR,           // Error (Stop, Hand)
};
```

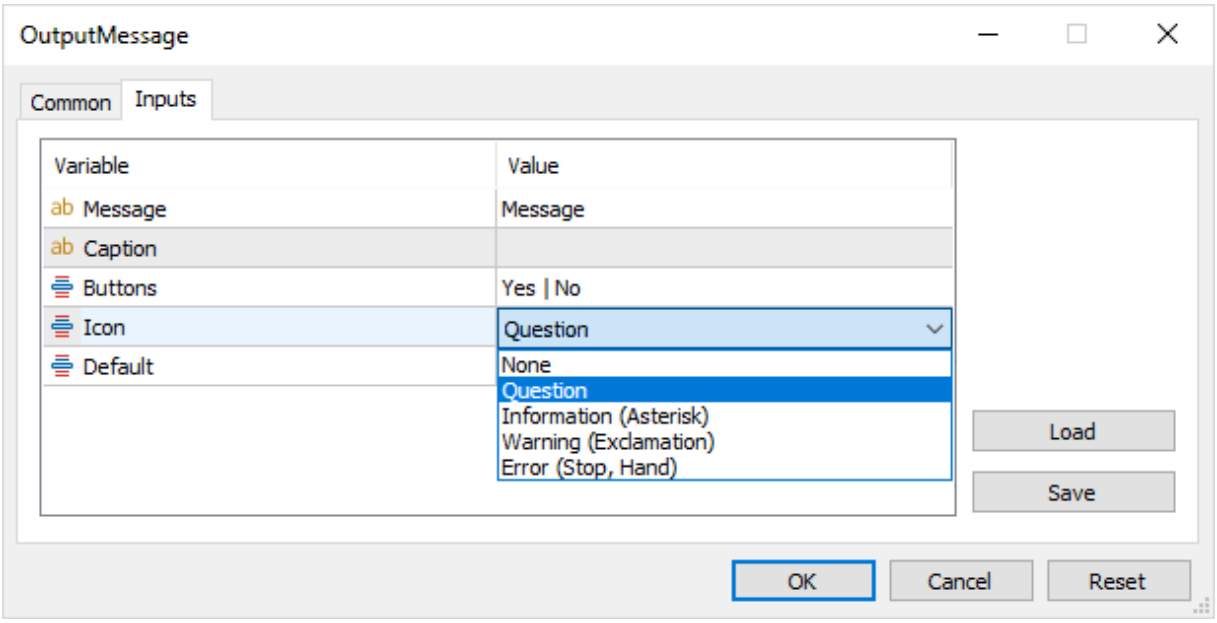
El resto se puede encontrar en el código fuente.

A continuación se utilizan como tipos de variables de entrada (con comentarios de elementos que proporcionan una presentación más fácil de usar en la interfaz de usuario).

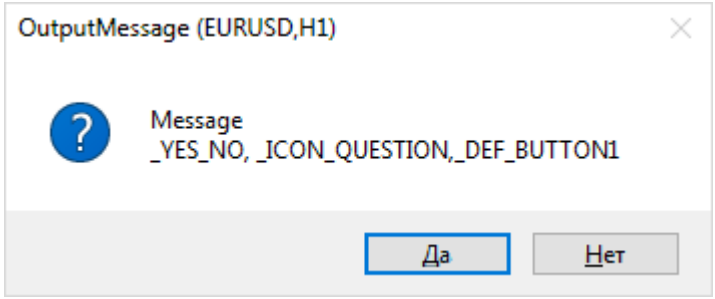
```
input string Message = "Message";
input string Caption = "";
input ENUM_MB_BUTTONS Buttons = _OK;
input ENUM_MB_ICONS Icon = _ICON_NONE;
input ENUM_MB_DEFAULT Default = _DEF_BUTTON1;

void OnStart()
{
   const string text = Message + "\n"
      + EnumToString(Buttons) + ", "
      + EnumToString(Icon) + ","
      + EnumToString(Default);
   ENUM_MB_RESULT result = (ENUM_MB_RESULT)
      MessageBox(text, StringLen(Caption) ? Caption : NULL, Buttons | Icon | Default)
   Print(EnumToString(result));
}
```

El script muestra el mensaje especificado en la ventana, junto con la configuración del cuadro de diálogo especificada. El resultado del cuadro de diálogo se muestra en el registro.

En las siguientes imágenes se muestra una captura de pantalla de la selección de opciones y el cuadro de diálogo resultante.



Cuadro de diálogo de propiedades de la ventana



Cuadro de diálogo de mensaje recibido

4.8.5 Alertas sonoras

Para trabajar con sonido, la API de MQL5 proporciona una función: *PlaySound*.

bool PlaySound(const string soundfile)

La función reproduce el archivo de sonido especificado en el formato *wav*.

Si el nombre del archivo se especifica sin una ruta (por ejemplo, «Ring.wav»), debe estar ubicado en la carpeta *Sounds* dentro del directorio de instalación del terminal. Si es necesario, puede organizar subcarpetas dentro de la carpeta *Sounds*. En tales casos, el nombre del archivo en el parámetro *soundfile* debe ir precedido de una ruta relativa. Por ejemplo, «Ejemplo/Ring.wav» se refiere a las carpetas y al archivo *Sounds/Example/Ring.wav* dentro del directorio de instalación del terminal.

Además, puede utilizar archivos de sonido ubicados en cualquier otra subcarpeta MQL5 del directorio de datos del terminal. Una ruta de este tipo debe ir precedida de una barra oblicua (una sola barra '/' o una doble barra invertida '\\'), que es el carácter delimitador que se utiliza entre niveles de carpetas adyacentes en el sistema de archivos. Por ejemplo, si el archivo de sonido *Demo.wav* está en *terminal_data_directory/MQL5/Files*, entonces en la llamada *PlaySound* escribiremos la ruta «/Archivos/Demo.wav».

Llamar a la función con un parámetro NULL detiene la reproducción del sonido. Llamar a una función con un nuevo archivo mientras el antiguo aún se está reproduciendo hará que el antiguo se interrumpa y el nuevo comience a reproducirse.

Además de los archivos ubicados en el sistema de archivos, una ruta a los recursos (bloques de datos incrustados en el programa MQL) se puede pasar a la función. En concreto, un desarrollador puede crear un recurso de sonido a partir de un archivo que esté disponible localmente en tiempo de compilación dentro de una «sandbox». Todos los recursos se encuentran dentro del archivo ex5, lo que garantiza que el usuario disponga de ellos y simplifica la distribución del programa como un único módulo.

Un artículo detallado sobre todas las formas de utilizar los recursos, incluidos no sólo el sonido, sino también las imágenes, los datos binarios y de texto arbitrarios y los programas dependientes (indicadores), se presenta en la sección correspondiente en la séptima parte del libro.

La función *PlaySound* devuelve *true* si se encuentra el archivo, o *false* en caso contrario. Tenga en cuenta que, aunque el archivo no sea de audio y no pueda reproducirse, la función devolverá *true*.

La reproducción del sonido se realiza de forma asíncrona, en paralelo con la ejecución de las siguientes instrucciones del programa. En otras palabras: la función devuelve el control al código de llamada inmediatamente después de la llamada, sin esperar a que se complete el efecto de audio.

En el probador de estrategias, la función *PlaySound* no se ejecuta.

El script *OutputSound.mq5* permite probar el funcionamiento de la función.

```
void OnStart()
{
   PRTF(PlaySound("new.txt"));
   PRTF(PlaySound("abracadabra.wav"));
   const uint start = GetTickCount();
   PRTF(PlaySound("request.wav"));
   PRTF(GetTickCount() - start);
}
```

El programa intenta reproducir varios archivos. El archivo «new.txt» existe (creado específicamente para las pruebas), el archivo «abracadabra.wav» no existe, y el archivo «request.wav» está incluido en la distribución estándar de MetaTrader 5. El tiempo de la última llamada a una función se mide utilizando un par de llamadas a *GetTickCount*.

Como resultado de la ejecución del script, obtenemos las siguientes entradas de registro:

```
PlaySound(new.txt)=true / ok
PlaySound(abracadabra.wav)=false / FILE_NOT_EXIST(5019)
PlaySound(request.wav)=true / ok
GetTickCount()-start=0 / ok
```

Se encontró el archivo «nuevo.txt» y, por tanto, la función devolvió *true*, aunque no produjo ningún sonido. Una llamada para un segundo archivo inexistente devolvió *false*, y el código de error en *_LastError* es 5019 (FILE_NOT_EXIST). Por último, la reproducción del último archivo (suponiendo que exista) debería tener éxito en todos los sentidos: la función devolverá *true*, y el terminal reproducirá el audio. El tiempo de procesamiento de la llamada es prácticamente nulo (la duración del sonido no importa).

4.9 Entorno de ejecución del programa MQL

Como sabemos, los textos fuente de un programa MQL después de la compilación en un código binario ejecutable con el formato *ex5* están listos para funcionar en el terminal o en agentes de prueba. Así, un terminal o un probador proporcionan un entorno común en el que «viven» los programas MQL.

Recuerde que el comprobador integrado sólo admite dos tipos de programas MQL: Asesores expertos e indicadores. Hablaremos en detalle sobre los tipos de programas MQL y sus características en la quinta parte del libro. Mientras tanto, en este capítulo nos centraremos en aquellas funciones de la API de MQL5 que son comunes a todos los tipos, y que le permiten analizar el entorno de ejecución y, hasta cierto punto, controlarlo.

La mayoría de las propiedades de entorno son de sólo lectura a través de las funciones *TerminalInfoInteger*, *TerminalInfoDouble*, *TerminalInfoString*, *MQLInfoInteger* y *MQLInfoString*. A partir de los nombres se puede entender que cada función devuelve valores de un tipo determinado. Una arquitectura de este tipo da lugar a que el significado aplicado de las propiedades combinadas en una función pueda ser muy diferente. Otra agrupación puede ser proporcionada por la implementación de su propia capa de objetos en MQL5 (se ofrecerá un ejemplo un poco más adelante, en la sección sobre el uso de las propiedades para la vinculación con el entorno del programa).

El conjunto especificado de funciones tiene una división lógica explícita en propiedades generales del terminal (con el prefijo «Terminal») y propiedades de un programa MQL independiente (con el prefijo «MQL»). Sin embargo, en muchos casos, es necesario analizar conjuntamente las características similares tanto del terminal como del programa. Por ejemplo, los permisos para utilizar una DLL o

realizar operaciones de trading se conceden tanto al terminal en su conjunto como a un programa específico. Por eso tiene sentido considerar las funciones dentro de un complejo, como un todo.

Sólo algunas de las propiedades de entorno asociadas a los códigos de error son «writable» (se puede escribir en ellas); en concreto, el restablecimiento de un error previo (*ResetLastError*) y el establecimiento de un error de usuario (*SetUserError*).

También veremos en este capítulo las funciones para cerrar el terminal dentro de un programa (*TerminalClose*, *SetReturnError*) y pausar el programa en el depurador (*Debug Break*).

4.9.1 Obtener una lista general de las propiedades del terminal y del programa

Las funciones integradas disponibles para obtener propiedades de entorno utilizan un enfoque genérico: las propiedades de cada tipo específico se combinan en una función independiente con un único argumento que especifica la propiedad solicitada. Hay enumeraciones definidas para identificar propiedades: cada elemento describe una propiedad.

Como veremos más adelante, este enfoque se utiliza a menudo en la API de MQL5 y en otras áreas, incluidas las áreas de aplicación. En concreto, se utilizan conjuntos de funciones similares para obtener las propiedades de cuentas de trading e instrumentos financieros.

Las propiedades de tres tipos simples, *int*, *double* y *string*, son suficientes para describir el entorno. No obstante, no sólo se presentan propiedades de enteros utilizando valores del tipo *int*, sino también banderas lógicas (en particular, permisos/prohibiciones, presencia de una conexión de red, etc.), así como otras enumeraciones integradas (por ejemplo, tipos de programas MQL y tipos de licencias).

Dada la división condicional en propiedades de terminales y propiedades de un programa MQL concreto, existen las siguientes funciones que describen el entorno.

```
int MQLInfoInteger(ENUM_MQL_INFO_INTEGER p)
int TerminalInfoInteger(ENUM_TERMINAL_INFO_INTEGER p)
double TerminalInfoDouble(ENUM_TERMINAL_INFO_DOUBLE p)
string MQLInfoString(ENUM_MQL_INFO_STRING p)
string TerminalInfoString(ENUM_TERMINAL_INFO_STRING p)
```

Estos prototipos asignan tipos de valores a tipos de enum. Por ejemplo, las propiedades del terminal del tipo *int* se resumen en ENUM_TERMINAL_INFO_INTEGER, y sus propiedades del tipo *double* se enumeran en ENUM_TERMINAL_INFO_DOUBLE, etc. La lista de enums disponibles y sus elementos puede consultarse en la documentación, en las secciones sobre Propiedades de los terminales y Programas MQL.

En las siguientes secciones veremos todas las propiedades, agrupadas en función de su finalidad. Pero aquí nos planteamos el problema de obtener una lista general de todas las propiedades existentes y sus valores. Esto suele ser necesario para identificar «cuellos de botella» o características del funcionamiento de los programas MQL en instancias específicas del terminal. Una situación bastante común es cuando un programa MQL funciona en un ordenador, pero no funciona en absoluto o el funcionamiento presenta algunos problemas en otro.

La lista de propiedades se actualiza constantemente a medida que se desarrolla la plataforma, por lo que es aconsejable realizar su solicitud no sobre la base de una lista programada en el código fuente, sino de forma automática.

En la sección Enumeraciones hemos creado una función de plantilla *EnumToArray* para obtener una lista completa de elementos de enumeración (archivo *EnumToArray.mqh*). También en esa sección introdujimos el script *ConversionEnum.mq5*, que utiliza el archivo de encabezado especificado. En el script se implementó una función de ayuda *process*, que recibía un array con códigos de elementos de enumeración y los mostraba en el registro. Tomaremos estos avances como punto de partida para seguir mejorando.

Tenemos que modificar la función *process* de tal forma que no sólo obtengamos una lista de los elementos de una enumeración determinada, sino que también consultemos las propiedades correspondientes utilizando una de las funciones de propiedades integradas.

Vamos a dar un nombre a la nueva versión del script, *Environment.mq5*.

Dado que las propiedades del entorno están dispersas en varias funciones diferentes (en este caso, cinco), es necesario descubrir cómo pasar a la nueva versión de la función *process* un puntero a la función integrada necesaria (véase la sección Punteros de función (typedef)). Sin embargo, MQL5 no permite asignar la dirección de una función integrada a un puntero de función. Esto sólo puede hacerse con una función de aplicación implementada en MQL5. Por lo tanto, crearemos funciones de envoltorio. Por ejemplo:

```
int _MQLInfoInteger(const ENUM_MQL_INFO_INTEGER p)
{
   return MQLInfoInteger(p);
}
// example of pointer type description
typedef int (*IntFuncPtr)(const ENUM_MQL_INFO_INTEGER property);
// initialization of pointer variables
IntFuncPtr ptr1 = _MQLInfoInteger;  // ok
IntFuncPtr ptr2 = MQLInfoInteger;   // compilation error
```

Arriba se muestra un «double» para *MQLInfoInteger* (obviamente, debería tener un nombre diferente, pero preferiblemente similar). Otras funciones se «empaquetan» de forma similar. Habrá cinco en total.

Si en la versión antigua de *process* sólo había un parámetro de plantilla especificando una enumeración; en la nueva necesitamos pasar también el tipo del valor de retorno (ya que MQL5 no «entiende» las palabras en el nombre de las enumeraciones): aunque la terminación «INTEGER» esté presente en el nombre ENUM_MQL_INFO_INTEGER, el compilador no es capaz de asociarlo con el tipo *int*).

Sin embargo, además de vincular los tipos del valor de retorno y de la enumeración, necesitamos pasar de algún modo a la función *process* un puntero a la función de envoltorio apropiada (una de las cinco que definimos anteriormente). Al fin y al cabo, el propio compilador no puede determinar mediante un argumento del tipo, por ejemplo, ENUM_MQL_INFO_INTEGER, que es necesario llamar a *MQLInfoInteger*.

Para resolver este problema se ha creado una estructura de plantilla especial que combina los tres factores.

```
template<typename E, typename R>
struct Binding
{
public:
   typedef R (*FuncPtr)(const E property);
   const FuncPtr f;
   Binding(FuncPtr p): f(p) { }
};
```

Los dos parámetros de la plantilla permiten especificar el tipo del puntero de la función (*FuncPtr*) con la combinación deseada de parámetros de resultado y de entrada. La instancia de la estructura tiene el campo *f* para un puntero a ese tipo recién definido.

Ahora, una nueva versión de la función *process* puede describirse del siguiente modo:

```
template<typename E, typename R>
void process(Binding<E, R> &b)
{
   E e = (E)0; // turn off the warning about the lack of initialization
   int array[];
   // get a list of enum elements into an array
   int n = EnumToArray(e, array, 0, USHORT_MAX);
   Print(typename(E), " Count=", n);
   ResetLastError();
   // display the name and value for each element,
   // obtained by calling a pointer in the Binding structure
   for(int i = 0; i < n; ++i)
   {
      e = (E)array[i];
      R r = b.f(e); // call the function, then parse _LastError
      const int snapshot = _LastError;
      PrintFormat("% 3d %s=%s", i, EnumToString(e), (string)r +
         (snapshot != 0 ? E2S(snapshot) + " (" + (string)snapshot + ")" : ""));
      ResetLastError();
   }
}
```

El argumento de entrada es la estructura *Binding*. Contiene un puntero a una función específica para obtener propiedades (este campo será rellenado por el código de llamada).

Esta versión del algoritmo registra el número de secuencia, el identificador de la propiedad y su valor. De nuevo, tenga en cuenta que el primer número de cada entrada contendrá el ordinal del elemento en la enumeración, no el valor (se pueden asignar valores a elementos con huecos). Opcionalmente puede añadir una salida de una variable *e* «en su forma pura» dentro de las instrucciones *print format*.

Además, puede modificar el proceso para que recoja en un array (u otro contenedor, como un mapa) los valores de las propiedades resultantes y los devuelva «fuera».

Sería un error en potencia hacer referencia al puntero de función directamente en la instrucción *print format* junto con el análisis del código de error *_LastError*. La cuestión es que la secuencia de evaluación de los argumentos de las funciones (véase la sección Parámetros y argumentos) y los operandos de una expresión (véase la sección Conceptos básicos) no está definido en este caso. Por lo tanto, cuando se llama a un puntero en la misma línea en la que se lee *_LastError*, el compilador puede

decidir ejecutar el segundo antes que el primero. Como resultado, veremos un código de error irrelevante (por ejemplo, de una llamada a una función anterior).

Pero eso no es todo. La variable integrada *_LastError* puede cambiar su valor casi en cualquier punto de la evaluación de una expresión si falla alguna operación. En concreto, la función *EnumToString* puede potencialmente generar un código de error si se pasa como argumento un valor que no está en la enumeración. En este fragmento, somos inmunes a este problema porque nuestra función *EnumToArray* devuelve un array sólo con elementos de enumeración comprobados (válidos). No obstante, en casos generales, en cualquier instrucción «compuesta», puede haber muchos lugares en los que *_LastError* se cambie. A este respecto, es conveniente fijar el código de error inmediatamente después de la acción que nos interesa (aquí se trata de una llamada a una función mediante un puntero), guardándolo en una variable intermedia *snapshot*.

Pero volvamos a la cuestión principal. Para terminar podemos organizar una llamada a la nueva función *process* a fin de obtener diversas propiedades del entorno de software.

```
void OnStart()
{
   process(Binding<ENUM_MQL_INFO_INTEGER, int>(_MQLInfoInteger));
   process(Binding<ENUM_TERMINAL_INFO_INTEGER, int>(_TerminalInfoInteger));
   process(Binding<ENUM_TERMINAL_INFO_DOUBLE, double>(_TerminalInfoDouble));
   process(Binding<ENUM_MQL_INFO_STRING, string>(_MQLInfoString));
   process(Binding<ENUM_TERMINAL_INFO_STRING, string>(_TerminalInfoString));
}
```

A continuación se muestra un fragmento de las entradas de registro generadas.

```
ENUM_MQL_INFO_INTEGER Count=15
  0 MQL_PROGRAM_TYPE=1
  1 MQL_DLLS_ALLOWED=0
  2 MQL_TRADE_ALLOWED=0
  3 MQL_DEBUG=1
...
  7 MQL_LICENSE_TYPE=0
...
ENUM_TERMINAL_INFO_INTEGER Count=50
  0 TERMINAL_BUILD=2988
  1 TERMINAL_CONNECTED=1
  2 TERMINAL_DLLS_ALLOWED=0
  3 TERMINAL_TRADE_ALLOWED=0
...
  6 TERMINAL_MAXBARS=100000
  7 TERMINAL_CODEPAGE=1251
  8 TERMINAL_MEMORY_PHYSICAL=4095
  9 TERMINAL_MEMORY_TOTAL=8190
 10 TERMINAL_MEMORY_AVAILABLE=7813
 11 TERMINAL_MEMORY_USED=377
 12 TERMINAL_X64=1
...
ENUM_TERMINAL_INFO_DOUBLE Count=2
  0 TERMINAL_COMMUNITY_BALANCE=0.0 (MQL5_WRONG_PROPERTY,4512)
  1 TERMINAL_RETRANSMISSION=0.0
ENUM_MQL_INFO_STRING Count=2
  0 MQL_PROGRAM_NAME=Environment
  1 MQL_PROGRAM_PATH=C:\Program Files\MT5East\MQL5\Scripts\MQL5Book\p4\Environment.ex
ENUM_TERMINAL_INFO_STRING Count=6
  0 TERMINAL_COMPANY=MetaQuotes Software Corp.
  1 TERMINAL_NAME=MetaTrader 5
  2 TERMINAL_PATH=C:\Program Files\MT5East
  3 TERMINAL_DATA_PATH=C:\Program Files\MT5East
  4 TERMINAL_COMMONDATA_PATH=C:\Users\User\AppData\Roaming\MetaQuotes\Terminal\Common
  5 TERMINAL_LANGUAGE=Russian
```

Estas y otras propiedades se describirán en las secciones siguientes.

Cabe señalar que algunas propiedades se heredan de etapas anteriores de desarrollo de la plataforma y se dejan sólo por compatibilidad. En concreto, la propiedad TERMINAL_X64 en *TerminalInfoInteger* devuelve una indicación de si el terminal es de 64 bits. En la actualidad, el desarrollo de versiones de 32 bits se ha interrumpido, por lo que esta propiedad es siempre igual a 1 (*true*).

4.9.2 Número de versión del terminal

Dado que el terminal se mejora constantemente y aparecen nuevas funciones en sus nuevas versiones, un programa MQL puede necesitar analizar la versión actual para aplicar diferentes opciones de algoritmos. Además, ningún programa es inmune a los errores, incluido el propio terminal. Por lo tanto, si se producen problemas, debe proporcionar una salida de diagnóstico que incluya la versión actual del terminal; ello puede ayudar a reproducir y corregir errores.

Puede obtener el número de versión del terminal utilizando la propiedad TERMINAL_BUILD en ENUM_TERMINAL_INFO_INTEGER.

```
if(TerminalInfoInteger(TERMINAL_BUILD) >= 3000)
{
   ...
}
```

Recordemos que el número de versión del compilador con el que se construye el programa está disponible en el código fuente a través de las definiciones de macro __MQLBUILD__ o __MQL5BUILD__ (véase Constantes predefinidas).

4.9.3 Tipo de programa y licencia

El mismo código fuente puede incluirse de algún modo en programas MQL de distintos tipos. Además de la opción de incluir los códigos fuente (directiva de preprocesador *#include*) en un producto común en la fase de compilación, también es posible ensamblar las bibliotecas, es decir, módulos binarios del programa conectados al programa principal en la fase de ejecución.

No obstante, algunas funciones sólo pueden utilizarse en determinados tipos de programas. Por ejemplo, la función *OrderCalcMargin* no puede utilizarse en indicadores. Aunque esta limitación no parece estar fundamentalmente justificada, el desarrollador de un algoritmo universal para calcular fondos de garantía, que se puede incorporar no sólo a los Asesores Expertos sino también a los indicadores, debería tener en cuenta este matiz y proporcionar un método de cálculo alternativo para los indicadores.

En la sección correspondiente de cada capítulo se ofrece una lista completa de las restricciones de los tipos de programas. En todos estos casos es importante conocer el tipo de programa «padre».

Para determinar el tipo de programa, existe la propiedad MQL_PROGRAM_TYPE en ENUM_MQL_INFO_INTEGER. Los posibles valores de las propiedades se describen en la enumeración ENUM_PROGRAM_TYPE.

Identificador	Valor	Descripción
PROGRAM_SCRIPT	1	Script
PROGRAM_EXPERT	2	Asesor Experto
PROGRAM_INDICATOR	4	Indicador
PROGRAM_SERVICE	5	Servicio

En el fragmento de registro de la sección anterior vimos que la propiedad PROGRAM_SCRIPT está establecida en 1 porque nuestra prueba es un script. Para obtener una descripción en forma de cadena puede utilizar la función *EnumToString*.

```
ENUM_PROGRAM_TYPE type = (ENUM_PROGRAM_TYPE)MQLInfoInteger(MQL_PROGRAM_TYPE);
Print(EnumToString(type));
```

Otra propiedad de un programa MQL que es conveniente analizar para habilitar/deshabilitar ciertas características es el tipo de licencia. Como sabe, los programas MQL pueden distribuirse libremente o dentro del Mercado MQL5. Además, el programa en la tienda se puede comprar o descargar como

versión demo. Estos factores son fáciles de comprobar y, si se desea, se pueden adaptar los algoritmos para ellos. Para estos fines existe la propiedad MQL_LICENSE_TYPE en ENUM_MQL_INFO_INTEGER, que utiliza la enumeración ENUM_LICENSE_TYPE como tipo.

Identificador	Valor	Descripción
LICENSE_FREE	0	Versión gratuita ilimitada
LICENSE_DEMO	1	Versión demo de un producto de pago del Mercado que sólo funciona en el probador de estrategias
LICENSE_FULL	2	Versión con licencia adquirida; permite al menos 5 activaciones (pueden aumentarlas el vendedor)
LICENSE_TIME	3	Versión de tiempo limitado (aún no implantada)

Es importante señalar aquí que la licencia se refiere al módulo binario ex5 desde el que se realiza la solicitud mediante *MQLInfoInteger(MQL_LICENSE_TYPE)*. Dentro de una biblioteca, esta función devolverá la licencia propia de la biblioteca, no el programa principal al que está enlazada la biblioteca.

Como ejemplo para probar las dos funciones de esta sección se incluye con el libro un sencillo servicio *EnvType.mq5*. No contiene ciclo de trabajo y, por lo tanto, terminará inmediatamente después de ejecutar las dos instrucciones en *OnStart*.

```
#property service

void OnStart()
{
   Print(EnumToString((ENUM_PROGRAM_TYPE)MQLInfoInteger(MQL_PROGRAM_TYPE)));
   Print(EnumToString((ENUM_LICENSE_TYPE)MQLInfoInteger(MQL_LICENSE_TYPE)));
}
```

Para simplificar su lanzamiento, es decir, para eliminar la necesidad de crear una instancia del servicio y ejecutarla a través del menú contextual del Navegador en el terminal, se propone utilizar el depurador: basta con abrir el código fuente en MetaEditor y ejecutar el comando *Depurar -> Empezar con datos reales* (F5, o botón en la barra de herramientas).

Deberíamos obtener las siguientes entradas de registro:

```
EnvType (debug)      PROGRAM_SERVICE
EnvType (debug)      LICENSE_FREE
```

Aquí se puede ver claramente que el tipo de programa es un servicio, y en realidad no hay licencia (uso libre).

4.9.4 Modos de funcionamiento del terminal y del programa

El entorno MetaTrader 5 ofrece una solución a diversas tareas en la intersección entre trading y programación, lo que requiere varios modos de funcionamiento tanto del propio terminal como de un programa específico.

Utilizando la API de MQL5 puede distinguir entre la actividad en línea regular y el backtesting (simulación en el pasado); entre la depuración del código fuente (con el fin de identificar posibles errores) y el análisis de rendimiento (búsqueda de cuellos de botella en el código), así como entre una copia local del terminal y la de la nube (MetaTrader VPS).

Los modos se describen mediante banderas, cada una de las cuales contiene un valor de tipo booleano: *true* o *false*.

Identificador	Descripción
MQL_DEBUG	El programa se ejecuta en modo depuración.
MQL_PROFILER	El programa funciona en modo de perfilaje de código.
MQL_TESTER	El programa funciona en el probador.
MQL_FORWARD	El programa se ejecuta en el proceso de simulación en el futuro (forward testing).
MQL_OPTIMIZATION	El programa se está ejecutando en el proceso de optimización.
MQL_VISUAL_MODE	El programa se ejecuta en modo de prueba visual.
MQL_FRAME_MODE	El Asesor Experto se ejecuta en el gráfico en el modo de recopilación de marcos de resultados de optimización.
TERMINAL_VPS	El terminal funciona en un servidor virtual MetaTrader Virtual Hosting (MetaTrader VPS)

Las banderas MQL_FORWARD, MQL_OPTIMIZATION y MQL_VISUAL_MODE implican la presencia de la bandera MQL_TESTER activada.

Algunas combinaciones de banderas son mutuamente excluyentes, es decir, no pueden activarse al mismo tiempo.

En concreto, la presencia de MQL_FRAME_MODE excluye MQL_TESTER, y viceversa. MQL_OPTIMIZATION excluye MQL_VISUAL_MODE, y MQL_PROFILER excluye MQL_DEBUG.

Estudiaremos todas las banderas relacionadas con las pruebas (MQL_TESTER, MQL_VISUAL_MODE) en las secciones dedicadas a los Asesores Expertos y, en parte, a los indicadores. Todo lo relacionado con la optimización del Asesor Experto (MQL_OPTIMIZATION, MQL_FORWARD, MQL_FRAME_MODE) se abordará en una sección por separado.

Ahora vamos a familiarizarnos con los principios de la lectura de banderas usando el ejemplo de los modos de depuración (MQL_DEBUG) y perfilado (MQL_PROFILER). Al mismo tiempo, recordaremos cómo se activan estos modos desde MetaEditor (para obtener más detalles, véase la documentación, en las secciones Depuración y Perfilaje).

Utilizaremos el script *EnvMode.mq5*.

```
void OnStart()
{
   PRTF(MQLInfoInteger(MQL_TESTER));
   PRTF(MQLInfoInteger(MQL_DEBUG));
   PRTF(MQLInfoInteger(MQL_PROFILER));
   PRTF(MQLInfoInteger(MQL_VISUAL_MODE));
   PRTF(MQLInfoInteger(MQL_OPTIMIZATION));
   PRTF(MQLInfoInteger(MQL_FORWARD));
   PRTF(MQLInfoInteger(MQL_FRAME_MODE));
}
```

Antes de ejecutar el programa, debe comprobar la configuración de depuración/perfilaje. Para ello, en MetaEditor, ejecute el comando *Herramientas -> Opciones* y compruebe los valores de los campos en la pestaña *Depuración/Perfilado*. Si la opción *Usar opciones especificadas* está activada, serán los valores de los campos subyacentes los que afectarán al gráfico del instrumento financiero y al marco temporal en el que se lanzará el programa. Si la opción está desactivada, se utilizará el primer instrumento financiero en *Market Watch* y el marco temporal H1.

En esta fase, la elección de la opción no es crítica.

Tras los preparativos, ejecute el script mediante el comando *Depurar -> Empezar con datos reales* (F5). Como el script sólo imprime en el registro las propiedades solicitadas (y no necesitamos puntos de interrupción en él), su ejecución será instantánea. Si es necesaria la depuración paso a paso, podríamos poner un punto de interrupción (F9) en cualquier sentencia del código fuente, y la ejecución del script se congelaría ahí durante el periodo que necesitemos, lo que permite estudiar el contenido de todas las variables en MetaEditor, y también movernos línea a línea (F10) a lo largo del algoritmo.

En el registro de MetaTrader 5 (pestaña Expertos), veremos lo siguiente:

```
MQLInfoInteger(MQL_TESTER)=0 / ok
MQLInfoInteger(MQL_DEBUG)=1 / ok
MQLInfoInteger(MQL_PROFILER)=0 / ok
MQLInfoInteger(MQL_VISUAL_MODE)=0 / ok
MQLInfoInteger(MQL_OPTIMIZATION)=0 / ok
MQLInfoInteger(MQL_FORWARD)=0 / ok
MQLInfoInteger(MQL_FRAME_MODE)=0 / ok
```

Se reinician las banderas de todos los modos, excepto MQL_DEBUG.

Ahora vamos a ejecutar el mismo script desde *Navegador* en MetaTrader 5 (sólo tiene que arrastrarlo con el ratón a cualquier gráfico). Obtendremos un conjunto casi idéntico de banderas, pero esta vez MQL_DEBUG será igual a 0 (porque el programa fue ejecutado de manera regular, y no bajo un depurador).

> Tenga en cuenta que el lanzamiento del programa con depuración va precedido de su recompilación en un modo especial cuando la información de servicio que permite la depuración se añade al archivo ejecutable. Dicho archivo binario es más grande y lento de lo habitual. Por lo tanto, una vez finalizada la depuración, antes de utilizarlo en operaciones de trading reales, transferirlo al cliente o cargarlo en el Mercado, el programa debe volver a compilarse con el comando *Archivo -> Compilar* (F7).
>
> El método de compilación no afecta directamente a la propiedad MQL_DEBUG. La versión de depuración del programa, como podemos ver, se puede iniciar en el terminal sin un depurador, y MQL_DEBUG se restablecerá en este caso. Dos macros integradas permiten determinar el método

de compilación: _DEBUG y _RELEASE (véase la sección Constantes predefinidas). Son constantes, no funciones, porque esta propiedad está «programada» en el programa en tiempo de compilación y no puede modificarse (a diferencia del entorno de ejecución).

Ahora vamos a ejecutar el comando *Depurar -> Comenzar el perfilado con datos reales* en MetaEditor. Por supuesto, no tiene sentido perfilar un script tan simple, pero nuestra tarea ahora es asegurarnos de que la bandera apropiada está activada en las propiedades del entorno. Efectivamente, frente a MQL_PROFILER ahora hay 1.

```
MQLInfoInteger(MQL_TESTER)=0 / ok
MQLInfoInteger(MQL_DEBUG)=0 / ok
MQLInfoInteger(MQL_PROFILER)=1 / ok
...
```

El lanzamiento del programa con perfilaje también va precedido de su recompilación en otro modo especial, que añade al archivo binario otra información de servicio necesaria para medir la velocidad de ejecución de las instrucciones. Tras analizar el informe del perfilador y solucionar los cuellos de botella, deberá recompilar el programa de la forma habitual.

En principio, la depuración y el perfilaje pueden realizarse tanto en línea como en el probador (MQL_TESTER) sobre datos históricos, pero el probador sólo admite Asesores Expertos e indicadores. Por lo tanto, es imposible ver la bandera MQL_TESTER o MQL_VISUAL_MODE establecida en el ejemplo de script.

Como sabe, MetaTrader 5 le permite probar los programas de trading en modo rápido (sin un gráfico) y en modo visual (en un gráfico separado). Es en el segundo caso cuando se habilitarán las propiedades MQL_VISUAL_MODE. Tiene sentido comprobarlo, en especial para desactivar las manipulaciones con objetos gráficos en ausencia de visualización.

Para depurar en modo visual utilizando el historial, primero debe activar la opción *Use visual mode for debugging on history* en el cuadro de diálogo de configuración de MetaEditor. Los programas analíticos (indicadores) se prueban siempre en modo visual.

Tenga en cuenta que la depuración en línea no es segura para Asesores Expertos de trading.

4.9.5 Permisos

MetaTrader 5 proporciona funciones para restringir la ejecución de ciertas acciones por parte de los programas MQL por razones de seguridad. Algunas de estas restricciones son de dos niveles, es decir, se establecen por separado para el terminal en su conjunto y para un programa específico. Los ajustes del terminal tienen prioridad o actúan como valores predeterminados para los ajustes de cualquier programa MQL. Por ejemplo, un operador de trading puede desactivar todas las operaciones de trading automatizadas marcando la casilla correspondiente en el cuadro de diálogo de configuración de MetaTrader 5. En este caso, los permisos de trading privados establecidos anteriormente para robots específicos en sus cuadros de diálogo pierden su validez.

En la API de MQL5, dichas restricciones (o viceversa, los permisos) están disponibles para su lectura a través de las funciones *TerminalInfoInteger* y *MQLInfoInteger*. Dado que tienen el mismo efecto en un programa MQL, el programa debe comprobar las prohibiciones generales y específicas con el mismo cuidado (para evitar generar un error al intentar realizar una acción ilegal). Por lo tanto, en esta sección se ofrece una lista de todas las opciones de los distintos niveles.

Todos los permisos son banderas booleanas, es decir, almacenan los valores de *true* o *false*.

Identificador	Descripción
TERMINAL_DLLS_ALLOWED	Permiso para utilizar la DLL
TERMINAL_TRADE_ALLOWED	Permiso para operar automáticamente en línea
TERMINAL_EMAIL_ENABLED	Permiso para enviar correos electrónicos (el servidor SMTP y el inicio de sesión deben especificarse en la configuración del terminal)
TERMINAL_FTP_ENABLED	Permiso para enviar archivos por FTP al servidor especificado (incluidos los informes para la cuenta de trading especificada en la configuración del terminal).
TERMINAL_NOTIFICATIONS_ENABLED	Permiso para enviar notificaciones push a un smartphone
MQL_DLLS_ALLOWED	Permiso para utilizar la DLL de este programa
MQL_TRADE_ALLOWED	Permiso para que un programa opere automáticamente
MQL_SIGNALS_ALLOWED	Permiso para que un programa trabaje con señales

El permiso para utilizar una DLL a nivel de terminal significa que, cuando se ejecuta un programa MQL que contiene un enlace a alguna biblioteca dinámica, la bandera 'Enable DLL Import' en la pestaña Dependencias se activará por defecto en su cuadro de diálogo de propiedades. Si la bandera está desactivada en la configuración del terminal, la opción en las propiedades del programa MQL estará desactivada por defecto. En cualquier caso, el usuario debe permitir las importaciones para el programa individual (hay una excepción para los scripts, que se aborda más adelante). De lo contrario, el programa no se ejecutará.

En otras palabras: las banderas TERMINAL_DLLS_ALLOWED y MQL_DLLS_ALLOWED pueden ser comprobadas tanto por un programa sin vinculación a una DLL, como por un programa con vinculación, pero para este programa, MQL_DLLS_ALLOWED debe ser inequívocamente igual a true (debido a que ya se ha iniciado). Así, como parte de los sistemas de software que requieren una DLL, probablemente tenga sentido proporcionar una utilidad independiente que controle el estado de la bandera y muestre diagnósticos para el usuario si se apaga repentinamente. Por ejemplo, un Asesor Experto puede necesitar un indicador que utilice una DLL. Entonces, antes de intentar cargar el indicador y obtener su manejador, el EA puede comprobar la bandera TERMINAL_DLLS_ALLOWED y generar una advertencia si la bandera se restablece.

En el caso de los scripts, el comportamiento es ligeramente diferente porque el cuadro de diálogo de configuración de scripts sólo se abre si la directiva *#property script_show_inputs* está presente en el código fuente. Si no lo está, el cuadro de diálogo aparece cuando se restablece la bandera TERMINAL_DLLS_ALLOWED en la configuración del terminal (y el usuario debe activar la bandera para que el script funcione). Cuando la bandera general TERMINAL_DLLS_ALLOWED está activada, el script se ejecuta sin confirmación del usuario, es decir, se asume que el valor de MQL_DLLS_ALLOWED es *true* (según TERMINAL_DLLS_ALLOWED).

> Cuando se trabaja en el probador, las banderas TERMINAL_TRADE_ALLOWED y MQL_TRADE_ALLOWED son siempre iguales a *true*. Sin embargo, en los indicadores, el acceso a todas las funciones de trading está prohibido independientemente de estas banderas. El comprobador no permite probar programas MQL con dependencias DLL.

Las banderas TERMINAL_EMAIL_ENABLED, TERMINAL_FTP_ENABLED y TERMINAL_NOTIFICATIONS_ENABLED son críticas para las funciones *send mail*, *SendFTP* y *send notification*, que se describen en la sección Funciones de red. La bandera MQL_SIGNALS_ALLOWED afecta a la disponibilidad de un grupo de funciones que gestionan la suscripción de señales de trading mql5.com (no tratadas en este libro). Su estado corresponde a la opción '*Allow changing signal settings*' en la pestaña *Common* de las propiedades del programa MQL.

Dado que la comprobación de algunas propiedades requiere un esfuerzo adicional, tiene sentido envolver las banderas en una clase que oculta múltiples llamadas a varias funciones del sistema en sus métodos. Esto es tanto más necesario por cuanto que algunos permisos no se limitan a las opciones anteriores. Por ejemplo, el permiso para operar se puede establecer (o eliminar) no sólo a nivel de terminal o programa MQL, sino también para un instrumento financiero individual, de acuerdo con su especificación por parte de su broker y las sesiones de intercambio. Por lo tanto, en este paso, presentaremos un borrador de la clase Permisos que sólo contendrá elementos familiares, y a continuación lo mejoraremos para API de aplicaciones particulares.

Gracias a la clase que actúa como capa del programa, el programador no tiene que recordar qué permisos están definidos para las funciones *TerminalInfo* y cuáles para las funciones *MqlInfo*.

El código fuente está en el archivo *EnvPermissions.mq5*.

```
class Permissions
{
public:
   static bool isTradeEnabled(const string symbol = NULL, const datetime session = 0)
   {
      // TODO: will be supplemented by applied checks of the symbol and sessions
      return PRTF(TerminalInfoInteger(TERMINAL_TRADE_ALLOWED))
          && PRTF(MQLInfoInteger(MQL_TRADE_ALLOWED));
   }
   static bool isDllsEnabledByDefault()
   {
      return (bool)PRTF(TerminalInfoInteger(TERMINAL_DLLS_ALLOWED));
   }
   static bool isDllsEnabled()
   {
      return (bool)PRTF(MQLInfoInteger(MQL_DLLS_ALLOWED));
   }

   static bool isEmailEnabled()
   {
      return (bool)PRTF(TerminalInfoInteger(TERMINAL_EMAIL_ENABLED));
   }

   static bool isFtpEnabled()
   {
      return (bool)PRTF(TerminalInfoInteger(TERMINAL_FTP_ENABLED));
   }

   static bool isPushEnabled()
   {
      return (bool)PRTF(TerminalInfoInteger(TERMINAL_NOTIFICATIONS_ENABLED));
   }

   static bool isSignalsEnabled()
   {
      return (bool)PRTF(MQLInfoInteger(MQL_SIGNALS_ALLOWED));
   }
};
```

Todos los métodos de la clase son estáticos y se invocan en *OnStart*.

```
void OnStart()
{
   Permissions::isTradeEnabled();
   Permissions::isDllsEnabledByDefault();
   Permissions::isDllsEnabled();
   Permissions::isEmailEnabled();
   Permissions::isPushEnabled();
   Permissions::isSignalsEnabled();
}
```

A continuación se muestra un ejemplo de los registros generados.

```
TerminalInfoInteger(TERMINAL_TRADE_ALLOWED)=1 / ok
MQLInfoInteger(MQL_TRADE_ALLOWED)=1 / ok
TerminalInfoInteger(TERMINAL_DLLS_ALLOWED)=0 / ok
MQLInfoInteger(MQL_DLLS_ALLOWED)=0 / ok
TerminalInfoInteger(TERMINAL_EMAIL_ENABLED)=0 / ok
TerminalInfoInteger(TERMINAL_NOTIFICATIONS_ENABLED)=0 / ok
MQLInfoInteger(MQL_SIGNALS_ALLOWED)=0 / ok
```

Para el autoaprendizaje, el script tiene incorporada (pero comentada) la capacidad de conectar DLL del sistema para leer el contenido del portapapeles de Windows. Consideraremos la creación y el uso de bibliotecas, en particular la directiva *#import*, en la séptima parte del libro, en la sección Bibliotecas.

Supongamos que la opción de importación global de DLL está desactivada en el terminal deshabilitado (esta es la configuración recomendada por razones de seguridad). Entonces, si las DLL están conectadas al script, sólo será posible ejecutar el script permitiendo la importación en su cuadro de diálogo de configuración individual, como resultado de lo cual *MQLInfoInteger(MQL_DLLS_ALLOWED)* devolverá 1 (*true*). Si se da el permiso global para la DLL, entonces obtenemos *TerminalInfoInteger(TERMINAL_DLLS_ALLOWED)=1*, y MQL_DLLS_ALLOWED heredará este valor.

4.9.6 Comprobación de las conexiones de red

Como sabe, la plataforma MetaTrader 5 es un sistema distribuido que incluye varios enlaces. Además del terminal de cliente y del servidor broker, incluye la comunidad MQL5, Market y servicios en la nube, entre otras muchas cosas. De hecho, la parte cliente también está distribuida, y consta de un terminal y agentes de prueba que pueden desplegarse en varios ordenadores de una red local. En este caso, la conexión entre los enlaces puede romperse por una razón u otra. Aunque la infraestructura de MetaTrader 5 intenta restaurar automáticamente su funcionalidad, no siempre es posible hacerlo rápidamente.

Por lo tanto, en los programas MQL hay que tener en cuenta la posibilidad de una pérdida de conexión. La API de MQL5 le permite controlar las conexiones más importantes: con el servidor de trading y con la comunidad MQL5. En *TerminalInfoInteger* están disponibles las siguientes propiedades:

Identificador	Descripción
TERMINAL_CONNECTED	Conexión con el servidor de trading
TERMINAL_PING_LAST	El último ping conocido al servidor de trading en microsegundos
TERMINAL_COMMUNITY_ACCOUNT	Disponibilidad de los datos de autorización MQL5.community en el terminal
TERMINAL_COMMUNITY_CONNECTION	Conexión con MQL5.community
TERMINAL_MQID	Disponibilidad de MetaQuotes ID para el envío de notificaciones push

Todas las propiedades excepto TERMINAL_PING_LAST son banderas booleanas. TERMINAL_PING_LAST contiene un valor de tipo *int*.

Además de la conexión, un programa MQL a menudo necesita asegurarse de que los datos que tiene están actualizados. En concreto, la bandera TERMINAL_CONNECTED comprobada no significa todavía

que las cotizaciones que le interesan estén sincronizadas con el servidor. Para ello, debe comprobar adicionalmente *SymbolIsSynchronized* o *SeriesInfoInteger(..., SERIES_SYNCHRONIZED)*. Estas características se tratarán en el capítulo sobre series temporales.

La función *TerminalInfoDouble* admite otra propiedad interesante: TERMINAL_RETRANSMISSION. Denota el porcentaje de paquetes de red reenviados en el protocolo TCP/IP para todas las aplicaciones y servicios en ejecución en este ordenador. Incluso en la red más rápida y mejor configurada, a veces se producen pérdidas de paquetes y, como resultado, no habrá confirmación de la entrega del paquete entre el destinatario y el remitente. En estos casos se vuelve a enviar el paquete perdido. El propio terminal no cuenta el indicador TERMINAL_RETRANSMISSION, sino que lo solicita una vez por minuto en el sistema operativo.

Un valor alto de esta métrica puede indicar problemas externos (conexión a Internet, su proveedor, red local o problemas informáticos), que pueden empeorar la calidad de la conexión del terminal.

Si existe una conexión confirmada con la comunidad (TERMINAL_COMMUNITY_CONNECTION), un programa MQL puede consultar el saldo actual del usuario llamando a *TerminalInfoDouble(TERMINAL_COMMUNITY_BALANCE)*. Esto le permite utilizar una suscripción automatizada a señales de trading de pago (la documentación de la API está disponible en la página web mql5.com).

Comprobemos las propiedades de la lista utilizando el script *EnvConnection.mq5*.

```
void OnStart()
{
   PRTF(TerminalInfoInteger(TERMINAL_CONNECTED));
   PRTF(TerminalInfoInteger(TERMINAL_PING_LAST));
   PRTF(TerminalInfoInteger(TERMINAL_COMMUNITY_ACCOUNT));
   PRTF(TerminalInfoInteger(TERMINAL_COMMUNITY_CONNECTION));
   PRTF(TerminalInfoInteger(TERMINAL_MQID));
   PRTF(TerminalInfoDouble(TERMINAL_RETRANSMISSION));
   PRTF(TerminalInfoDouble(TERMINAL_COMMUNITY_BALANCE));
}
```

He aquí un ejemplo de registro (los valores coincidirán con su configuración).

```
TerminalInfoInteger(TERMINAL_CONNECTED)=1 / ok
TerminalInfoInteger(TERMINAL_PING_LAST)=49082 / ok
TerminalInfoInteger(TERMINAL_COMMUNITY_ACCOUNT)=0 / ok
TerminalInfoInteger(TERMINAL_COMMUNITY_CONNECTION)=0 / ok
TerminalInfoInteger(TERMINAL_MQID)=0 / ok
TerminalInfoDouble(TERMINAL_RETRANSMISSION)=0.0 / ok
TerminalInfoDouble(TERMINAL_COMMUNITY_BALANCE)=0.0 / ok
```

4.9.7 Recursos informáticos: memoria, disco y CPU

Como todos los programas, las aplicaciones MQL consumen recursos del ordenador, incluida memoria, espacio en disco y CPU. Teniendo en cuenta que el propio terminal consume muchos recursos (debido, en especial, a la posible descarga de cotizaciones y ticks para múltiples instrumentos financieros con un largo historial), a veces es necesario analizar y controlar la situación en cuanto a la proximidad de los límites disponibles.

La API de MQL5 proporciona varias propiedades que le permiten calcular los recursos máximos alcanzables y gastados. Las propiedades se resumen en las enumeraciones ENUM_MQL_INFO_INTEGER y ENUM_TERMINAL_INFO_INTEGER.

Identificador	Descripción
MQL_MEMORY_LIMIT	Cantidad máxima posible de memoria dinámica para un programa MQL en Kb
MQL_MEMORY_USED	Memoria utilizada por un programa MQL en Mb
MQL_HANDLES_USED	Número de objetos de clase
TERMINAL_MEMORY_PHYSICAL	RAM física del sistema en Mb
TERMINAL_MEMORY_TOTAL	Memoria (física+archivo de intercambio, es decir, virtual) disponible para el proceso del terminal (agente) en Mb
TERMINAL_MEMORY_AVAILABLE	Memoria libre del proceso del terminal (agente) en Mb, parte de TOTAL
TERMINAL_MEMORY_USED	Memoria utilizada por el terminal (agente) en Mb, parte de TOTAL
TERMINAL_DISK_SPACE	Espacio libre en disco, teniendo en cuenta posibles cuotas para la carpeta MQL5/Files del terminal (agente), en Mb
TERMINAL_CPU_CORES	Número de núcleos de procesador del sistema
TERMINAL_OPENCL_SUPPORT	Versión OpenCL admitida como 0x00010002 = 1.2; «0» significa que OpenCL no está admitido

La cantidad máxima de memoria disponible para un programa MQL se describe mediante la propiedad MQL_MEMORY_LIMIT. Esta es la única propiedad de la lista que utiliza kilobytes (Kb). Todas las demás se devuelven en megabytes (Mb). Por regla general, MQL_MEMORY_LIMIT es igual a TERMINAL_MEMORY_TOTAL, es decir, toda la memoria disponible en el ordenador puede asignarse a un programa MQL por defecto. No obstante, el terminal, en particular su implementación en la nube para MetaTrader VPS, y los agentes de prueba en la nube pueden limitar la memoria para un solo programa MQL. Así, MQL_MEMORY_LIMIT será significativamente menor que TERMINAL_MEMORY_TOTAL.

Dado que Windows normalmente crea un archivo de intercambio que es igual en tamaño a la memoria física (RAM), la propiedad TERMINAL_MEMORY_TOTAL puede ser de hasta 2 veces el tamaño de TERMINAL_MEMORY_PHYSICAL.

Toda la memoria virtual disponible TERMINAL_MEMORY_TOTAL se divide entre la memoria utilizada (TERMINAL_MEMORY_USED) y la aún libre (TERMINAL_MEMORY_AVAILABLE).

El libro viene con el script *EnvProvision.mq5*, que registra todas las propiedades especificadas.

```
void OnStart()
{
   PRTF(MQLInfoInteger(MQL_MEMORY_LIMIT)); // Kb!
   PRTF(MQLInfoInteger(MQL_MEMORY_USED));
   PRTF(TerminalInfoInteger(TERMINAL_MEMORY_PHYSICAL));
   PRTF(TerminalInfoInteger(TERMINAL_MEMORY_TOTAL));
   PRTF(TerminalInfoInteger(TERMINAL_MEMORY_AVAILABLE));
   PRTF(TerminalInfoInteger(TERMINAL_MEMORY_USED));
   PRTF(TerminalInfoInteger(TERMINAL_DISK_SPACE));
   PRTF(TerminalInfoInteger(TERMINAL_CPU_CORES));
   PRTF(TerminalInfoInteger(TERMINAL_OPENCL_SUPPORT));

   uchar array[];
   PRTF(ArrayResize(array, 1024 * 1024 * 10)); // allocate 10 Mb
   PRTF(MQLInfoInteger(MQL_MEMORY_USED));
   PRTF(TerminalInfoInteger(TERMINAL_MEMORY_AVAILABLE));
   PRTF(TerminalInfoInteger(TERMINAL_MEMORY_USED));
}
```

Tras la salida inicial de las propiedades, asignamos 10 Mb para el array y volvemos a comprobar la memoria. A continuación se muestra un ejemplo de resultado (usted tendrá sus propios valores).

```
MQLInfoInteger(MQL_MEMORY_LIMIT)=8388608 / ok
MQLInfoInteger(MQL_MEMORY_USED)=1 / ok
TerminalInfoInteger(TERMINAL_MEMORY_PHYSICAL)=4095 / ok
TerminalInfoInteger(TERMINAL_MEMORY_TOTAL)=8190 / ok
TerminalInfoInteger(TERMINAL_MEMORY_AVAILABLE)=7842 / ok
TerminalInfoInteger(TERMINAL_MEMORY_USED)=348 / ok
TerminalInfoInteger(TERMINAL_DISK_SPACE)=4528 / ok
TerminalInfoInteger(TERMINAL_CPU_CORES)=4 / ok
TerminalInfoInteger(TERMINAL_OPENCL_SUPPORT)=0 / ok
ArrayResize(array,1024*1024*10)=10485760 / ok
MQLInfoInteger(MQL_MEMORY_USED)=11 / ok
TerminalInfoInteger(TERMINAL_MEMORY_AVAILABLE)=7837 / ok
TerminalInfoInteger(TERMINAL_MEMORY_USED)=353 / ok
```

Observe que la memoria virtual total (8190) es el doble de la memoria física (4095). La cantidad de memoria disponible para el script es de 8388608 Kb, que es casi igual a toda la memoria de 8190 Mb. La memoria del sistema libre (7842) y utilizada (348) también suman 8190.

Si antes de asignar memoria para un array, el programa MQL ocupaba 1 Mb; después de asignarla ocupa ya 11 Mb. Mientras tanto, la cantidad de memoria ocupada por el terminal sólo aumentó en 5 Mb (de 348 a 353), ya que algunos recursos se reservaron de antemano.

4.9.8 Especificaciones de la pantalla

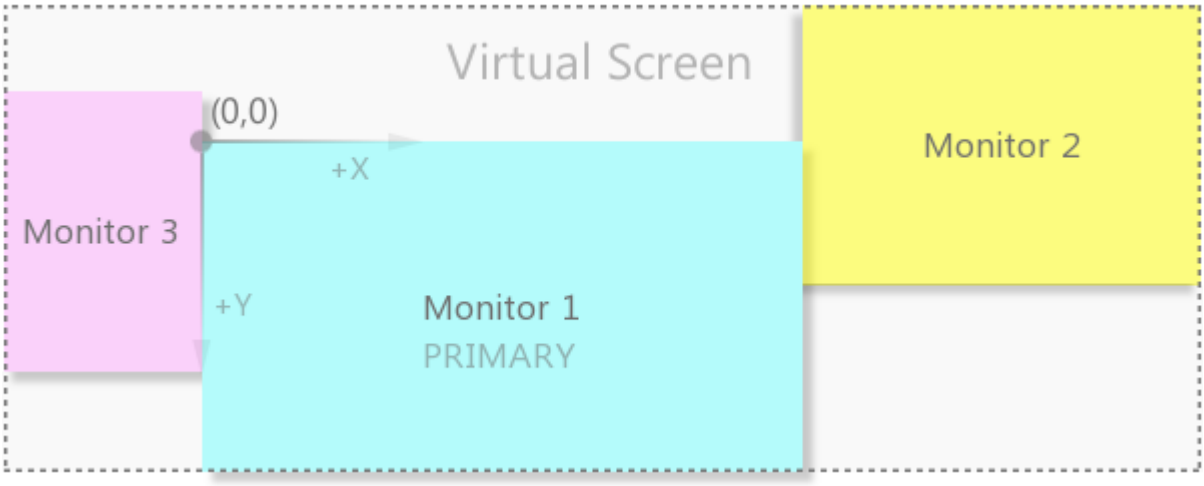
Varias propiedades proporcionadas por la función *TerminalInfoInteger* se refieren al subsistema de vídeo del ordenador.

Identificador	Descripción
TERMINAL_SCREEN_DPI	La resolución de la información transmitida a la pantalla se mide en número de puntos por pulgada lineal (DPI, Dots Per Inch).
TERMINAL_SCREEN_LEFT	Coordenada izquierda de la pantalla virtual
TERMINAL_SCREEN_TOP	Coordenada superior de la pantalla virtual
TERMINAL_SCREEN_WIDTH	Anchura de la pantalla virtual
TERMINAL_SCREEN_HEIGHT	Altura de la pantalla virtual
TERMINAL_LEFT	Coordenada izquierda del terminal respecto a la pantalla virtual
TERMINAL_TOP	Coordenada superior del terminal respecto a la pantalla virtual
TERMINAL_RIGHT	Coordenada derecha del terminal respecto a la pantalla virtual
TERMINAL_BOTTOM	Coordenada inferior del terminal respecto a la pantalla virtual

Conociendo el parámetro TERMINAL_SCREEN_DPI puede establecer las dimensiones de objetos gráficos para que se vean de la misma manera en monitores con resoluciones diferentes. Por ejemplo, si desea crear un botón con un tamaño visible de X centímetros, puede especificarlo como el número de puntos en la pantalla (píxeles) utilizando la siguiente función:

```
int cm2pixels(const double x)
{
   static const double inch2cm = 2.54; // 1 inch equals 2.54 cm
   return (int)(x / inch2cm * TerminalInfoInteger(TERMINAL_SCREEN_DPI));
}
```

La pantalla virtual es un cuadro delimitador (bounding box) de todos los monitores. Si hay más de un monitor en el sistema y el orden de su disposición difiere estrictamente de izquierda a derecha, entonces la coordenada izquierda de la pantalla virtual puede resultar negativa, y el centro (punto de referencia) estará en el borde de dos monitores (en la esquina superior izquierda del monitor principal).



Pantalla virtual desde varios monitores

Si el sistema tiene un monitor, el tamaño de la pantalla virtual se corresponde totalmente con él.

Las coordenadas del terminal no tienen en cuenta su posible maximización actual (es decir, si la ventana principal está maximizada, las propiedades devuelven el tamaño no maximizado, aunque el terminal esté expandido a todo el monitor).

En el script *EnvScreen.mq5*, compruebe las propiedades de la pantalla de lectura.

```
void OnStart()
{
   PRTF(TerminalInfoInteger(TERMINAL_SCREEN_DPI));
   PRTF(TerminalInfoInteger(TERMINAL_SCREEN_LEFT));
   PRTF(TerminalInfoInteger(TERMINAL_SCREEN_TOP));
   PRTF(TerminalInfoInteger(TERMINAL_SCREEN_WIDTH));
   PRTF(TerminalInfoInteger(TERMINAL_SCREEN_HEIGHT));
   PRTF(TerminalInfoInteger(TERMINAL_LEFT));
   PRTF(TerminalInfoInteger(TERMINAL_TOP));
   PRTF(TerminalInfoInteger(TERMINAL_RIGHT));
   PRTF(TerminalInfoInteger(TERMINAL_BOTTOM));
}
```

He aquí un ejemplo de las entradas de registro resultantes.

```
TerminalInfoInteger(TERMINAL_SCREEN_DPI)=96 / ok
TerminalInfoInteger(TERMINAL_SCREEN_LEFT)=0 / ok
TerminalInfoInteger(TERMINAL_SCREEN_TOP)=0 / ok
TerminalInfoInteger(TERMINAL_SCREEN_WIDTH)=1440 / ok
TerminalInfoInteger(TERMINAL_SCREEN_HEIGHT)=900 / ok
TerminalInfoInteger(TERMINAL_LEFT)=126 / ok
TerminalInfoInteger(TERMINAL_TOP)=41 / ok
TerminalInfoInteger(TERMINAL_RIGHT)=1334 / ok
TerminalInfoInteger(TERMINAL_BOTTOM)=836 / ok
```

Además de los tamaños generales de la pantalla y de la ventana del terminal, los programas MQL necesitan con bastante frecuencia analizar el tamaño actual del gráfico (ventana hija dentro del terminal). Para ello existe un conjunto especial de funciones (en particular, *ChartGetInteger*), de las que hablaremos en la sección Gráficos.

4.9.9 Propiedades del terminal y de la cadena de programa

Las funciones *MQLInfoString* y *TerminalInfoString* se pueden utilizar para averiguar diversas propiedades de cadena del terminal y el programa MQL.

Identificador	Descripción
MQL_PROGRAM_NAME	El nombre del programa MQL en ejecución
MQL_PROGRAM_PATH	Ruta para este programa MQL en ejecución
TERMINAL_LANGUAGE	Idioma del terminal
TERMINAL_COMPANY	Nombre de la empresa (broker)
TERMINAL_NAME	Nombre del terminal
TERMINAL_PATH	La carpeta desde la que se inicia el terminal
TERMINAL_DATA_PATH	La carpeta donde se almacenan los datos del terminal
TERMINAL_COMMONDATA_PATH	La carpeta compartida de todos los terminales cliente instalados en el ordenador

El nombre del programa en ejecución (MQL_PROGRAM_NAME) suele coincidir con el nombre del módulo principal (archivo mq5), pero puede diferir. En concreto, si su código fuente se compila en una biblioteca que se ha importado en otro programa MQL (Asesor Experto, indicador, script o servicio), entonces la propiedad MQL_PROGRAM_NAME devolverá el nombre del programa principal, no de la biblioteca (la biblioteca no es un programa independiente que se pueda ejecutar).

Hablamos de la disposición de las carpetas de terminales de trabajo en Trabajar con archivos. Utilizando las propiedades enumeradas, puede averiguar dónde está instalado el terminal (TERMINAL_PATH), así como encontrar los datos de trabajo de la instancia actual del terminal (TERMINAL_DATA_PATH) y de todas las instancias (TERMINAL_COMMONDATA_PATH).

Un simple script *EnvDescription.mq5* registra todas estas propiedades.

```
void OnStart()
{
   PRTF(MQLInfoString(MQL_PROGRAM_NAME));
   PRTF(MQLInfoString(MQL_PROGRAM_PATH));
   PRTF(TerminalInfoString(TERMINAL_LANGUAGE));
   PRTF(TerminalInfoString(TERMINAL_COMPANY));
   PRTF(TerminalInfoString(TERMINAL_NAME));
   PRTF(TerminalInfoString(TERMINAL_PATH));
   PRTF(TerminalInfoString(TERMINAL_DATA_PATH));
   PRTF(TerminalInfoString(TERMINAL_COMMONDATA_PATH));
}
```

A continuación se muestra un ejemplo de resultado:

```
MQLInfoString(MQL_PROGRAM_NAME)=EnvDescription / ok
MQLInfoString(MQL_PROGRAM_PATH)= »
» C:\Program Files\MT5East\MQL5\Scripts\MQL5Book\p4\EnvDescription.ex5 / ok
TerminalInfoString(TERMINAL_LANGUAGE)=Russian / ok
TerminalInfoString(TERMINAL_COMPANY)=MetaQuotes Software Corp. / ok
TerminalInfoString(TERMINAL_NAME)=MetaTrader 5 / ok
TerminalInfoString(TERMINAL_PATH)=C:\Program Files\MT5East / ok
TerminalInfoString(TERMINAL_DATA_PATH)=C:\Program Files\MT5East / ok
TerminalInfoString(TERMINAL_COMMONDATA_PATH)= »
» C:\Users\User\AppData\Roaming\MetaQuotes\Terminal\Common / ok
```

El idioma de interfaz del terminal puede encontrarse no sólo como una cadena en la propiedad TERMINAL_LANGUAGE, sino también como un número de página de códigos (véase la propiedad TERMINAL_CODEPAGE en la siguiente sección).

4.9.10 Propiedades personalizadas: límite de barras e idioma de la interfaz

Entre las propiedades del terminal, hay dos propiedades especiales que el usuario puede cambiar interactivamente. Entre ellas se incluye el número máximo por defecto de barras mostradas en cada gráfico (corresponde al valor del campo *Max. bars in the window* en el cuadro de diálogo *Options*, así como el idioma de la interfaz (seleccionado mediante el comando *Ver -> Languages*).

Identificador	Descripción
TERMINAL_MAXBARS	Número máximo de barras en el gráfico
TERMINAL_CODEPAGE	Número de página del código del idioma seleccionado en el terminal de cliente

Tenga en cuenta que el valor TERMINAL_MAXBARS establece el límite superior para la visualización de barras, pero, de hecho, su número puede ser menor si la profundidad del historial de cotizaciones disponibles no es suficiente en algún marco temporal. Por otra parte, la longitud del historial también puede superar el límite TERMINAL_MAXBARS especificado. A continuación, puede hallar el número de barras potencialmente disponibles utilizando la función del grupo de propiedades series temporales: *SeriesInfoInteger* con la propiedad SERIES_BARS_COUNT. Tenga en cuenta que el valor TERMINAL_MAXBARS afecta directamente al consumo de RAM.

4.9.11 Vincular un programa a propiedades en tiempo de ejecución

Como ejemplo de trabajo con las propiedades descritas en las secciones anteriores, veamos la popular tarea de vincular un programa MQL a un entorno de hardware para protegerlo de la copia. Cuando el programa se distribuye a través de MQL5 Market, la vinculación la proporciona el propio servicio. Sin embargo, si el programa se desarrolla a medida, puede vincularse al número de cuenta, al nombre del cliente o a las propiedades disponibles del terminal (ordenador). La primera no siempre es conveniente, porque muchos operadores de trading tienen varias cuentas reales (probablemente con distintos brokers), por no hablar de las cuentas demo con un periodo de validez limitado. La segunda puede ser ficticia o demasiado corriente. Por lo tanto, implementaremos un algoritmo prototipo para vincular un programa a un conjunto seleccionado de propiedades del entorno. Los esquemas de seguridad más serios probablemente podrían utilizar una DLL y leer directamente las etiquetas de hardware del

dispositivo desde Windows, pero no todos los clientes aceptarán ejecutar bibliotecas potencialmente inseguras.

Nuestra opción de protección se presenta en el script *EnvSignature.mq5*. El script calcula hashes a partir de las propiedades dadas del entorno y crea una firma única (huella) basada en ellas.

El hashing es un procesamiento especial de información arbitraria, como resultado del cual se crea un nuevo bloque de datos que tiene las siguientes características (están garantizadas por el algoritmo utilizado):

- La coincidencia de los valores hash de dos conjuntos de datos originales significa, con una probabilidad de casi el 100 %, que los datos son idénticos (la probabilidad de una coincidencia aleatoria es insignificante).
- Si los datos originales cambian, su valor hash también cambiará.
- Es imposible restaurar matemáticamente los datos originales a partir del valor hash (permanecen secretos) a menos que se realice una enumeración completa de los posibles valores iniciales (si su tamaño inicial aumenta y no hay información sobre su estructura, el problema es irresoluble en un futuro previsible).
- El tamaño del hash es fijo (no depende de la cantidad de datos iniciales).

Supongamos que una de las propiedades del entorno está descrita por la cadena «TERMINAL_LANGUAGE=German». Se puede obtener con una simple sentencia como la siguiente (simplificada):

```
string language = EnumToString(TERMINAL_LANGUAGE) +
            "=" + TerminalInfoString(TERMINAL_LANGUAGE);
```

El idioma real coincidirá con la configuración. Teniendo una hipotética función *Hash*, podemos calcular la firma.

```
string signature = Hash(language);
```

Si hay más propiedades, simplemente repetimos el procedimiento para todas ellas, o solicitamos un hash a partir de las cadenas combinadas (hasta ahora esto es pseudocódigo, no forma parte del programa real).

```
string properties[];
// fill in the property lines as you wish
// ...
string signature;
for(int i = 0; i < ArraySize(properties); ++i)
{
   signature += properties[i];
}
return Hash(signature);
```

La firma recibida puede ser comunicada por el usuario al desarrollador del programa, que la «firmará» de forma especial, al recibir una cadena de validación adecuada sólo para esta firma. La firma también se basa en hashing y requiere el conocimiento de algún secreto (frase de contraseña), conocido sólo por el desarrollador y codificado en el programa (para la fase de verificación).

El desarrollador pasará la cadena de validación al usuario, que podrá ejecutar el programa especificando esta cadena en los parámetros.

Cuando se lanza sin una cadena de validación, el programa debe generar una nueva firma para el entorno actual, imprimirla en el registro y salir (esta información debe pasarse al desarrollador). Con una cadena de validación no válida, el programa debe mostrar un mensaje de error y salir.

Se pueden proporcionar varios modos de lanzamiento para el propio desarrollador: con una firma, pero sin una cadena de validación (para generar la última), o con una firma y una cadena de validación (aquí el programa volverá a firmar la firma y la comparará con la cadena de validación especificada sólo para comprobar).

Calculemos cuán selectiva será dicha protección. Al fin y al cabo, la vinculación aquí no se realiza a un único identificador de nada.

En la siguiente tabla se ofrecen estadísticas sobre dos características: el tamaño de la pantalla y la memoria RAM. Obviamente, los valores cambiarán con el tiempo, pero la distribución aproximada seguirá siendo la misma: unos pocos valores característicos serán los más populares, mientras que algunos «nuevos» avanzados y otros «antiguos» que van saliendo de la circulación conformarán «colas» decrecientes.

Pantalla	**1920x1080**	**1536x864**	**1440x900**	**1366x768**	**800x600**
RAM	**21 %**	**7 %**	**5 %**	**10 %**	**4 %**
4 Gb 20 %	4.20	1.40	1.00	2.0	0.8
8 Gb 20 %	4.20	1.40	1.00	2.0	0.8
16 Gb 15 %	3.15	1.05	0.75	1.5	0.6
32 Gb 10 %	2.10	0.70	0.50	1.0	0.4
64 Gb 5 %.	1.05	0.35	0.25	0.5	0.2

Preste atención a las celdas con los valores más grandes, porque significan las mismas firmas (a menos que introduzcamos un elemento de aleatoriedad en ellas, que se discutirá más adelante). En este caso, las dos combinaciones de características de la esquina superior izquierda son las más probables, con un 4.2 % cada una. Pero estas son solo dos características. Si añade el idioma de la interfaz, la zona horaria, el número de núcleos y la ruta de datos de trabajo (preferiblemente compartida, ya que contiene el nombre de usuario de Windows) al entorno evaluado, el número de posibles coincidencias disminuirá notablemente.

Para el hashing, utilizamos la función integrada *CryptEncode* (se describirá en la sección Criptografía) que admita el método hash SHA256. Como su nombre indica, produce un hash de 256 bits, es decir, 32 bytes. Si necesitáramos mostrárselo al usuario, entonces lo traduciríamos a texto en representación hexadecimal y obtendríamos una cadena de 64 caracteres de longitud.

Para que la firma sea más corta, la convertiremos utilizando la codificación Base64 (también es compatible con la función *CryptEncode* y su homóloga *CryptDecode*), lo que dará como resultado una cadena de 44 caracteres de longitud. A diferencia de una operación hash unidireccional, la codificación Base64 es reversible, es decir, los datos originales pueden recuperarse a partir de ella.

Las operaciones principales se implementan mediante la clase *EnvSignature*, que define la cadena *data* que debe acumular ciertos fragmentos que describen el entorno. La interfaz pública consiste en varias versiones sobrecargadas de la función *append* para añadir cadenas con propiedades de entorno. Esencialmente, unen el nombre de la propiedad solicitada y su valor utilizando como enlace algún elemento abstracto devuelto por el método virtual 'pepper'. La clase derivada lo definirá como una cadena específica (pero puede estar vacía).

```
class EnvSignature
{
private:
   string data;
protected:
   virtual string pepper() = 0;
public:
   bool append(const ENUM_TERMINAL_INFO_STRING e)
   {
      return append(EnumToString(e) + pepper() + TerminalInfoString(e));
   }
   bool append(const ENUM_MQL_INFO_STRING e)
   {
      return append(EnumToString(e) + pepper() + MQLInfoString(e));
   }
   bool append(const ENUM_TERMINAL_INFO_INTEGER e)
   {
      return append(EnumToString(e) + pepper()
        + StringFormat("%d", TerminalInfoInteger(e)));
   }
   bool append(const ENUM_MQL_INFO_INTEGER e)
   {
      return append(EnumToString(e) + pepper()
        + StringFormat("%d", MQLInfoInteger(e)));
   }
```

Para añadir una cadena arbitraria a un objeto, existe un método genérico *append*, que se llama en los métodos anteriores.

```
   bool append(const string s)
   {
      data += s;
      return true;
   }
```

Opcionalmente, el desarrollador puede añadir, por así, «sal» a los datos hash. Se trata de un array con datos generados aleatoriamente, lo que complica aún más la inversión del hash. Cada generación de la firma será diferente de la anterior, aunque el entorno permanezca constante. La implementación de esta característica, así como de otros aspectos de protección más específicos (como el uso de cifrado simétrico y el cálculo dinámico del secreto) se dejan para un estudio independiente.

Dado que el entorno se compone de propiedades bien conocidas (su lista está limitada por las constantes de la API de MQL5), y no todas ellas son suficientemente únicas, nuestra defensa, tal y como la hemos calculado, puede generar las mismas firmas para distintos usuarios si no utilizamos la sal. La coincidencia de firmas no permitirá identificar el origen de la fuga de licencias si ésta se ha producido.

Por lo tanto, puede aumentar la eficacia de la protección cambiando el método de presentación de las propiedades antes del hash para cada cliente. Por supuesto, el método en sí no debe divulgarse. En el ejemplo analizado, esto implica cambiar el contenido del método *pepper* y volver a compilar el producto, lo cual puede resultar caro, pero permite evitar el uso de sal al azar.

Con la cadena de propiedades rellenada, podemos generar una firma. Para ello se utiliza el método *emit*.

```
string emit() const
{
   uchar pack[];
   if(StringToCharArray(data + secret(), pack, 0,
      StringLen(data) + StringLen(secret()), CP_UTF8) <= 0) return NULL;

   uchar key[], result[];
   if(CryptEncode(CRYPT_HASH_SHA256, pack, key, result) <= 0) return NULL;
   Print("Hash bytes:");
   ArrayPrint(result);

   uchar text[];
   CryptEncode(CRYPT_BASE64, result, key, text);
   return CharArrayToString(text);
}
```

El método añade un cierto secreto (una secuencia de bytes que sólo conoce el desarrollador y que se encuentra dentro del programa) a los datos y calcula el hash de la cadena compartida. El secreto se obtiene del método virtual *secret*, que también definirá la clase derivada.

El array de bytes resultante con el hash se codifica en una cadena utilizando Base64.

Ahora viene la función de clase más importante: *check*. Es esta función la que implementa la firma desde el desarrollador y la comprueba desde el usuario.

```
bool check(const string sig, string &validation)
{
   uchar bytes[];
   const int n = StringToCharArray(sig + secret(), bytes, 0,
      StringLen(sig) + StringLen(secret()), CP_UTF8);
   if(n <= 0) return false;

   uchar key[], result1[], result2[];
   if(CryptEncode(CRYPT_HASH_SHA256, bytes, key, result1) <= 0) return false;

   /*
     WARNING
     The following code should only be present in the developer utility.
     The program supplied to the user must compile without this if.
   */
   #ifdef I_AM_DEVELOPER
   if(StringLen(validation) == 0)
   {
      if(CryptEncode(CRYPT_BASE64, result1, key, result2) <= 0) return false;
      validation = CharArrayToString(result2);
      return true;
   }
   #endif
   uchar values[];
   // the exact length is needed to not append terminating '0'
   if(StringToCharArray(validation, values, 0,
      StringLen(validation)) <= 0) return false;
   if(CryptDecode(CRYPT_BASE64, values, key, result2) <= 0) return false;

   return ArrayCompare(result1, result2) == 0;
}
```

Durante el funcionamiento normal (para el usuario), el método calcula el hash a partir de la firma recibida, complementada con el secreto, y lo compara con el valor de la cadena de validación (primero debe descodificarse de Base64 en la representación binaria en bruto del hash). Si los dos hashes coinciden, la validación es correcta: la cadena de validación coincide con el conjunto de propiedades. Obviamente, una cadena de validación vacía (o una cadena introducida al azar) no pasará la prueba.

En la máquina del desarrollador, la macro I_AM_DEVELOPER debe definirse en el código fuente de la utilidad de firma, lo que provoca que una cadena de validación vacía se trate de forma diferente. En este caso, el hash resultante se codifica en Base64, y esta cadena se pasa a través del parámetro *validation*. De este modo, la utilidad podrá mostrar al desarrollador una cadena de validación ya preparada para la firma dada.

Para crear un objeto, se necesita una determinada clase derivada que defina cadenas con el secreto y con «pepper».

```
// WARNING: change the macro to your own set of random bytes
#define PROGRAM_SPECIFIC_SECRET "<PROGRAM-SPECIFIC-SECRET>"
// WARNING: choose your characters to link in pairs name'='value
#define INSTANCE_SPECIFIC_PEPPER "=" // obvious single sign is selected for demo
// WARNING: the following macro needs to be disabled in the real product,
//          it should only be in the signature utility
#define I_AM_DEVELOPER
#ifdef I_AM_DEVELOPER
#define INPUT input
#else
#define INPUT const
#endif

INPUT string Signature = "";
INPUT string Secret = PROGRAM_SPECIFIC_SECRET;
INPUT string Pepper = INSTANCE_SPECIFIC_PEPPER;

class MyEnvSignature : public EnvSignature
{
protected:
   virtual string secret() override
   {
      return Secret;
   }
   virtual string pepper() override
   {
      return Pepper;
   }
};
```

Escojamos rápidamente algunas propiedades para rellenar la firma.

```
void FillEnvironment(EnvSignature &env)
{
   // the order is not important, you can mix
   env.append(TERMINAL_LANGUAGE);
   env.append(TERMINAL_COMMONDATA_PATH);
   env.append(TERMINAL_CPU_CORES);
   env.append(TERMINAL_MEMORY_PHYSICAL);
   env.append(TERMINAL_SCREEN_DPI);
   env.append(TERMINAL_SCREEN_WIDTH);
   env.append(TERMINAL_SCREEN_HEIGHT);
   env.append(TERMINAL_VPS);
   env.append(MQL_PROGRAM_TYPE);
}
```

Ahora todo está listo para probar nuestro esquema de protección en la función *OnStart*. Pero primero, veamos las variables de entrada. Dado que el mismo programa se compilará en dos versiones, para el usuario final y para el desarrollador, existen dos conjuntos de variables de entrada: para la introducción de los datos de registro por parte del usuario y para la generación de estos datos a partir de la firma del desarrollador. Las variables de entrada destinadas al desarrollador se han descrito anteriormente utilizando la macro INPUT. Sólo la cadena de validación está disponible para el usuario.

```
input string Validation = "";
```

Cuando la cadena esté vacía, el programa recogerá los datos del entorno, generará una nueva firma y la imprimirá en el registro. Esto completa el trabajo del script, ya que aún no se ha confirmado el acceso al código útil.

```
void OnStart()
{
   MyEnvSignature env;
    string signature;
   if(StringLen(Signature) > 0)
   {
     // ... here will be the code to be signed by the author
   }
   else
   {
      FillEnvironment(env);
      signature = env.emit();
   }

   if(StringLen(Validation) == 0)
   {
      Print("Validation string from developer is required to run this script");
      Print("Environment Signature is generated for current state...");
      Print("Signature:", signature);
      return;
   }
   else
   {
     // ... check the validation string here
   }
   Print("The script is validated and running normally");
   // ... actual working code is here
}
```

Si se rellena la variable *Validation*, se comprueba su conformidad con la firma y se finaliza el trabajo en caso de fallo.

```
   if(StringLen(Validation) == 0)
   {
      ...
   }
   else
   {
      validation = Validation; // need a non-const argument
      const bool accessGranted = env.check(Signature, validation);
      if(!accessGranted)
      {
         Print("Wrong validation string, terminating");
         return;
      }
      // success
   }
   Print("The script is validated and running normally");
   // ... actual working code is here
}
```

Si no hay discrepancias, el algoritmo pasa al código de trabajo del programa.

Por parte del desarrollador (en la versión del programa que se diseñó con la macro I_AM_DEVELOPER), se puede introducir una firma. Restauramos el estado del objeto *MyEnvSignature* utilizando la firma y calculamos la cadena de validación.

```
void OnStart()
{
   ...
   if(StringLen(Signature) > 0)
   {
      #ifdef I_AM_DEVELOPER
      if(StringLen(Validation) == 0)
      {
         string validation;
         if(env.check(Signature, validation))
           Print("Validation:", validation);
         return;
      }
      signature = Signature;
      #endif
   }
   ...
```

El desarrollador no sólo puede especificar la firma, sino también validarla: en este caso, la ejecución del código continuará en modo usuario (a efectos de depuración).

Si lo desea, puede simular un cambio en el entorno, por ejemplo, de la siguiente manera:

```
   FillEnvironment(env);
   // artificially make a change in the environment (add a time zone)
   // env.append("Dummy" + (string)(TimeGMTOffset() - TimeDaylightSavings()));
   const string update = env.emit();
   if(update != signature)
   {
      Print("Signature and environment mismatch");
      return;
   }
```

Veamos algunos registros de prueba.

Cuando ejecute por primera vez el script *EnvSignature.mq5*, el «usuario» verá algo como el siguiente registro (los valores variarán debido a las diferencias de entorno):

```
Hash bytes:
  4 249 194 161 242  28  43  60 180 195  54 254  97 223 144 247 216 103 238 245 244 2
Validation string from developer is required to run this script
Environment Signature is generated for current state...
Signature:BPnCofIcKzy0wzb+Yd+Q99hn7vX04AdEZf34hhtmypk=
```

Envía la firma generada al «desarrollador» (no hay usuarios reales durante la prueba, por lo que se citan todos los roles de «usuario» y «desarrollador»), que la introduce en la utilidad de firma (compilada con la macro I_AM_DEVELOPER), en el parámetro *Signature*. Como resultado, el programa generará una cadena de validación:

```
Validation:YBpYpQ0tLIpUhBslIw+AsPhtPG48b0qut9igJ+Tk1fQ=
```

El «desarrollador» lo devuelve al «usuario», y éste, al introducirlo en el parámetro *Validation*, obtendrá el script activado:

```
Hash bytes:
  4 249 194 161 242  28  43  60 180 195  54 254  97 223 144 247 216 103 238 245 244 2
The script is validated and running normally
```

Para demostrar la eficacia de la protección, dupliquemos el script como servicio: para ello, copiemos el archivo en la carpeta *MQL5/Services/MQL5Book/p4/* y sustituyamos la siguiente línea en el código fuente:

```
#property script_show_inputs
```

con la siguiente línea:

```
#property service
```

Vamos a compilar el servicio, crear y ejecutar su instancia, y especificar la cadena de validación recibida previamente en los parámetros de entrada. Como resultado, el servicio abortará (antes de llegar a las sentencias con el código requerido) con el siguiente mensaje:

```
Hash bytes:
147 131  69  39  29 254  83 141  90 102 216 180 229 111   2 246 245  19  35 205 223 1
Wrong validation string, terminating
```

La cuestión es que entre las propiedades del entorno hemos utilizado la cadena MQL_PROGRAM_TYPE. Por lo tanto, una licencia emitida para un tipo de programa no funcionará para otro tipo de programa, aunque se esté ejecutando en el ordenador del mismo usuario.

4.9.12 Comprobar el estado del teclado

La función *TerminalInfoInteger* permite conocer el estado de las teclas de control, también llamadas virtuales. Se trata, en concreto, de *Ctrl*, *Alt*, *Shift*, *Enter*, *Ins*, *Del*, *Esc*, flechas, etc. Se llaman virtuales porque los teclados, por regla general, ofrecen varias formas de generar la misma acción de control. Por ejemplo, *Ctrl*, *Shift* y *Alt* se duplican a la izquierda y a la derecha de la barra espaciadora, mientras que el cursor puede desplazarse tanto por las teclas dedicadas como por las principales cuando se pulsa *Fn*. Por lo tanto, esta función no puede distinguir entre métodos de control a nivel físico (por ejemplo, *Shift* izquierdo y derecho).

La API define constantes para las siguientes claves:

Identificador	Descripción
TERMINAL_KEYSTATE_LEFT	Flecha izquierda
TERMINAL_KEYSTATE_UP	Flecha arriba
TERMINAL_KEYSTATE_RIGHT	Flecha derecha
TERMINAL_KEYSTATE_DOWN	Flecha abajo
TERMINAL_KEYSTATE_SHIFT	Mayús
TERMINAL_KEYSTATE_CONTROL	Ctrl
TERMINAL_KEYSTATE_MENU	Windows
TERMINAL_KEYSTATE_CAPSLOCK	BloqMayús
TERMINAL_KEYSTATE_NUMLOCK	BloqNum
TERMINAL_KEYSTATE_SCRLOCK	BloqDespl
TERMINAL_KEYSTATE_ENTER	Intro
TERMINAL_KEYSTATE_INSERT	Insert
TERMINAL_KEYSTATE_DELETE	Supr
TERMINAL_KEYSTATE_HOME	Inicio
TERMINAL_KEYSTATE_END	Fin
TERMINAL_KEYSTATE_TAB	Tab
TERMINAL_KEYSTATE_PAGEUP	Repág
TERMINAL_KEYSTATE_PAGEDOWN	Avpág
TERMINAL_KEYSTATE_ESCAPE	Esc

La función devuelve un valor entero de dos bytes que informa del estado actual de la clave solicitada mediante un par de bits.

El bit menos significativo lleva la cuenta de las pulsaciones desde la última llamada a una función. Por ejemplo, si *TerminalInfoInteger(TERMINAL_KEYSTATE_ESCAPE)* devolvió 0 en algún momento, y luego

el usuario pulsó *Escape*, entonces en la siguiente llamada, *TerminalInfoInteger(TERMINAL_KEYSTATE_ESCAPE)* devolverá 1. Si se vuelve a pulsar la tecla, el valor volverá a ser 0.

Para las teclas encargadas de conmutar los modos de entrada, como *CapsLock*, *NumLock* y *crollLock*, la posición del bit indica si el modo correspondiente está activado o desactivado.

El bit más significativo del segundo byte (0x8000) se activa si la tecla está pulsada (y no soltada) en ese momento.

Esta función no puede utilizarse para realizar un seguimiento de la pulsación de teclas alfanuméricas y funcionales. Para ello, es necesario aplicar el manejador *OnChartEvent* e interceptar los mensajes con el código CHARTEVENT_KEYDOWN en el programa. Tenga en cuenta que los eventos se generan en el gráfico y sólo están disponibles para los Asesores Expertos y los indicadores. Los programas de otros tipos (scripts y servicios) no admiten el modelo de programación de eventos.

El script *EnvKeys.mq5* incluye un bucle a través de todas las constantes TERMINAL_KEYSTATE.

```
void OnStart()
{
   for(ENUM_TERMINAL_INFO_INTEGER i = TERMINAL_KEYSTATE_TAB;
      i <= TERMINAL_KEYSTATE_SCRLOCK; ++i)
   {
      const string e = EnumToString(i);
      // skip values that are not enum elements
      if(StringFind(e, "ENUM_TERMINAL_INFO_INTEGER") == 0) continue;
      PrintFormat("%s=%4X", e, (ushort)TerminalInfoInteger(i));
   }
}
```

Puede experimentar con las pulsaciones y activar o desactivar los modos de teclado para ver cómo cambian los valores en el registro.

Por ejemplo, si las mayúsculas están desactivadas por defecto, veremos el siguiente registro:

```
TERMINAL_KEYSTATE_SCRLOCK= 0
```

Si pulsamos la tecla ScrollLock y, sin soltarla, volvemos a ejecutar el script, obtenemos el siguiente registro:

```
TERMINAL_KEYSTATE_CAPSLOCK=8001
```

Es decir, el modo ya está activado y la tecla, pulsada. Vamos a soltar la tecla, y la próxima vez el script volverá:

```
TERMINAL_KEYSTATE_SCRLOCK= 1
```

El modo ha permanecido activado, pero la tecla se soltó.

TerminalInfoInteger no es adecuado para comprobar el estado de las teclas (TERMINAL_KEYSTATE_XYZ) en los indicadores dependientes creados por la llamada *iCustom* o *IndicatorCreate*. En ellos, la función siempre devuelve 0, incluso si el indicador se añadió al gráfico utilizando *ChartIndicatorAdd*.

Además, la función no funciona cuando el gráfico del programa MQL no está activo (el usuario ha cambiado a otro). MQL5 no proporciona medios para el control permanente del teclado.

4.9.13 Comprobar el estado del programa MQL y motivo de finalización

Ya nos hemos encontrado con la función *IsStopped* en diferentes ejemplos a lo largo del libro. Se trata de una función a la que hay que llamar de vez en cuando en los casos en que el programa MQL realiza cálculos largos. Esto permite comprobar si el usuario inició el cierre del programa (es decir, si intentó eliminarlo del gráfico).

bool IsStopped() ≡ bool _StopFlag

La función devuelve *true* si el programa ha sido interrumpido por el usuario (por ejemplo, pulsando el botón Borrar en el cuadro de diálogo abierto por el comando Lista de expertos del menú contextual).

El programa dispone de 3 segundos para pausar correctamente los cálculos, guardar los resultados intermedios si es necesario y completar su trabajo. Si esto no ocurre, el programa será retirado del gráfico por la fuerza.

En lugar de la función *IsStopped*, puede comprobar el valor de la variable *_StopFlag* integrada.

El script de prueba *EnvStop.mq5* emula largos cálculos en bucle: búsqueda de números primos. Las condiciones para salir del bucle *while* se escriben utilizando la función *IsStopped*. Por lo tanto, cuando el usuario borra el script, el bucle se interrumpe de la forma habitual y el registro muestra las estadísticas del registro de números primos encontrados (el script también podría guardar los números en un archivo).

```
bool isPrime(int n)
{
   if(n < 1) return false;
   if(n <= 3) return true;
   if(n % 2 == 0) return false;
   const int p = (int)sqrt(n);
   int i = 3;
   for( ; i <= p; i += 2)
   {
      if(n % i == 0) return false;
   }

   return true;
}

void OnStart()
{
   int count = 0;
   int candidate = 1;

   while(!IsStopped()) // try to replace it with while(true)
   {
      // emulate long calculations
      if(isPrime(candidate))
      {
         Comment("Count:", ++count, ", Prime:", candidate);
      }
      ++candidate;
      Sleep(10);
   }
   Comment("");
   Print("Total found:", count);
}
```

Si sustituimos la condición de bucle por *true* (bucle infinito), el script dejará de responder a la petición de parada del usuario y se descargará del gráfico forzosamente. Como resultado, veremos el error «Terminación anormal» en el registro, y el comentario en la esquina superior izquierda de la ventana permanece sin limpiar. Así, todas las instrucciones que en este ejemplo simbolizan guardar datos y borrar recursos ocupados (y esto podría ser, por ejemplo, borrar sus propios objetos gráficos de la ventana) son ignoradas.

Una vez enviada una solicitud de parada al programa (y el valor *_StopFlag* sea igual a *true*), se puede averiguar el motivo de la finalización mediante la función *UninitializeReason*.

Lamentablemente, esta función sólo está disponible para los Asesores Expertos y los indicadores.

int UninitializeReason() ≡ int _UninitReason

La función devuelve uno de los códigos predefinidos que describen los motivos de la desinicialización.

Constante	Valor	Descripción
REASON_PROGRAM	0	*ExpertRemove* : función sólo disponible en Asesores Expertos y scripts a la que se llama
REASON_REMOVE	1	Programa eliminado del gráfico
REASON_RECOMPILE	2	Programa recompilado
REASON_CHARTCHANGE	3	Símbolo del gráfico o período cambiado
REASON_CHARTCLOSE	4	Gráfico cerrado
REASON_PARAMETERS	5	Parámetros de entrada del programa modificados
REASON_ACCOUNT	6	Se conecta otra cuenta o se produce una reconexión al servidor de trading
REASON_TEMPLATE	7	Otra plantilla de gráfico aplicada
REASON_INITFAILED	8	*OnInit* devuelve una bandera de error
REASON_CLOSE	9	Terminal cerrado

En lugar de una función, puede acceder a la variable global integrada *_UninitReason*.

El código de motivo de desinicialización también se pasa como parámetro a la función *OnDeinit* de manejo de eventos.

Más tarde, al estudiar Funciones de inicio y parada del programa, veremos un indicador (*Indicators/MQL5Book/p5/LifeCycle.mq5*) y un Asesor Experto (*Experts/MQL5Book/p5/LifeCycle.mq5*) que registran los motivos de desinicialización y permiten explorar el comportamiento de los programas en función de las acciones del usuario.

4.9.14 Cierre programático del terminal y establecimiento de un código de retorno

La API de MQL5 contiene varias funciones no sólo para leer, sino también para modificar el entorno del programa. Uno de los más radicales es *TerminalClose*. Mediante esta función, un programa MQL puede cerrar el terminal (¡sin confirmación del usuario!).

bool TerminalClose(int retcode)

La función tiene un parámetro *retcode* que es el código devuelto por el proceso terminal64.exe al sistema operativo Windows. Estos códigos pueden analizarse en archivos por lotes (*.bat y *.cmd), así como en «shell scripts» (Windows Script Host (WSH), que admite VBScript y JScript, o Windows PowerShell (WPS), con archivos .ps*) y otras herramientas de automatización (por ejemplo, el programador integrado de Windows, el subsistema de soporte de Linux en Windows con archivos *.sh, etc.).

La función no detiene inmediatamente el terminal, sino que envía una orden de terminación al terminal.

Si el resultado de la llamada es *true*, significa que el comando ha sido «aceptado para su consideración» con éxito, y el terminal intentará cerrarse lo más rápido posible, pero correctamente

(generando una notificación y deteniendo otros programas MQL en ejecución). En el código de llamada, por supuesto, también se deben hacer todos los preparativos para la terminación inmediata del trabajo (en concreto, se deben cerrar todos los archivos abiertos previamente), y después de la llamada a la función, se debe devolver el control al terminal.

Otra función asociada al código de retorno del proceso es *SetReturnError*. Permite preasignar este código sin enviar un comando de cierre inmediato.

void SetReturnError(int retcode)

La función establece el código que el proceso de terminal devolverá al sistema Windows después de cerrarse.

Tenga en cuenta que no es necesario cerrar el terminal a la fuerza mediante la función *TerminalClose*. El cierre regular del terminal por parte del usuario también se producirá con el código especificado. Además, este código entrará en el sistema si el terminal se cierra debido a un error crítico inesperado.

Si la función *SetReturnError* ha sido llamada repetidamente y/o desde diferentes programas MQL, el terminal devolverá el último código establecido.

Vamos a probar estas funciones utilizando el script *EnvClose.mq5*.

```
#property script_show_inputs

input int ReturnCode = 0;
input bool CloseTerminalNow = false;

void OnStart()
{
   if(CloseTerminalNow)
   {
      TerminalClose(ReturnCode);
   }
   else
   {
      SetReturnError(ReturnCode);
   }
}
```

Para probarlo en acción, también necesitamos el archivo *envrun.bat* (ubicado en la carpeta *MQL5/Files/MQL5Book/*).

```
terminal64.exe
@echo Exit code: %ERRORLEVEL%
```

De hecho, sólo lanza el terminal, y tras su finalización muestra el código resultante en la consola. El archivo debe colocarse en la carpeta del terminal (o la instancia actual de MetaTrader 5 de entre varias instaladas en el sistema debe registrarse en la variable de sistema PATH).

Por ejemplo, si iniciamos el terminal utilizando el archivo bat, y ejecutamos el script *EnvClose.mq5*, por ejemplo, con los parámetros *ReturnCode=100* y *CloseTerminalNow=true*, veremos algo así en la consola:

```
Microsoft Windows [Version 10.0.19570.1000]
(c) 2020 Microsoft Corporation. All rights reserved.
C:\Program Files\MT5East>envrun
C:\Program Files\MT5East>terminal64.exe
Exit code: 100
C:\Program Files\MT5East>
```

Como recordatorio, MetaTrader 5 admite varias opciones cuando se ejecuta desde la línea de comandos (ver detalles en la sección de documentación Ejecución de la plataforma de trading). Así, es posible organizar, por ejemplo, pruebas por lotes de varios Asesores Expertos o configuraciones, así como el cambio secuencial entre miles de cuentas supervisadas, lo que sería poco realista de lograr con el funcionamiento paralelo constante de tantas instancias en un ordenador.

4.9.15 Tratamiento de errores en tiempo de ejecución

Cualquier programa escrito correctamente para compilar sin errores no es, a pesar de todo, inmune a los errores de ejecución. Dichos errores pueden producirse tanto por un descuido del desarrollador como por circunstancias imprevistas surgidas en el entorno del software (como perder la conexión a Internet, quedarse sin memoria, etc.). Pero no menos probable es la situación en la que el error se produce debido a una aplicación incorrecta del programa. En todos estos casos, el programa debe ser capaz de analizar la esencia del problema y procesarlo adecuadamente.

Cada sentencia MQL5 es una fuente potencial de errores en tiempo de ejecución. Si se produce un error de este tipo, el terminal guarda un código descriptivo en la variable especial *_LastError*. Asegúrese de analizar el código inmediatamente después de cada sentencia, ya que posibles errores en sentencias posteriores pueden sobrescribir este valor.

Tenga en cuenta que hay una serie de errores críticos que abortarán inmediatamente la ejecución del programa cuando se produzcan:

- División por cero
- Índice fuera de rango
- Puntero de objeto incorrecto

Para obtener una lista completa de los códigos de error y su significado, consulte la documentación.

En la sección Abrir y cerrar archivos hemos abordado ya el problema del diagnóstico de errores como parte de la escritura de una macro PRTF útil. Allí, en particular, hemos visto un archivo de encabezado auxiliar *MQL5/Include/MQL5Book/MqlError.mqh*, en el que la enumeración MQL_ERROR permite convertir fácilmente el código de error numérico en un nombre utilizando *EnumToString*.

```
enum MQL_ERROR
{
   SUCCESS = 0,
   INTERNAL_ERROR = 4001,
   WRONG_INTERNAL_PARAMETER = 4002,
   INVALID_PARAMETER = 4003,
   NOT_ENOUGH_MEMORY = 4004,
   ...
   // start of area for errors defined by the programmer (see next section)
   USER_ERROR_FIRST = 65536,
};
#define E2S(X) EnumToString((MQL_ERROR)(X))
```

Aquí, como parámetro *X* de la macro *E2S*, deberíamos tener la variable *_LastError* o su función equivalente *GetLastError*.

int GetLastError() ≡ int _LastError

La función devuelve el código del último error que se ha producido en las sentencias del programa MQL. Inicialmente, mientras no haya errores, el valor es 0. La diferencia entre leer *_LastError* y llamar a la función *GetLastError* es puramente sintáctica (elija la opción adecuada según el estilo preferido).

Debe tenerse en cuenta que la ejecución regular de sentencias sin errores no restablece el código de error, y tampoco lo hará llamar a *GetLastError*.

Por lo tanto, si hay una secuencia de acciones en la que sólo una activará una bandera de error, esta bandera será devuelta por la función para las acciones posteriores (con éxito). Por ejemplo:

```
// _LastError = 0 by default
action1; // ok, _LastError doe not change
action2; // error, _LastError = X
action3; // ok, _LastError does not change, i.e. is still equal to X
action4; // another error, _LastError = Y
action5; // ok, _LastError does not change, that is, it is still equal to Y
action6; // ok, _LastError does not change, that is, it is still equal to Y
```

Este comportamiento dificultaría la localización de la zona problemática. Para evitarlo, existe una función *ResetLastError* independiente que restablece la variable *_LastError* a 0.

void ResetLastError()

La función pone a cero el valor de la variable integrada *_LastError*.

Se recomienda llamar a la función antes de cualquier acción que pueda dar lugar a un error y después de la cual usted va a analizar los errores por medio de *GetLastError*.

Un buen ejemplo de uso de ambas funciones es la macro PRTF ya mencionada (archivo PRTF.mqh). Su código se muestra a continuación:

```
#include <MQL5Book/MqlError.mqh>

#define PRTF(A) ResultPrint(#A, (A))

template<typename T>
T ResultPrint(const string s, const T retval = NULL)
{
   const int snapshot = _LastError; // recording _LastError at input
   const string err = E2S(snapshot) + "(" + (string)snapshot + ")";
   Print(s, "=", retval, " / ", (snapshot == 0 ? "ok" : err));
   ResetLastError(); // clear the error flag for the next calls
   return retval;
}
```

El propósito de la macro y de la función *ResultPrint* incluida en ella es registrar el valor pasado, que es el código de error actual, y borrar inmediatamente el código de error. Así, la aplicación sucesiva de PRTF sobre una serie de sentencias garantiza siempre que el error (o la indicación de éxito) que se imprime en el registro corresponde a la última sentencia con la que se obtuvo el valor del parámetro *retval*.

Necesitamos guardar *_LastError* en la variable local intermedia *snapshot* porque *_LastError* puede cambiar su valor en casi cualquier punto de la evaluación de una expresión si falla alguna operación. En este ejemplo concreto, la macro E2S utiliza la función *EnumToString*, que puede generar su propio código de error si se pasa como argumento un valor que no está en la enumeración. A continuación, en las subsiguientes partes de la misma expresión, al formar una cadena, no veremos el error inicial sino el planteado.

Puede haber varios lugares en cualquier sentencia en los que *_LastError* cambie de repente. A este respecto, conviene registrar el código de error inmediatamente después de la acción deseada.

4.9.16 Errores definidos por el usuario

El desarrollador puede utilizar la variable integrada *_LastError* para sus propios fines aplicados. Esto se facilita mediante la función *SetUserError*.

void SetUserError(ushort user_error)

La función establece la variable integrada *_LastError* al valor *ERR_USER_ERROR_FIRST* + *user_error*, donde ERR_USER_ERROR_FIRST es 65536. Todos los códigos por debajo de este valor se reservan para errores del sistema.

Utilizando este mecanismo, puede eludir parcialmente la limitación de MQL5 asociada al hecho de que no se admiten excepciones en el lenguaje.

A menudo, las funciones utilizan el valor de retorno como señal de error. Sin embargo, hay algoritmos en los que la función debe devolver un valor del tipo de la aplicación. Hablemos de *double*. Si la función tiene un rango de definición de menos a más infinito, cualquier valor que elijamos para indicar un error (por ejemplo, 0) será indistinguible del resultado real del cálculo. En el caso de *double*, por supuesto, existe la opción de devolver un valor NaN (Not a Number, véase la sección Comprobación de la normalidad de los números reales). Pero, ¿y si la función devuelve una estructura o un objeto de clase? Una de las posibles soluciones es devolver el resultado a través de un parámetro por referencia o puntero, pero tal forma imposibilita el uso de funciones como operandos de expresiones.

En el contexto de las clases, consideremos las funciones especiales denominadas «constructores». Estas funciones devuelven una nueva instancia del objeto. Sin embargo, a veces las circunstancias impiden construir el objeto completo, y entonces el código de llamada parece obtener el objeto pero no debe utilizarlo. Es bueno si la clase puede proporcionar un método adicional que le permita comprobar la utilidad del objeto. Pero como enfoque alternativo uniforme (por ejemplo, que abarque todas las clases), podemos utilizar *SetUserError*.

En la sección Sobrecarga de operadores encontramos la clase *Matrix*. La complementaremos con métodos para calcular el determinante y la matriz inversa, y luego la utilizaremos para demostrar los errores del usuario (véase el archivo *Matrix.mqh*). Se han definido operadores sobrecargados para matrices, lo que permite combinarlos en cadenas de operadores en una única expresión, y por tanto sería inconveniente implementar una comprobación de posibles errores en ella.

> Nuestra clase *Matrix* es una implementación alternativa personalizada para el tipo de objeto integrado recientemente añadido en MQL5 *matriz*.

Comenzamos validando los parámetros de entrada en los constructores de la clase principal *Matrix*. Si alguien intenta crear una matriz de tamaño cero, vamos a establecer un error personalizado ERR_USER_MATRIX_EMPTY (uno de los varios proporcionados).

```
enum ENUM_ERR_USER_MATRIX
{
   ERR_USER_MATRIX_OK = 0,
   ERR_USER_MATRIX_EMPTY =  1,
   ERR_USER_MATRIX_SINGULAR = 2,
   ERR_USER_MATRIX_NOT_SQUARE = 3
};

class Matrix
{
   ...
public:
   Matrix(const int r, const int c) : rows(r), columns(c)
   {
      if(rows <= 0 || columns <= 0)
      {
         SetUserError(ERR_USER_MATRIX_EMPTY);
      }
      else
      {
         ArrayResize(m, rows * columns);
         ArrayInitialize(m, 0);
      }
   }
```

Estas nuevas operaciones sólo están definidas para matrices cuadradas, así que vamos a crear una clase derivada con una restricción de tamaño apropiada.

```
class MatrixSquare : public Matrix
{
public:
   MatrixSquare(const int n, const int _ = -1) : Matrix(n, n)
   {
      if(_ != -1 && _ != n)
      {
         SetUserError(ERR_USER_MATRIX_NOT_SQUARE);
      }
   }
   ...
```

El segundo parámetro en el constructor debería estar ausente (se asume que es igual al primero), pero lo necesitamos porque la clase *Matrix* tiene un método de transposición de plantilla en el que todos los tipos de T deben admitir un constructor con dos parámetros enteros.

```
class Matrix
{
   ...
   template<typename T>
   T transpose() const
   {
      T result(columns, rows);
      for(int i = 0; i < rows; ++i)
      {
         for(int j = 0; j < columns; ++j)
         {
            result[j][i] = this[i][(uint)j];
         }
      }
      return result;
   }
```

Debido al hecho de que hay dos parámetros en el constructor *MatrixSquare*, también tenemos que comprobar su igualdad obligatoria. Si no son iguales, activamos el error ERR_USER_MATRIX_NOT_SQUARE.

Finalmente, durante el cálculo de la matriz inversa, podemos encontrar que la matriz es degenerada (el determinante es 0). El error ERR_USER_MATRIX_SINGULAR está reservado para este caso.

```
class MatrixSquare : public Matrix
{
public:
   ...
   MatrixSquare inverse() const
   {
      MatrixSquare result(rows);
      const double d = determinant();
      if(fabs(d) > DBL_EPSILON)
      {
         result = complement().transpose<MatrixSquare>() * (1 / d);
      }
      else
      {
         SetUserError(ERR_USER_MATRIX_SINGULAR);
      }
      return result;
   }

   MatrixSquare operator!() const
   {
      return inverse();
   }
   ...
```

Para la salida visual de errores, se ha añadido un método estático al registro, lo que devuelve la enumeración ENUM_ERR_USER_MATRIX, que es fácil de pasar a *EnumToString*:

```
   static ENUM_ERR_USER_MATRIX lastError()
   {
      if(_LastError >= ERR_USER_ERROR_FIRST)
      {
         return (ENUM_ERR_USER_MATRIX)(_LastError - ERR_USER_ERROR_FIRST);
      }
      return (ENUM_ERR_USER_MATRIX)_LastError;
   }
```

El código completo de todos los métodos se encuentra en el archivo adjunto.

Comprobaremos los códigos de error de la aplicación en el script de prueba *EnvError.mq5*.

Primero, asegurémonos de que la clase funciona: invierta la matriz y compruebe que el producto de la matriz original y la invertida es igual a la matriz identidad.

```
void OnStart()
{
   Print("Test matrix inversion (should pass)");
   double a[9] =
   {
      1,  2,  3,
      4,  5,  6,
      7,  8,  0,
   };

   ResetLastError();
   Matrix SquaremA(a);   // assign data to the original matrix
   Print("Input");
   mA.print();
   MatrixSquare mAinv(3);
   mainv = !mA;          // invert and store in another matrix
   Print("Result");
   mAinv.print();

   Print("Check inverted by multiplication");
   Matrix Squaretest(3); // multiply the first by the second
   test = mA * mAinv;
   test.print();         // get identity matrix
   Print(EnumToString(Matrix::lastError())); // ok
   ...
```

Este fragmento de código genera las siguientes entradas de registro:

```
Test matrix inversion (should pass)
Input
1.00000 2.00000 3.00000
4.00000 5.00000 6.00000
7.00000 8.00000 0.00000
Result
-1.77778  0.88889 -0.11111
 1.55556 -0.77778  0.22222
-0.11111  0.22222 -0.11111
Check inverted by multiplication
 1.00000 +0.00000  0.00000
 -0.00000   1.00000  +0.00000
0.00000 0.00000 1.00000
ERR_USER_MATRIX_OK
```

Nótese que en la matriz identidad, debido a errores de coma flotante, algunos elementos cero son en realidad valores muy pequeños cercanos a cero, y por tanto tienen signo.

A continuación, veamos cómo maneja el algoritmo la matriz degenerada.

```
   Print("Test matrix inversion (should fail)");
   double b[9] =
   {
     -22, -7, 17,
     -21, 15,  9,
     -34,-31, 33
   };

   MatrixSquare mB(b);
   Print("Input");
   mB.print();
   ResetLastError();
   Print("Result");
   (!mB).print();
   Print(EnumToString(Matrix::lastError())); // singular
   ...
```

Los resultados se presentan a continuación:

```
Test matrix inversion (should fail)
Input
-22.00000  -7.00000  17.00000
-21.00000  15.00000   9.00000
-34.00000 -31.00000  33.00000
Result
0.0 0.0 0.0
0.0 0.0 0.0
0.0 0.0 0.0
ERR_USER_MATRIX_SINGULAR
```

En este caso, simplemente mostramos una descripción del error. Pero en un programa real, debería ser posible elegir una opción de continuación, dependiendo de la naturaleza del problema.

Por último, simularemos situaciones para los dos errores aplicados restantes.

```
   Print("Empty matrix creation");
   MatrixSquare m0(0);
   Print(EnumToString(Matrix::lastError()));

   Print("'Rectangular' square matrix creation");
   MatrixSquare r12(1, 2);
   Print(EnumToString(Matrix::lastError()));
}
```

Aquí describimos una matriz vacía y una matriz supuestamente cuadrada pero con tamaños diferentes.

```
Empty matrix creation
ERR_USER_MATRIX_EMPTY
'Rectangular' square matrix creation
ERR_USER_MATRIX_NOT_SQUARE
```

En estos casos no podemos evitar crear un objeto porque el compilador lo hace automáticamente.

Por supuesto, esta prueba viola claramente los contratos (las especificaciones de datos y acciones que las clases y los métodos «consideran» válidos). Sin embargo, en la práctica, los argumentos se obtienen a menudo de otras partes del código, en el curso del procesamiento de grandes datos de «terceros», y detectar las desviaciones con respecto a las expectativas no es tan fácil.

La capacidad de un programa para «digerir» datos incorrectos sin consecuencias fatales es el indicador más importante de su calidad, junto con la producción de resultados correctos para datos de entrada correctos.

4.9.17 Gestión de depuración

El depurador integrado en MetaEditor permite establecer puntos de interrupción en el código fuente, que son las líneas en las que se debe suspender la ejecución del programa. A veces este sistema falla, es decir, la pausa no funciona, y entonces puede utilizar la función *DebugBreak* que fuerza explícitamente la parada.

void DebugBreak()

La llamada a la función pone en pausa el programa y activa la ventana del editor en el modo de depuración, con todas las herramientas para ver las variables y la pila de llamadas y para continuar la ejecución paso a paso.

La ejecución del programa se interrumpe sólo si el programa se lanza desde el editor en el modo de depuración (mediante los comandos *Depurar -> Empezar con datos reales* o *Empezar con datos históricos*). En todos los demás modos, incluido el inicio normal (en el terminal) y la creación de perfiles, la función no tiene ningún efecto.

4.9.18 Variables predefinidas

Cada programa MQL tiene un determinado conjunto general de variables globales proporcionadas por el terminal: ya hemos abordado la mayoría de ellas en las secciones anteriores, y a continuación se muestra una tabla resumen. Casi todas las variables son de sólo lectura. La excepción es la variable *_LastError*, que puede restablecerse mediante la función *ResetLastError*.

Variable	Valor
_LastError	Valor del último error, análogo de la función *GetLastError*
_StopFlag	Bandera de parada del programa, análoga de la función *IsStopped*
_UninitReason	Código de motivo de desinicialización del programa, análogo de la función *UninitializeReason*
_RandomSeed	Estado interno actual del generador de enteros pseudoaleatorios
_IsX64	Bandera de un terminal de 64 bits, análoga de *TerminalInfoInteger* para la propiedad TERMINAL_X64

Además, para los programas MQL que se ejecutan en el contexto gráfico de un gráfico, como Asesores Expertos, scripts e indicadores, el lenguaje proporciona variables predefinidas con propiedades de gráfico (tampoco pueden modificarse desde el programa).

Variable	Valor
_Symbol	Nombre del símbolo del gráfico actual, análogo de la función *Symbol*
_Period	Este es el marco temporal del gráfico actual, análogo de la función *Period*
_Digits	El número de decimales en el precio del símbolo del gráfico actual, análogo de la función *Digits*
_Point	Tamaño del punto en los precios del símbolo actual (en la divisa de cotización), análogo de la función *Point*
_AppliedTo	Tipo de datos sobre los que se calcula el indicador (sólo para indicadores)

4.9.19 Constantes predefinidas del lenguaje MQL5

En esta sección se describen todas las constantes definidas por el entorno de ejecución para cualquier programa. Ya hemos visto algunos de ellos en secciones anteriores. Algunas constantes están relacionadas con aspectos de programación MQL5 aplicados, que se presentarán en capítulos posteriores.

Constante	Descripción	Valor
CHARTS_MAX	El número máximo posible de gráficos abiertos simultáneamente	100
clrNONE	Sin color	-1 (0xFFFFFFFF)
EMPTY_VALUE	Valor vacío en el búfer del indicador	DBL_MAX
INVALID_HANDLE	Manejador no válido	-1
NULL	Cualquier tipo null	0
WHOLE_ARRAY	El número de elementos hasta el final del array, es decir, se procesará todo el array.	-1
WRONG_VALUE	Una constante se puede convertir implícitamente a cualquier tipo de enumeración	-1

Como se muestra en el capítulo Archivos, se puede utilizar la constante INVALID_HANDLE para validar los descriptores de archivo.

La constante WHOLE_ARRAY está pensada para funciones que trabajan con arrays que requieren especificar el número de elementos de los arrays procesados: Si es necesario procesar todos los valores del array desde la posición especificada hasta el final, especifique el valor WHOLE_ARRAY.

La constante EMPTY_VALUE suele asignarse a aquellos elementos en búferes indicadores, que no deben dibujarse en el gráfico. En otras palabras: esta constante significa un valor vacío. Más adelante describiremos cómo puede sustituirse para un búfer indicador específico por otro valor, por ejemplo, 0.

La constante WRONG_VALUE está pensada para aquellos casos en los que se requiere designar un valor de enumeración incorrecto.

Además, dos constantes tienen valores diferentes según el método de compilación.

Constante	Descripción
IS_DEBUG_MODE	Un atributo de ejecutar un programa mq5 en el modo de depuración: es distinto de cero en el modo de depuración y 0 en caso contrario.
IS_PROFILE_MODE	Un atributo de ejecutar un programa mq5 en el modo de perfilaje: es distinto de cero en el modo de perfilaje y 0 en caso contrario

La constante IS_PROFILE_MODE permite cambiar el funcionamiento del programa para la correcta recopilación de información en el modo de perfilaje. La creación de perfiles permite medir el tiempo de ejecución de fragmentos individuales del programa (funciones y líneas individuales).

El compilador establece el valor de la constante IS_PROFILE_MODE durante la compilación. Normalmente, se fija en 0. Cuando el programa se lanza en un modo de perfilaje, se realiza una compilación especial y, en este caso, se utiliza un valor distinto de cero en lugar de IS_PROFILE_MODE.

La constante IS_DEBUG_MODE funciona de forma similar: es igual a 0 como resultado de la compilación nativa y es mayor que 0 tras la compilación de depuración. Es útil en los casos en los que es necesario modificar ligeramente el funcionamiento del programa MQL con fines de verificación; por ejemplo, para enviar información adicional al registro o para crear objetos gráficos auxiliares en el gráfico.

El preprocesador define las constantes _DEBUG y _RELEASE que tienen un significado similar (véase Constantes predefinidas del preprocesador).

Se puede obtener información más detallada sobre el modo de funcionamiento del programa en tiempo de ejecución mediante la función *MQLInfoInteger* (véase Modos de funcionamiento del terminal y del programa). En concreto, la versión de depuración de un programa puede ejecutarse sin depurador.

4.10 Matrices y vectores

El lenguaje MQL5 proporciona tipos de datos de objeto especiales: matrices y vectores. Estos tipos pueden utilizarse para resolver una gran variedad de problemas matemáticos: proporcionan métodos para escribir código conciso y comprensible cercano a la notación matemática de ecuaciones lineales o diferenciales.

Todos los lenguajes de programación admiten el concepto de array, que es un conjunto de múltiples elementos. La mayoría de los algoritmos, sobre todo en el trading algorítmico, se construyen sobre la base de estructuras o arrays de tipo numérico (*int*, *double*). Se puede acceder a los elementos del array por índice, lo que permite realizar operaciones dentro de bucles. Como sabemos, los arrays pueden tener una, dos o más dimensiones.

Las tareas relativamente sencillas de almacenamiento y procesamiento de datos suelen realizarse mediante arrays, pero cuando se trata de problemas matemáticos complejos, el gran número de bucles anidados dificulta el trabajo con los arrays, tanto en términos de programación como de lectura de código. Incluso las operaciones más sencillas de álgebra lineal requieren mucho código y una buena comprensión de las matemáticas. Esta tarea puede simplificarse mediante el paradigma funcional de la

programación, que se plasma en funciones de métodos matriciales y vectoriales. Estas acciones realizan muchas acciones rutinarias «entre bastidores».

Las tecnologías modernas, como el aprendizaje automático, las redes neuronales y los gráficos 3D, hacen un amplio uso de la resolución de problemas con álgebra lineal, que utiliza operaciones con vectores y matrices. Los nuevos tipos de datos se han añadido a MQL5 para trabajar de forma rápida y cómoda con este tipo de objetos.

> En el momento de escribir este libro, el conjunto de funciones para trabajar con matrices y vectores se estaba desarrollando activamente, por lo que es posible que aquí no se mencionen muchas novedades interesantes. Siga las notas de la versión y la sección de artículos del sitio *mql5.com*.

En este capítulo vamos a ver una breve descripción. Para obtener más detalles sobre matrices y vectores, consulte la sección de ayuda correspondiente Métodos matriciales y vectoriales.

También se supone que el lector está familiarizado con la teoría del álgebra lineal. En caso necesario, siempre puede recurrir a bibliografía de referencia y manuales en Internet.

4.10.1 Tipos de matrices y vectores

Un vector es un array unidimensional de tipo real o complejo, mientras que una matriz es un array bidimensional de tipo real o complejo. Así, la lista de tipos numéricos válidos para los elementos de estos objetos incluye *double* (considerado el tipo por defecto), *float* y *complex*.

Desde el punto de vista del álgebra lineal (¡pero no del compilador!), un número primo es también un vector mínimo, y un vector, a su vez, puede considerarse un caso especial de matriz.

El vector, dependiendo del tipo de elementos, se describe utilizando una de las palabras clave *vector* (con o sin sufijo):

- *vector* es un vector con elementos de tipo *double*
- *vectorf* es un vector con elementos de tipo *float*
- *vectorc* es un vector con elementos de tipo *complex*

Aunque los vectores pueden ser verticales y horizontales, MQL5 no hace tal división. La orientación requerida del vector viene determinada (implícitamente) por la posición del vector en la expresión.

Se definen las siguientes operaciones en vectores: suma y multiplicación, así como el método Norm (con la correspondiente *norma*), que obtiene el módulo o longitud del vector.

Puede pensar en una matriz como en un array, donde el primer índice es el número de fila y el segundo índice, el número de columna. No obstante, la numeración de filas y columnas, a diferencia del álgebra lineal, empieza por cero, como en los arrays.

$$A=\begin{pmatrix} a_{11}\, a_{12} \ldots a_{1n} \\ a_{21}\, a_{22} \ldots a_{2n} \\ \ldots\ldots\ldots\ldots\ldots \\ a_{m1}\, a_{m2} \ldots a_{mn} \end{pmatrix}$$

Las dos dimensiones de las matrices también se denominan ejes y se numeran del siguiente modo: 0 para el eje horizontal (a lo largo de las filas) y 1 para el eje vertical (a lo largo de las columnas). Los

números de eje se utilizan en muchas funciones matriciales. En concreto, cuando hablamos de dividir una matriz en partes, dividir horizontalmente significa cortar entre filas, y dividir verticalmente significa cortar entre columnas.

Dependiendo del tipo de elementos, la matriz se describe utilizando una de las palabras clave *matrix* (con o sin sufijo):

- *matrix* es una matriz con elementos de tipo *double*
- *matrixf* es una matriz con elementos de tipo *float*
- *matrixc* es una matriz con elementos de tipo *complex*

Para su aplicación en funciones de plantilla, puede utilizar la notación *matrix<double>* , *matrix<float>* , *matrix<complex>* , *vector<double>* , *vector<float>* , *vector<complex>* en lugar de los tipos correspondientes.

```
vectorf v_f1 = {0, 1, 2, 3,};
vector<float> v_f2 = v_f1;
matrix m = {{0, 1}, {2, 3}};

void OnStart()
{
   Print(v_f2);
   Print(m);
}
```

Cuando se registran, las matrices y los vectores se imprimen como secuencias de números separados por comas y encerrados entre corchetes.

```
[0,1,2,3]
[[0,1]
 [2,3]]
```

Para las matrices se definen las siguientes operaciones algebraicas:

- Suma de matrices del mismo tamaño
- Multiplicación de matrices de un tamaño adecuado, cuando el número de columnas de la primera matriz debe ser igual al número de filas de la segunda matriz.
- Multiplicación de una matriz por un vector columna y multiplicación de un vector fila por una matriz según las reglas de multiplicación de matrices (un vector es, en este sentido, un caso especial de matriz).
- Multiplicación de una matriz por un número

Además, los tipos *matrix* y *vector* tienen métodos integrados que corresponden a análogos de la biblioteca NumPy (un paquete popular para el aprendizaje automático en Python), por lo que puede obtener más pistas en la documentación y en los ejemplos de la biblioteca. Encontrará una lista completa de métodos en la correspondiente sección de ayuda de MQL5.

Por desgracia, MQL5 no permite convertir matrices y vectores de un tipo a otro (por ejemplo, de *double* a *float*). Además, el compilador no trata automáticamente a un vector como una matriz (con una columna o fila) en expresiones en las que se espera una matriz. Esto significa que el concepto de herencia (característico de la programación orientada a objetos) entre matrices y vectores no existe, a pesar de la aparente relación entre estas estructuras.

4.10.2 Creación e inicialización de matrices y vectores

Hay varias formas de declarar e inicializar matrices y vectores, que pueden dividirse en varias categorías según su finalidad.

- Declaración sin especificar el tamaño
- Declaración con el tamaño especificado
- Declaración con inicialización
- Métodos de creación estática
- Métodos no estáticos de (re)configuración e inicialización

El método de creación más sencillo es una declaración sin especificar el tamaño, es decir, sin asignar memoria para los datos. Para ello, basta con especificar el tipo y el nombre de la variable:

```
matrix         matrix_a;   // matrix of type double
matrix<double> matrix_a1;  // double type matrix inside function or class templates
matrix<float>  matrix_a2;  // float matrix
vector         vector_v;   // vector of type double
vector<double> vector_v1;  // another notation of a double-type vector creation
vector<float>  vector_v2;  // vector of type float
```

A continuación, puede cambiar el tamaño de los objetos creados y rellenarlos con los valores deseados. También pueden utilizarse en métodos matriciales y vectoriales integrados para obtener los resultados de los cálculos. Todos estos métodos se analizarán por grupos en las secciones de este capítulo.

Puede declarar una matriz o un vector con un tamaño especificado. Esto asignará memoria pero sin ninguna inicialización. Para ello, después del nombre de la variable entre paréntesis, especifique el tamaño o tamaños (para una matriz, el primero es el número de filas y el segundo, el de columnas):

```
matrix         matrix_a(128, 128);      // you can specify as parameters
matrix<double> matrix_a1(nRows, nCols); // both constants and variables
matrix<float>  matrix_a2(nRows, 1);     // analog of column vector
vector         vector_v(256);
vector<double> vector_v1(nSize);
vector<float>  vector_v2(nSize +16);    // expression as a parameter
```

La tercera forma de crear objetos es mediante una declaración con inicialización. Los tamaños de matrices y vectores en este caso vienen determinados por la secuencia de inicialización indicada entre llaves:

```
matrix         matrix_a = {{0.1, 0.2, 0.3}, {0.4, 0.5, 0.6}};
matrix<double> matrix_a1 =matrix_a;     // must be matrices of the same type
matrix<float>  matrix_a2 = {{1, 2}, {3, 4}};
vector         vector_v = {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5};
vector<double> vector_v1 = {1, 5, 2.4, 3.3};
vector<float>  vector_v2 =vector_v1;    // must be vectors of the same type
```

También existen métodos estáticos para crear matrices y vectores de un tamaño especificado con inicialización de una determinada manera (específicamente para una u otra forma canónica). Todos ellos se enumeran a continuación y tienen prototipos similares (los vectores sólo se diferencian de las matrices en la ausencia de una segunda dimensión).

static matrix<T> matrix<T>::Eye∫Tri(const ulong rows, const ulong cols, const int diagonal = 0);
static matrix<T> matrix<T>::Identity∫Ones∫Zeros(const ulong rows, const ulong cols);
static matrix<T> matrix<T>::Full(const ulong rows, const ulong cols, const double value);

- *Eye* construye una matriz con unos en la diagonal especificada y ceros en el resto.
- *Tri* construye una matriz con unos en y por debajo de la diagonal especificada y ceros en el resto.
- *Identity* construye una matriz de identidad del tamaño especificado.
- *Ones* construye una matriz (o vector) llena de unos.
- *Zeros* construye una matriz (o vector) llena de ceros.
- *Full* construye una matriz (o vector) con el valor dado en todos los elementos.

Si es necesario, puede convertir cualquier matriz existente en una matriz de identidad, para lo cual debe aplicar un método no estático *Identity* (sin parámetros).

Demostremos los métodos en acción:

```
matrix         matrix_a = matrix::Eye(4, 5, 1);
matrix<double> matrix_a1 = matrix::Full(3, 4, M_PI);
matrixf        matrix_a2 = matrixf::Identity(5, 5);
matrixf<float> matrix_a3 = matrixf::Ones(5, 5);
matrix         matrix_a4 = matrix::Tri(4, 5, -1);
vector         vector_v = vector::Ones(256);
vectorf        vector_v1 = vector<float>::Zeros(16);
vector<float>  vector_v2 = vectorf::Full(128, float_value);
```

Además, existen métodos no estáticos para inicializar una matriz/vector con valores dados: *Init* y *Fill*.

void matrix<T>::Init(const ulong rows, const ulong cols, func_reference rule = NULL, ...)
void matrix<T>::Fill(const T value)

Una ventaja importante del método *Init* (que también está presente para los constructores) es la posibilidad de especificar en los parámetros una función de inicialización para rellenar los elementos de una matriz/vector según una ley determinada (véase el ejemplo siguiente).

Se puede pasar una referencia a dicha función después de los tamaños especificando su identificador sin comillas en el parámetro *rules* (no se trata de un puntero en el sentido de *typedef (*pointer)(...)* y tampoco una cadena con un nombre).

La función de inicialización debe tener una referencia al objeto que se está rellenando como primer parámetro y también puede tener parámetros adicionales: en este caso, los valores para los mismos se pasan a *Init* o a un constructor tras la referencia a la función. Si no se especifica el enlace *rule*, se creará simplemente una matriz de las dimensiones especificadas.

El método *Init* también permite cambiar la configuración de la matriz.

Veamos todo lo anterior con pequeños ejemplos.

```
matrix m(2, 2);
m.Fill(10);
Print("matrix m \n", m);
/*
  matrix m
  [[10,10]
  [10,10]]
*/
m.Init(4, 6);
Print("matrix m \n", m);
/*
  matrix m
  [[10,10,10,10,0.0078125,32.00000762939453]
  [0,0,0,0,0,0]
  [0,0,0,0,0,0]
  [0,0,0,0,0,0]]
*/
```

Aquí se utilizó el método *Init* para redimensionar una matriz ya inicializada, lo que ha dado lugar a que los nuevos elementos se rellenen con valores aleatorios.

La siguiente función rellena la matriz con números que aumentan exponencialmente:

```
template<typename T>
void MatrixSetValues(matrix<T> &m, const T initial = 1)
{
   T value = initial;
   for(ulong r = 0; r < m.Rows(); r++)
   {
      for(ulong c = 0; c < m.Cols(); c++)
      {
         m[r][c] = value;
         value *= 2;
      }
   }
}
```

A continuación, puede utilizarse para crear una matriz.

```
void OnStart()
{
   matrix M(3, 6, MatrixSetValues);
   Print("M = \n", M);
}
```

El resultado de la ejecución es:

```
M =
[[1,2,4,8,16,32]
 [64,128,256,512,1024,2048]
 [4096,8192,16384,32768,65536,131072]]
```

En este caso, los valores para el parámetro de la función de inicialización no se especificaron a continuación de su identificador en la llamada al constructor, por lo que se utilizó el valor por defecto

(1). Pero podemos, por ejemplo, pasar un valor inicial de -1 para la misma *MatrixSetValues*, lo que llenará la matriz con una fila negativa.

```
matrix M(3, 6, MatrixSetValues, -1);
```

4.10.3 Copiar matrices, vectores y arrays

La forma más sencilla y habitual de copiar matrices y vectores es mediante el operador de asignación '='.

```
matrix a = {{2, 2}, {3, 3}, {4, 4}};
matrix b = a + 2;
matrix c;
Print("matrix a \n", a);
Print("matrix b \n", b);
c.Assign(b);
Print("matrix c \n", c);
```

Este fragmento genera las siguientes entradas de registro:

```
matrix a
[[2,2]
 [3,3]
 [4,4]]
matrix b
[[4,4]
 [5,5]
 [6,6]]
matrix c
[[4,4]
 [5,5]
 [6,6]]
```

Los métodos *Copy* y *Assign* también pueden utilizarse para copiar matrices y vectores. La diferencia entre *Assign* y *Copy* es que *Assign* permite copiar no sólo matrices, sino también matrices.

bool matrix<T>::Copy(const matrix<T> &source)
bool matrix<T>::Assign(const matrix<T> &source)
bool matrix<T>::Assign(const T &array[])

También existen métodos y prototipos similares para los vectores.

A través de *Assign* es posible escribir un vector en una matriz: el resultado será una matriz de una fila.

bool matrix<T>::Assign(const vector<T> &v)

También puede asignar una matriz a un vector: se desenvolverá, es decir, todas las filas de la matriz se alinearán en una fila (equivalente a llamar al método *Flat*).

bool vector<T>::Assign(const matrix<T> &m)

En el momento de escribir este capítulo, no había ningún método en MQL5 para exportar una matriz o vector a un array, aunque existe un mecanismo para «transferir» los datos (véase el método *Swap*).

En el siguiente ejemplo se muestra cómo se copia un array de enteros *int_arr* en una matriz de tipo *double*. En este caso, la matriz resultante se ajusta automáticamente al tamaño del array copiado.

```
matrix double_matrix = matrix::Full(2, 10, 3.14);
Print("double_matrix before Assign() \n", double_matrix);
int int_arr[5][5] = {{1, 2}, {3, 4}, {5, 6}};
Print("int_arr: ");
ArrayPrint(int_arr);
double_matrix.Assign(int_arr);
Print("double_matrix after Assign(int_arr) \n", double_matrix);
```

Tenemos la siguiente salida en el registro:

```
double_matrix before Assign()
[[3.14,3.14,3.14,3.14,3.14,3.14,3.14,3.14,3.14,3.14]
 [3.14,3.14,3.14,3.14,3.14,3.14,3.14,3.14,3.14,3.14]]

int_arr:
     [,0][,1][,2][,3][,4]
[0,]   1   2   0   0   0
[1,]   3   4   0   0   0
[2,]   5   6   0   0   0
[3,]   0   0   0   0   0
[4,]   0   0   0   0   0

double_matrix after Assign(int_arr)
[[1,2,0,0,0]
 [3,4,0,0,0]
 [5,6,0,0,0]
 [0,0,0,0,0]
 [0,0,0,0,0]]
```

Así, el método *Assign* puede utilizarse para pasar de arrays a matrices con conversión automática de tamaño y tipo.

Una forma más eficaz (rápida y sin necesidad de copiar) de transferir datos entre matrices, vectores y arrays es utilizar los métodos *Swap*.

bool matrix<T>::Swap(vector<T> &vec)
bool matrix<T>::Swap(matrix<T> &vec)
bool matrix<T>::Swap(T &arr[])
bool vector<T>::Swap(vector<T> &vec)
bool vector<T>::Swap(matrix<T> &vec)
bool vector<T>::Swap(T &arr[])

Funcionan de forma similar a *ArraySwap*: los punteros internos a búferes con datos dentro de dos objetos se intercambian. Como resultado, los elementos de una matriz o vector desaparecen en el objeto de origen y aparecen en el array receptor, o viceversa: pasan del array a la matriz o vector.

El método *Swap* permite trabajar con arrays dinámicos, incluidos los multidimensionales. Se aplica una determinada condición a los tamaños constantes de las dimensiones más altas de un array multidimensional (*array[][N1][N2]...*): El producto de estas dimensiones debe ser múltiplo del tamaño de la matriz o del vector. Así, un array de [][2][3] se redistribuye en bloques de 6 elementos. Por lo tanto, es intercambiable con matrices y vectores de tamaño 6, 12, 18, etc.

4.10.4 Copiar series temporales en matrices y vectores

El método *matrix<T>::CopyRates* copia las series temporales con el historial de cotizaciones directamente en una matriz o vector. Este método funciona de forma similar a las funciones que veremos en detalle en la Parte 5, en el capítulo sobre series temporales, a saber: *CopyRates* y *funciones Copy* independientes para cada campo de la estructura *MqlRates*.

```
bool matrix<T>::CopyRates(const string symbol, ENUM_TIMEFRAMES tf, ulong rates_mask,
 ulong start, ulong count)
bool matrix<T>::CopyRates(const string symbol, ENUM_TIMEFRAMES tf, ulong rates_mask,
 datetime from, ulong count)
bool matrix<T>::CopyRates(const string symbol, ENUM_TIMEFRAMES tf, ulong rates_mask,
 datetime from, datetime to)
```

En los parámetros, debe especificar el símbolo, el marco temporal y el intervalo de barras solicitado, bien por número y cantidad, o bien por intervalo de fechas. Los datos se copian de forma que el elemento más antiguo se coloque al principio de la matriz/vector.

El parámetro *rates_mask* especifica una combinación de banderas de la enumeración ENUM_COPY_RATES con un conjunto de campos disponibles. La combinación de banderas permite obtener varias series temporales del historial en una sola solicitud. En este caso, el orden de las filas en la matriz corresponderá al orden de los valores en la enumeración ENUM_COPY_RATES; en particular, la fila con datos *High* en la matriz siempre estará por encima de la fila con datos *Low*.

Al copiar a un vector, sólo se puede especificar un valor de la enumeración ENUM_COPY_RATES. De lo contrario, se producirá un error.

Identificador	Valor	Descripción
COPY_RATES_OPEN	1	*Open* precios
COPY_RATES_HIGH	2	*High* precios
COPY_RATES_LOW	4	*Low* precios
COPY_RATES_CLOSE	8	*Close* precios
COPY_RATES_TIME	16	Horas de apertura de barra
COPY_RATES_VOLUME_TICK	32	Volúmenes de ticks
COPY_RATES_VOLUME_REAL	64	Volúmenes reales
COPY_RATES_SPREAD	128	Diferenciales
		Combinaciones
COPY_RATES_OHLC	15	*Open, High, Low, Close*
COPY_RATES_OHLCT	31	*Open, High, Low, Close, Time*

Veremos un ejemplo de utilización de esta función en la sección Resolución de ecuaciones.

4.10.5 Copiar el historial de ticks en matrices o vectores

Como en el caso de las barras, puede copiar los ticks en un vector o matriz. Esto se consigue mediante las sobrecargas de los métodos *CopyTicks* y *CopyTicksRange*, que funcionan sobre una base similar a las funciones *CopyTicks* y *CopyTicksRange* pero reciben datos en el llamador. Estas funciones se describirán en detalle en la Parte 5, en la sección sobre arrays de ticks reales en estructuras *MqlTick*. Aquí sólo mostraremos los prototipos y mencionaremos los puntos principales.

```
bool matrix<T>::CopyTicks(const string symbol, uint flags, ulong from_msc, uint count)
bool vector<T>::CopyTicks(const string symbol, uint flags, ulong from_msc, uint count)
bool matrix<T>::CopyTicksRange(const string symbol, uint flags, ulong from_msc, ulong to_msc)
bool matrix<T>::CopyTicksRange(const string symbol, uint flags, ulong from_msc, ulong to_msc)
```

El parámetro *symbol* establece el nombre del instrumento financiero para el que se solicitan los ticks. El intervalo de ticks puede especificarse de distintas maneras:

- En *CopyTicks*, puede especificarse como un número de ticks (el parámetro *count*), a partir de algún momento (*from_msc*), en milisegundos.
- En *CopyTicksRange*, puede ser un intervalo de dos puntos en el tiempo (de *from_msc* a *to_msc*).

La composición de los datos copiados acerca de cada tick se especifica en el parámetro *flags* como una máscara de bits de valores de la enumeración ENUM_COPY_TICKS.

Identificador	Valor	Descripción
COPY_TICKS_INFO	1	Ticks generados por los cambios *Bid* y/o *Ask*
COPY_TICKS_TRADE	2	Ticks generados por los cambios *Last* y *Volume*
COPY_TICKS_ALL	3	Todos los ticks
COPY_TICKS_TIME_MS	1 << 8	Tiempo en milisegundos
COPY_TICKS_BID	1 << 9	*Bid* precio
COPY_TICKS_ASK	1 << 10	*Ask* precio
COPY_TICKS_LAST	1 << 11	*Last* precio
COPY_TICKS_VOLUME	1 << 12	Volumen
COPY_TICKS_FLAGS	1 << 13	Banderas de ticks

Los tres primeros bits (byte bajo) determinan el conjunto de ticks solicitados, y los restantes bits (byte alto) determinan las propiedades de estos ticks.

Las banderas de byte alto sólo pueden combinarse para matrices, ya que en el vector sólo se coloca una fila con los valores de un campo concreto de todos los ticks. Por lo tanto, sólo debe seleccionarse un bit del byte más significativo para rellenar el vector.

Al seleccionar varias propiedades de ticks en el proceso de rellenar la matriz, el orden de las filas en ella se corresponderá con el orden de los elementos en la enumeración. Por ejemplo, el precio *Bid* aparecerá siempre más arriba en la fila (con un índice menor) que la fila con precios *Ask*.

Un ejemplo de trabajo con ambos, ticks y vectores, se mostrará en la sección sobre aprendizaje automático.

4.10.6 Evaluación de expresiones con matrices y vectores

Puede realizar operaciones matemáticas elemento por elemento (utilice operadores) sobre matrices y vectores, como sumas, restas, multiplicaciones y divisiones. Para estas operaciones, ambos objetos deben ser del mismo tipo y tener las mismas dimensiones. Cada miembro de la matriz/vector interactúa con el elemento correspondiente de la segunda matriz/vector.

Como segundo término (multiplicador, sustraendo o divisor), también se puede utilizar un escalar del tipo correspondiente (*double*, *float* o *complex*). En este caso, cada elemento de la matriz o vector se procesará teniendo en cuenta ese escalar.

```
matrix matrix_a = {{0.1, 0.2, 0.3}, {0.4, 0.5, 0.6}};
matrix matrix_b = {{1, 2, 3}, {4, 5, 6}};
matrix matrix_c1 = matrix_a + matrix_b;
matrix matrix_c2 = matrix_b - matrix_a;
matrix matrix_c3 = matrix_a * matrix_b;   // Hadamard product (element-by-element)
matrix matrix_c4 = matrix_b / matrix_a;
matrix_c1 = matrix_a + 1;
matrix_c2 = matrix_b - double_value;
matrix_c3 = matrix_a * M_PI;
matrix_c4 = matrix_b / 0.1;
matrix_a += matrix_b;                     // operations "in place" are possible
matrix_a /= 2;
```

Las operaciones in situ modifican la matriz (o vector) original colocando en ella el resultado, a diferencia de las operaciones binarias normales en las que los operandos no se modifican y se crea un nuevo objeto para el resultado.

Además, las matrices y los vectores pueden pasarse como parámetros a la mayoría de las funciones matemáticas. En este caso, la matriz o el vector se procesan elemento a elemento. Por ejemplo:

```
matrix a = {{1, 4}, {9, 16}};
Print("matrix a=\n", a);
a = MathSqrt(a);
Print("MatrSqrt(a)=\n", a);
/*
   matrix a=
   [[1,4]
    [9,16]]
   MatrSqrt(a)=
   [[1,2]
    [3,4]]
*/
```

En el caso de *MathMod* y *MathPow*, el segundo parámetro puede ser un escalar, una matriz o un vector del tamaño adecuado.

4.10.7 Manipulación de matrices y vectores

Cuando se trabaja con matrices y vectores se pueden realizar manipulaciones básicas sin necesidad de cálculos. Al principio de la lista figuran exclusivamente métodos matriciales, mientras que los cuatro últimos también son aplicables a vectores.

- *Transpose*: transposición de matrices
- *Col, Row, Diag*: extraer y fijar filas, columnas y diagonales por número
- *TriL, TriU*: obtener la matriz triangular inferior y superior por el número de la diagonal
- *SwapCols, SwapRows*: reordenar filas y columnas indicadas por números
- *Flat*: establecer y obtener un elemento de matriz mediante un índice
- *Reshape*: remodelar una matriz «in situ»
- *Split, Hsplit, Vsplit*: dividir una matriz en varias submatrices
- *resize*: redimensionar una matriz o un vector «in situ»
- *Compare, CompareByDigits*: comparar dos matrices o dos vectores con una precisión dada de números reales
- *Sort*: ordenar «in situ» (permutación de elementos) y obteniendo un vector o matriz de índices
- *clip*: limitar el rango de valores de los elementos «in situ»

Tenga en cuenta que no se proporciona la división de vectores.

A continuación se muestran los métodos prototipo para matrices.

```
matrix<T> matrix<T>::Transpose()
vector matrix<T>::Col∫Row(const ulong n)
void matrix<T>::Col∫Row(const vector v, const ulong n)
vector matrix<T>::Diag(const int n = 0)
void matrix<T>::Diag(const vector v, const int n = 0)
matrix<T> matrix<T>::TriL∫TriU(const int n = 0)
bool matrix<T>::SwapCols∫SwapRows(const ulong n1, const ulong n2)
T matrix<T>::Flat(const ulong i)
bool matrix<T>::Flat(const ulong i, const T value)
bool matrix<T>::Resize(const ulong rows, const ulong cols, const ulong reserve = 0)
void matrix<T>::Reshape(const ulong rows, const ulong cols)
ulong matrix<T>::Compare(const matrix<T> &m, const T epsilon)
ulong matrix<T>::CompareByDigits(const matrix &m, const int digits)
bool matrix<T>::Split(const ulong nparts, const int axis, matrix<T> &splitted[])
void matrix<T>::Split(const ulong &parts[], const int axis, matrix<T> &splitted[])
bool matrix<T>::Hsplit∫Vsplit(const ulong nparts, matrix<T> &splitted[])
void matrix<T>::Hsplit∫Vsplit(const ulong &parts[], matrix<T> &splitted[])
void matrix<T>::Sort(func_reference compare = NULL, T context)
void matrix<T>::Sort(const int axis, func_reference compare = NULL, T context)
matrix<T> matrix<T>::Sort(func_reference compare = NULL, T context)
matrix<T> matrix<T>::Sort(const int axis, func_reference compare = NULL, T context)
bool matrix<T>::Clip(const T min, const T max)
```

Para los vectores existe un conjunto más reducido de métodos.

bool vector<T>::Resize(const ulong size, const ulong reserve = 0)
ulong vector<T>::Compare(const vector<T> &v, const T epsilon)
ulong vector<T>::CompareByDigits(const vector<T> &v, const int digits)
void vector<T>::Sort(func_reference compare = NULL, T context)
vector vector<T>::Sort(func_reference compare = NULL, T context)
bool vector<T>::Clip(const T min, const T max)

Ejemplo de transposición de matrices:

```
matrix a = {{0, 1, 2}, {3, 4, 5}};
Print("matrix a \n", a);
Print("a.Transpose() \n", a.Transpose());
/*
   matrix a
   [[0,1,2]
    [3,4,5]]
   a.Transpose()
   [[0,3]
    [1,4]
    [2,5]]
*/
```

Varios ejemplos de ajuste de diferentes diagonales utilizando el método *Diag*:

```
vector v1 = {1, 2, 3};
matrix m1;
m1.Diag(v1);
Print("m1\n", m1);
/*
   m1
   [[1,0,0]
    [0,2,0]
    [0,0,3]]
*/

matrix m2;
m2.Diag(v1, -1);
Print("m2\n", m2);
/*
   m2
   [[0,0,0]
    [1,0,0]
    [0,2,0]
    [0,0,3]]
*/

matrix m3;
m3.Diag(v1, 1);
Print("m3\n", m3);
/*
   m3
   [[0,1,0,0]
    [0,0,2,0]
    [0,0,0,3]]
*/
```

Modificación de la configuración de la matriz mediante *Reshape*:

```
matrix matrix_a = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}};
Print("matrix_a\n", matrix_a);
/*
   matrix_a
   [[1,2,3]
    [4,5,6]
    [7,8,9]
    [10,11,12]]
*/

matrix_a.Reshape(2, 6);
Print("Reshape(2,6)\n", matrix_a);
/*
   Reshape(2,6)
   [[1,2,3,4,5,6]
    [7,8,9,10,11,12]]
*/

matrix_a.Reshape(3, 5);
Print("Reshape(3,5)\n", matrix_a);
/*
   Reshape(3,5)
   [[1,2,3,4,5]
    [6,7,8,9,10]
    [11,12,0,3,0]]
*/

matrix_a.Reshape(2, 4);
Print("Reshape(2,4)\n", matrix_a);
/*
   Reshape(2,4)
   [[1,2,3,4]
    [5,6,7,8]]
*/
```

Aplicaremos la división de matrices en submatrices en un ejemplo cuando realicemos la Resolución de ecuaciones.

Los métodos *Col* y *Row* permiten no sólo obtener columnas o filas de una matriz por su número, sino también insertarlas «in situ» en matrices previamente definidas. En este caso no cambiarán ni las dimensiones de la matriz ni los valores de los elementos fuera del vector columna (para el caso *Col*) o de un vector fila (para el caso *Row*).

Si cualquiera de estos dos métodos se aplica a una matriz cuyas dimensiones aún no se han establecido, se creará una matriz nula de tamaño [N * M], donde N y M se definen de forma diferente para *Col* y *Row*, en función de la longitud del vector y del índice de columna o fila dado:

- Para *Col*, N es la longitud del vector de columnas y M es en 1 mayor que el índice especificado de la columna insertada
- Para *Row*, N es en 1 mayor que el índice especificado de la fila insertada y M es la longitud del vector de la fila

En el momento de escribir este capítulo, MQL5 no proporcionaba métodos para la inserción completa de filas y columnas con la expansión de los elementos subsiguientes, ni para excluir filas y columnas especificadas.

4.10.8 Productos de matrices y vectores

La multiplicación de matrices es una de las operaciones básicas de diversos métodos numéricos. Por ejemplo, suele utilizarse al aplicar métodos de propagación hacia delante y hacia atrás en capas de redes neuronales.

También se pueden atribuir varios tipos de convoluciones a la categoría de productos matriciales. El grupo de tales funciones en MQL5 tiene este aspecto:

- *MatMul*: producto matricial de dos matrices
- *Power*: elevar una matriz cuadrada a la potencia entera especificada
- *Inner*: producto interior de dos matrices
- *Outer*: producto exterior de dos matrices o dos vectores
- *Kron*: producto Kronecker de dos matrices, una matriz y un vector, un vector y una matriz, o dos vectores
- *CorrCoef*: calcular la correlación de Pearson entre filas o columnas de una matriz, o entre vectores
- *Cov*: calcular la matriz de covarianza de filas o columnas de una matriz, o entre dos vectores
- *Correlate*: calcular la correlación mutua (correlación cruzada) de dos vectores
- *Convolve*: calcular la convolución lineal discreta de dos vectores
- *Dot*: producto escalar de dos vectores

Para dar una idea general de cómo manejar estos métodos, daremos sus prototipos (en el siguiente orden: de matriz, pasando por matriz-vector mixto, a vector).

```
matrix<T> matrix<T>::MatMul(const matrix<T> &m)
matrix<T> matrix<T>::Power(const int power)
matrix<T> matrix<T>::Inner(const matrix<T> &m)
matrix<T> matrix<T>::Outer(const matrix<T> &m)
matrix<T> matrix<T>::Kron(const matrix<T> &m)
matrix<T> matrix<T>::Kron(const vector<T> &v)
matrix<T> matrix<T>::CorrCoef(const bool rows = true)
matrix<T> matrix<T>::Cov(const bool rows = true)
matrix<T> vector<T>::Cov(const vector<T> &v)
T vector<T>::CorrCoef(const vector<T> &v)
vector<T> vector<T>::Correlate(const vector<T> &v, ENUM_VECTOR_CONVOLVE mode)
vector<T> vector<T>::Convolve(const vector<T> &v, ENUM_VECTOR_CONVOLVE mode)
matrix<T> vector<T>::Outer(const vector<T> &v)
matrix<T> vector<T>::Kron(const matrix<T> &m)
matrix<T> vector<T>::Kron(const vector<T> &v)
T vector<T>::Dot(const vector<T> &v)
```

He aquí un ejemplo sencillo del producto matricial de dos matrices utilizando el método *MatMul*:

```
matrix a = {{1, 0, 0},
            {0, 1, 0}};
matrix b = {{4, 1},
            {2, 2},
            {1, 3}};
matrix c1 = a.MatMul(b);
matrix c2 = b.MatMul(a);
Print("c1 = \n", c1);
Print("c2 = \n", c2);
/*
   c1 =
   [[4,1]
    [2,2]]
   c2 =
   [[4,1,0]
    [2,2,0]
    [1,3,0]]
*/
```

Las matrices de la forma A[M,N] * B[N,K] = C[M,K] pueden multiplicarse, es decir, el número de columnas de la primera matriz debe ser igual al número de filas de la segunda matriz. Si las dimensiones no son coherentes, el resultado es una matriz vacía.

Al multiplicar una matriz y un vector se permiten dos opciones:

- El vector horizontal (fila) se multiplica por la matriz de la derecha, la longitud del vector es igual al número de filas de la matriz.
- La matriz se multiplica por un vector vertical (columna) de la derecha, la longitud del vector es igual al número de columnas de la matriz.

Los vectores también pueden multiplicarse entre sí. En *MatMul*, esto equivale siempre al producto punto (el método *Dot*) de un vector fila por un vector columna, y la opción cuando se multiplica un vector columna por un vector fila y se obtiene una matriz se admite en otro método: *Outer*.

Vamos a demostrar el producto *Outer* del vector *v5* por el vector *v3*, y en orden inverso. En ambos casos, un vector columna está implícito a la izquierda, y un vector fila está implícito a la derecha.

```
vector v3 = {1, 2, 3};
vector v5 = {1, 2, 3, 4, 5};
Print("v5 = \n", v5);
Print("v3 = \n", v3);
Print("v5.Outer(v3) = m[5,3] \n", v5.Outer(v3));
Print("v3.Outer(v5) = m[3,5] \n", v3.Outer(v5));
/*
   v5 =
   [1,2,3,4,5]
   v3 =
   [1,2,3]
   v5.Outer(v3) = m[5,3]
   [[1,2,3]
    [2,4,6]
    [3,6,9]
    [4,8,12]
    [5,10,15]]
   v3.Outer(v5) = m[3,5]
   [[1,2,3,4,5]
    [2,4,6,8,10]
    [3,6,9,12,15]]
*/
```

4.10.9 Transformaciones (descomposición) de matrices

Las transformaciones matriciales son las operaciones más utilizadas cuando se trabaja con datos. No obstante, muchas transformaciones complejas no pueden realizarse analíticamente y con absoluta precisión.

Las transformaciones matriciales (o, dicho de otro modo, las descomposiciones) son métodos que descomponen una matriz en sus partes componentes, lo que facilita el cálculo de operaciones matriciales más complejas. Los métodos de descomposición de matrices, también llamados métodos de factorización de matrices, son la base de los algoritmos de álgebra lineal, como la resolución de sistemas de ecuaciones lineales y el cálculo de la inversa de una matriz o determinante.

En concreto, la descomposición en valores singulares (SVD, por sus siglas en inglés) es muy utilizada en el aprendizaje automático, ya que permite representar la matriz original como producto de otras tres matrices. La descomposición SVD se utiliza para resolver diversos problemas, desde la aproximación por mínimos cuadrados hasta la compresión y el reconocimiento de imágenes.

Lista de métodos disponibles:

- *Cholesky*: calcular la descomposición de Cholesky
- *Eig*: calcular los valores propios y los vectores propios derechos de una matriz cuadrada
- *Eig Vals*: calcular los valores propios de la matriz común
- *LU*: implementar la factorización LU de una matriz como producto de una matriz triangular inferior y una matriz triangular superior
- *LUP*: implementar la factorización LUP con rotación parcial, que es una factorización LU sólo con permutaciones de filas, PA=LU
- *QR*: aplicar la factorización QR de la matriz

- *SVD*: descomposición en valores singulares

A continuación se presentan los prototipos de métodos.

```
bool matrix<T>::Cholesky(matrix<T> &L)
bool matrix<T>::Eig(matrix<T> &eigen_vectors, vector<T> &eigen_values)
bool matrix<T>::EigVals(vector<T> &eigen_values)
bool matrix<T>::LU(matrix<T> &L, matrix<T> &U)
bool matrix<T>::LUP(matrix<T> &L, matrix<T> &U, matrix<T> &P)
bool matrix<T>::QR(matrix<T> &Q, matrix<T> &R)
bool matrix<T>::SVD(matrix<T> &U, matrix<T> &V, vector<T> &singular_values)
```

Veamos un ejemplo de descomposición de valores singulares con el método SVD (véase. archivo *MatrixSVD.mq5*). En primer lugar, inicializamos la matriz original.

```
matrix a = {{0, 1, 2, 3, 4, 5, 6, 7, 8}};
a = a - 4;
a.Reshape(3, 3);
Print("matrix a \n", a);
```

Seguidamente hacemos una descomposición SVD:

```
matrix U, V;
vector singular_values;
a.SVD(U, V, singular_values);
Print("U \n", U);
Print("V \n", V);
Print("singular_values = ", singular_values);
```

Comprobemos la expansión, que debe cumplir la siguiente igualdad: *U * "singular diagonal" * V = A*.

```
matrix matrix_s;
matrix_s.Diag(singular_values);
Print("matrix_s \n", matrix_s);
matrix matrix_vt = V.Transpose();
Print("matrix_vt \n", matrix_vt);
matrix matrix_usvt = (U.MatMul(matrix_s)).MatMul(matrix_vt);
Print("matrix_usvt \n", matrix_usvt);
```

Vamos a comprobar la matriz resultante y la original en busca de errores.

```
ulong errors = (int)a.Compare(matrix_usvt, 1e-9);
Print("errors=", errors);
```

El registro debería tener este aspecto:

```
matrix a
[[-4,-3,-2]
 [-1,0,1]
 [2,3,4]]
U
[[-0.7071067811865474,0.5773502691896254,0.408248290463863]
 [-6.827109697437648e-17,0.5773502691896253,-0.8164965809277256]
 [0.7071067811865472,0.5773502691896255,0.4082482904638627]]
V
[[0.5773502691896258,-0.7071067811865474,-0.408248290463863]
 [0.5773502691896258,1.779939029415334e-16,0.8164965809277258]
 [0.5773502691896256,0.7071067811865474,-0.408248290463863]]
singular_values = [7.348469228349533,2.449489742783175,3.277709923350408e-17]

matrix_s
[[7.348469228349533,0,0]
 [0,2.449489742783175,0]
 [0,0,3.277709923350408e-17]]
matrix_vt
[[0.5773502691896258,0.5773502691896258,0.5773502691896256]
 [-0.7071067811865474,1.779939029415334e-16,0.7071067811865474]
 [-0.408248290463863,0.8164965809277258,-0.408248290463863]]
matrix_usvt
[[-3.999999999999997,-2.999999999999999,-2]
 [-0.9999999999999981,-5.977974170712231e-17,0.9999999999999974]
 [2,2.999999999999999,3.999999999999996]]
errors=0
```

Otro caso práctico de aplicación del método *Convolve* se incluye en el ejemplo de Métodos de aprendizaje automático.

4.10.10 Obtener estadísticas

Los métodos enumerados a continuación están diseñados para obtener estadísticas descriptivas de matrices y vectores. Todas ellas se aplican a un vector o a una matriz en su conjunto, así como a un eje de matriz determinado (horizontal o verticalmente). Cuando se aplican por completo a un objeto, estas funciones devuelven un escalar (singular). Cuando se aplica a una matriz a lo largo de cualquiera de los ejes, se devuelve un vector.

Aspecto general de los prototipos:

```
T vector<T>::Method(const vector<T> &v)
T matrix<T>::Method(const matrix<T> &m)
vector<T> matrix<T>::Method(const matrix<T> &m, const int axis)
```

Lista de métodos:

- *ArgMax, ArgMin*: devuelve índices de valores máximos y mínimos
- *Max, Min*: devuelve los valores máximo y mínimo
- *Ptp*: devuelve un rango de valores
- *Sum, Prod*: calcula la suma o el producto de elementos

- *CumSum, CumProd*: calcula la suma o el producto acumulados de elementos
- *Median, Mean, Average*: calcula la mediana, la media aritmética o la media aritmética ponderada
- *Std, Var*: calcula la desviación típica y la varianza
- *Percentile, Quantile*: calcular percentiles y cuantiles
- *RegressionMetric*: calcula una de las métricas de regresión predefinidas, como los errores de desviación de la línea de regresión sobre los datos matriciales/vectoriales

En el archivo *MatrixStdPercentile.mq5* se ofrece un ejemplo de cálculo de la desviación estándar y los percentiles para el rango de barras (en puntos) del símbolo y el marco temporal actuales.

```
input int BarCount = 1000;
input int BarOffset = 0;

void OnStart()
{
   // getting current chart quotes
   matrix rates;
   rates.CopyRates(_Symbol, _Period, COPY_RATES_OPEN | COPY_RATES_CLOSE,
      BarOffset, BarCount);
   // calculating price increments on bars
   vector delta = MathRound((rates.Row(1) - rates.Row(0)) / _Point);
   // debug print of initial bars
   rates.Resize(rates.Rows(), 10);
   Normalize(rates);
   Print(rates);
   // printing increment metrics
   PRTF((int)delta.Std());
   PRTF((int)delta.Percentile(90));
   PRTF((int)delta.Percentile(10));
}
```

Registro:

```
(EURUSD,H1)   [[1.00832,1.00808,1.00901,1.00887,1.00728,1.00577,1.00485,1.00652,1.005
(EURUSD,H1)    [1.00808,1.00901,1.00887,1.00728,1.00577,1.00485,1.00655,1.00537,1.004
(EURUSD,H1)   (int)delta.Std()=163 / ok
(EURUSD,H1)   (int)delta.Percentile(90)=170 / ok
(EURUSD,H1)   (int)delta.Percentile(10)=-161 / ok
```

4.10.11 Características de matrices y vectores

El siguiente grupo de métodos permite obtener las principales características de las matrices:

- *Rows, Cols*: número de filas y columnas de la matriz
- *Norm*: una de las normas de matriz predefinidas (ENUM_MATRIX_NORM)
- *Cond*: número de condición de la matriz
- *Det*: determinante de una matriz cuadrada no degenerada
- *SLogDet*: calcula el signo y el logaritmo del determinante de la matriz
- *Rank*: rango de la matriz
- *Trace*: suma de los elementos a lo largo de las diagonales de la matriz (traza)

- *Spectrum*: espectro de una matriz como conjunto de sus valores propios

Además, se definen las siguientes características para los vectores:

- *Size*: longitud del vector
- *Norm*: una de las normas de vector predefinidas (ENUM_VECTOR_NORM)

Los tamaños de los objetos (así como la indexación de los elementos que los componen) utilizan valores del tipo *ulong*.

```
ulong matrix<T>::Rows()
ulong matrix<T>::Cols()
ulong vector<T>::Size()
```

La mayoría de las demás características son números reales.

```
double vector<T>::Norm(const ENUM_VECTOR_NORM norm, const int norm_p = 2)
double matrix<T>::Norm(const ENUM_MATRIX_NORM norm)
double matrix<T>::Cond(const ENUM_MATRIX_NORM norm)
double matrix<T>::Det()
double matrix<T>::SLogDet(int &sign)
double matrix<T>::Trace()
```

El rango y el espectro son, respectivamente, un número entero y un vector.

```
int matrix<T>::Rank()
vector matrix<T>::Spectrum()
```

Ejemplo de cálculo del rango de una matriz:

```
matrix a = matrix::Eye(4, 4);
Print("matrix a (eye)\n", a);
Print("a.Rank()=", a.Rank());

a[3, 3] = 0;
Print("matrix a (defective eye)\n", a);
Print("a.Rank()=", a.Rank());

matrix b = matrix::Ones(1, 4);
Print("b \n", b);
Print("b.Rank()=", b.Rank());

matrix zeros = matrix::Zeros(4, 1);
Print("zeros \n", zeros);
Print("zeros.Rank()=", zeros.Rank());
```

Y este es el resultado de la ejecución del script:

```
matrix a (eye)
[[1,0,0,0]
 [0,1,0,0]
 [0,0,1,0]
 [0,0,0,1]]
a.Rank()=4

matrix a (defective eye)
[[1,0,0,0]
 [0,1,0,0]
 [0,0,1,0]
 [0,0,0,0]]
a.Rank()=3

b
[[1,1,1,1]]
b.Rank()=1

zeros
[[0]
 [0]
 [0]
 [0]]
zeros.Rank()=0
```

4.10.12 Resolución de ecuaciones

En los métodos de aprendizaje automático y problemas de optimización, a menudo es necesario encontrar una solución a un sistema de ecuaciones lineales. MQL5 contiene cuatro métodos que permiten resolver este tipo de ecuaciones en función del tipo de matriz.

- *Solve* resuelve una ecuación matricial lineal o un sistema de ecuaciones algebraicas lineales.
- *LstSq* resuelve de forma aproximada un sistema de ecuaciones algebraicas lineales (para matrices no cuadradas o degeneradas).
- *Inv* calcula una matriz inversa multiplicativa con respecto a una matriz cuadrada no singular mediante el método de Jordan-Gauss.
- *PInv* calcula la matriz pseudoinversa según el método de Moore-Penrose.

A continuación se presentan los prototipos de métodos.

vector<T> matrix<T>::Solve(const vector<T> b)
vector<T> matrix<T>::LstSq(const vector<T> b)
matrix<T> matrix<T>::Inv()
matrix<T> matrix<T>::PInv()

Los métodos *Solve* y *LstSq* implican la solución de un sistema de ecuaciones de la forma *A*X=B*, donde A es una matriz, B es un vector pasado por un parámetro con los valores de la función (o «variable dependiente»).

Vamos a aplicar el método *LstSq* para resolver un sistema de ecuaciones, que es un modelo de trading de cartera ideal (en nuestro caso, analizaremos una cartera de las principales divisas Forex). Para ello,

en un número determinado de barras «históricas», necesitamos encontrar aquellos tamaños de lote para cada divisa con los que la línea de equilibrio tiende a ser una línea recta en constante crecimiento.

Indiquemos el par de divisas *i*-ésimo como S_i. Su cotización en la barra con el índice *k* es igual a $S_i[k]$. La numeración de las barras irá del pasado al futuro, como en las matrices y vectores rellenados por el método *CopyRates*. Así, el inicio de las cotizaciones recogidas para entrenar el modelo corresponde a la barra marcada con el número 0, pero en la línea de tiempo será la barra histórica más antigua (de las que procesemos, según la configuración del algoritmo). Las barras a la derecha (hacia el futuro) a partir de ella se numeran 1, 2 y así sucesivamente, hasta el número total de barras en las que el usuario ordenará el cálculo.

La variación del precio de un símbolo entre la barra 0 y la barra N determina la ganancia (o pérdida) por tiempo de la barra N.

Teniendo en cuenta el conjunto de divisas, obtenemos, por ejemplo, la siguiente ecuación de beneficios para la 1ª barra:

```
(S1[1] - S1[0]) * X1 + (S2[1] - S2[0]) * X2 + ... + (Sm[1] - Sm[0]) * Xm = B
```

Aquí, *m* es el número total de caracteres, X_i es el tamaño de lote de cada símbolo, y *B* es el beneficio flotante (saldo condicional, si fija el beneficio).

Para simplificar, abreviemos la notación. Pasemos de valores absolutos a incrementos de precio ($A_i[k] = S_i[k]-S_i[0]$). Teniendo en cuenta el movimiento a través de las barras, obtendremos varias expresiones para la curva de equilibrio virtual:

```
A1[1] * X1 + A2[1] * X2 + ... + Am[1] * Xm = B[1]
A1[2] * X1 + A2[2] * X2 + ... + Am[2] * Xm = B[2]
                          ...
A1[K] * X1 + A2[K] * X2 + ... + Am[K] * Xm = B[K]
```

El éxito del trading se caracteriza por un beneficio constante en cada barra, es decir, el modelo para el vector derecho *B* es una función creciente de forma monótona, idealmente una línea recta.

Pongamos en práctica este modelo y seleccionemos los coeficientes *X* para él basándonos en las cotizaciones. Como aún no conocemos las API de la aplicación, no codificaremos una estrategia de trading completa. Vamos simplemente a crear un gráfico de equilibrio virtual utilizando la función *GraphPlot* del archivo de encabezado estándar *Graphic.mqh* (ya la hemos utilizado para demostrar funciones matemáticas).

El código fuente completo del nuevo ejemplo se encuentra en el script *MatrixForexBasket.mq5*.

En los parámetros de entrada, deje que el usuario elija el número total de barras para el muestreo de datos (*BarCount*), así como el número de barras dentro de esta selección (*BarOffset*) en el que termina el pasado condicional y comienza el futuro condicional.

Se construirá un modelo sobre el pasado condicional (se resolverá el sistema de ecuaciones lineales anterior) y se realizará una prueba forward sobre el futuro condicional.

```
input int BarCount = 20;  // BarCount (known "history" and "future")
input int BarOffset = 10; // BarOffset (where "future" starts)
input ENUM_CURVE_TYPE CurveType = CURVE_LINES;
```

Para rellenar el vector con un equilibrio ideal, escribimos la función *ConstantGrow*: se utilizará más adelante durante la inicialización.

```
void ConstantGrow(vector &v)
{
   for(ulong i = 0; i < v.Size(); ++i)
   {
      v[i] = (double)(i + 1);
   }
}
```

La lista de instrumentos negociados (principales pares de Forex) se establece al principio de la función *OnStart*. Edítela para adaptarla a sus necesidades y a su entorno de negociación.

```
void OnStart()
{
   const string symbols[] =
   {
      "EURUSD", "GBPUSD", "USDJPY", "USDCAD",
      "USDCHF", "AUDUSD", "NZDUSD"
   };
   const int size = ArraySize(symbols);
   ...
```

Vamos a crear la matriz *rates* en la que se añadirán las cotizaciones de los símbolos, el vector *model* con la curva de balance deseada, y el vector auxiliar *close* para una petición símbolo a símbolo de los precios de cierre de barra (los datos de éste se copiarán en las columnas de la matriz *rates*).

```
   matrix rates(BarCount, size);
   vector model(BarCount - BarOffset, ConstantGrow);
   vector close;
```

En un bucle de símbolos, copiamos los precios de cierre en el vector *close*, calculamos los incrementos de precio y los escribimos en la columna correspondiente de la matriz *rates*.

```
   for(int i = 0; i < size; i++)
   {
      if(close.CopyRates(symbols[i], _Period, COPY_RATES_CLOSE, 0, BarCount))
      {
         // calculate increments (profit on all and on each bar in one line)
         close -= close[0];
         // adjust the profit to the pip value
         close *= SymbolInfoDouble(symbols[i], SYMBOL_TRADE_TICK_VALUE) /
            SymbolInfoDouble(symbols[i], SYMBOL_TRADE_TICK_SIZE);
         // place the vector in the matrix column
         rates.Col(close, i);
      }
      else
      {
         Print("vector.CopyRates(%d, COPY_RATES_CLOSE) failed. Error ",
            symbols[i], _LastError);
         return;
      }
   }
   ...
```

Veremos el cálculo del valor de un punto de precio (en la moneda de depósito) en la Parte 5.

También es importante tener en cuenta que las barras con los mismos índices pueden tener diferentes marcas de tiempo en diferentes instrumentos financieros: por ejemplo, si había un día festivo en uno de los países y el mercado estaba cerrado (fuera de Forex, los símbolos pueden, en teoría, tener diferentes horarios para las sesiones de trading). Para resolver este problema, necesitaríamos un análisis más profundo de las cotizaciones, teniendo en cuenta las horas de las barras y su sincronización antes de insertarlas en la matriz *rates*. No lo hacemos aquí para mantener la simplicidad, y también porque el mercado Forex opera según las mismas reglas la mayor parte del tiempo.

Dividimos la matriz en dos partes: la parte inicial se utilizará para encontrar una solución (ello emula la optimización sobre la historia), y la parte posterior se utilizará para una prueba forward (cálculo de posteriores cambios en el equilibrio).

```
matrix split[];
if(BarOffset > 0)
{
   // training on BarCount - BarOffset bars
   // check on BarOffset bars
   ulong parts[] = {BarCount - BarOffset, BarOffset};
   rates.Split(parts, 0, split);
}

// solve the system of linear equations for the model
vector x = (BarOffset > 0) ? split[0].LstSq(model) : rates.LstSq(model);
Print("Solution (lots per symbol): ");
Print(x);
...
```

Ahora que ya tenemos una solución, vamos a crear la curva de equilibrio para todas las barras de la muestra (la parte «histórica» ideal estará al principio, y después comenzará la parte «futura», que no se utilizó para ajustar el modelo).

```
vector balance = vector::Zeros(BarCount);
for(int i = 1; i < BarCount; ++i)
{
   balance[i] = 0;
   for(int j = 0; j < size; ++j)
   {
      balance[i] += (float)(rates[i][j] * x[j]);
   }
}
...
```

Vamos a calcular la calidad de la solución mediante el criterio R2.

```
   if(BarOffset > 0)
   {
      // make a copy of the balance
      vector backtest = balance;
      // select only "historical" bars for backtesting
      backtest.Resize(BarCount - BarOffset);
      // bars for the forward test have to be copied manually
      vector forward(BarOffset);
      for(int i = 0; i < BarOffset; ++i)
      {
         forward[i] = balance[BarCount - BarOffset + i];
      }
      // compute regression metrics independently for both parts
      Print("Backtest R2 = ", backtest.RegressionMetric(REGRESSION_R2));
      Print("Forward R2 = ", forward.RegressionMetric(REGRESSION_R2));
   }
   else
   {
      Print("R2 = ", balance.RegressionMetric(REGRESSION_R2));
   }
   ...
```

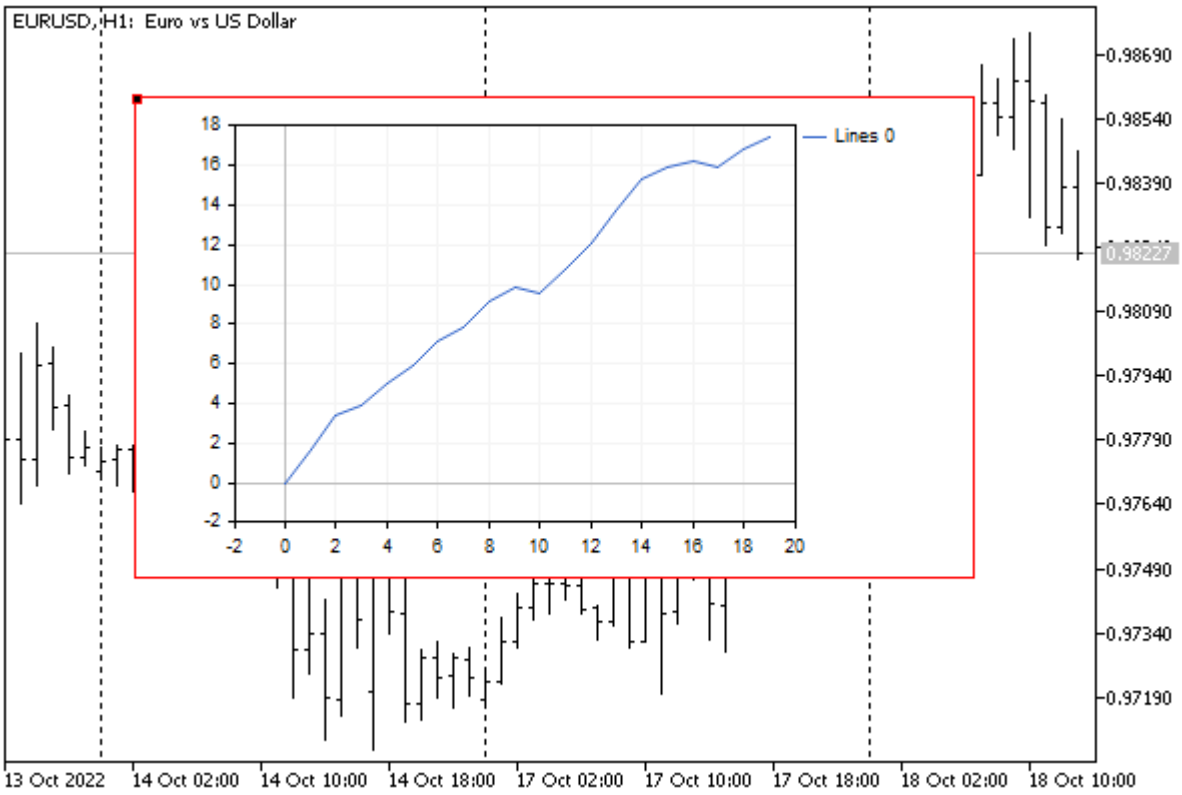
Para visualizar la curva de equilibrio en un gráfico es necesario transferir los datos de un vector a un array.

```
   double array[];
   balance.Swap(array);

   // print the values of the changing balance with an accuracy of 2 digits
   Print("Balance: ");
   ArrayPrint(array, 2);

   // draw the balance curve in the chart object ("backtest" and "forward")
   GraphPlot(array, CurveType);
}
```

A continuación se muestra un ejemplo de registro obtenido al ejecutar el script en EURUSD,H1.

```
Solution (lots per symbol):
[-0.0057809334,-0.0079846876,0.0088985749,-0.0041461736,-0.010710154,-0.0025694175,0.
Backtest R2 = 0.9896645616246145
Forward R2 = 0.8667852183780984
Balance:
 0.00  1.68  3.38  3.90  5.04  5.92  7.09  7.86  9.17  9.88
 9.55 10.77 12.06 13.67 15.35 15.89 16.28 15.91 16.85 16.58
```

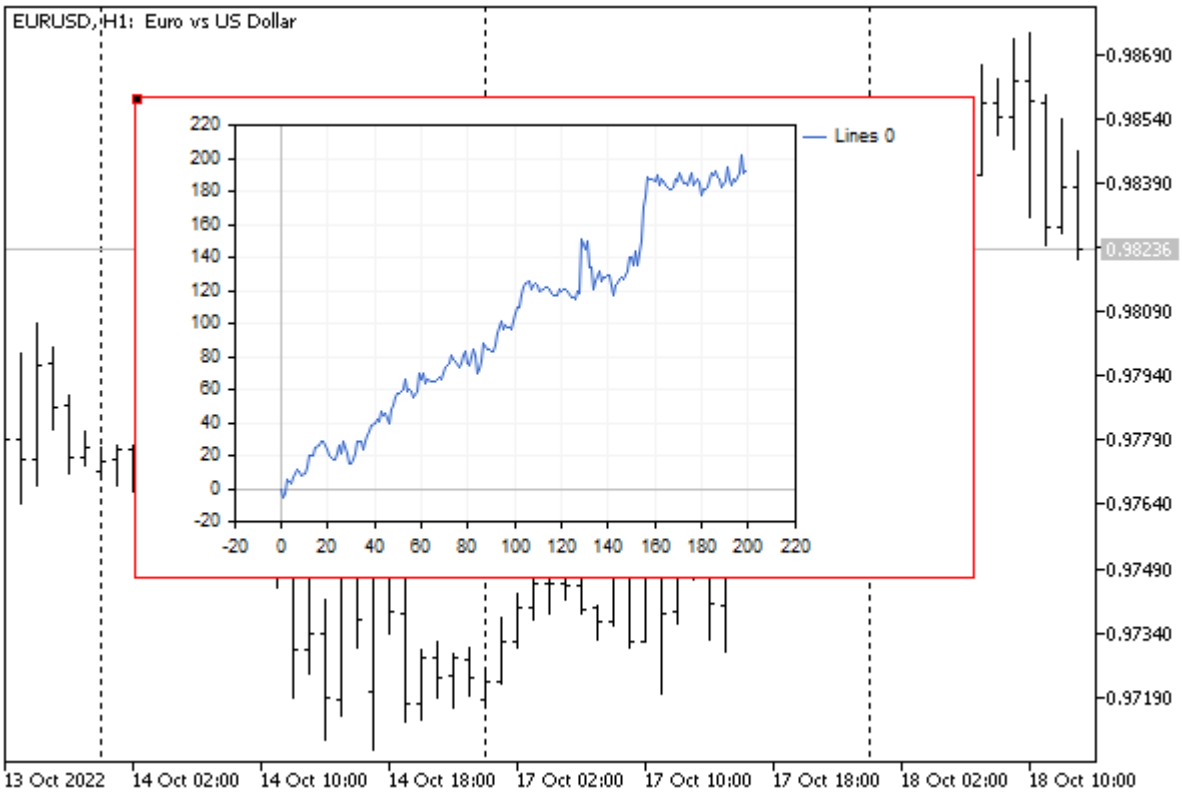
Y este es el aspecto de la curva de equilibrio virtual.



Balance virtual del trading de una cartera de divisas por lotes según la decisión

La mitad izquierda tiene una forma más uniforme y un R2 más alto, lo que no es sorprendente porque el modelo (*X* variables) se ajustó específicamente para ello.

Sólo por interés, aumentaremos 10 veces la profundidad de entrenamiento y verificación, es decir, fijaremos en los parámetros *BarCount* = *200* y *BarOffset* = *100*. Obtendremos una nueva imagen.



Balance virtual del trading de una cartera de divisas por lotes según la decisión

La parte del «futuro» parece menos suave, e incluso podemos decir que tenemos suerte de que siga creciendo, a pesar de un modelo tan simple. Por regla general, durante la prueba forward, la curva de equilibrio virtual se degrada de forma significativa y empieza a descender.

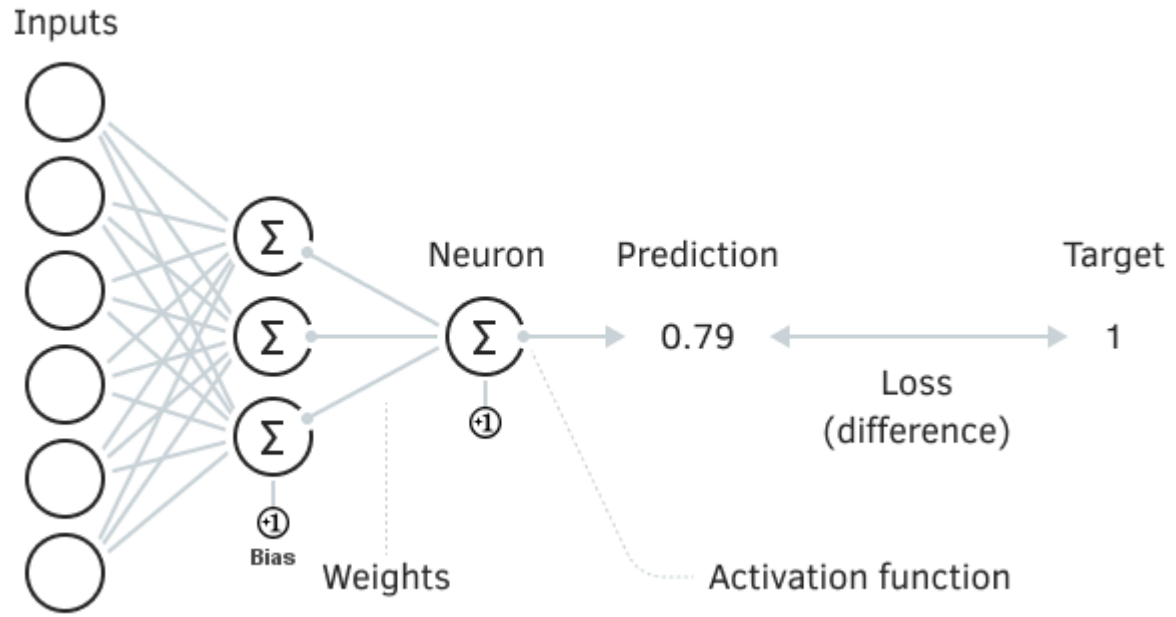
Es importante señalar que, para probar el modelo, tomamos los valores obtenidos en *X* de la solución «tal cual» del sistema, mientras que en la práctica tendremos que normalizarlos a los lotes mínimos y al paso de lote, lo que afectará negativamente a los resultados y los acercará más a la realidad.

4.10.13 Métodos de aprendizaje automático

Entre los métodos integrados de matrices y vectores, hay varios que son demandados en tareas de aprendizaje automático, en concreto en la implementación de redes neuronales.

Como su nombre indica, una red neuronal es un conjunto de muchas neuronas, que son las células de cálculo primitivas. Son primitivas en el sentido de que realizan cálculos bastante sencillos: por regla general, una neurona tiene un conjunto de coeficientes de peso que se aplican a determinadas señales de entrada, tras lo cual la suma ponderada de las señales se introduce en la función, que es un convertidor no lineal.

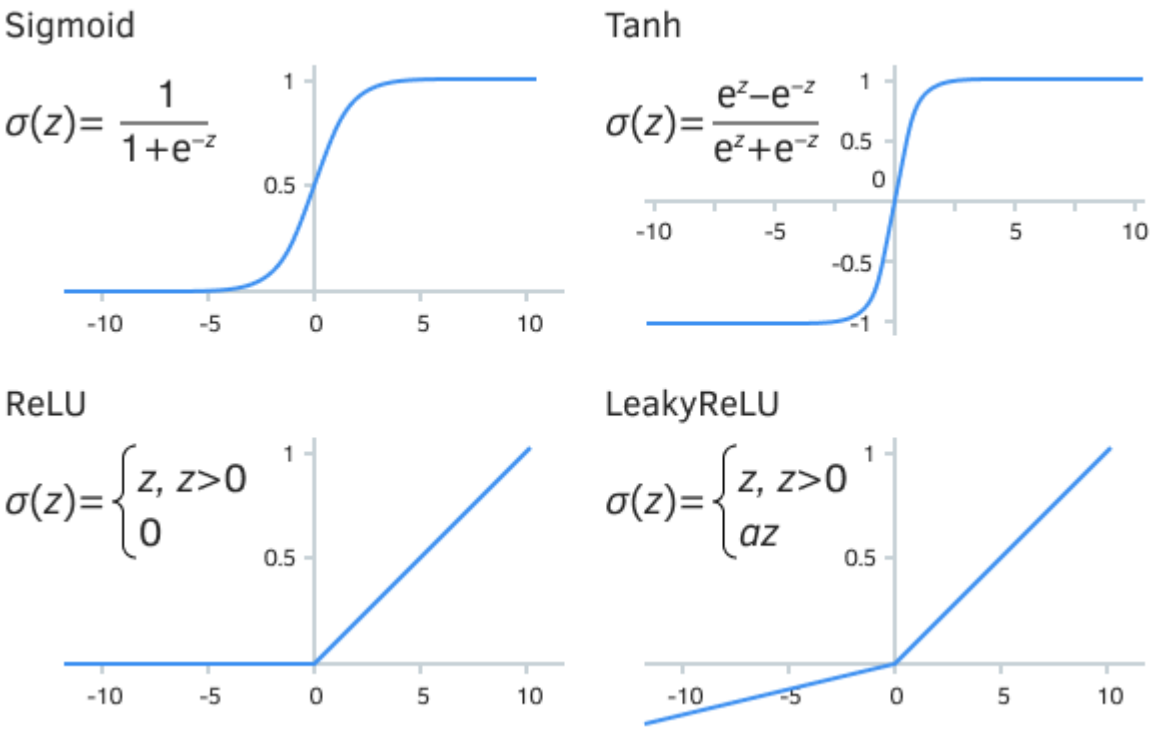
El uso de una función de activación amplifica las señales débiles y limita las demasiado fuertes, evitando el paso a la saturación (desbordamiento de los cálculos reales). Sin embargo, lo más importante es que la no linealidad dota a la red de nuevas capacidades de cálculo, lo que permite resolver problemas más complicados.



Red neuronal elemental

La potencia de las redes neuronales se manifiesta combinando un gran número de neuronas y estableciendo conexiones entre ellas. Normalmente, las neuronas se organizan en capas (que pueden compararse con matrices o vectores), incluidas aquellas que tienen conexiones recursivas (recurrentes), y también pueden tener funciones de activación que difieren en su efecto. Esto permite analizar datos volumétricos mediante diversos algoritmos, en concreto mediante la búsqueda de patrones ocultos en ellos.

Obsérvese que, si no fuera por la no linealidad de cada neurona, una red neuronal multicapa podría representarse de forma equivalente como una sola capa, cuyos coeficientes se obtienen por el producto matricial de todas las capas ($W_{total} = W_1 * W_2 * \ldots * W_{I-D}$, donde 1..L es el número de capas). Y esto sería un simple sumador lineal. Por tanto, se corrobora matemáticamente la importancia de las funciones de activación.



Algunas de las funciones de activación más famosas

Una de las principales clasificaciones de las redes neuronales las divide según el algoritmo de aprendizaje utilizado en redes de aprendizaje supervisado y no supervisado. Las supervisadas requieren que un experto humano proporcione los resultados deseados para el conjunto de datos original (por ejemplo, marcadores discretos del estado de un sistema de trading, o indicadores numéricos de incrementos de precios implícitos). Las redes no supervisadas identifican por sí solas conjuntos en los datos.

En cualquier caso, la tarea de entrenar una red neuronal consiste en buscar parámetros que minimicen el error en las muestras de entrenamiento y de prueba, para lo cual se utiliza la función de pérdida: ésta proporciona una estimación cualitativa o cuantitativa del error entre el objetivo y la respuesta recibida de la red.

> Los aspectos más importantes de la correcta aplicación de las redes neuronales incluyen la selección de predictores informativos e independientes entre sí (características analizadas), la transformación de los datos (normalización y limpieza) según las especificidades del algoritmo de aprendizaje, y la optimización de la arquitectura y el tamaño de la red. Tenga en cuenta que el uso de algoritmos de aprendizaje automático no garantiza el éxito.

Aquí no entraremos en la teoría de las redes neuronales, su clasificación y las tareas típicas que deben resolver, ya que se trata de un tema demasiado amplio. Los interesados pueden encontrar artículos en la web mql5.com y en otras fuentes.

MQL5 proporciona tres métodos de aprendizaje automático que han pasado a formar parte de la API de matrices y vectores.

- *Activation* calcula los valores de la función de activación.
- *Derivative* calcula los valores de la derivada de la función de activación.
- *Loss* calcula el valor de la función de pérdida.

Las derivadas de las funciones de activación permiten actualizar eficazmente los parámetros del modelo en función del error que cambia durante el proceso de aprendizaje.

Los dos primeros métodos escriben el resultado en el vector/matriz pasado y devuelven un indicador de éxito (*true* o *false*), y la función de pérdida devuelve un número. Veamos sus prototipos (bajo el tipo *object<T>* hemos marcado ambos, *matrix<T>* y *vector<T>*):

```
bool object<T>::Activation(object<T> &out, ENUM_ACTIVATION_FUNCTION activation)
bool object<T>::Derivative(object<T> &out, ENUM_ACTIVATION_FUNCTION loss)
T object<T>::Loss(const object<T> &target, ENUM_LOSS_FUNCTION loss)
```

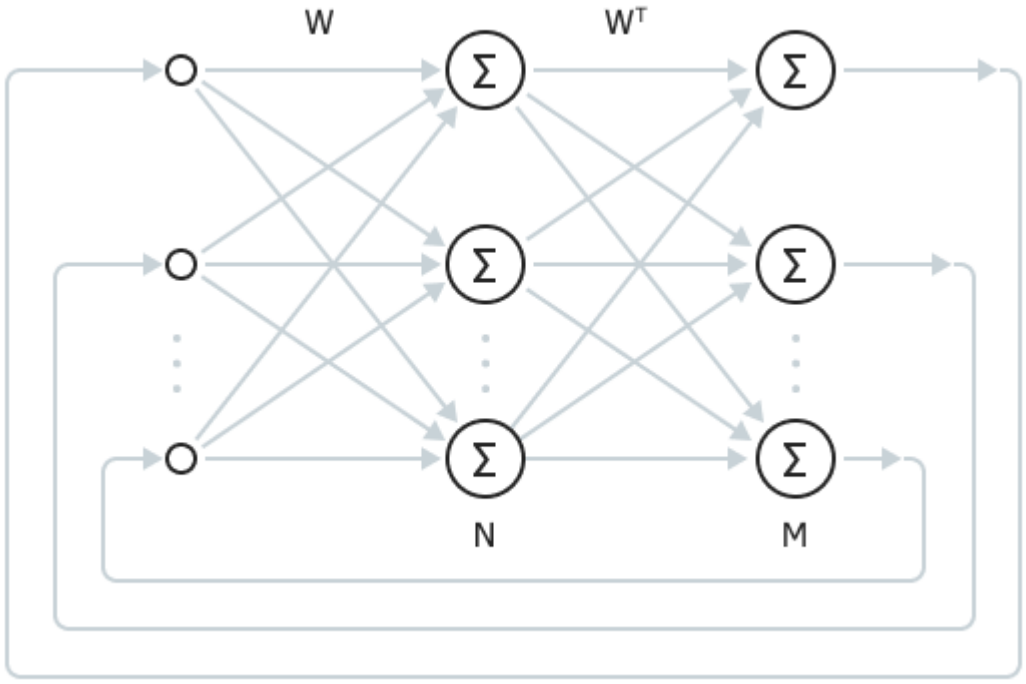
Algunas funciones de activación permiten establecer un parámetro con un tercer argumento opcional.

Consulte la documentación de MQL5 para ver la lista de funciones de activación admitidas en la enumeración ENUM_ACTIVATION_FUNCTION y las funciones de pérdida en la enumeración ENUM_LOSS_FUNCTION.

Como ejemplo introductorio, consideremos el problema de analizar el flujo de ticks reales. Algunos operadores consideran que los ticks son ruido basura, mientras que otros practican trading de alta frecuencia basado en ticks. Existe la presunción de que los algoritmos de alta frecuencia, por regla general, dan ventaja a los grandes operadores y se basan únicamente en el tratamiento informático de la información sobre precios. Basándonos en esto, plantearemos la hipótesis de que existe un efecto de memoria a corto plazo en el flujo de ticks, debido a los actuales robots activos de los creadores de mercado. A continuación, se puede utilizar un método de aprendizaje automático para encontrar esta dependencia y predecir varios ticks futuros.

> El aprendizaje automático siempre implica plantear hipótesis, sintetizar un modelo para las mismas y probarlas en la práctica. Obviamente, no siempre se obtienen hipótesis productivas. Se trata de un largo proceso de ensayo y error, en el que el fracaso es fuente de mejoras y nuevas ideas.

Utilizaremos uno de los tipos más sencillos de redes neuronales: la Memoria Asociativa Bidireccional (BAM, por sus siglas en inglés). Una red de este tipo sólo tiene dos capas: entrada y salida. Se forma una determinada respuesta (asociación) en la salida en respuesta a la señal de entrada. El tamaño de las capas puede variar. Cuando los tamaños son iguales, el resultado es una red de Hopfield.



Memoria asociativa bidireccional totalmente conectada

Utilizando una red de este tipo, compararemos N ticks previos recientes y M ticks próximos previstos, formando una muestra de entrenamiento desde el pasado cercano hasta una profundidad determinada. Los ticks se introducirán en la red como incrementos de precio positivos o negativos convertidos a valores binarios [+1, -1] (las señales binarias son la forma canónica de codificación en las redes BAM y Hopfield).

La ventaja más importante de BAM es el proceso de aprendizaje casi instantáneo (en comparación con la mayoría de los demás métodos iterativos), que consiste en calcular la matriz de pesos. Mas abajo le ofrecemos la fórmula.

Sin embargo, esta simplicidad también tiene un inconveniente: la capacidad de BAM (el número de imágenes que puede recordar) está limitada al tamaño de capa más pequeño, siempre que se cumpla la condición de una distribución especial de +1 y -1 en los vectores de la muestra de entrenamiento.

Así, para nuestro caso, la red generalizará todas las secuencias de ticks de la muestra de entrenamiento y luego, en el curso de un trabajo regular, pasará a una u otra imagen almacenada, en función de la secuencia de nuevos ticks presentada. El resultado en la práctica depende de un gran número de factores, como el tamaño y la configuración de la red, las características del flujo de ticks actual, etc.

Como se supone que el flujo de ticks sólo tiene memoria a corto plazo, es conveniente volver a entrenar la red en tiempo real o casi, ya que el entrenamiento se reduce en realidad a varias operaciones matriciales.

Así, para que la red recuerde las imágenes asociativas (en nuestro caso, el pasado y el futuro del flujo de ticks), se requiere la siguiente ecuación:

$$\mathbf{W} = \Sigma_i (\mathbf{A}_i^T \mathbf{B}_i)$$

donde W es la matriz de pesos de la red. La suma se realiza sobre todos los productos por pares de los vectores de entrada A_i y los correspondientes vectores de salida B_i.

A continuación, cuando la red está en funcionamiento, suministramos la imagen de entrada a la primera capa, le aplicamos la matriz W y activamos así la segunda capa, en la que se calcula la función de activación de cada neurona. Seguidamente, utilizando la matriz W^T transpuesta, la señal se propaga de nuevo a la primera capa, donde también se aplican funciones de activación en las neuronas. En este momento, la imagen de entrada ya no llega a la primera capa, es decir, el proceso oscilatorio libre continúa en la red. Continúa hasta que los cambios en la señal de las neuronas de la red se estabilizan (es decir, se vuelven inferiores a un determinado valor predeterminado).

En este estado, la segunda capa de la red contiene la imagen de salida asociada encontrada: la predicción.

Vamos a implementar este escenario de aprendizaje automático en el script *MatrixMachineLearning.mq5*.

En los parámetros de entrada, puede establecer el número total de últimos ticks (*TicksToLoad*) solicitados al historial, y cuántos de ellos se asignan a las pruebas (*TicksToTest*). En consecuencia, el modelo (ponderaciones) se basará en ticks (*TicksToLoad* - *TicksToTest*).

```
input int TicksToLoad = 100;
input int TicksToTest = 50;
input int PredictorSize = 20;
input int ForecastSize = 10;
```

Además, en las variables de entrada, se seleccionan los tamaños del vector de entrada (el número de ticks conocidos *PredictorSize*) y del vector de salida (el número de ticks futuros *ForecastSize*).

Los ticks se solicitan al principio de la función *OnStart*. En este caso, sólo trabajamos con los precios de *Ask*. No obstante, también puede añadir el proceso *Bid* y *Last*, junto con los volúmenes.

```
void OnStart()
{
   vector ticks;
   ticks.CopyTicks(_Symbol, COPY_TICKS_ALL | COPY_TICKS_ASK, 0, TicksToLoad);
   ...
```

Dividamos los ticks en conjuntos de entrenamiento y de prueba.

```
   vector ask1(n - TicksToTest);
   for(int i = 0; i < n - TicksToTest; ++i)
   {
      ask1[i] = ticks[i];
   }

   vector ask2(TicksToTest);
   for(int i = 0; i < TicksToTest; ++i)
   {
      ask2[i] = ticks[i + TicksToLoad - TicksToTest];
   }
   ...
```

Para calcular los incrementos de precio utilizamos el método *Convolve* con un vector adicional {+1, -1}. Tenga en cuenta que el vector con incrementos será 1 elemento más corto que el original.

```
    vector differentiator = {+1, -1};
    vector deltas = ask1.Convolve(differentiator, VECTOR_CONVOLVE_VALID);
    ...
```

La convolución según el algoritmo VECTOR_CONVOLVE_VALID significa que sólo se tienen en cuenta los solapamientos completos de los vectores (es decir, el vector más pequeño se desplaza de forma secuencial a lo largo del más grande sin salirse de sus límites). Otros tipos de convoluciones permiten que los vectores se solapen sólo con un elemento, o con la mitad de los elementos (en este caso, los elementos restantes están más allá del vector correspondiente y los valores de la convolución muestran efectos de borde).

Para convertir los valores continuos de los incrementos en impulsos unitarios (positivos y negativos dependiendo del signo del elemento inicial del vector), utilizaremos una función auxiliar *Binary* (no mostrada aquí): devuelve una nueva copia del vector en la que cada elemento es +1 o -1.

```
    vector inputs = Binary(deltas);
```

Basándonos en la secuencia de entrada recibida, utilizamos la función *TrainWeights* para calcular la matriz de pesos de la red neuronal W. Más adelante estudiaremos la estructura de esta función. Por ahora, preste atención a que se le pasan los parámetros *PredictorSize* y *ForecastSize*, lo que permite dividir una secuencia continua de ticks en conjuntos de vectores de entrada y salida emparejados según el tamaño de las capas BAM de entrada y salida, respectivamente.

```
    matrix W = TrainWeights(inputs, PredictorSize, ForecastSize);
    Print("Check training on backtest: ");
    CheckWeights(W, inputs);
    ...
```

Inmediatamente después de entrenar la red, comprobamos su precisión en el conjunto de entrenamiento, sólo para asegurarnos de que la red ha sido entrenada. Para ello se utiliza la función *CheckWeights*.

No obstante, es más importante comprobar cómo se comporta la red con datos de prueba desconocidos. Para ello, vamos a diferenciar y «binarizar» el segundo vector *ask2*, y después a enviarlo también a *CheckWeights*.

```
    vector test = Binary(ask2.Convolve(differentiator, VECTOR_CONVOLVE_VALID));
    Print("Check training on forwardtest: ");
    CheckWeights(W, test);
    ...
}
```

Es hora de familiarizarse con la función *TrainWeights*, en la que definimos las matrices A y B para «cortar» vectores de la secuencia de entrada pasada, es decir, del vector *data*.

```
template<typename T>
matrix<T> TrainWeights(const vector<T> &data, const uint predictor, const uint respon
   const uint start = 0, const uint _stop = 0, const uint step = 1)
{
   const uint sample = predictor + responce;
   const uint stop = _stop <= start ? (uint)data.Size() : _stop;
   const uint n = (stop - sample + 1 - start) / step;
   matrix<T> A(n, predictor), B(n, responce);

   ulong k = 0;
   for(ulong i = start; i < stop - sample + 1; i += step, ++k)
   {
      for(ulong j = 0; j < predictor; ++j)
      {
         A[k][j] = data[start + i * step + j];
      }
      for(ulong j = 0; j < responce; ++j)
      {
         B[k][j] = data[start + i * step + j + predictor];
      }
   }
   ...
```

Cada patrón A sucesivo se obtiene a partir de ticks consecutivos en cantidad igual a *predictor*, y el patrón futuro correspondiente se obtiene a partir de los siguientes elementos *response*. Mientras la cantidad total de datos lo permita, esta ventana se desplaza hacia la derecha, un elemento cada vez, formando más pares nuevos de imágenes. Las imágenes se numeran por filas y los ticks que aparecen en ellas se numeran por columnas.

A continuación, debemos asignar memoria a la matriz de pesos W y llenarla utilizando métodos matriciales: multiplicamos secuencialmente las filas de A y B utilizando *Outer*, y luego realizamos la suma matricial.

```
   matrix<T> W = matrix<T>::Zeros(predictor, responce);

   for(ulong i = 0; i < k; ++i)
   {
      W += A.Row(i).Outer(B.Row(i));
   }

   return W;
}
```

La función *CheckWeights* realiza acciones similares para una red neuronal, cuyos coeficientes de peso se pasan ya preparados en el primer argumento W. Los tamaños de los vectores de entrenamiento se extraen de la propia matriz W.

```
template<typename T>
void CheckWeights(const matrix<T> &W,
   const vector<T> &data,
   const uint start = 0, const uint _stop = 0, const uint step = 1)
{
   const uint predictor = (uint)W.Rows();
   const uint responce = (uint)W.Cols();
   const uint sample = predictor + responce;
   const uint stop = _stop <= start ? (uint)data.Size() : _stop;
   const uint n = (stop - sample + 1 - start) / step;
   matrix<T> A(n, predictor), B(n, responce);

   ulong k = 0;
   for(ulong i = start; i < stop - sample + 1; i += step, ++k)
   {
      for(ulong j = 0; j < predictor; ++j)
      {
         A[k][j] = data[start + i * step + j];
      }
      for(ulong j = 0; j < responce; ++j)
      {
         B[k][j] = data[start + i * step + j + predictor];
      }
   }

   const matrix<T> w = W.Transpose();
   ...
```

En este caso, las matrices A y B no se forman para calcular W, sino que actúan como «proveedoras» de vectores para las pruebas. También necesitamos una copia transpuesta de W para calcular las señales de retorno de la segunda capa de red a la primera.

El número de iteraciones durante las cuales se permiten procesos transitorios en la red, hasta la convergencia, está limitado por la constante *limit*.

```
   const uint limit = 100;

   int positive = 0;
   int negative = 0;
   int average = 0;
```

Las variables *positive*, *negative*, y *average* son necesarias para calcular las estadísticas de predicciones acertadas y fallidas a fin de evaluar la calidad del entrenamiento.

Además, la red se activa en un bucle sobre pares de patrones de prueba y se toma su respuesta final. Cada vector de entrada siguiente se escribe en el vector ***a***y la capa de salida ***b*** se rellena con ceros. A continuación, se lanzan iteraciones para la transmisión de la señal de ***a*** a ***b*** utilizando la matriz W y aplicando la función de activación AF_TANH, así como para la señal de realimentación de ***b*** a ***a***, y también el uso de AF_TANH. El proceso continúa hasta alcanzar *limit* bucles (lo cual es improbable) o hasta que se cumpla la condición de convergencia, bajo la cual los vectores ***a*** y ***b*** de estado de las neuronas prácticamente no cambian (aquí utilizamos el método *Compare* y copias auxiliares de ***x*** e ***y*** de la iteración anterior).

```
   for(ulong i = 0; i < k; ++i)
   {
      vector a = A.Row(i);
      vector b = vector::Zeros(responce);
      vector x, y;
      uint j = 0;

      for( ; j < limit; ++j)
      {
         x = a;
         y = b;
         a.MatMul(W).Activation(b, AF_TANH);
         b.MatMul(w).Activation(a, AF_TANH);
         if(!a.Compare(x, 0.00001) && !b.Compare(y, 0.00001)) break;
      }

      Binarize(a);
      Binarize(b);
      ...
```

Tras alcanzar un estado estable, transferimos los estados de las neuronas de continuo (real) a binario +1 y -1 utilizando la función *Binarize* (es similar a la función *Binary* mencionada anteriormente, pero cambia el estado del vector en su lugar).

Ahora, sólo tenemos que contar el número de coincidencias en la capa de salida con el vector objetivo. Para ello, realice una multiplicación escalar de vectores. Un resultado positivo significa que el número de aciertos supera al de errores. El recuento total de aciertos se acumula en «media».

```
      const int match = (int)(b.Dot(B.Row(i)));
      if(match > 0) positive++;
      else if(match < 0) negative++;

      average += match; // 0 in match means 50/50 precision (i.e. random guessing)
   }
```

Una vez completado el ciclo para todas las muestras de prueba, se muestran las estadísticas.

```
   float skew = (float)average / k; // average number of matches per vector

   PrintFormat("Count=%d Positive=%d Negative=%d Accuracy=%.2f%%",
      k, positive, negative, ((skew + responce) / 2 / responce) * 100);
}
```

El script también incluye la función *RunWeights*, que representa una ejecución de trabajo de la red neuronal (por su matriz de pesos W) para el vector en línea de los últimos ticks *predictor*. La función devolverá un vector con los ticks futuros estimados.

```
template<typename T>
vector<T> RunWeights(const matrix<T> &W, const vector<T> &data)
{
   const uint predictor = (uint)W.Rows();
   const uint responce = (uint)W.Cols();
   vector a = data;
   vector b = vector::Zeros(responce);

   vector x, y;
   uint j = 0;
   const uint limit = LIMIT;
   const matrix<T> w = W.Transpose();

   for( ; j < limit; ++j)
   {
      x = a;
      y = b;
      a.MatMul(W).Activation(b, AF_TANH);
      b.MatMul(w).Activation(a, AF_TANH);
      if(!a.Compare(x, 0.00001) && !b.Compare(y, 0.00001)) break;
   }

   Binarize(b);

   return b;
}
```

Al final de *OnStart*, pausamos la ejecución durante 1 segundo (para esperar nuevos ticks con un cierto grado de probabilidad), solicitamos los últimos ticks *PredictorSize + 1* (no olvidemos +1 para la diferenciación), y hacemos predicciones para ellos en línea.

```
void OnStart()
{
   ...
   Sleep(1000);
   vector ask3;
   ask3.CopyTicks(_Symbol, COPY_TICKS_ALL | COPY_TICKS_ASK, 0, PredictorSize + 1);
   vector online = Binary(ask3.Convolve(differentiator, VECTOR_CONVOLVE_VALID));
   Print("Online: ", online);
   vector forecast = RunWeights(W, online);
   Print("Forecast: ", forecast);
}
```

Al ejecutar el script con la configuración predeterminada en EURUSD el viernes por la tarde, se obtuvieron los siguientes resultados:

```
Check training on backtest:
Count=20 Positive=20 Negative=0 Accuracy=85.50%
Check training on forwardtest:
Count=20 Positive=12 Negative=2 Accuracy=58.50%
Online: [1,1,1,1,-1,-1,-1,1,-1,1,1,-1,1,1,-1,-1,1,1,-1,-1]
Forecast: [-1,1,-1,1,-1,-1,1,1,-1,1]
```

El símbolo y la hora no se mencionan, ya que la situación del mercado puede afectar significativamente a la aplicabilidad del algoritmo y a la configuración específica de la red. Cuando el mercado esté abierto, cada vez que ejecute el script obtendrá nuevos resultados a medida que entren más y más ticks. Este es un comportamiento esperado coherente con la hipótesis de formación de memoria a corto plazo.

Como podemos ver, la precisión de entrenamiento es aceptable, pero disminuye notablemente en los datos de prueba y puede caer por debajo del 50 %.

En este punto, pasamos sin problemas de la programación al campo de la investigación científica. El conjunto de herramientas de aprendizaje automático integrado en MQL5 le permite implementar muchas otras configuraciones de analizadores y redes neuronales, con diferentes estrategias de trading y principios de preparación de los datos iniciales.

Parte 5. Creación de programas de aplicación en MQL5

En esta parte, estudiaremos detenidamente aquellas secciones de la API que están relacionadas con la resolución de problemas aplicados del trading algorítmico: análisis y procesamiento de datos financieros, su visualización y marcado mediante objetos gráficos, automatización de acciones rutinarias e interacción interactiva con el usuario.

Comencemos con los principios generales de la creación de programas MQL, sus tipos, características y el modelo de eventos en el terminal. A continuación, abordaremos el acceso a series temporales, el trabajo con gráficos y los objetos gráficos. Por último, vamos a analizar los principios de creación y uso de cada tipo de programa MQL por separado.

Los usuarios activos de MetaTrader 5 sin duda recuerdan que el terminal admite cinco tipos de programas:

- Indicadores técnicos para calcular indicadores arbitrarios en forma de series temporales, con la posibilidad de visualizarlos en la ventana principal del gráfico o en un panel independiente (subventana);
- Asesores Expertos que ofrecen trading automático o semiautomático;
- Scripts para realizar tareas auxiliares puntuales bajo demanda;
- Servicios para realizar tareas en segundo plano en modo continuo;
- Bibliotecas, que son módulos compilados con una funcionalidad específica y separada, que se conectan a otros tipos de programas MQL durante su carga, de forma dinámica (lo que distingue fundamentalmente a las bibliotecas de los archivos de encabezado que se incluyen de forma estática en la etapa de compilación).

En las partes anteriores del libro, a medida que dominábamos los fundamentos de la programación y las funciones integradas comunes, ya tuvimos que recurrir a la implementación de scripts y servicios a modo de ejemplo. Se eligieron estos tipos de programas por ser más sencillos que los demás. Ahora los describiremos a propósito y les añadiremos indicadores más funcionales y populares.

Con la ayuda de indicadores y gráficos aprenderemos algunas técnicas que también serán aplicables a los Asesores Expertos. No obstante, pospondremos el desarrollo propiamente dicho de los Asesores Expertos, que es una tarea más compleja en su esencia, y lo trasladaremos a la siguiente parte del libro, la Parte 6, que incluye no sólo la ejecución automática de órdenes y la formalización de estrategias de trading, sino también su backtesting (simulación en el pasado) y optimización.

En cuanto a los indicadores, MetaTrader 5 es conocido por venir con un conjunto de indicadores estándar integrados. En esta parte aprenderemos a utilizarlos mediante programación, así como a crear nuestros propios indicadores tanto desde cero como basándonos en otros indicadores.

Todos los servicios, scripts, Asesores Expertos e indicadores compilados se muestran en el Navegador en MetaTrader 5. Las bibliotecas no son programas independientes, y por lo tanto no tienen una rama dedicada en la jerarquía, aunque, por supuesto, ello sería conveniente desde el punto de vista de la gestión uniforme de todos los módulos binarios. Como veremos más adelante, los programas que dependen de una determinada biblioteca no pueden ejecutarse sin ella. Pero ahora puede comprobar la existencia de la biblioteca sólo en el gestor de archivos.

Programación en MQL5 para Traders: códigos fuente del libro. Parte 5:

Los ejemplos del libro también están disponibles en el proyecto público \MQL5\Shared Projects\MQL5Book

5.1 Principios generales de ejecución de programas MQL

Todos los programas MQL pueden dividirse a grandes rasgos en varios grupos según sus capacidades y características.

La mayoría de los programas, como Asesores Expertos, indicadores y scripts, funcionan en el contexto de un gráfico. En otras palabras: comienzan a ejecutarse sólo después de que se adjuntan a uno de los gráficos abiertos utilizando el comando del menú contextual *Attach to Chart* en el árbol *Navegador* o arrastrando y soltando desde *Navegador* al gráfico.

En cambio, los servicios no pueden colocarse en el gráfico, ya que están diseñados para realizar acciones largas y cíclicas en segundo plano. Por ejemplo, en un servicio, puede crear un símbolo personalizado y, a continuación, recibir sus datos y seguir actualizándolos en un bucle sin fin mediante funciones de red. Otra aplicación lógica de un servicio es la supervisión de la cuenta de trading y la conexión de red, como parte de una solución que notifica al usuario los problemas de comunicación.

Es importante tener en cuenta que los indicadores y los Asesores Expertos se guardan en el gráfico entre las sesiones de trabajo del terminal. En otras palabras: si, por ejemplo, un usuario ejecuta un indicador en el gráfico y luego, sin borrarlo explícitamente, cierra MetaTrader 5, la próxima vez que se inicie el terminal, el indicador se restaurará junto con el gráfico, incluyendo todos sus ajustes.

Por cierto, vincular indicadores y Asesores Expertos al gráfico es la base de las plantillas (véase la Documentación). El usuario puede crear un conjunto de programas para utilizarlos en un gráfico, configurarlos y guardar el conjunto en un archivo especial con la extensión *tpl*. Para ello se utiliza el comando del menú contextual *Plantillas -> Guardar*. Después, puede aplicar la plantilla a cualquier gráfico nuevo (comando *Plantillas -> Subir*) y ejecutar todos los programas vinculados. Por defecto, las plantillas se almacenan en el directorio *MQL5/Profiles/Templates/*.

Otra consecuencia de adjuntar a un gráfico es que el cierre de un gráfico da lugar a la descarga de todos los programas MQL que se colocaron en él. Sin embargo, MetaTrader 5 guarda todos los gráficos cerrados de una manera específica (al menos durante un tiempo) y, por lo tanto, si el gráfico se cerró por accidente, se puede restaurar junto con todos los programas (y objetos gráficos) mediante el comando *Archivo -> Abrir remoto*.

Si por alguna razón el terminal falla al cargar los archivos de gráficos, se perderá todo el estado de los programas MQL (configuración y ubicación). Básicamente, lo mismo se aplica a los objetos gráficos : los programas pueden añadirlos para sus propias necesidades y esperar que estos objetos se encuentren en el gráfico. Haga copias de seguridad de los gráficos. Cada gráfico es un archivo con la extensión *chr*. Estos archivos se almacenan por defecto en el directorio *MQL5/Profiles/Charts/Default/*. Este es el perfil estándar que se crea al instalar la plataforma. Puede crear otros perfiles con el comando de menú *Archivo -> Perfiles* y luego pasar de uno a otro (consulte la Documentación).

Si es necesario, puede detener un Asesor Experto y eliminarlo del gráfico utilizando el comando del menú contextual *Lista de expertos* (al que se llama pulsando el botón derecho del ratón en la ventana del gráfico). Esto abre el cuadro de diálogo *Expertos* con una lista de todos los Asesores Expertos que se están ejecutando en el terminal. En esta lista, seleccione un Asesor Experto que ya no necesite y pulse *Eliminar*.

Los indicadores también pueden eliminarse explícitamente mediante un comando similar del menú contextual *Lista de indicadores* que abre un cuadro de diálogo con una lista de indicadores que se ejecutan en el gráfico actual, en el que puede seleccionar un indicador específico y hacer clic en el botón *Eliminar*. Además, la mayoría de los indicadores muestran en el gráfico diversas construcciones

gráficas, como líneas e histogramas, que también pueden borrarse mediante los comandos correspondientes del menú contextual.

A diferencia de los indicadores y los Asesores Expertos, los scripts no están permanentemente vinculados a un gráfico. En el modo estándar, el script se elimina automáticamente del gráfico una vez finalizada la tarea que se le ha asignado, si se trata de una acción puntual. Si un script tiene un bucle para acciones periódicas y repetitivas, continuará su trabajo, por supuesto, hasta que el bucle se interrumpa de un modo u otro, pero no más allá del final de la sesión. Cerrar el terminal provoca que el script se separe del gráfico. Después de reiniciar MetaTrader 5, los scripts no se restauran en los gráficos.

> Tenga en cuenta que, si cambia el gráfico a otro símbolo o marco temporal, el script que se esté ejecutando en él se descargará. Sin embargo, los indicadores y Asesores Expertos seguirán funcionando, aunque se reiniciarán. Las reglas de inicialización para ellos son diferentes. Estos detalles se abordarán en la sección Funciones de inicio y parada de programas de varios tipos.

En el gráfico pueden colocarse sólo un Asesor Experto, sólo un script y cualquier número de indicadores. El Asesor Experto, el script y todos los indicadores trabajarán en paralelo (simultáneamente).

En cuanto a los servicios, sus instancias creadas y en ejecución se restauran automáticamente tras cargar el terminal. La instancia de servicio puede detenerse o eliminarse mediante el menú contextual de la sección *Services* de la ventana *Navegador*.

En la siguiente tabla se resumen de forma sintética las propiedades descritas anteriormente:

Tipo de programa	Enlace al gráfico	Cantidad en el gráfico	Recuperación de la sesión
Indicador	Requerido	Múltiples	Con gráfico o plantilla
Asesor Experto	Requerido	Máximo 1	Con gráfico o plantilla
Script	Requerido	Máximo 1	No se admite
Servicio	No se admite	0	Con terminal

Todos los programas MQL se ejecutan en el terminal del cliente y, por tanto, sólo funcionan mientras el terminal está abierto. Para un control constante del programa sobre la cuenta, utilice un servidor privado virtual (VPS).

5.1.1 Diseño de programas MQL de varios tipos

El tipo de programa es una propiedad fundamental en MQL5. A diferencia de C++ u otros lenguajes de programación de propósito general, en los que cualquier programa puede desarrollarse en direcciones arbitrarias, por ejemplo añadiendo una interfaz gráfica o cargando datos desde un servidor a través de la red, los programas MQL se dividen en determinados grupos en función de su finalidad. Por ejemplo, el análisis técnico de series temporales con visualización se realiza mediante indicadores, pero éstos no permiten realizar operaciones de trading. A su vez, las funciones de la API de trading están disponibles para los Asesores Expertos, pero carecen de búferes de indicadores (arrays para dibujar líneas).

Por lo tanto, al resolver un problema específico aplicado, el desarrollador debe descomponerlo en partes, y la funcionalidad de cada parte debe encajar en la especialización de un tipo separado. Por

supuesto, en casos sencillos basta con un único programa MQL, pero a veces la solución técnica óptima no es evidente. Por ejemplo, ¿cómo implementaría el trazado de un gráfico Renko: como un indicador, como un símbolo personalizado generado por el servicio, o puede que como cálculos específicos directamente en el Asesor Experto de trading? Todas las opciones son posibles.

El tipo de programa MQL se caracteriza por varios factores.

En primer lugar, cada tipo de programa tiene una carpeta separada en el directorio de trabajo MQL5. Ya hemos mencionado este hecho en la introducción a la Parte 1 y enumerado las carpetas. Así, para los indicadores, Asesores Expertos, scripts y servicios, las carpetas designadas son *Indicators*, *Experts*, *Scripts* y *Services*, respectivamente. La subcarpeta *Libraries* está reservada para las bibliotecas de la carpeta MQL5. En cada una de ellas, puede organizar un árbol de carpetas anidadas de configuración arbitraria.

El archivo binario (el programa terminado con la extensión *ex5*), que es el resultado de compilar el archivo mq5, se genera en el mismo directorio que el archivo fuente mq5. No obstante, debemos mencionar también los proyectos en MetaEditor (archivos con la extensión *mqproj*), que analizaremos en el capítulo Proyectos. Cuando se desarrolla un proyecto, se crea un producto acabado en un directorio junto al proyecto. Al crear un programa desde el Asistente MQL5 en MetaEditor (comando *Archivo -> Nuevo*), el archivo fuente se coloca por defecto en la carpeta correspondiente al tipo de programa. Si copia accidentalmente un programa en el directorio equivocado, no ocurrirá nada terrible: no se convertirá, por ejemplo, de un Asesor Experto en un indicador, o viceversa. Se puede mover a la ubicación deseada directamente en el editor, dentro de la ventana *Navigator* o en un administrador de archivos externo. En *Navigator*, cada tipo de programa se muestra con un icono especial.

> La ubicación de un programa dentro del directorio MQL5 en una subcarpeta dedicada a un tipo concreto no determina el tipo de este programa MQL en particular. El tipo se determina en función del contenido del archivo ejecutable, que, a su vez, es formado por el compilador a partir de directivas de propiedades y sentencias del código fuente.
>
> La jerarquía de carpetas por tipos de programa se utiliza por comodidad. Se recomienda ceñirse a ella, salvo cuando se trate de un grupo de proyectos relacionados (con programas de distinto tipo), que es más lógico almacenar en un directorio distinto.

En segundo lugar, cada tipo de programa se caracteriza por ser compatible con un conjunto limitado y específico de eventos del sistema que activan el programa. Obtendremos una visión general de las funciones de gestión de eventos en una sección aparte. Para recibir eventos de un tipo específico en un programa es necesario describir una función manejadora con un prototipo predefinido (nombre, lista de parámetros, valor de retorno).

Por ejemplo, ya hemos visto que, en los scripts y servicios, el trabajo se inicia en la función *OnStart*, y puesto que es la única que existe, puede denominarse el «punto de entrada» principal a través del cual el terminal transfiere el control al código de la aplicación. En otros tipos de programas, la situación es algo más complicada. En general, observamos que un tipo de programa se caracteriza por un determinado conjunto de manejadores, algunos de los cuales pueden ser obligatorios y otros opcionales (pero, al mismo tiempo, inaceptables para otros tipos de programas). En concreto, un indicador requiere la función *OnCalculate* (sin ella, un indicador no compilará y el compilador generará un error). Sin embargo, esta función no se utiliza en los Asesores Expertos.

En tercer lugar, algunos tipos de programas requieren directivas especiales de *#property*. En el capítulo Propiedades generales de los programas vimos ya directivas que pueden utilizarse en todo tipo de programas. Sin embargo, existen otras directivas especializadas. Por ejemplo, en las tareas con servicios, que hemos mencionado, nos encontramos con la directiva *#property service*, que convierte el

programa en un servicio. Sin ella, ni siquiera colocando el programa en la carpeta *Services* podrá ejecutarse en segundo plano.

Del mismo modo, la directiva *#property library* desempeña un papel definitorio en la creación de bibliotecas. Todas estas propiedades de las directivas se abordarán en las secciones correspondientes a cada tipo de programa.

La combinación de directivas y manejadores de eventos se tiene en cuenta a la hora de establecer un tipo de programa MQL en el siguiente orden (de arriba a abajo hasta la primera coincidencia):

- indicador: la presencia del manejador *OnCalculate*
- biblioteca: *#property library*
- script: la presencia del manejador *OnStart* y la ausencia de *#property service*
- servicio: la presencia del manejador *OnStart* y *#property service*
- Asesor Experto: la presencia de cualquier otro manejador

Un ejemplo del efecto que estas propiedades tienen sobre el compilador se ofrecerá en la sección Visión general de las funciones de gestión de eventos.

> Por todo lo anterior, hay que tener en cuenta un punto más. El tipo de programa viene determinado por el módulo principal compilado: un archivo con la extensión mq5, en el que se pueden incluir otras fuentes de otros directorios mediante la directiva *#include*. Todas las funciones incluidas de esta forma se tienen en cuenta al mismo nivel que las que están presentes directamente en el archivo mq5 principal.
>
> Por otra parte, las directivas *#property* sólo tienen efecto cuando se colocan en el archivo mq5 principal compilado. Si las directivas aparecen en archivos incluidos en el programa utilizando *#include*, serán ignoradas.

El archivo principal mq5 no tiene que contener literalmente funciones manejadoras de eventos. Es perfectamente aceptable colocar parte o todo el algoritmo en archivos de encabezado mqh y luego incluirlos en uno o más programas. Por ejemplo, podemos implementar el manejador *OnStart* con un conjunto de acciones útiles en un archivo mqh y utilizarlo a través de *#include* dentro de dos programas separados: un script y un servicio.

Mientras tanto, fijémonos en que la presencia de manejadores de eventos comunes no es el único motivo para separar fragmentos de algoritmos comunes en un archivo de encabezado. Puede utilizar la misma fórmula de cálculo, por ejemplo, en un indicador y en un Asesor Experto, dejando sus manejadores de eventos en los módulos principales del programa.

> Aunque es habitual referirse a los archivos de inclusión como archivos de encabezado y darles la extensión *mqh*, esto no es técnicamente necesario. Es bastante aceptable (aunque no recomendable) incluir otro archivo mq5 o, por ejemplo, un archivo txt en un archivo mq5. Pueden contener algún código heredado o, digamos, la inicialización de ciertos arrays con constantes. La inclusión de otro archivo mq5 no lo convierte en el principal.
>
> Debe asegurarse de que sólo se introduzcan en el programa las funciones de gestión de eventos características del tipo de programa específico, y de que no haya duplicados entre ellas (como sabe, las funciones se identifican mediante una combinación de nombres y una lista de parámetros: la sobrecarga de funciones sólo se permite con un conjunto diferente de parámetros). Esto suele conseguirse utilizando varias directivas del preprocesador. Por ejemplo, si definimos la macro *#define OnStart OnStartPrevious* antes de incluir un archivo de script mq5 de terceros en algunos de nuestros programas, convertiremos realmente la función *OnStart* descrita en él en

OnStartPrevious, y podremos llamarla como de costumbre desde nuestros propios manejadores de eventos.

Sin embargo, este enfoque sólo tiene sentido en casos excepcionales, cuando el código fuente del archivo mq5 incluido no puede modificarse por algún motivo; en concreto, cuando no puede estructurarse con la selección de algoritmos de interés en funciones o clases en archivos de encabezado independientes.

Según el principio de interacción con el usuario, los programas MQL pueden dividirse en interactivos y funcionales.

Los programas interactivos (indicadores y Asesores Expertos) pueden procesar eventos, los cuales se producen en el entorno de software en respuesta a acciones del usuario, tales como pulsar botones en el teclado, mover el ratón, cambiar el tamaño de la ventana, así como muchos otros eventos, por ejemplo, relacionados con la recepción de datos de cotización o con acciones del temporizador.

Los programas de utilidades (servicios y scripts) se guían únicamente por las variables de entrada establecidas en el momento del lanzamiento, y no responden a eventos del sistema.

Aparte de todos los tipos de programas están las bibliotecas. Siempre se ejecutan como parte de otro tipo de programa MQL (uno de los cuatro principales), por lo que no tienen ninguna característica o comportamiento distintivo. En concreto, no pueden recibir directamente eventos del terminal y no disponen de hilos propios (véase la sección siguiente). La misma biblioteca puede estar conectada a muchos programas, y esto ocurre de forma dinámica en el momento de lanzar cada programa padre. En la sección sobre bibliotecas aprenderemos a describir la API exportada de una biblioteca y a importarla a un programa padre.

5.1.2 Hilos

De forma simplificada, un programa puede representarse como una secuencia de sentencias que un desarrollador ha generado para un ordenador. El principal ejecutor de sentencias en un ordenador es la unidad central de procesamiento. Los ordenadores modernos suelen estar equipados con procesadores de varios núcleos, lo que equivale a tener varios procesadores. Sin embargo, el número de programas que un usuario puede querer ejecutar en paralelo es prácticamente ilimitado. Así, el número de programas es siempre varias veces superior al de núcleos/procesadores disponibles. Por eso, cada núcleo divide en realidad su tiempo de trabajo entre varios programas diferentes: asignará 1 milisegundo para ejecutar las sentencias de un programa, luego 1 milisegundo para las sentencias de otro, luego para las de terceros, y así sucesivamente, en círculo. Como la conmutación se produce muy rápidamente, el usuario no lo nota, ya que parece que todos los programas se ejecutan en paralelo y simultáneamente.

Para que el procesador pueda suspender la ejecución de las sentencias de un programa y luego reanudar su trabajo desde el lugar en el que lo dejó (después de haber pasado tranquilamente a las sentencias de otros programas «paralelos»), debe ser capaz de guardar y restaurar de algún modo el estado intermedio de cada programa: la sentencia actual, las variables, los posibles archivos abiertos, las conexiones de red, etc. Todo este conjunto de recursos y datos que un programa necesita para ejecutarse con normalidad, junto con su posición actual en la secuencia de sentencias, se denomina contexto de ejecución del programa. El sistema operativo, de hecho, está diseñado para crear dichos contextos para cada programa a petición del usuario (o de otros programas). Cada uno de estos contextos activos se denomina hilo. Muchos programas requieren muchos hilos para sí mismos porque su funcionalidad implica mantener varias actividades en paralelo. MetaTrader 5 también requiere

múltiples hilos para cargar las cotizaciones para múltiples símbolos, trazar gráficos y responder a las acciones del usuario. Además, también se asignan hilos independientes a los programas MQL.

El entorno de ejecución de programas MQL no asigna más de un hilo a cada programa. Los Asesores Expertos, scripts y servicios reciben estrictamente un hilo cada uno. En cuanto a los indicadores, se asigna un flujo para todos los indicadores que trabajan con un instrumento financiero. Además, el mismo hilo se encarga de mostrar los gráficos del símbolo correspondiente, por lo que no es recomendable ocuparlo con cálculos pesados. De lo contrario, la interfaz de usuario dejará de responder: las acciones del usuario se procesarán con retraso, o la ventana dejará de responder incluso. Los hilos de todos los demás tipos de programas MQL no están vinculados a una interfaz y, por lo tanto, pueden cargar el procesador con cualquier tarea compleja.

Una de las propiedades importantes de un hilo se deriva de su definición y finalidad: sólo admite la ejecución secuencial de sentencias especificadas una tras otra. Sólo se ejecuta una sentencia en un hilo en cada momento. Si se escribe un bucle infinito en el programa, el hilo se atascará en esta instrucción y nunca llegará a las instrucciones por debajo de ella. Los cálculos largos también pueden crear el efecto de un bucle sin fin: cargarán el procesador e impedirán que se realicen otras acciones, cuyos resultados puede que el usuario esté esperando. Por eso, la eficacia de los cálculos en los indicadores es importante para el buen funcionamiento de la interfaz gráfica.

No obstante, en otros tipos de programas MQL, se debe prestar atención a la disposición de los hilos. En las siguientes secciones nos familiarizaremos con las funciones especiales de manejo de eventos que son los puntos de entrada a los programas MQL. Un modelo de un solo hilo significa que, durante el procesamiento de un evento, el programa es inmune a otros eventos que potencialmente podrían ocurrir al mismo tiempo. Por lo tanto, el terminal organiza una cola de eventos para cada programa. Abordaremos este punto con más detalle en la próxima sección.

Para experimentar los efectos del hilo único en la práctica, veremos un ejemplo sencillo en la sección Limitaciones y ventajas de los indicadores (*IndBarIndex.mq5*). Hemos elegido indicadores para este fin porque no sólo comparten un hilo para cada símbolo, sino que además muestran los resultados directamente en el gráfico, lo que hace que el problema potencial sea más evidente.

5.1.3 Visión general de las funciones de gestión de eventos

La transferencia de control a los programas MQL, es decir, su ejecución, se produce mediante la llamada a funciones especiales por parte de los agentes de pruebas o del terminal, que el desarrollador MQL define en su código de aplicación para procesar eventos predefinidos. Dichas funciones deben tener un prototipo especificado, que incluya un nombre, una lista de parámetros (número, tipos y orden) y un tipo de retorno.

El nombre de cada función corresponde al significado del evento, con la adición del prefijo *On*. Por ejemplo, *OnStart* es la función principal para «iniciar» scripts y servicios; es llamada por el terminal en el momento en que se coloca el script en el gráfico o se lanza la instancia del servicio.

Para los propósitos de este libro, nos referiremos a un evento y a su correspondiente manejador por el mismo nombre.

En la siguiente tabla se enumeran todos los tipos de eventos y los programas que los admiten (: indicador; : Asesor Experto; : script; : servicio). En las secciones de los respectivos tipos de programas se ofrece una descripción detallada de los eventos. Muchos factores pueden causar eventos de inicialización y desinicialización: colocar el programa en el gráfico, cambiar su

configuración, cambiar el símbolo o marco temporal del gráfico (o plantilla, o perfil), cambiar la cuenta, etc. (véase el capítulo Funciones de inicio y parada de programas de varios tipos).

Tipo de programa Evento/manejador					Descripción
OnStart	-	-	●	●	Iniciar/Ejecutar
OnInit	+	+	-	-	Inicialización tras la carga (véanse los detalles en la sección Funciones de inicio y parada de programas de varios tipos)
OnDeinit	+	+	-	-	Desinicialización antes de parar y descargar
OnTick	-	+	-	-	Obtener un nuevo precio (tick)
OnCalculate	●	-	-	-	Solicitud para recalcular el indicador debido a la recepción de un nuevo precio o a la sincronización de precios antiguos.
OnTimer	+	+	-	-	Activación del temporizador con una frecuencia determinada
OnTrade	-	+	-	-	Finalización de una operación de trading en el servidor
OnTradeTransaction	-	+	-	-	Modificación del estado de la cuenta de trading (órdenes, transacciones, posiciones)
OnBookEvent	+	+	-	-	Variación en el libro de órdenes
OnChartEvent	+	+	-	-	Acción del usuario o del programa MQL sobre el gráfico
OnTester	-	+	-	-	Fin de un único pase de comprobador
OnTesterInit	-	+	-	-	Inicialización antes de la optimización
OnTesterDeinit	-	+	-	-	Desinicialización tras la optimización
OnTesterPass	-	+	-	-	Recepción de datos de optimización del agente de pruebas

Los manejadores obligatorios están marcados con el símbolo '●', y los manejadores opcionales están marcados con '+'.

Aunque las funciones del manejador están pensadas principalmente para ser llamadas por tiempo de ejecución, también puede llamarlas desde su propio código fuente. Por ejemplo, si un Asesor Experto necesita realizar algún cálculo basado en las cotizaciones disponibles inmediatamente después del inicio, e incluso en ausencia de ticks (por ejemplo, los fines de semana), puede llamar a *OnTick* antes de salir de *OnInit*. Como alternativa, sería lógico separar el cálculo en una función independiente y llamarla tanto desde *OnInit* como desde *OnTick*. No obstante, es deseable realizar el trabajo de la función de inicialización rápidamente, y si el cálculo es largo, debería realizarse en un temporizador.

Todos los programas MQL (excepto las bibliotecas) deben tener al menos un manejador de eventos. De lo contrario, el compilador generará un error «función de gestión de eventos no encontrada».

La presencia de algunas funciones de manejador determina el tipo del programa en ausencia de directivas #property que establezcan otro tipo. Por ejemplo, tener el manejador *OnCalculate* conduce a la generación del indicador (incluso si se encuentra en otra carpeta, por ejemplo, scripts o Asesores Expertos). La presencia del manejador *OnStart* (si no existe *OnCalculate*) significa la creación de un script. Al mismo tiempo, si el indicador, además de *OnCalculate*, va a enfrentarse a *OnStart*, obtenemos una advertencia del compilador «función OnStart definida en el programa que no sea script».

El libro incluye dos archivos: *AllInOne.mq5* y *AllInOne.mqh*. El archivo de encabezado describe plantillas casi vacías de todos los manejadores de eventos principales. No contienen nada excepto la salida del nombre del manejador al registro. Examinaremos la sintaxis y los detalles específicos del uso de cada uno de los manejadores en las secciones sobre tipos específicos de programas MQL. El propósito de este archivo es proporcionar un campo para experimentar con la compilación de diferentes tipos de programas, dependiendo de la presencia de ciertos manejadores y directivas de propiedades (*#property*).

Algunas combinaciones pueden dar lugar a errores o advertencias.

Si la compilación se ha realizado correctamente, el tipo de programa resultante se registra automáticamente después de cargarlo mediante la siguiente línea:

```
const string type =
   PRTF(EnumToString((ENUM_PROGRAM_TYPE)MQLInfoInteger(MQL_PROGRAM_TYPE)));
```

Estudiamos el enum ENUM_PROGRAM_TYPE y la función *MQLInfoInteger* en la sección Tipo de programa y licencia.

El archivo *AllInOne.mq5*, que incluye *AllInOne.mqh*, se encuentra inicialmente en el directorio *MQL5Book/Scripts/p5/*, pero puede copiarse a cualquier otra carpeta, incluidas las ramas vecinas de *Navigator* (por ejemplo, a una carpeta de Asesores Expertos o indicadores). Dentro del archivo, en los comentarios, se dejan opciones para conectar determinadas configuraciones de ensamblado del programa. Por defecto, si no edita el archivo, apostará por un Asesor Experto.

```
//+------------------------------------------------------------------+
//| Uncomment the following line to get the service                  |
//| NB: also activate #define _OnStart OnStart                       |
//+------------------------------------------------------------------+
//#property service

//+------------------------------------------------------------------+
//| Uncomment the following line to get a library                    |
//+------------------------------------------------------------------+
//#property library

//+------------------------------------------------------------------+
//| Uncomment the following line to get a script or                  |
//| service (#property service must be enabled)                      |
//+------------------------------------------------------------------+
//#define _OnStart OnStart

//+------------------------------------------------------------------+
//| Uncomment one of the following two lines for the indicator       |
//+------------------------------------------------------------------+
//#define _OnCalculate1 OnCalculate
//#define _OnCalculate2 OnCalculate

#include <MQL5Book/AllInOne.mqh>
```

Si adjuntamos el programa al gráfico, obtendremos una entrada en el registro:

```
EnumToString((ENUM_PROGRAM_TYPE)MQLInfoInteger(MQL_PROGRAM_TYPE))=PROGRAM_EXPERT / ok
OnInit
OnChartEvent
OnTick
OnTick
OnTick
...
```

Además, lo más probable es que se genere un flujo de registros desde el manejador *OnTick* si el mercado está abierto.

Si duplica el archivo mq5 con un nombre diferente y, por ejemplo, anula el comentario de la directiva *#property service*, el compilador generará el servicio, pero devolverá algunas advertencias.

```
no OnStart function defined in the script
OnInit function is useless for scripts
OnDeinit function is useless for scripts
```

La primera de ellas, sobre la ausencia de la función *OnStart*, es realmente significativa, porque cuando se crea una instancia de servicio, no se llamará a ninguna función en ella, sino que sólo se inicializarán las variables globales. No obstante, debido a esto, el diario (pestaña *Experts* del terminal) seguirá imprimiendo el tipo PROGRAM_SERVICE. Sin embargo, por regla general, en los servicios, así como en los scripts, se supone que la función *OnStart* está presente.

Las otras dos advertencias surgen porque nuestro archivo de encabezado contiene manejadores para todas las ocasiones, y el compilador nos recuerda que *OnInit* y *OnDeinit* no tienen sentido (no serán llamados por el terminal y ni siquiera se incluirán en la imagen binaria del programa). Por supuesto, en

los programas reales no debería haber tales advertencias, es decir, todos los manejadores deberían estar implicados, y todo lo superfluo debería eliminarse del código fuente, ya sea física o lógicamente, utilizando directivas de preprocesador para la compilación condicional.

Si crea otra copia de AllInOne.mq5 y activa no sólo la directiva *#property service* sino también la macro *#define _OnStart OnStart*, obtendrá como resultado de su compilación un servicio totalmente operativo. Cuando se lance, no sólo mostrará el nombre de su tipo, sino también el nombre del manejador activado *OnStart*.

La macro debía poder activar o desactivar el manejador estándar *OnStart* si así lo deseaban. En el texto *AllInOne.mqh*, esta función se describe del siguiente modo:

```
void _OnStart() // "extra" underline makes the function customized
{
   Print(__FUNCTION__);
}
```

El nombre que comienza con un guion bajo hace que no sea un manejador estándar, sino simplemente una función definida por el usuario con un prototipo similar. Cuando incluimos una macro, durante la compilación el compilador sustituye *_OnStart* por *OnStart*, y el resultado ya es un manejador estándar. Si nombramos explícitamente la función *OnStart*, entonces, según las prioridades de las características que determinan el tipo del programa MQL (véase la sección Características de programas MQL de varios tipos), no le permitiría obtener una plantilla de Asesor Experto (porque *OnStart* identifica el programa como un script o servicio).

Una compilación personalizada similar con macros *_OnCalculate1* o *_OnCalculate2* requiere «ocultar» opcionalmente el manejador con un nombre estándar *OnCalculate*: de lo contrario, si estuviera presente, siempre obtendríamos un indicador.

Si en la siguiente copia del programa activa la macro *#define _OnCalculate1 OnCalculate*, obtendrá un indicador de ejemplo (aunque esté vacío y no haga nada). Como veremos más adelante, existen dos formas diferentes del manejador *OnCalculate* para los indicadores, en relación con las cuales se presentan bajo nombres numerados (*_OnCalculate1* y *_OnCalculate2*). Si ejecuta el indicador en el gráfico, podrá ver en el registro los nombres de los eventos *OnCalculate* (a la llegada de los ticks) y *OnChartEvent* (por ejemplo, al hacer clic con el ratón).

Al compilar el indicador, el compilador generará dos advertencias:

```
no indicator window property is defined, indicator_chart_window is applied
no indicator plot defined for indicator
```

Esto se debe a que los indicadores, como herramientas de visualización de datos, requieren algunos ajustes específicos en su código que no están aquí. En esta fase en que empezamos a familiarizarnos un poco con los distintos tipos de programas, esto no es importante, pero más adelante aprenderemos a describir sus propiedades y arrays en los indicadores, que determinan qué y cómo debe visualizarse en el gráfico. Entonces estos avisos desaparecerán.

Cola de eventos

Cuando se produce un nuevo evento, debe enviarse a todos los programas MQL que se ejecutan en el gráfico correspondiente. Debido al modelo de ejecución de un solo hilo de los programas MQL (véase la sección Hilos), puede ocurrir que el siguiente evento llegue cuando todavía se está procesando el anterior. Para estos casos, el terminal mantiene una cola de eventos para cada programa MQL interactivo. Todos los eventos que contiene se procesan uno tras otro por orden de recepción.

Las colas de eventos tienen un tamaño limitado. Por lo tanto, un programa escrito de forma irracional puede provocar un desbordamiento de la cola a causa de acciones lentas. En caso de desbordamiento, los nuevos eventos se descartan sin ser puestos en cola.

No procesar los eventos lo suficientemente rápido puede afectar negativamente a la experiencia del usuario o a la calidad de los datos (imagine que graba cambios de Profundidad de Mercado y se salta algunos mensajes). Para resolver este problema, puede buscar algoritmos más eficientes o utilizar el funcionamiento en paralelo de varios programas MQL interconectados (por ejemplo, asignar cálculos a un indicador, y leer sólo datos ya preparados en un Asesor Experto).

Hay que tener en cuenta que el terminal no coloca todos los eventos en la cola, sino que opera de forma selectiva. Algunos tipos de eventos se procesan según el principio «no más de un evento de este tipo en la cola». Por ejemplo, si ya existe el evento *OnTick* en la cola, o se está procesando, entonces un nuevo evento *OnTick* no se pone en cola. Si ya existe el evento *OnTimer* o un evento de cambio de gráfico en la cola, los nuevos eventos de estos tipos también se descartan (ignoran). Se trata de una instancia específica del programa. Otros programas menos «ocupados» recibirán este mensaje.

No proporcionamos una lista completa de dichos tipos de eventos porque esta optimización al omitir eventos «solapados» puede ser modificada por los desarrolladores del terminal.

El enfoque para organizar el trabajo de los programas en respuesta a eventos entrantes se denomina dirigido por eventos. También se puede denominar asíncrono, ya que la puesta en cola de un evento en la cola del programa y su extracción (junto con el procesamiento) se producen en momentos diferentes (idealmente, separados por un intervalo microscópico, pero lo ideal no siempre es alcanzable). Sin embargo, de los cuatro tipos de programas MQL, sólo los indicadores y los Asesores Expertos siguen plenamente este enfoque. Los scripts y servicios tienen, de hecho, sólo la función principal, que, cuando es llamada, debe realizar rápidamente la acción requerida y completarla o iniciar un bucle sin fin para mantener alguna actividad (por ejemplo, leer datos de la red) hasta que el usuario se detenga. Hemos visto ejemplos de este tipo de bucles:

```
while(!IsStopped())
{
  useful code
   ...
   Sleep(...);
}
```

En este tipo de bucles es importante no olvidar utilizar *Sleep* con algún periodo para compartir los recursos de la CPU con otros programas. El valor del periodo se selecciona en función de la intensidad estimada de la actividad que se está llevando a cabo.

Este enfoque puede denominarse cíclico o síncrono, o incluso en tiempo real, ya que se puede seleccionar el periodo de reposo para proporcionar una frecuencia constante de tratamiento de datos, por ejemplo:

```
int rhythm = 100; // 100 ms, 10 times per sec
while(!IsStopped())
{
   const int start = (int)GetTickCount();
  useful code
   ...
   Sleep(rhythm - ((int)GetTickCount() - start));
}
```

Por supuesto, el «código útil» debe caber en el marco asignado.

En cambio, con el enfoque por eventos, no se sabe de antemano cuándo funcionará la próxima vez el trozo de código (manejador). Por ejemplo, en un mercado rápido, durante las noticias, los ticks pueden llegar en lotes, y por la noche pueden estar ausentes durante segundos enteros. En el caso límite, tras el último tick del viernes por la noche, el siguiente cambio de precio de algún instrumento financiero no puede emitirse hasta el lunes por la mañana, por lo que los eventos *OnTick* estarán ausentes durante dos días. En otras palabras: en los eventos (y en los momentos de activación de los manejadores de eventos) no hay regularidad, no hay un horario claro.

Pero si es necesario, puede combinar ambos viajes. En concreto, el evento temporizador (*OnTimer*) proporciona regularidad, y el desarrollador puede generar periódicamente eventos personalizados para un gráfico dentro de un bucle (por ejemplo, la intermitencia de una etiqueta de advertencia).

5.1.4 Funciones de inicio y parada de programas de varios tipos

En programación, el término inicialización se utiliza en muchos contextos diferentes. En MQL5 también existe cierta ambigüedad. En la sección Inicialización hemos utilizado ya esta palabra para referirnos a la fijación de los valores iniciales de las variables. Luego discutimos el evento de inicialización *OnInit* en indicadores y Asesores Expertos. Aunque el significado de ambas inicializaciones es similar (llevar el programa a un estado de trabajo), en realidad significan diferentes etapas de la preparación de un programa MQL para el lanzamiento: sistema y aplicación.

El ciclo de vida de un programa MQL terminado puede representarse mediante los siguientes pasos principales:

1. Carga: lectura de un programa desde un archivo en la memoria del terminal: incluye instrucciones, datos predefinidos (literales), recursos y bibliotecas. Aquí es donde entran en juego las directivas *#property*.
2. Asignación de memoria para variables globales y establecimiento de sus valores iniciales: es la inicialización del sistema realizada por el tiempo de ejecución. Recordemos que en la sección Inicialización, mientras estudiábamos paso a paso el inicio del programa bajo el depurador, vimos que la entrada @global_initializations estaba en la pila. Este era el bloque de código para este elemento, que fue creado implícitamente por el compilador. Si el programa utiliza objetos globales de clases o estructuras, se llamará a sus constructores en esta fase.
3. Llamada al manejador de eventos *OnInit* (si existe): lleva a cabo una inicialización aplicada de nivel superior, por lo que cada programa la realiza de forma independiente, según sea necesario. Por ejemplo, puede ser la asignación dinámica de memoria para arrays de objetos, para los que, por una razón u otra, sea necesario utilizar constructores paramétricos en lugar de constructores predeterminados. Como sabemos, la asignación automática de memoria para arrays utiliza sólo constructores predeterminados, y por lo tanto no se pueden inicializar en el paso anterior (2). También puede ser abrir archivos, llamar a funciones integradas de la API para activar los modos de gráfico necesarios, etc.

4. Bucle hasta que el usuario cierre el programa o el terminal o realice cualquier otra acción que requiera reinicialización (ver más adelante):
   - llamando a otros manejadores a medida que se produzcan los eventos apropiados.
5. Llamada al manejador de eventos *OnDeinit* (si existe) al detectar un intento de cerrar el programa por parte del usuario o programáticamente (la función correspondiente ExpertRemove sólo está disponible en Asesores Expertos y scripts).
6. Finalización: liberar la memoria asignada y otros recursos que el programador no consideró necesario liberar en *OnDeinit*. Si el programa utiliza POO, los destructores de objetos globales y estáticos se llaman aquí.
7. Descarga del programa.

Los scripts y servicios a priori no tienen manejadores *OnInit* y *OnDeinit*, y por lo tanto los pasos 3 y 5 no existen para ellos, y el paso 4 degenera en una sola llamada a *OnStart*.

La inicialización del sistema (paso 2) es inseparable de la carga, es decir, siempre a continuación de ella. La finalización siempre precede a la descarga. No obstante, los indicadores y los Asesores Expertos pasan por las etapas de carga y descarga de manera diferente en distintas situaciones. Por lo tanto, las llamadas a *OnInit* y *OnDeinit* (pasos 3 y 5) son los puntos de referencia en los que es posible proporcionar una inicialización y desinicialización aplicada coherente de Asesores Expertos e indicadores.

La carga de indicadores y Asesores Expertos se realiza en los siguientes casos:

Caso		
El usuario inicia el programa en el gráfico	+	+
Lanzamiento del terminal (si el programa se estaba ejecutando en el gráfico antes del cierre anterior del terminal)	+	+
Carga de una plantilla (si la plantilla contiene un programa adjunto al gráfico)	+	+
Modificación del perfil (si el programa está vinculado a uno de los gráficos del perfil)	+	+
Tras la recompilación correcta, si el programa se adjuntó al gráfico	+	+
Cambio de la cuenta activa	+	+
Modificación del símbolo o el período del gráfico al que está vinculado el indicador	+	-
Modificación de los parámetros de entrada del indicador	+	-
Conexión a la cuenta (autorización), aunque el número de cuenta no haya cambiado	-	+

De forma más compacta, se puede formular la siguiente regla: los Asesores Expertos no pasan por el ciclo de vida completo, es decir, no se recargan cuando cambia el símbolo o marco temporal del gráfico, así como cuando cambian los parámetros de entrada.

Por lo tanto, puede observarse una asimetría similar al descargar los programas. Las razones para descargar indicadores y Asesores Expertos son:

Caso		
Eliminación del programa del gráfico	+	+
Cierre del terminal (cuando el programa está conectado al gráfico)	+	+
Carga de una plantilla en el gráfico en el que se ejecuta el programa	+	+
Cierre del gráfico en el que se está ejecutando el programa	+	+
Modificación del perfil si el programa está vinculado a uno de los gráficos del perfil	+	+
Modificación de la cuenta a la que está conectado el terminal	+	+
Modificación del símbolo y/o el periodo del gráfico al que está vinculado el indicador	+	-
Modificación de los parámetros de entrada del indicador	+	-
Vinculación del mismo u otro EA al gráfico en el que ya se está ejecutando el EA actual	-	+
Llamada a la función ExpertRemove	-	+

El motivo de la desinicialización se puede encontrar en el programa utilizando la función *UninitializeReason* o la bandera *_UninitReason* (véase la sección Comprobar el estado y el motivo de parada de un programa MQL).

Tenga en cuenta que cuando cambia el símbolo o el marco temporal del gráfico, así como cuando cambia los parámetros de entrada, el Asesor Experto permanece en memoria, es decir, los pasos 6-7 (finalización y descarga) y los pasos 1-2 (carga y asignación de memoria primaria) no se ejecutan, por lo tanto, los valores de las variables globales y estáticas no se restablecen. En este caso, los manejadores *OnDeinit* y *OnInit* son llamados de forma secuencial en el antiguo y en el nuevo símbolo o marco temporal, respectivamente (o en la antigua y en la nueva configuración).

Una consecuencia de que las variables globales no se borren en los Asesores Expertos es que el código de desinicialización *_UninitReason* permanece inalterado para su análisis en el manejador *OnInit*. El nuevo código se escribirá en la variable sólo en caso del siguiente evento, justo antes de la llamada a *OnDeinit*.

Todos los eventos recibidos para el Asesor Experto antes del final de la función *OnInit* se omiten.

Cuando se inicia el programa MQL por primera vez, aparece el cuadro de diálogo de configuración entre los pasos 1 y 2. Al cambiar los parámetros de entrada, el cuadro de diálogo de configuración se encaja en el bucle general de diferentes maneras en función del tipo de programa: para los indicadores aparece todavía antes del paso 2, y para los Asesores Expertos, antes del paso 3.

El libro va acompañado de un indicador y una plantilla de Asesor Experto que lleva por título *LifeCycle.mq5* y registra los pasos globales de inicialización o finalización en manejadores *OnInit/OnDeinit*. Coloque programas en el gráfico y vea qué eventos se producen en respuesta a diversas acciones del usuario: carga o descarga, cambio de parámetros, cambio de símbolos o marcos temporales.

El script se carga sólo cuando se añade al gráfico. Si un script se está ejecutando en bucle, recompilarlo no provoca un reinicio.

El servicio se carga y descarga mediante los comandos del menú contextual de la interfaz del terminal. Cuando se recompila un servicio que ya está en funcionamiento, éste se reinicia. Recuerde que las instancias activas de los servicios se cargan automáticamente cuando se inicia el terminal, y se descargan cuando se cierra.

En las dos secciones siguientes examinaremos las características del lanzamiento de diferentes programas MQL a nivel de manejadores de eventos.

5.1.5 Eventos de referencia de indicadores y Asesores Expertos: OnInit y OnDeinit

En los programas MQL interactivos (indicadores y Asesores Expertos) el entorno genera dos eventos para preparar el lanzamiento (*OnInit*) y la parada (*OnDeinit*). No existe este tipo de eventos en scripts y servicios porque no aceptan eventos asíncronos: después de pasar el control a su único manejador de eventos *OnStart* y hasta el final del trabajo, el contexto de ejecución del hilo de script/servicio se encuentra en el código del programa MQL. En cambio, para los indicadores y los Asesores Expertos, el curso normal de trabajo supone que el entorno llamará repetidamente a sus funciones específicas de manejo de eventos (hablaremos de ellas en las secciones sobre indicadores y Asesores Expertos), y cada vez, una vez emprendidas las acciones necesarias, los programas devolverán el control al terminal para la espera en reposo de nuevos eventos.

```
int OnInit()
```

La función *OnInit* es un controlador del evento del mismo nombre, que se genera después de cargar un Asesor Experto o un indicador. La función sólo puede definirse en función de las necesidades.

La función debe devolver uno de los valores de la enum ENUM_INIT_RETCODE.

Identificador	Descripción
INIT_SUCEDED	Inicialización correcta, puede continuar la ejecución del programa; corresponde al valor 0
INIT_FAILED	Inicialización fallida, no puede continuar la ejecución debido a errores fatales (por ejemplo, no fue posible crear un archivo o un indicador auxiliar); valor 1
INIT_PARAMETERS_INCORRECT	Conjunto incorrecto de parámetros de entrada, la ejecución del programa es imposible
INIT_AGENT_NOT_SUITABLE	Código específico para trabajar en el probador: por alguna razón, este agente no es adecuado para las pruebas (por ejemplo, no tiene suficiente RAM, no es compatible con OpenCL, etc.)

Si *OnInit* devuelve cualquier código de retorno distinto de cero, significa que la inicialización no ha tenido éxito, y entonces se genera el evento *Deinit*, con el código de motivo de desinicialización REASON_INITFAILED (véase más abajo).

La función *OnInit* puede declararse con un tipo de resultado *void*: en este caso, la inicialización siempre se considera correcta.

> En el manejador *OnInit* es importante comprobar que toda la información necesaria del entorno está presente, y si no está disponible, aplazar las acciones preparatorias para los próximos eventos de llegada de ticks o temporizadores. La cuestión es que, cuando se inicia el terminal, el evento *OnInit* a menudo se dispara antes de que se establezca una conexión con el servidor y, por lo tanto, muchas propiedades de los instrumentos financieros y de una cuenta de trading siguen siendo desconocidas. En concreto, el valor de un pip de un símbolo concreto puede devolverse como cero.

void OnDeinit(const int reason)

La función *OnDeinit* (si está definida) se llama cuando se desinicializa el Asesor Experto o el indicador. La función es opcional.

El parámetro *reason* contiene el código de motivo de desinicialización. Los valores posibles se muestran en la siguiente tabla:

Constante	Valor	Descripción
REASON_PROGRAM	0	El Asesor Experto ha detenido la operación por llamada a la función *ExpertRemove*
REASON_REMOVE	1	Programa eliminado del gráfico
REASON_RECOMPILE	2	Programa recompilado
REASON_CHARTCHANGE	3	Periodo o símbolo de gráfico modificado
REASON_CHARTCLOSE	4	Gráfico cerrado
REASON_PARAMETERS	5	Parámetros de entrada modificados
REASON_ACCOUNT	6	Se ha activado otra cuenta o se ha producido una reconexión al servidor de trading debido a un cambio en la configuración de la cuenta.
REASON_TEMPLATE	7	Plantilla de gráfico diferente aplicada
REASON_INITFAILED	8	*OnInit* el manejador ha devuelto un valor no nulo
REASON_CLOSE	9	Terminal cerrado

El mismo código puede obtenerse en cualquier parte del programa utilizando la función *UninitializeReason* si la bandera de parada *_StopFlag* está activada en el programa MQL.

El archivo *AllInOne.mqh* tiene la clase *Finalizer* que le permite «enganchar» el código de desinicialización en el destructor a través de la llamada *UninitializeReason*. Debemos obtener el mismo valor en el manejador *OnDeinit*.

```
class Finalizer
{
   static const Finalizer f;
public:
   ~Finalizer()
   {
      PRTF(EnumToString((ENUM_DEINIT_REASON)UninitializeReason()));
   }
};

static const Finalizer Finalizer::f;
```

Para facilitar la traducción de los códigos en una representación de cadena (nombres de motivos) mediante *EnumToString*, en el archivo *Uninit.mqh* se describe la enumeración ENUM_DEINIT_REASON con constantes de la tabla anterior. El registro mostrará entradas como la siguiente:

```
OnDeinit DEINIT_REASON_REMOVE
EnumToString((ENUM_DEINIT_REASON)UninitializeReason())=DEINIT_REASON_REMOVE / ok
```

Cuando se modifica el símbolo o el marco temporal del gráfico en el que se encuentra el indicador, éste se descarga y se vuelve a cargar. En este caso, la secuencia de activación del evento *OnDeinit* en la copia antigua y *OnInit* no se define en la copia nueva. Esto se debe a las particularidades del procesamiento asíncrono de eventos por parte del terminal. En otras palabras: puede que no sea del todo lógico que se cargue e inicialice una nueva copia antes de que se descargue por completo la

anterior. Si el indicador realiza algún ajuste del gráfico en *OnInit* (por ejemplo, crea un objeto gráfico), entonces, sin adoptar medidas especiales, la copia descargada puede «limpiar» inmediatamente el gráfico (borrar el objeto, considerándolo como propio). En el caso concreto de los objetos gráficos, existe una solución particular: se puede dar a los objetos nombres que incluyan los prefijos de símbolo y marco temporal (así como la suma de comprobación de los valores de las variables de entrada), pero en el caso general no funcionará. Para darle una solución universal al problema debería implementarse algún tipo de mecanismo de sincronización, por ejemplo, en variables globales o recursos.

Al probar los indicadores en el probador, los desarrolladores de MetaTrader 5 decidieron no generar el evento *OnDeinit*. Su idea es que el indicador puede crear algunos objetos gráficos, que normalmente elimina en el manejador *OnDeinit*, pero al usuario le gustaría verlos una vez finalizada la prueba. De hecho, el autor de un programa MQL puede, si lo desea, proporcionar un comportamiento similar y dejar los objetos con una comprobación positiva del modo *MQLInfoInteger(MQL_TESTER)*. Esto es extraño ya que el manejador *OnDeinit* es llamado después de la prueba del Asesor Experto, y el Asesor Experto puede borrar objetos de la misma manera en *OnDeinit*. Ahora, sólo para los indicadores, resulta que el comportamiento regular del manejador *OnDeinit* no se puede garantizar en el probador. Además, no se realizan otras finalizaciones; por ejemplo, no se llama a los destructores de objetos globales.

Por lo tanto, si necesita realizar un cálculo estadístico, guardar un archivo u otra acción después de la ejecución de prueba que originalmente estaba prevista para el indicador de *OnDeinit*, tendrá que transferir los algoritmos del indicador al Asesor Experto.

5.1.6 Función principal de scripts y servicios: OnStart

Los programas de utilidades (scripts y servicios) se ejecutan en el terminal llamando a su función de gestión de eventos únicos *OnStart*.

void OnStart()

La función no tiene parámetros y no devuelve ningún valor. Sólo sirve como punto de entrada al programa de aplicación desde el lado del terminal.

Los scripts están pensados, por regla general, para acciones puntuales realizadas en un gráfico (más adelante estudiaremos todas las posibilidades que ofrece la API de gráficos). Por ejemplo, se puede utilizar un script a fin de establecer una parrilla de órdenes o, a la inversa, para cerrar todas las posiciones abiertas rentables, para aplicar automáticamente marcas con objetos gráficos o para ocultar temporalmente todos los objetos.

En los scripts puede utilizar acciones constantes envueltas en un bucle infinito, en el que, como se ha mencionado anteriormente, debe comprobar siempre la señal de stop (*_StopFlag*) y liberar periódicamente el procesador (*Sleep*). Hay que recordar aquí que cuando apague y encienda el terminal, el script tendrá que ejecutarse de nuevo.

Por lo tanto, para esa actividad constante, si no está directamente relacionada con el horario, es mejor utilizar el servicio. La técnica estándar en la implementación del servicio es simplemente un bucle «infinito».

En las partes anteriores del libro, casi todos los ejemplos se implementaban como scripts. Un ejemplo de servicio es el programa *GlobalsWithCondition.mq5* de la sección Sincronización de programas mediante variables globales. Veremos otro ejemplo en la siguiente sección sobre la detención de Asesores Expertos y scripts utilizando la función *ExpertRemove*.

5.1.7 Eliminación programática de Asesores Expertos y scripts: ExpertRemove

Si es necesario, el desarrollador puede organizar la parada y descarga de programas MQL de dos tipos: Asesores expertos y scripts. Para ello se utiliza la función *ExpertRemove*.

void ExpertRemove()

La función no tiene parámetros y no devuelve ningún valor. Envía una petición al entorno de ejecución del programa MQL para borrar el programa actual. De hecho, esto lleva a fijar la bandera *_StopFlag* y detiene la recepción (y el procesamiento) de todos los eventos posteriores. Después de eso, el programa dispone de tres segundos para completar correctamente su trabajo: liberar recursos, romper bucles en algoritmos, etc. Si el programa no lo hace, se descargará forzosamente, con la pérdida de datos intermedios.

Esta función no funciona en indicadores y servicios (el programa sigue ejecutándose).

Para cada llamada a la función, el registro contendrá la entrada «ExpertRemove() function called».

Esta función se utiliza principalmente en los Asesores Expertos que no pueden interrumpirse de ninguna otra forma. En el caso de los scripts suele ser más fácil romper el bucle (si lo hay) con la sentencia *break*. Pero si los bucles están anidados, o el algoritmo utiliza muchas llamadas de unas funciones a otras, es más fácil tener en cuenta la bandera de parada en diferentes niveles en las condiciones para continuar los cálculos, y en caso de una situación errónea, establecer esta bandera utilizando *ExpertRemove*. Si no utiliza esta bandera integrada, en todo caso, tendría que introducir una variable global con el mismo propósito.

El script *ScriptRemove.mq5* proporciona el ejemplo de uso *ExpertRemove*.

La clase *ProblemSource* emula un problema potencial en el funcionamiento del algoritmo, lo que lleva a la necesidad de descargar el script. *ExpertRemove* se llama aleatoriamente en su constructor.

```
class ProblemSource
{
public:
   ProblemSource()
   {
      // simulating a problem during object creation, for example,
      // with the capture of some resources, such as a file, etc.
      if(rand() > 20000)
      {
         ExpertRemove(); // will set _StopFlag to true
      }
   }
};
```

Más adelante se crean objetos de esta clase a nivel global y dentro de la función auxiliar.

```
ProblemSource global; // object may throw an error

void SubFunction()
{
   ProblemSource local; //object may throw an error
   // simulate some work (we need to check the integrity of the object!)
   Sleep(1000);
}
```

Ahora utilizamos *SubFunction* en la operación *OnStart*, dentro del bucle con la condición *IsStopped*.

```
void OnStart()
{
   int count = 0;
   // loop until stopped by the user or the program itself
   while(!IsStopped())
   {
      SubFunction();
      Print(++count);
   }
}
```

He aquí un ejemplo de registro (cada ejecución será diferente debido a la aleatoriedad):

```
1
2
3
ExpertRemove() function called
4
```

Tenga en cuenta que, si se produce un error al crear el objeto global, el bucle nunca se ejecutará.

Dado que los Asesores Expertos pueden funcionar en el probador, la función *ExpertRemove* también puede utilizarse en el probador. Su efecto depende del lugar de la llamada a la función. Si esto se hace dentro del manejador *OnInit*, la función cancelará la prueba, es decir, una ejecución del probador en el conjunto actual de los parámetros del Asesor Experto. Dicha terminación se trata como un error de inicialización. Si se llama a *ExpertRemove* en cualquier otro lugar del algoritmo, la prueba del Asesor Experto se interrumpirá antes de tiempo, pero se procesará de forma regular, con las llamadas *OnDeinit* y *OnTester*. En este caso se obtendrán las estadísticas de trading acumuladas y el valor del criterio de optimización, teniendo en cuenta que el tiempo de servidor emulado *TimeCurrent* no alcanza la fecha de finalización en la configuración del probador.

5.2 Scripts y servicios

En este capítulo, resumiremos y presentaremos toda la información técnica sobre los scripts y servicios que ya hemos empezado a conocer en las partes anteriores del libro.

Los scripts y los servicios tienen los mismos principios para organizar y ejecutar el código del programa. Como sabemos, su función principal *OnStart* también es la única. Los scripts y servicios no pueden procesar otros eventos.

Sin embargo, hay un par de diferencias significativas. Los scripts se ejecutan en el contexto de un gráfico y tienen acceso directo a sus propiedades a través de variables integradas tales como *_Symbol*,

_Period y *_Point*, entre otras. Los estudiaremos en la sección Propiedades de los gráficos. Los servicios, por su parte, trabajan por su cuenta, sin estar vinculados a ninguna ventana, aunque tienen la capacidad de analizar todos los gráficos mediante funciones especiales (las mismas *funciones de gráficos* puede utilizarse en otros tipos de programas: scripts, indicadores y Asesores Expertos).

Por otra parte, las instancias creadas del servicio son restauradas automáticamente por el terminal en las siguientes sesiones. En otras palabras: el servicio, una vez iniciado, permanece siempre en funcionamiento hasta que el usuario lo detiene. Por el contrario, el script se borra cuando se apaga el terminal o se cierra el gráfico.

> Tenga en cuenta que el servicio se ejecuta en el terminal, como todos los demás tipos de programas MQL, y por lo tanto cerrar el terminal también detiene el servicio. El servicio activo se reanudará la próxima vez que inicie el terminal. El funcionamiento ininterrumpido de los programas MQL sólo puede garantizarse mediante un terminal en funcionamiento constante, por ejemplo, en un VPS.

En scripts y servicios, puede establecer las propiedades generales de los programas utilizando las directivas *#property*. Además de ellas, hay propiedades que son específicas de los scripts y los servicios; hablaremos de ellas en las dos secciones siguientes.

Los scripts que se están ejecutando actualmente en los gráficos aparecen en la misma lista que muestra los Asesores Expertos en ejecución: en el cuadro de diálogo *Experts* abierto con el comando *Lista de expertos* del menú contextual del gráfico. A partir de ahí, pueden ser retirados a la fuerza del gráfico.

Los servicios sólo pueden gestionarse desde la ventana *Navigator*.

5.2.1 Scripts

Un script es un programa MQL con el manejador único *OnStart*, siempre y cuando no haya una directiva *#property service* (de lo contrario se obtiene un servicio; véase la siguiente sección).

Por defecto, el script comienza a ejecutarse inmediatamente cuando se coloca en el gráfico. El desarrollador puede pedirle al usuario que confirme el inicio añadiendo la directiva *#property script_show_confirm* al principio del archivo. En este caso, el terminal mostrará un mensaje con la pregunta «¿Está seguro de que desea ejecutar 'programa' en el gráfico 'símbolo, marco temporal'?» y los botones *Sí* y *No*.

Los scripts, al igual que otros programas, pueden tener variables de entrada. Sin embargo, en el caso de los scripts, el cuadro de diálogo de introducción de parámetros no se muestra por defecto, aunque el script defina *input*s. Para asegurarse de que el cuadro de diálogo de propiedades se abre antes de ejecutar el script, debe aplicarse la directiva *#property script_show_inputs*. Tiene prioridad sobre *script_show_confirm*, es decir, la salida del cuadro de diálogo desactiva la petición de confirmación (ya que el propio cuadro de diálogo actúa con un papel similar). La directiva llama a un cuadro de diálogo aunque no haya variables de entrada. Puede utilizarse para mostrarle al usuario la descripción y la versión del producto (se muestran en la pestaña *Common*).

En la siguiente tabla se muestran las opciones de combinación para la directiva *#property* y su efecto en el programa.

Directiva Efecto	script_show_confirm	script_show_inputs
Lanzamiento inmediato	No	No
Solicitud de confirmación	Sí	No
Apertura del cuadro de diálogo de propiedades	irrelevante	Sí

Un ejemplo sencillo de script con directivas se encuentra en el archivo *ScriptNoComment.mq5*. El objetivo del script es el siguiente: a veces, los programas MQL dejan comentarios innecesarios en la esquina superior izquierda del gráfico. Estos comentarios se almacenan en archivos chr junto con el gráfico, por lo que se restauran incluso después de reiniciar el terminal. Este script le permite borrar un comentario o establecerlo en un valor arbitrario. Si usted *Assign hotkey* a un script utilizando el comando del menú contextual *Navegador*, será posible limpiar el comentario del gráfico actual con un solo clic.

Originalmente, las directivas *script_show_confirm* y *script_show_inputs* se desactivan al convertirse en comentarios en línea. Puede experimentar con diferentes combinaciones de directivas eliminando los comentarios de uno en uno o al mismo tiempo.

```
//#property script_show_confirm
//#property script_show_inputs

input string Text = "";

void OnStart()
{
   Comment(""); // clean up the comment
}
```

5.2.2 Servicios

Un servicio es un programa MQL con un único manejador *OnStart* y la directiva *#property service*.

Recuerde que, tras la compilación correcta del servicio, debe crear y configurar su instancia (una o varias) mediante el comando *Añadir servicio* en el menú contextual de la ventana *Navegador*.

Como ejemplo de un servicio, vamos a resolver un pequeño problema práctico que suele surgir entre los desarrolladores de programas MQL. Muchos de ellos practican la vinculación de sus programas al número de cuenta del usuario. No se trata necesariamente de un producto de pago, sino que puede referirse a la distribución entre amigos y conocidos para recopilar estadísticas o configuraciones de éxito. Al mismo tiempo, el usuario puede registrar cuentas demo además de una cuenta real operativa. La vida útil de estas cuentas suele ser limitada, por lo que resulta bastante incómodo actualizar el enlace de las mismas cada dos semanas. Para ello, es necesario editar el código fuente, compilar y volver a enviar el programa.

En lugar de ello, podemos desarrollar un servicio que registre en variables globales (o archivos) los números de cuentas en las que se ha implementado una conexión con éxito desde el terminal dado.

La tecnología de vinculación se basa en el cifrado por pares (o, alternativamente, hashing) de los números de cuenta: la cuenta de inicio de sesión antigua y la cuenta de inicio de sesión nueva. La

cuenta anterior debe ser una cuenta maestra (a la que se «emite» el enlace condicional) para que la firma común del par extienda los derechos de uso del producto a la nueva cuenta. La clave es un secreto conocido sólo dentro de los programas (se supone que todos ellos se suministran de forma cerrada y compilada). El resultado de la operación será una cadena con el formato *Base64*. La implementación utiliza funciones de la API de MQL5, algunas de las cuales aún están pendientes de estudio; en concreto, la obtención de un número de cuenta a través de la función de encriptación *AccountInfoInteger* y *CryptEncode*. La conexión con el servidor se comprueba mediante la función *TerminalInfoInteger* (véase Comprobación de las conexiones de red).

El servicio no está obligado a saber qué cuentas son principales y cuáles son adicionales. Sólo necesita «firmar» de manera especial pares de cuentas que hayan iniciado sesión sucesivamente. Pero un programa de aplicación específico debería complementar el proceso de comprobación de su «licencia»: además de comparar la cuenta corriente con la cuenta maestra, debería repetir el algoritmo de servicio creando un par [cuenta maestra; cuenta corriente], calculando la firma cifrada para él y comprobando si se encuentra entre las variables globales.

Será posible robar dicha licencia transfiriéndola a otro ordenador sólo si se conecta a la misma cuenta en modo de negociación (no inversor). Un usuario sin escrúpulos, por supuesto, puede crear cuentas demo para otras personas. Por lo tanto, es deseable mejorar la protección. En la implementación actual, la variable global simplemente se hace temporal, es decir, se borra junto con el final de la sesión de terminal, pero esto no impide su posible copia.

Como medidas adicionales, es posible, por ejemplo, cifrar la hora de su creación en la firma y prever la caducidad de los derechos cada día (o con otra frecuencia). Otra opción es generar un número aleatorio cuando se inicia el servicio y añadirlo a la información firmada junto con los números de cuenta. Este número sólo se conoce dentro del servicio, pero puede traducirlo a los programas MQL interesados en los gráficos mediante la función *EventChartCustom*. Así, la firma seguirá siendo válida en esta instancia del terminal hasta el final de la sesión. Cada sesión generará y enviará un nuevo número aleatorio, por lo que no funcionará para otros terminales. Por último, la opción más sencilla y cómoda sería probablemente añadir a la firma la hora de inicio del sistema: *(TimeLocal() - GetTickCount() / 1000)* o su derivado.

De los distintos tipos de programas MQL, sólo algunos siguen ejecutándose entre cambios de cuenta y permiten aplicar este esquema de protección. Dado que es necesario proteger de manera uniforme los programas MQL de todo tipo, incluidos los indicadores y los Asesores Expertos (que se recargan cuando se cambia la cuenta), tiene sentido confiar esta tarea a un servicio. A continuación, el servicio, que se ejecuta constantemente desde que se carga el terminal hasta que se cierra, controlará los inicios de sesión y generará firmas de autorización.

El código fuente del servicio figura en el archivo *MQL5/Services/MQL5Book/p5/ServiceAccount.mq5*. Los parámetros de entrada especifican la cuenta maestra y el prefijo de las variables globales en las que se almacenarán las firmas. En los programas reales, las listas de cuentas maestras deberían estar codificadas en el código fuente, y en lugar de variables globales, es mejor utilizar archivos en la carpeta *Common* para abarcar también el probador.

```
#property service

input long MasterAccount = 123456789;
input string Prefix = "!A_";
```

La función principal del servicio realiza su trabajo de la siguiente manera: en un bucle sin fin con pausas de 1 segundo, rastreamos los cambios de cuenta y guardamos el último número, creamos una firma para el par y la escribimos en una variable global. La firma se crea mediante la función *Cipher*.

```
void OnStart()
{
   static long account = 0; // previous login

   for(; !IsStopped(); )
   {
      // require connection, successful login and full access (not investor)
      const bool c = TerminalInfoInteger(TERMINAL_CONNECTED)
                  && AccountInfoInteger(ACCOUNT_TRADE_ALLOWED);
      const long a = c ? AccountInfoInteger(ACCOUNT_LOGIN) : 0;

      if(account != a) // account changed
      {
         if(a != 0) // current account
         {
            if(account != 0) // previous account
            {
               // transfer authorization from one to another
               const string signature = Cipher(account, a);
               PrintFormat("Account %I64d registered by %I64d: %s",
                  a, account, signature);
               // saving a record about the connection of accounts
               if(StringLen(signature) > 0)
               {
                  GlobalVariableTemp(Prefix + signature);
                  GlobalVariableSet(Prefix + signature, account);
               }
            }
            else // the first account is authorized, now waiting for the second one
            {
               PrintFormat("New account %I64d detected", a);
            }
            // remember the last active account
            account = a;
         }
      }
      Sleep(1000);
   }
}
```

La función *Cipher* utiliza una unión especial *ByteOverlay2* para representar un par de números de cuenta (de tipo *long*) como un array de bytes, que se pasa para su cifrado en *CryptEncode* (aquí se elige el método de cifrado CRYPT_DES, pero puede sustituirse por el hashing CRYPT_AES128, CRYPT_AES256 o simplemente CRYPT_HASH_SHA256 (con el secreto como «sal»), si no se requiere la recuperación de información de la «firma»).

```
template<typename T>
union ByteOverlay2
{
   T values[2];
   uchar bytes[sizeof(T) * 2];
   ByteOverlay2(const T v1, const T v2) { values[0] = v1; values[1] = v2; }
};

string Cipher(const long data1, const long data2)
{
   // TODO: replace the secret with your passphrase
   // TODO: CRYPT_AES128/CRYPT_AES256 methods require 16/32 byte arrays
   const static uchar secret[] = {'S', 'E', 'C', 'R', 'E', 'T', '0'};
   ByteOverlay2<long> bo(data1, data2);
   uchar result[];
   if(CryptEncode(CRYPT_DES, bo.bytes, secret, result) > 0)
   {
      uchar dummy[], text[];
      if(CryptEncode(CRYPT_BASE64, result, dummy, text) > 0)
      {
         return CharArrayToString(text);
      }
   }
   return NULL;
}
```

Entonces, cualquier programa en el terminal puede comprobar si hay «licencias» para la cuenta corriente en las variables globales. Para ello se utilizan las funciones *CheckAccounts* y *IsCurrentAccountAuthorizedByMaster*, que se muestran en el servicio sólo a efectos de demostración.

Las funciones *CheckAccounts* realizan una comprobación de todas las cuentas maestras codificadas para encontrar las que coinciden con la actual.

```
bool CheckAccounts()
{
   const long accounts[] = {MasterAccount}; // TODO: to fill array with constants
   for(int i = 0; i < ArraySize(accounts); ++i)
   {
      if(IsCurrentAccountAuthorizedByMaster(accounts[i])) return true;
   }
   return false;
}
```

IsCurrentAccountAuthorizedByMaster toma el número de una cuenta maestra como parámetro, recrea una «firma» para ella en un par con la cuenta actual y analiza las coincidencias.

```
bool IsCurrentAccountAuthorizedByMaster(const long data)
{
   const long a = AccountInfoInteger(ACCOUNT_LOGIN);
   if(a == data) return true; // direct match
   const string s = Cipher(data, a); // recalculating "signature"
   if(a != 0 && GlobalVariableGet(Prefix + s) == a)
   {
      Print("Sub-License is active: ", s);
      return true;
   }
   return false;
}
```

Supongamos que se permite la ejecución de programas en la cuenta 123456789 y que ésta actualmente está activa. Al iniciarse, el servicio responderá con una entrada de registro:

```
New account 123456789 detected
```

Si a continuación cambiamos el número de cuenta, por ejemplo, a 5555555, obtendremos la siguiente firma:

```
Account 5555555 registered by 123456789: jdVKxUswBiNlZzDAnV3yxw==
```

Si detenemos e iniciamos de nuevo el servicio, veremos la verificación de la cuenta 5555555 en acción (llamando a la función *CheckAccounts* incrustada para demostración al principio *OnStart*).

```
Sub-License is active: jdVKxUswBiNlZzDAnV3yxw==
Account 123456789 registered by 5555555: ZWcwwJ1d8seN1UrFSzAGIw==
```

La licencia ha funcionado para la nueva cuenta. Si vuelve a cambiar, se generará un «pase» de la cuenta actual a la anterior (esto es consecuencia de que el servicio no «sabe» qué cuentas son principales y cuáles temporales, y lo más probable es que dicha «firma» no sea necesaria en los programas).

Para autorizar indirectamente una nueva cuenta, tendrá que volver a conectarse a la cuenta maestra y sólo entonces cambiar a la nueva: esto creará otra variable global con el par cifrado [cuenta maestra; nueva cuenta].

Esta versión del servicio no comprueba que la cuenta maestra sea real y la cuenta dependiente sea demo. Cada una de estas restricciones se puede añadir.

5.2.3 Restricciones para scripts y servicios

Todas las funciones incluidas en el grupo para trabajar con indicadores están prohibidas en scripts y servicio. Estas funciones se describirán en el correspondiente capítulo:

- *SetIndexBuffer*
- *IndicatorSetDouble*
- *IndicatorSetInteger*
- *IndicatorSetString*
- *PlotIndexSetDouble*
- *PlotIndexSetInteger*

- *PlotIndexSetString*
- *PlotIndexGetInteger*

Además, en scripts y servicios, no tiene sentido utilizar las funciones de manejador *OnTimer* (como cualquier otro manejador) y temporizador:

- *EventSetMillisecondTimer*
- *EventSetTimer*
- *EventKillTimer*

Dado que los scripts y los servicios no son compatibles con el probador, no pueden utilizar las funciones *Probador*;provocarán errores ERR_FUNCTION_NOT_ALLOWED (4014).

5.3 Series temporales

Las series temporales son arrays de datos en los que los índices de los elementos corresponden a muestras temporales ordenadas. Debido a las especificidades de la aplicación del terminal, casi toda la información que necesita un operador de trading se proporciona en forma de series temporales. Se trata, en particular, de arrays de cotizaciones, ticks, lecturas de indicadores técnicos, etc. La gran mayoría de los programas MQL también trabajan con estos datos, por lo que se les ha asignado un grupo de funciones en la API de MQL5 que estudiaremos en esta sección.

La forma de acceder a los arrays en MQL5 permite a los desarrolladores establecer una de las dos direcciones de indexación:

- Normal (hacia delante): la numeración de los elementos va desde el principio de la matriz hasta el final (de los recuentos antiguos a los nuevos)
- Inversa (series temporales): la numeración va del final del array al principio (de los recuentos nuevos a los antiguos)

Ya hemos tratado este tema en la sección Dirección de indexación de arrays como en las series temporales.

El cambio del modo de indexación se realiza mediante la función *ArraySetAsSeries* y no afecta a la disposición física del array en la memoria. Sólo cambia la forma de acceder a los elementos por número: en la indexación normal obtenemos el i-ésimo elemento como *array[i]*, mientras que, en el modo de series temporales, la fórmula equivalente es *array[N - i - 1]*, donde N es el tamaño del array (se llama «equivalente» porque el desarrollador de la aplicación no necesita hacer ese recálculo en todas partes, ya que lo hace automáticamente el terminal si se establece el modo de indexación de series temporales para el array). Esto se ilustra en la siguiente tabla (para un array de caracteres de 10 elementos).

Elementos del array	A	B	C	D	E	F	G	H	I	J
Índice regular	0	1	2	3	4	5	6	7	8	9
Índice como en series temporales	9	8	7	6	5	4	3	2	1	0

Recordemos que la indexación de arrays comienza siempre desde cero.

Cuando se trata de arrays de cotizaciones y otros datos actualizados constantemente, los nuevos elementos se añaden físicamente al final del array. Sin embargo, desde el punto de vista del trading,

hay que tener en cuenta los datos más recientes y tomarlos como punto de partida al analizar la historia. Por eso es conveniente tener siempre la barra actual (la última) bajo el índice 0, y contar las anteriores desde ella hacia el pasado. Así, obtenemos la indexación de series temporales.

Por defecto, los arrays se indexan de izquierda a derecha. Si imaginamos que dicho array se muestra en un gráfico estándar de MetaTrader 5, entonces puramente visual, el elemento con índice 0 estará en la posición del extremo izquierdo, y el último, en la extrema derecha. En las series temporales con indexación inversa, el elemento 0º corresponde a la posición más a la derecha, y el último elemento corresponde a la posición más a la izquierda. Dado que las series temporales almacenan el historial de los datos de precios de los instrumentos financieros en relación con el tiempo, los datos más recientes de las mismas están siempre a la derecha de los antiguos.

El elemento con el índice cero en el array de series temporales contiene información sobre la última cotización del símbolo. La barra cero suele estar incompleta mientras sigue formándose.

Otra característica de una serie temporal de cotizaciones es su periodo, es decir, el intervalo de tiempo entre lecturas adyacentes. Este periodo también se denomina «marco temporal» y puede reformularse con mayor precisión. El marco temporal es un periodo de tiempo durante el cual se forma una barra de cotizaciones, y su inicio y final están alineados en tiempo absoluto con el mismo paso. Por ejemplo, en el marco temporal de «1 hora» (H1), las barras comienzan estrictamente a los 0 minutos de cada hora del día. El comienzo de cada uno de estos periodos se incluye en la barra actual, y el final pertenece a la barra siguiente.

En el capítulo Símbolos y marcos temporales se proporciona una lista completa de los marcos temporales estándar.

En el marco del concepto de series temporales, por regla general, los búferes de indicadores técnicos también funcionan, pero estudiaremos sus características más adelante.

Si es necesario, en cualquier programa MQL puede solicitar los valores de las series temporales para cualquier símbolo y marco temporal, así como los valores de los indicadores calculados para cualquier símbolo y marco temporal. Estos datos se obtienen utilizando las funciones *Copy*, entre las que hay varios arrays de lectura de precios de distintos tipos por separado (por ejemplo, *Open*, *High*, *Low*, *Close*) o arrays de estructuras *MqlRates* que contienen todas las características de cada barra.

Barras y ticks

Además de barras con cotizaciones, MetaTrader 5 proporciona a los usuarios y programas MQL la capacidad de analizar los ticks, que son cambios elementales en el precio, sobre la base de los cuales se construyen las barras. Cada tick contiene la hora con precisión de milisegundos, varios tipos de precios (*Bid*, *Ask*, *Last*) y banderas que describen la esencia de los cambios, así como el volumen de negociación de la transacción. Estudiaremos la estructura *MqlTick* correspondiente un poco más adelante, en el capítulo Trabajar con arrays de ticks reales.

Dependiendo del tipo de instrumento de trading, las barras pueden construirse a partir de los precios de *Bid* o *Last*. En concreto, *Last* ofrece precios para instrumentos cotizados, que también difunden los precios de Profundidad de Mercado. Para los instrumentos no bursátiles, como Forex o CFD, se utiliza el precio *Bid*.

Los periodos en los que no hubo variaciones de precios no generan barras. Así es como se presenta el precio en MetaTrader 5. Por ejemplo, si el marco temporal es igual a 1 día (D1), entonces un par de barras para el fin de semana, por regla general, están ausentes, y el lunes sigue inmediatamente al viernes.

Aparece una barra de cotización si se ha producido al menos un tick en el intervalo de tiempo correspondiente. Al mismo tiempo, la hora de apertura de la barra siempre se alinea estrictamente con el borde del período, aunque el primer tick haya llegado más tarde (como suele ocurrir). Por ejemplo, la primera barra M1 del día puede formarse a las 00:05 si no hubo ticks durante 4 minutos después de medianoche, y el cambio de precio se produjo a las 00:05:15 (es decir, en el 15º segundo del quinto minuto). Así, un tick se incluye en una barra determinada en función de la siguiente relación de marcas de tiempo: $T_{open} <= T_{tick} < T_{open} + P$, donde T_{open} es la hora de apertura de la barra, T_{tick} es la hora de tick, $T_{open} + P$ es la hora de apertura de la siguiente barra potencial después del periodo P (se llama barra «potencial» porque su presencia depende de otros ticks).

5.3.1 Símbolos y marcos temporales

Las series temporales con cotizaciones se identifican mediante dos parámetros: nombre del símbolo (instrumento financiero) y marco temporal (periodo).

El usuario puede ver la lista de símbolos en la ventana *Market Watch* y editarla basándose en la lista general proporcionada por el bróker (cuadro de diálogo *Symbols*). Para los programas MQL existe un conjunto de funciones que pueden utilizarse para hacer lo mismo: buscar en todos los símbolos, averiguar sus propiedades y añadir símbolos a, o eliminarlos de, *Market Watch*. Estas características serán objeto de un capítulo aparte.

Sin embargo, para solicitar series temporales, basta con conocer el nombre del símbolo: se trata de una cadena que contiene la designación de un instrumento financiero existente. Por ejemplo, puede ser fijado por el usuario en la variable de entrada. Además, el símbolo del gráfico actual se puede encontrar a partir de la variable integrada *_Symbol* (o la función *Symbol*), pero para nuestra comodidad, todas las funciones de series temporales admiten la convención de que el valor NULL también corresponde al símbolo del gráfico actual.

Pasemos ahora a los marcos temporales. Existen 21 marcos temporales estándar definidos en el sistema, y cada uno se especifica mediante un elemento de la enumeración especial ENUM_TIMEFRAMES.

Identificador	Valor (hexadecimal)	Descripción
PERIOD_CURRENT	0	Período del gráfico actual
PERIOD_M1	1 (0x1)	1 minuto
PERIOD_M2	2 (0x2)	2 minutos
PERIOD_M3	3 (0x3)	3 minutos
PERIOD_M4	4 (0x4)	4 minutos
PERIOD_M5	5 (0x5)	5 minutos
PERIOD_M6	6 (0x6)	6 minutos
PERIOD_M10	10 (0xA)	10 minutos
PERIOD_M12	12 (0xC)	12 minutos

Identificador	Valor (hexadecimal)	Descripción
PERIOD_M15	15 (0xF)	15 minutos
PERIOD_M20	20 (0x14)	20 minutos
PERIOD_M30	30 (0x1E)	30 minutos
PERIOD_H1	16385 (0x4001)	1 hora
PERIOD_H2	16386 (0x4002)	2 horas
PERIOD_H3	16387 (0x4003)	3 horas
PERIOD_H4	16388 (0x4004)	4 horas
PERIOD_H6	16390 (0x4006)	6 horas
PERIOD_H8	16392 (0x4008)	8 horas
PERIOD_H12	16396 (0x400C)	12 horas
PERIOD_D1	16408 (0x4018)	1 día
PERIOD_W1	32769 (0x8001)	1 semana
PERIOD_MN1	49153 (0xC001)	1 mes

Como vimos en la sección sobre variables predefinidas, el programa puede conocer el periodo del gráfico actual a partir de la variable integrada *_Period* (o la función *Period*). Es fácil ver en la columna de valores que pasar cero a las funciones integradas que aceptan un marco temporal significará el periodo del gráfico actual.

El valor de los marcos temporales en minutos es el mismo que el número de minutos que contienen (por ejemplo, 30 significa M30). Para los marcos temporales en horas se activa el bit 0x4000, y el byte inferior contiene el número de horas (por ejemplo, 0x4003 para H3). El periodo de día D1 se codifica como 24 horas, es decir, 0x4018 (0x18 es igual a 24). Por último, los marcos temporales semanal y mensual tienen sus propios bits distintivos 0x8000 y 0xC000, respectivamente, como indicadores de unidad, y el recuento (en el byte bajo) es 1 en ambos casos.

Para una cómoda conversión de elementos de enumeración en cadenas y viceversa, se adjunta al libro un archivo de encabezado *Periods.mqh* (ya lo hemos utilizado en el ejemplo de trabajo con archivos, y lo utilizaremos en futuros ejemplos). Una de sus funciones, *StringToPeriod*, utiliza en su algoritmo las características antes descritas de la representación interna en bits de los elementos de enumeración.

```
#define PERIOD_PREFIX_LENGTH 7 // StringLen("PERIOD_")

// getting the abbreviated name of the period without the "PERIOD_" prefix
string PeriodToString(const ENUM_TIMEFRAMES tf = PERIOD_CURRENT)
{
   const static int prefix = StringLen("PERIOD_");
   return StringSubstr(EnumToString(tf == PERIOD_CURRENT ? _Period : tf),
      PERIOD_PREFIX_LENGTH);
}

// get the period value by full (PERIOD_H4) or short (H4) name
ENUM_TIMEFRAMES StringToPeriod(string name)
{
   if(StringLen(name) < 2) return 0;
   // converting full name "PERIOD_TN" to short "TN" if needed
   if(StringLen(name) > PERIOD_PREFIX_LENGTH)
   {
     name = StringSubstr(name, PERIOD_PREFIX_LENGTH);
   }
   // convert the digital ending "N" to a number, skip "T"
   const int count = (int)StringToInteger(StringSubstr(name, 1));
   // clear possible error WRONG_STRING_PARAMETER(5040)
   // for example, if the input string is "MN1", then N1 is not a number for StringTo
   ResetLastError();
   switch(name[0])
   {
      case 'M':
         if(!count) return PERIOD_MN1;
         return (ENUM_TIMEFRAMES)count;
      case 'H':
         return (ENUM_TIMEFRAMES)(0x4000 + count);
      case 'D':
         return PERIOD_D1;
      case 'W':
         return PERIOD_W1;
   }
   return 0;
}
```

Tenga en cuenta que las variables *_Symbol* y *_Period* contienen datos reales sólo en los programas MQL que se ejecutan en gráficos, incluidos scripts, Asesores Expertos e indicadores. En los servicios, estas variables están vacías y, por lo tanto, para acceder a las series temporales, debe establecer explícitamente el nombre del símbolo y el período u obtenerlos de alguna manera desde el exterior.

La propiedad que define un marco temporal es su duración (duración de la barra). MQL5 le permite obtener el número de segundos que forman una barra de un marco temporal específico utilizando la función *PeriodSeconds*.

int PeriodSeconds(ENUM_TIMEFRAMES period = PERIOD_CURRENT)

El parámetro *period* especifica el periodo como un elemento de la enumeración ENUM_TIMEFRAMES. Si no se especifica el parámetro se devuelve el número de segundos del período del gráfico actual en el que se está ejecutando el programa.

Veremos ejemplos de utilización de la función en el indicador *IndDeltaVolume.mq5* de la sección Esperar datos y gestionar la visibilidad, así como en el indicador *UseM1MA.mq5* de la sección Utilización de los indicadores integrados.

Para generar marcos temporales de duración no estándar que no estén incluidos en la lista especificada, la API de MQL5 proporciona símbolos personalizados; sin embargo, estos no permiten operar como en los gráficos estándar sin modificar los Asesores Expertos.

Además, es importante tener en cuenta que en MetaTrader 5 la duración de las barras dentro de una determinada serie temporal o en un gráfico es siempre la misma. Por lo tanto, para construir gráficos en los que las barras se forman no en función del tiempo, sino a medida que se acumulan otros parámetros, en concreto, los volúmenes (gráficos Equivolume) o el movimiento de los precios en una dirección en pasos fijos (Renko), puede desarrollar sus propias soluciones basadas en indicadores (por ejemplo, con el tipo de representación DRAW_CANDLES o DRAW_BARS) o utilizando símbolos personalizados.

5.3.2 Aspectos técnicos de la organización y el almacenamiento de series temporales

Antes de proceder a las cuestiones prácticas de la utilización de las funciones de la API de MQL5 diseñadas para trabajar con series temporales, debemos considerar los fundamentos técnicos de la recepción de datos de cotización desde el servidor y su almacenamiento en MetaTrader 5.

Antes de que los datos de precios estén disponibles en el terminal para su visualización en gráficos y su transferencia a programas MQL, se descargan del servidor y se preparan de una manera especial. El mecanismo de acceso al servidor para obtener datos no depende de cómo se haya iniciado la solicitud: por el usuario al navegar por el gráfico o mediante programación a través del lenguaje MQL5.

Los datos llegan del servidor en un formato comprimido: se trata de bloques de barras de minutos comprimidos económicamente que, sin embargo, no son las barras M1 habituales.

Los datos recibidos del servidor se descomprimen automáticamente y se guardan en un formato intermedio especial HCC. Los datos de cada símbolo se escriben en una carpeta independiente *{terminal_dir}/bases/{server_name}/history/{symbol_name}*. Por ejemplo, los datos sobre EURUSD del servidor de trading demo de MetaQuotes pueden ubicarse en la carpeta *C:/Program Files/MetaTrader 5/bases/MetaQuotes-Demo/history/EURUSD/*.

Los datos se escriben en archivos con la extensión *.hcc: cada archivo almacena los datos de barras de un minuto correspondientes a un año. Por ejemplo, el archivo 2021.hcc de la carpeta EURUSD contiene barras de minutos EURUSD para 2021. Estos archivos se utilizan para preparar los datos de precios de todos los plazos y no están destinados al acceso directo.

Los archivos de servicios en formato HCC actúan como fuente de datos para trazar los datos de precios de periodos concretos. Sólo se crean a petición de un gráfico o un programa MQL y se guardan para su uso posterior en archivos con la extensión *.hc.

Para cada marco temporal, los datos se preparan independientemente de otros marcos temporales. Las normas de generación y disponibilidad de datos son las mismas para todos los marcos temporales, incluido M1. Es decir, a pesar de que la unidad de almacenamiento de datos en el formato HCC es una barra de minutos, su presencia no significa la presencia y disponibilidad de datos del marco temporal M1 en el mismo volumen en el formato HC.

Para ahorrar recursos, los datos de los marcos temporales se cargan y almacenan en la RAM sólo cuando es necesario: si no hay accesos a los datos durante mucho tiempo, se descargan de la RAM (pero permanecen en el archivo). Esto puede provocar un aumento del tiempo de ejecución de la siguiente solicitud de series temporales si no se ha utilizado durante mucho tiempo. Todas las series temporales populares, en particular aquellas cuyos gráficos están abiertos, están disponibles casi instantáneamente si el ordenador dispone de recursos suficientes.

La recepción de nuevos datos del servidor provoca la actualización automática de los datos de precios utilizados en el formato HC para todos los marcos temporales y el recálculo de todos los indicadores dependientes.

Cuando un programa MQL accede a los datos para un símbolo y un marco temporal concretos, existe la posibilidad de que las series temporales requeridas aún no se hayan generado o sincronizado con el servidor de trading (por ejemplo, que hayan aparecido en él precios actualizados). En este caso, deberá aplicar la espera para la disponibilidad de datos de una forma u otra.

Para los scripts, la única solución es utilizar bucles, ya que no tienen otra opción debido a la falta de manejo de eventos. En el caso de los indicadores, estos algoritmos, al igual que cualquier otro ciclo de espera, no se recomiendan categóricamente, ya que provocan una interrupción en el cálculo de todos los indicadores y del resto del procesamiento de los datos de precios de un símbolo determinado.

Para los Asesores Expertos y los indicadores, es mejor utilizar el modelo de procesamiento de eventos. Si, al procesar un evento *OnTick* u *OnCalculate* no ha conseguido obtener todos los datos necesarios de las series temporales requeridas, entonces debe salir del manejador de eventos y esperar a que aparezcan durante las siguientes llamadas del manejador.

Número máximo de barras

Debe tenerse en cuenta que el número máximo de barras que se calculará para cada par símbolo/marco temporal solicitado no supera el valor del parámetro *Max. bars in chart* en el cuadro de diálogo *Options* del terminal. Por lo tanto, este parámetro impone restricciones no sólo a los gráficos de cualquier marco temporal, sino también a todos los programas MQL.

Esta limitación tiene como principal objetivo ahorrar recursos. Al establecer valores grandes de este parámetro, hay que tener en cuenta que, si hay un historial suficientemente profundo de datos de precios para marcos temporales inferiores, el consumo de memoria para almacenar series temporales y búferes de indicadores puede ascender a cientos de megabytes y ocupar toda la RAM.

La modificación del límite de barras sólo surte efecto tras reiniciar el terminal de cliente. Afecta a la cantidad de datos solicitados al servidor para construir el número necesario de barras de marcos temporales de trabajo.

El límite fijado por el parámetro no es rígido y puede superarse en determinados casos. Por ejemplo, si al principio de la sesión, el historial de cotizaciones para un marco temporal específico es suficiente para seleccionar todo el límite, a medida que se forman nuevas barras, su número puede llegar a ser mayor que el valor actual del parámetro. El número real de barras disponibles se devuelve mediante las funciones *Bars/iBars*.

5.3.3 Obtención de características de arrays de precios

Antes de leer arrays de series temporales debemos asegurarnos de que están disponibles y de que poseen las características requeridas. La función *SeriesInfoInteger* recupera las propiedades básicas, como la profundidad del historial disponible en el terminal y en el servidor, el número de barras construidas para una combinación específica de símbolo/período, y la ausencia de discrepancias en las cotizaciones entre el terminal y el servidor.

La función tiene dos formas: la primera devuelve directamente el valor solicitado (de tipo *long*) y la segunda utiliza el cuarto parámetro *result* pasado por referencia. En este caso, el segundo formulario devuelve una señal de éxito (*true*) o de errores (*false*). En cualquier caso, el código de error puede encontrarse utilizando la función *GetLastError*.

```
long SeriesInfoInteger(const string symbol, ENUM_TIMEFRAMES timeframe,
ENUM_SERIES_INFO_INTEGER property)
bool SeriesInfoInteger(const string symbol, ENUM_TIMEFRAMES timeframe,
ENUM_SERIES_INFO_INTEGER property, long &result)
```

La función permite averiguar una de las propiedades de la serie temporal para el símbolo y el marco temporal especificados o para todo el historial del símbolo. La propiedad solicitada se identifica mediante el tercer argumento de tipo ENUM_SERIES_INFO_INTEGER. Esta enumeración incluye todas las propiedades disponibles:

Identificador	Descripción	Tipo de propiedad
SERIES_BARS_COUNT	Número de barras por símbolo/período, véase *Bars*	long
SERIES_FIRSTDATE	Primera cita por símbolo/período	datetime
SERIES_LASTBAR_DATE	Hora de apertura de la última barra por símbolo/período	datetime
SERIES_SYNCHRONIZED	Signo de sincronización de datos por símbolo/período en el terminal y en el servidor	bool
SERIES_SERVER_FIRSTDATE	Primera fecha de la historia por símbolo en el servidor independientemente del periodo	datetime
SERIES_TERMINAL_FIRSTDATE	Primera fecha del historial por símbolo en el terminal de cliente independientemente del periodo	datetime

Dependiendo de la esencia de la propiedad, el valor resultante debe convertirse en un valor de un tipo específico (véase la columna *Property type*).

Todas las propiedades se devuelven en el momento actual.

El script *SeriesInfo.mq5* proporciona un ejemplo de consulta de todas las propiedades.

```
void OnStart()
{
   PRTF(SeriesInfoInteger(NULL, 0, SERIES_BARS_COUNT));
   PRTF((datetime)SeriesInfoInteger(NULL, 0, SERIES_FIRSTDATE));
   PRTF((datetime)SeriesInfoInteger(NULL, 0, SERIES_LASTBAR_DATE));
   PRTF((bool)SeriesInfoInteger(NULL, 0, SERIES_SYNCHRONIZED));
   PRTF((datetime)SeriesInfoInteger(NULL, 0, SERIES_SERVER_FIRSTDATE));
   PRTF((datetime)SeriesInfoInteger(NULL, 0, SERIES_TERMINAL_FIRSTDATE));
   PRTF(SeriesInfoInteger("ABRACADABRA", 0, SERIES_BARS_COUNT));
}
```

He aquí un ejemplo del resultado obtenido en EURUSD, H1, en el servidor MQ Demo:

```
SeriesInfoInteger(NULL,0,SERIES_BARS_COUNT)=10001 / ok
(datetime)SeriesInfoInteger(NULL,0,SERIES_FIRSTDATE)=2020.03.02 10:00:00 / ok
(datetime)SeriesInfoInteger(NULL,0,SERIES_LASTBAR_DATE)=2021.10.08 14:00:00 / ok
(bool)SeriesInfoInteger(NULL,0,SERIES_SYNCHRONIZED)=false / ok
(datetime)SeriesInfoInteger(NULL,0,SERIES_SERVER_FIRSTDATE)=1971.01.04 00:00:00 / ok
(datetime)SeriesInfoInteger(NULL,0,SERIES_TERMINAL_FIRSTDATE)=2016.06.01 00:00:00 / o
SeriesInfoInteger(ABRACADABRA,0,SERIES_BARS_COUNT)=0 / MARKET_UNKNOWN_SYMBOL(4301)
```

5.3.4 Número de barras disponibles (Bars/iBars)

Una forma más corta de averiguar el número total de barras de una serie temporal por símbolo/periodo la proporcionan las funciones *Bars* y *iBars* (no hay diferencia entre ellas ya que *iBars* está disponible por compatibilidad con MQL4).

int Bars(const string symbol, ENUM_TIMEFRAMES timeframe)
int iBars(const string symbol, ENUM_TIMEFRAMES timeframe)

Las funciones devuelven el número de barras disponibles para el programa MQL para el símbolo y periodo dados. Este valor está influenciado por el parámetro *Max. bars in chart* en el terminal *Options* (véase la nota en la sección Aspectos técnicos de la organización y el almacenamiento de series temporales). Por ejemplo, si se descarga un historial en el terminal, que para un marco temporal específico es de 20 000 barras, pero el límite está fijado en 10 000 barras en los ajustes, entonces el segundo valor será decisivo. Inmediatamente después del lanzamiento del terminal, las funciones devolverán el número de 10 000 barras, pero a medida que se formen nuevas barras, aumentará (si la memoria libre lo permite). En MQL5, este límite se puede encontrar llamando a *TerminalInfoInteger(TERMINAL_MAXBARS)*.

Además, la función *Bars* tiene una segunda opción que le permite averiguar el número de barras en el intervalo entre dos fechas.

int Bars(const string symbol, ENUM_TIMEFRAMES timeframe, datetime start, datetime stop)

Sólo se consultan aquellas barras cuya hora de apertura esté comprendida en el rango entre *start* y *stop* (ambos inclusive). No importa en qué orden se especifiquen *start* y *stop*: la función analizará las cotizaciones de un tiempo menor a uno mayor.

Si los datos de la serie temporal con los parámetros especificados aún no se han generado o sincronizado con el servidor de operaciones de trading en el momento en que se llama a la función *Bars/iBars*, la función devolverá null. En este caso, el atributo de error en *_LastError* también será 0 (no hay error porque, simplemente, los datos aún no se han descargado o no están listos). Después de

recibir 0, compruebe la sincronización de un marco temporal específico utilizando *SeriesInfoInteger(..., SERIES_SYNCHRONIZED)* o la sincronización del símbolo mediante la función especial *SymbolIsSynchronized*.

En el script *SeriesBars.mq5* de la siguiente sección se muestran ejemplos de cómo trabajar con las funciones, junto con la función *iBarShift* asociada.

5.3.5 Índice de la barra de búsqueda por tiempo (iBarShift)

La función *iBarShift* proporciona el número de barra de la hora especificada. En este caso, la numeración de las barras se entiende siempre como en las series temporales, es decir, el índice 0 corresponde a la barra más a la derecha, la más reciente, y los valores aumentan a medida que se avanza de derecha a izquierda (hacia el pasado).

```
int iBarShift(const string symbol, ENUM_TIMEFRAMES timeframe, datetime time, bool exact = false)
```

La función devuelve el índice de la barra en la serie temporal para el par especificado de parámetros *symbol/timeframe*, en el que cae el valor del parámetro *time*. Cada barra se caracteriza por una hora de apertura y una duración común a todas las barras de la serie, es decir, por un periodo. Por ejemplo, en un marco temporal por horas, una barra marcada con una hora de apertura de 13:00 dura desde las 13:00:00 hasta las 13:59:59 (incluyendo todo el último minuto y segundo).

Si no hay ninguna barra para la hora especificada (por ejemplo, la hora cae en horas o días en las que no hay trading), la función se comporta de forma diferente dependiendo del parámetro exacto: si *precise* = *true*, la función devolverá -1; si *exact=false*, devolverá el índice de la barra más cercana cuya hora de apertura sea menor que la especificada. En el caso de que no exista tal barra, es decir, que no haya historial antes de la hora especificada, la función devolverá -1. Pero aquí hay un matiz.

> ¡Atención! Si la función *iBarShift* devuelve un número de barra específico, es decir, un valor distinto de -1, esto no significa que el siguiente intento de acceder a las series temporales por este índice podrá obtener los precios u otras características de esta barra. En concreto, esto puede ocurrir si el índice de la barra solicitada supera el límite de barras de la ventana del terminal (*TerminalInfoInteger(TERMINAL_MAXBARS)*). Esto puede ocurrir a medida que se forman nuevas barras: entonces las barras más antiguas pueden desplazarse más allá del límite hacia la izquierda y quedar fuera de la ventana de visibilidad, aunque nominalmente pueden permanecer en la memoria durante algún tiempo. El desarrollador debe comprobar siempre estas situaciones.

Vamos a comprobar el rendimiento de las funciones *Bars/iBars* y (véase la sección anterior) *iBarShift* utilizando el script *SeriesBars.mq5*.

```
void OnStart()
{
   const datetime target = PRTF(ChartTimeOnDropped());
   PRTF(iBarShift(NULL, 0, target));
   PRTF(iBarShift(NULL, 0, target, true));
   PRTF(iBarShift(NULL, 0, TimeCurrent()));
   PRTF(Bars(NULL, 0, target, TimeCurrent()));
   PRTF(Bars(NULL, 0, TimeCurrent(), target));
   PRTF(iBars(NULL, 0));
   PRTF(Bars(NULL, 0));
   PRTF(Bars(NULL, 0, 0, TimeCurrent()));
   PRTF(Bars(NULL, 0, TimeCurrent(), TimeCurrent()));
}
```

Aquí nos encontramos con otra función desconocida *ChartTimeOnDropped* (lo describiremos más adelante): devuelve el tiempo de una barra específica (en el gráfico activo) a la que el script de *Navigator* fue arrastrado y soltado con el ratón. En primer lugar, vamos a arrastrar el script a la zona del gráfico donde hay cotizaciones.

Se crearán las siguientes entradas en el registro (los números serán diferentes, de acuerdo con su configuración, acciones y la hora actual):

```
ChartTimeOnDropped()=2021.10.01 09:00:00 / ok
iBarShift(NULL,0,target)=125 / ok
iBarShift(NULL,0,target,true)=125 / ok
iBarShift(NULL,0,TimeCurrent())=0 / ok
Bars(NULL,0,target,TimeCurrent())=126 / ok
Bars(NULL,0,TimeCurrent(),target)=126 / ok
iBars(NULL,0)=10004 / ok
Bars(NULL,0)=10004 / ok
Bars(NULL,0,0,TimeCurrent())=10004 / ok
Bars(NULL,0,TimeCurrent(),TimeCurrent())=0 / ok
```

En este caso, el script se arrastró a una barra con la hora 2021.10.01 09:00 (se utilizó un marco temporal horario). Según *iBarShift*, esta hora correspondía al número de barra 125.

El número de barras desde la barra bajo el ratón hasta la última (hora actual) era 126. Esto se combina con el número de barra 125, ya que la numeración empieza por 0.

El número total de barras del gráfico, obtenido de diferentes formas (*iBars*, *Bars* sin intervalo de fechas, y *Bars* con un intervalo completo de 0 al momento actual *TimeCurrent*), es igual a 10004. La configuración del terminal tenía un límite de 10 000 pero se han formado 4 barras horarias adicionales durante la sesión.

El número de la barra en la que cae el tiempo actual *iBarShift(..., TimeCurrent())* es siempre 0 para un símbolo y un marco temporal existentes, siempre que *exact* = *false*. Si *exact* = *true*, entonces a veces podemos obtener -1 ya que el tiempo del servidor aumenta cuando llegan los ticks de todos los instrumentos del mercado, y el símbolo actual puede no objeto de trading temporalmente. Entonces la hora del servidor puede adelantarse en más de una barra, y para *TimeCurrent* no hay ninguna nueva barra que acertar exactamente.

Si arrastramos y soltamos el script en el área vacía a la derecha de la última barra actual (es decir, en el futuro), obtenemos algo como esto:

```
ChartTimeOnDropped()=2021.10.09 02:30:00 / ok
iBarShift(NULL,0,target)=0 / ok
iBarShift(NULL,0,target,true)=-1 / ok
Bars(NULL,0,target,TimeCurrent())=0 / ok
Bars(NULL,0,TimeCurrent(),target)=0 / ok
iBars(NULL,0)=10004 / ok
Bars(NULL,0)=10004 / ok
Bars(NULL,0,0,TimeCurrent())=10004 / ok
Bars(NULL,0,TimeCurrent(),TimeCurrent())=0 / ok
```

La función *iBarShift* en el modo de búsqueda de cualquier barra anterior (*exact* = *false*) devuelve 0 porque la barra actual es la más próxima al futuro. Sin embargo, una búsqueda exacta (*exact* = *true*) da el resultado -1. Además, las funciones *Bars* que cuentan barras en el rango desde el tiempo actual hasta el futuro «objetivo» ahora devuelven 0 (no hay barras allí todavía).

La función *iBarShift* es especialmente útil para escribir programas MQL multidivisa. Muy a menudo, los calendarios de trading de los distintos instrumentos financieros no coinciden, por lo que, para una hora determinada, puede existir una barra en un símbolo y no existir en otro. Utilizando la función *iBarShift* en el modo de búsqueda de barras más cercano (anterior), siempre puede obtener índices de barras con precios que fueron relevantes para diferentes símbolos en el mismo momento. Por regla general, incluso para los símbolos de Forex, los índices de las barras históricas para el mismo tiempo pueden diferir.

Por ejemplo, las siguientes instrucciones registrarán diferentes números de barras y sus números en el mismo intervalo de fechas para tres símbolos: EURUSD, XAUUSD, USDRUB en el marco temporal de una hora (servidor MQ Demo):

```
PRTF(Bars("EURUSD", PERIOD_H1, D'2021.05.01', D'2021.09.01')); // 2087
PRTF(Bars("XAUUSD", PERIOD_H1, D'2021.05.01', D'2021.09.01')); // 1991
PRTF(Bars("USDRUB", PERIOD_H1, D'2021.05.01', D'2021.09.01')); // 694
PRTF(iBarShift("EURUSD", PERIOD_H1, D'2021.09.01')); // 671
PRTF(iBarShift("XAUUSD", PERIOD_H1, D'2021.09.01')); // 638
PRTF(iBarShift("USDRUB", PERIOD_H1, D'2021.09.01')); // 224
```

5.3.6 Visión general de las funciones Copy para obtener arrays de cotizaciones

La API de MQL5 contiene varias funciones para leer series temporales de cotizaciones en arrays. Sus nombres figuran en la siguiente tabla.

Función	Acción
CopyRates	Obtener el historial de cotizaciones en un array de estructuras *MqlRates*
CopyTime	Obtener el histórico de horarios de apertura de barras en un array del tipo *datetime*
CopyOpen	Obtener el histórico de precios de apertura de barras en un array del tipo *double*
CopyHigh	Obtener el historial de precios máximos de barra en un array del tipo *double*
CopyLow	Obtener el historial de precios de barras bajas en un array del tipo *double*
CopyClose	Obtener el histórico de precios de cierre de barras en un array del tipo *double*
CopyTickVolume	Obtener el historial de volúmenes de ticks en un array del tipo *long*
CopyRealVolume	Obtener el histórico de volúmenes de intercambio en un array del tipo *long*
CopySpread	Obtener el historial de diferenciales en un array del tipo *int*

Todas las funciones toman como los dos primeros parámetros el nombre del símbolo y el período deseados, que pueden representarse condicionalmente mediante el siguiente pseudocódigo:

```
int Copy***(const string symbol, ENUM_TIMEFRAMES timeframe, ...)
```

Además, todas las funciones tienen tres variantes del prototipo, que difieren en la forma de establecer el rango solicitado:

- Índice inicial de barras y número de barras: *Copy***(..., int offset, int count, ...)*
- Hora de inicio del intervalo y número de barras: *Copy***(..., datetime start, int count, ...)*
- Horas de inicio y fin del intervalo: *Copy***(..., datetime start, datetime stop, ...)*

Al mismo tiempo, la notación de parámetros implica que los datos solicitados tienen una dirección de indexación como en una serie temporal, es decir, la posición *offset* con índice 0 almacena los datos de la barra incompleta actual, y el aumento de índices corresponde a moverse más profundamente en el historial de precios. Debido a ello, en particular a la segunda opción, el número indicado de barras *count* contará hacia atrás desde el inicio del intervalo *offset*, es decir, en el sentido de disminución del tiempo.

La tercera opción proporciona flexibilidad adicional: no importa en qué orden se especifiquen las fechas de inicio y fin (*start/stop*), ya que las funciones devolverán en cualquier caso datos en el intervalo que va de la fecha menor a la mayor. Las barras adecuadas se seleccionan de forma que su tiempo de apertura se encuentre entre los recuentos de tiempo *start/stop* o sea igual a uno de ellos, es decir, se considera el rango [*start; stop*] incluyendo los límites.

El desarrollador determina qué opción de función elegir en función de lo que sea más importante: obtener un número garantizado de elementos (por ejemplo, para algoritmos de aprendizaje automático) o cubrir un intervalo de fechas específico (por ejemplo, con un comportamiento uniforme predeterminado del mercado).

La precisión de representación del tiempo en el tipo *datetime* es de 1 segundo. Los valores *start/stop* no tienen que redondearse al tamaño del punto. Por ejemplo, el intervalo de 14:59 a 16:01 le permitirá

seleccionar dos barras en el marco temporal H1 para las 15:00 y las 16:00. Un intervalo degenerado con etiquetas iguales y redondeadas, por ejemplo, 15:00 en cotizaciones H1, corresponde a una barra.

Puede solicitar barras en el marco temporal diario aunque haya horas/minutos/segundos distintos de cero en los parámetros de inicio/parada (a pesar de que las etiquetas de las barras en el marco temporal D1 tengan la hora 00:00). En este caso, sólo aquellas barras D1 que tengan una hora de apertura posterior al mínimo de *start/stop* y hasta el máximo de *start/stop* (la igualdad con las etiquetas de las barras diarias es imposible en este caso, ya que el tiempo requerido contiene horas/minutos/segundos). Por ejemplo, entre D'2021.09.01 12:00' y D'2021.09.03 07:00', hay dos horarios de apertura de barras D1: D'2021.09.02' y D'2021.09.03'. Estas barras se incluirán en el resultado. La barra D'2021.09.01' tiene una hora de apertura de 00:00, que es anterior al inicio del intervalo, por lo que se descarta. La barra D'2021.09.03' se incluye en el resultado, a pesar de que sólo 7 horas de la mañana de ese día entraron en el intervalo. En cambio, una solicitud de varias horas dentro de un día, por ejemplo, entre D'2021.09.01 12:00' y D'2021.09.01 15:00', no cubrirá una única barra del día (la hora de apertura de la barra D'2021.09.01' no entra en este intervalo), y por tanto el array receptor estará vacío.

La única diferencia entre todas las funciones de la tabla es el tipo de array que recibe los datos, que se pasa como último parámetro por referencia. Por ejemplo, la función *CopyRates* coloca los datos solicitados en un array de estructuras *MqlRates*, y la función *CopyTime* coloca los horarios de apertura de las barras en un array del tipo *datetime*, y así sucesivamente.

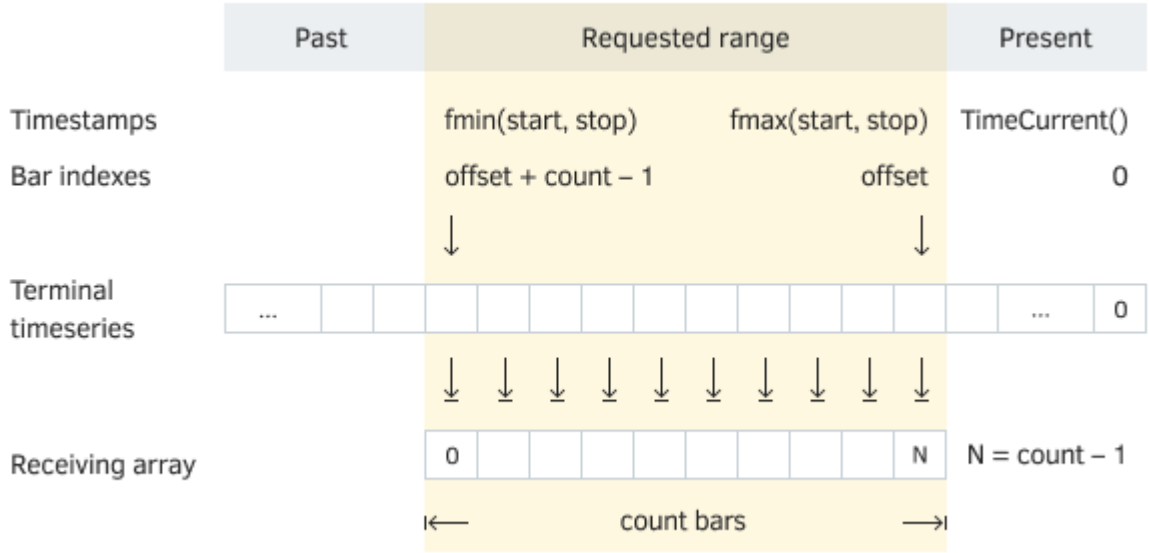
Así, los prototipos de funciones comunes pueden representarse de la siguiente manera:

```
int Copy***(const string symbol, ENUM_TIMEFRAMES timeframe, int offset, int count, type &result[])
int Copy***(const string symbol, ENUM_TIMEFRAMES timeframe, datetime start, int count, type &result[])
int Copy***(const string symbol, ENUM_TIMEFRAMES timeframe, datetime start, datetime stop, type &result[])
```

Aquí, el *type* coincide con cualquiera de los tipos *MqlRates*, *datetime*, *double*, *long* o *int*, dependiendo de la función específica.

Las funciones devuelven el número de elementos copiados en el array o -1 en caso de error. En concreto, obtendremos -1 si no hay datos en el servidor en el intervalo solicitado, o el intervalo está fuera del número máximo de barras del gráfico (*TerminalInfoInteger(TERMINAL_MAXBARS)*).

Es importante señalar que, en el array receptor, los datos recibidos siempre se colocan físicamente en orden cronológico, del pasado al futuro. Así, si se utiliza la indexación estándar para el array receptor (es decir, la función *ArraySetAsSeries*), entonces el elemento del índice 0 será el más antiguo y el último el más reciente. Si la instrucción se ejecutó para el array *ArraySetAsSeries(result, true)*, entonces la numeración se realizará en orden inverso, como en una serie temporal: el elemento 0º será el más nuevo del rango, y el último elemento será el más antiguo. Esto se ilustra en la siguiente figura.



Series temporales terminales y array receptor

Si tiene éxito, el número especificado de elementos de la propia serie temporal (interna) del terminal se copiará en el array de destino. Cuando se solicitan datos por intervalo de fechas (*start/stop*), el número de elementos del array resultante se determinará indirectamente, en función del contenido del historial de este intervalo. Por lo tanto, para copiar un número desconocido de valores, se recomienda utilizar arrays dinámicos: las funciones de copia asignan de forma independiente el tamaño necesario de los arrays de destino (el tamaño puede aumentar o disminuir).

Si necesita copiar un número conocido de elementos o lo hace con frecuencia, como cada vez que llama a *OnTick* en Asesores Expertos u *OnCalculate* en los indicadores, es mejor utilizar arrays distribuidos estáticamente. El hecho es que las operaciones de asignación de memoria para arrays dinámicos requieren tiempo adicional y pueden afectar al rendimiento, especialmente durante las pruebas y la optimización.

Se accede a las series temporales de forma diferente para los distintos tipos de programas MQL si los datos solicitados aún no están listos. Por ejemplo, en las funciones de indicadores*, Copy* personalizados devuelven inmediatamente un error, ya que los indicadores se ejecutan en el hilo de interfaz común del terminal y no pueden esperar a recibir datos (se supone que los indicadores solicitarán datos durante las siguientes llamadas de sus manejadores de eventos, y las series temporales ya se habrán descargado y construido para entonces). Además, en el capítulo dedicado a los indicadores, aprenderemos que para acceder a las cotizaciones del gráfico «nativo» sobre el que se sitúa el indicador, no es necesario utilizar las funciones *Copy*, ya que todas las series temporales se pasan automáticamente a través de los parámetros del array del manejador *OnCalculate*.

Cuando se accede desde Asesores Expertos y scripts se realizan varios intentos de recibir datos con una breve pausa (con una espera dentro de la función), que da tiempo a cargar y calcular las series temporales que faltan. La función devolverá la cantidad de datos que estarán listos para cuando expire este tiempo de espera, pero la carga del historial continuará, y la siguiente petición similar devolverá más datos.

En cualquier caso, debe estar preparado para que la función *Copy* devuelva un error en lugar de datos (existen diferentes motivos: fallo de conexión, falta de datos solicitados, carga del procesador si se solicitan muchas series temporales nuevas en paralelo): analice la causa del problema en el código (_LastError) y vuelva a intentarlo más tarde, corrija la configuración o informe al usuario.

La presencia de un símbolo en *Market Watch* no es una condición necesaria para solicitar series temporales utilizando las funciones de *Copy*; no obstante, para los símbolos incluidos en esta ventana, las consultas tienden a ejecutarse más rápidamente porque algunos datos ya han sido descargados del servidor y probablemente calculados para los periodos solicitados. Aprenderemos a añadir caracteres a *Market Watch* mediante programación en la sección Editar la lista de Observación de Mercado.

Para explicar el funcionamiento de las funciones en la práctica, veamos el script *SeriesCopy.mq5*. Contiene múltiples llamadas a la función *CopyTime*, que le permite ver visualmente cómo se correlacionan las marcas de tiempo y los números de barra.

El script define un array dinámico *times* para recibir datos. Todas las solicitudes se realizan para el símbolo «EURUSD» y el marco temporal H1.

```
void OnStart()
{
   datetime times[];
```

Para empezar, se solicitan 10 barras, desde el 5 de septiembre de 2021 hacia el pasado. Como ese día es domingo, las barras anteriores fueron el viernes 3 (véase el registro más abajo).

```
   PRTF(CopyTime("EURUSD", PERIOD_H1, D'2021.09.05', 10, times)); // 10 / ok
   ArrayPrint(times);
   /*
   [0] 2021.09.03 14:00 2021.09.03 15:00 2021.09.03 16:00 2021.09.03 17:00 2021.09.03 18:00
   [5] 2021.09.03 19:00 2021.09.03 20:00 2021.09.03 21:00 2021.09.03 22:00 2021.09.03 23:00
   */
```

La salida del array se realiza por defecto en orden cronológico (a pesar de que los parámetros de la función se establecen en el sistema de coordenadas inverso: como en una serie temporal). Cambiemos el orden de indexación en el array receptor y obtengamos una nueva salida.

```
   PRTF(ArraySetAsSeries(times, true)); // true / ok
   ArrayPrint(times);
   /*
   [0] 2021.09.03 23:00 2021.09.03 22:00 2021.09.03 21:00 2021.09.03 20:00 2021.09.03 19:00
   [5] 2021.09.03 18:00 2021.09.03 17:00 2021.09.03 16:00 2021.09.03 15:00 2021.09.03 14:00
   */
```

Para los próximos experimentos, restableceremos el orden habitual.

```
   PRTF(ArraySetAsSeries(times, false)); // true / ok
```

Ahora vamos a solicitar un número indefinido de barras entre dos puntos temporales (el número es desconocido, porque las vacaciones pueden estar en el intervalo, por ejemplo). Lo haremos de dos maneras: en el primer caso, indicaremos el intervalo del futuro al pasado, y en el segundo, del pasado al futuro. Los resultados coinciden.

```
   //                                    FROM                 TO
   PRTF(CopyTime("EURUSD", PERIOD_H1, D'2021.09.06 03:00', D'2021.09.05 03:00', times
   ArrayPrint(times)  //                 FROM                 TO
   PRTF(CopyTime("EURUSD", PERIOD_H1, D'2021.09.05 03:00', D'2021.09.06 03:00', times
   ArrayPrint(times);
   /*
   CopyTime(EURUSD,PERIOD_H1,D'2021.09.06 03:00',D'2021.09.05 03:00',times)=4 / ok
   2021.09.06 00:00 2021.09.06 01:00 2021.09.06 02:00 2021.09.06 03:00
   CopyTime(EURUSD,PERIOD_H1,D'2021.09.05 03:00',D'2021.09.06 03:00',times)=4 / ok
   2021.09.06 00:00 2021.09.06 01:00 2021.09.06 02:00 2021.09.06 03:00
   */
```

Al imprimir los arrays podemos ver que son idénticos. Volvamos al modo de indexación de series temporales y discutamos una cuestión más.

```
   PRTF(ArraySetAsSeries(times, true)); // true / ok
   ArrayPrint(times);
   // 2021.09.06 03:00 2021.09.06 02:00 2021.09.06 01:00 2021.09.06 00:00
```

Aunque las dos marcas de tiempo están separadas por 24 horas, lo que implica obtener 25 elementos en el array (recuerde que el principio y el final se procesan de forma inclusiva), el resultado sólo contiene 4 barras. El hecho es que el 5 de septiembre cae en domingo y, por lo tanto, de todo el intervalo, el trading se ha llevado a cabo únicamente en las horas matinales del día 6.

Observe también que el tamaño del array receptor se ha reducido automáticamente de 10 a 4 elementos.

Por último, solicitaremos 10 barras, a partir de la barra 100ª (los resultados obtenidos dependerán de su hora actual y del historial disponible).

```
   PRTF(CopyTime("EURUSD", PERIOD_H1, 100, 10, times)); // 10 / ok
   ArrayPrint(times);
   /*
   [0] 2021.10.04 19:00 2021.10.04 18:00 2021.10.04 17:00 2021.10.04 16:00 2021.10.04
   [5] 2021.10.04 14:00 2021.10.04 13:00 2021.10.04 12:00 2021.10.04 11:00 2021.10.04
   */
}
```

Debido a la indexación como en una serie temporal, el array se muestra en orden cronológico inverso.

5.3.7 Obtener cotizaciones como un array de estructuras MqlRates

Para solicitar un array de cotizaciones que incluya todas las características de la barra, utilice la función *CopyRates*, que tiene múltiples sobrecargas.

```
int CopyRates(const string symbol, ENUM_TIMEFRAMES timeframe, int offset, int count, MqlRates &rates[])
int CopyRates(const string symbol, ENUM_TIMEFRAMES timeframe, datetime start, int count, MqlRates &rates[])
int CopyRates(const string symbol, ENUM_TIMEFRAMES timeframe, datetime start, datetime stop, MqlRates &rates[])
```

La función obtiene en el array *rates* datos históricos para los parámetros especificados: símbolo, marco temporal e intervalo temporal especificados mediante números de barra o valores *start/stop* del tipo *datetime*.

La función devuelve el número de elementos de array copiados, o -1 en caso de error, cuyo código puede encontrarse en *_LastError*. En concreto, se producirá un error si se especifica un símbolo inexistente, el intervalo no contiene datos en el servidor o supera el límite del número de barras del gráfico (*TerminalInfoInteger(TERMINAL_MAXBARS)*).

Los conceptos básicos para trabajar con esta función son comunes a todas las funciones de *Copy* y se han expuesto en la sección Visión general de las funciones Copy para obtener arrays de cotizaciones.

La estructura de tipos en línea *MqlRates* se describe del siguiente modo:

```
struct MqlRates
{
   datetime time;         // bar opening time
   double   open;         // opening price
   double   high;         // maximum price per bar
   double   low;          // minimum price per bar
   double   close;        // closing price
   long     tick_volume;  // tick volume per bar
   int      spread;       // minimum spread per bar in points
   long     real_volume;  // exchange volume per bar
};
```

Vamos a intentar aplicar la función para calcular el tamaño medio de las barras en el script *SeriesStats.mq5*. En las variables de entrada, ofreceremos la posibilidad de seleccionar un símbolo de trabajo, un marco temporal, el número de barras analizadas y el desplazamiento inicial hacia el pasado (0 significa análisis desde la barra actual).

```
input string WorkSymbol = NULL; // Symbol (leave empty for current)
input ENUM_TIMEFRAMES TimeFrame = PERIOD_CURRENT;
input int BarOffset = 0;
input int BarCount = 10000;

void OnStart()
{
   MqlRates rates[];
   double range = 0, move = 0; // calculate the range and price movement in bars

   PrintFormat("Requesting %d bars on %s %s",
      BarCount, StringLen(WorkSymbol) > 0 ? WorkSymbol : _Symbol,
      EnumToString(TimeFrame == PERIOD_CURRENT ? _Period : TimeFrame));

   // request all information about BarCount bars to the MqlRates array
   const int n = PRTF(CopyRates(WorkSymbol, TimeFrame, BarOffset, BarCount, rates));

   // in the loop we calculate the average for the range and movement
   for(int i = 0; i < n; ++i)
   {
      range += (rates[i].high - rates[i].low) / n;
      move += (fmax(rates[i].open, rates[i].close)
             - fmin(rates[i].open, rates[i].close)) / n;
   }

   PrintFormat("Stats per bar: range=%f, movement=%f", range, move);
   PrintFormat("Dates: %s - %s",
      TimeToString(rates[0].time), TimeToString(rates[n - 1].time));
}
```

Habiendo lanzado el script en el gráfico EURUSD,H1, podemos obtener aproximadamente el siguiente resultado:

```
Requesting 100000 bars on EURUSD PERIOD_H1
CopyRates(WorkSymbol,TimeFrame,BarOffset,BarCount,rates)=20018 / ok
Stats per bar: range=0.001280, movement=0.000621
Dates: 2018.07.19 15:00 - 2021.10.11 17:00
```

Como el terminal tenía un límite de 20 000 barras, una petición de 100 000 barras sólo podía devolver 20 018 (el límite y las barras recién formadas tras el inicio de la sesión). El primer elemento del array (con índice 0) contiene una barra con la hora 2018.07.19 15:00, y el último, - 2021.10.11 17:00.

Según las estadísticas, el rango medio de la barra durante este tiempo fue de 128 puntos, y el movimiento entre la apertura y el cierre fue de 62 puntos.

Cuando solicite información utilizando una fecha de inicio y de fin (*start/stop*) tenga en cuenta que ambos límites se tratan de forma inclusiva. Por lo tanto, para establecer un intervalo correspondiente a cualquier barra de un marco temporal superior, hay que restar 1 segundo del borde derecho. Aplicaremos esta técnica en el ejemplo *SeriesSpread.mq5* de la sección Lectura de precio, volumen, diferencial y hora por índice de barras.

5.3.8 Solicitud independiente de arrays de precios, volúmenes, diferenciales, hora

En lugar de consultar todas las características de codificación como un array *MqlRates*, puede leer sólo los datos de un campo concreto (precio, volumen, diferencial u hora) en un array independiente. Para ello se definen varias funciones que operan según los principios generales expuestos en la sección Visión general de las funciones Copy para obtener arrays de cotizaciones.

En el siguiente diagrama se combinan las descripciones de todos los prototipos.

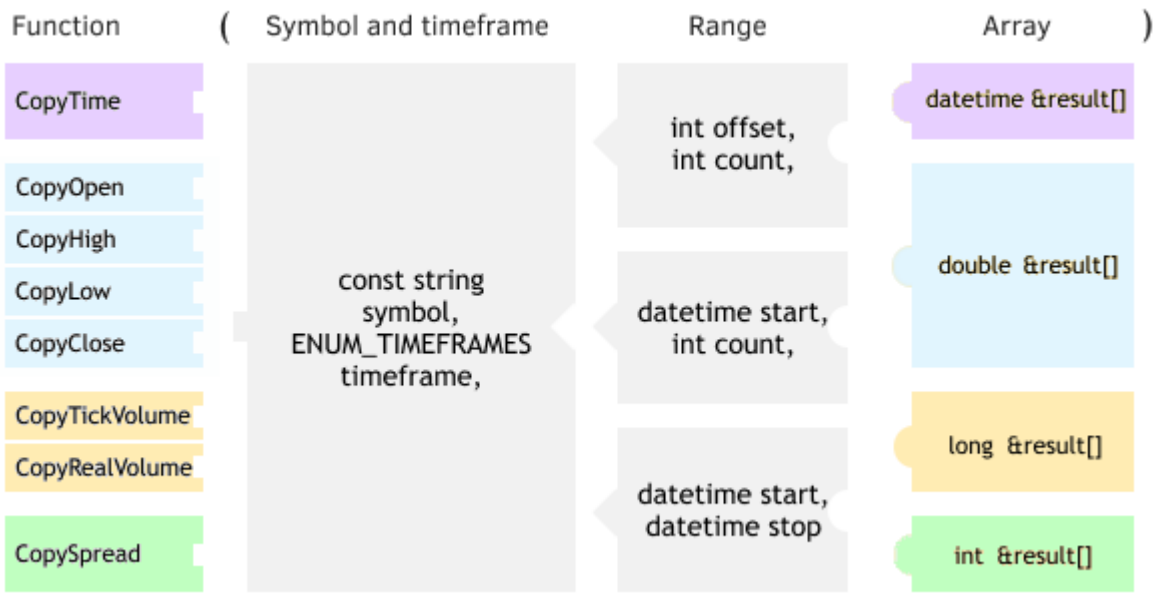


Diagrama prototipo de las funciones Copy

El script *SeriesRates.mq5* utiliza las funciones de copia de precios OHLC para compararlas con el resultado de la llamada a *CopyRates*.

```
void OnStart()
{
   const int N = 10;
   MqlRates rates[];

   // request and display all information about N bars from the MqlRates array
   PRTF(CopyRates("EURUSD", PERIOD_D1, D'2021.10.01', N, rates));
   ArrayPrint(rates);

   // now request OHLC prices separately
   double open[], high[], low[], close[];
   PRTF(CopyOpen("EURUSD", PERIOD_D1, D'2021.10.01', N, open));
   PRTF(CopyHigh("EURUSD", PERIOD_D1, D'2021.10.01', N, high));
   PRTF(CopyLow("EURUSD", PERIOD_D1, D'2021.10.01', N, low));
   PRTF(CopyClose("EURUSD", PERIOD_D1, D'2021.10.01', N, close));

   // compare prices obtained by different methods
   for(int i = 0; i < N; ++i)
   {
      if(rates[i].open != open[i]
      || rates[i].high != high[i]
      || rates[i].low != low[i]
      || rates[i].close != close[i])
      {
         // we shouldn't be here
         Print("Data mismatch at ", i);
         return;
      }
   }

   Print("Copied OHLC arrays match MqlRates array"); // success: there is a match
}
```

Después de ejecutar el script, obtenemos las siguientes entradas en el registro:

```
CopyRates(EURUSD,PERIOD_D1,D'2021.10.01',N,rates)=10 / ok
[hora] [apertura] [máximo] [mínimo] [cierre] [volumen_tic] [diferencial] [volumen_rea
[0] 2021.09.20 00:00:00 1.17272 1.17363 1.17004 1.17257 58444 0 0
[1] 2021.09.21 00:00:00 1.17248 1.17486 1.17149 1.17252 58514 0 0
[2] 2021.09.22 00:00:00 1.17240 1.17555 1.16843 1.16866 72571 0 0
[3] 2021.09.23 00:00:00 1.16860 1.17501 1.16835 1.17381 68536 0 0
[4] 2021.09.24 00:00:00 1.17379 1.17476 1.17007 1.17206 51401 0 0
[5] 2021.09.27 00:00:00 1.17255 1.17255 1.16848 1.16952 57807 0 0
[6] 2021.09.28 00:00:00 1.16940 1.17032 1.16682 1.16826 64793 0 0
[7] 2021.09.29 00:00:00 1.16825 1.16901 1.15894 1.15969 68964 0 0
[8] 2021.09.30 00:00:00 1.15963 1.16097 1.15626 1.15769 68517 0 0
[9] 2021.10.01 00:00:00 1.15740 1.16075 1.15630 1.15927 66777 0 0
CopyOpen(EURUSD,PERIOD_D1,D'2021.10.01',N,open)=10 / ok
CopyHigh(EURUSD,PERIOD_D1,D'2021.10.01',N,high)=10 / ok
CopyLow(EURUSD,PERIOD_D1,D'2021.10.01',N,low)=10 / ok
CopyClose(EURUSD,PERIOD_D1,D'2021.10.01',N,close)=10 / ok
Los arrays OHLC copiados coinciden con el array MqlRates.
```

Recordemos que el volumen de ticks del campo *tick_volume* es un simple contador de ticks para un periodo. El volumen de intercambio en el campo *real_volume* es igual a cero para los instrumentos sin intercambio (así como para EURUSD, en este caso).

Otro ejemplo de uso de la función *CopyTime* se proporciona en el script *SeriesCopy.mq5* en la sección Visión general de las funciones Copy para obtener arrays de cotizaciones.

5.3.9 Lectura de precio, volumen, diferencial y hora por índice de barras

A veces se necesita información, no sobre una secuencia de barras, sino sobre una sola barra. En teoría, esto puede hacerse utilizando las funciones *Copy* anteriormente comentadas, especificando en ellas la cantidad (parámetro *count*) igual a 1, pero esto no es muy conveniente. Una versión más sencilla la ofrecen las siguientes funciones, que devuelven un valor de un tipo determinado para una barra por su número en la serie temporal.

Todas las funciones tienen un prototipo similar pero diferentes nombres y tipos de retorno. Históricamente, los nombres comienzan con el prefijo *i*, es decir, tienen la forma *iValue* (estas funciones pertenecen a un gran grupo de indicadores técnicos integrados: al fin y al cabo, las características de las cotizaciones son la fuente principal para el análisis técnico, y casi todos los indicadores son sus derivados, de ahí la letra *i*).

```
type iValue(const string symbol, ENUM_TIMEFRAMES timeframe, int offset)
```

Aquí, type corresponde a uno de los tipos *datetime*, *double*, *long* o *int*, dependiendo de la función específica. El símbolo y el marco temporal identifican la serie temporal solicitada. El índice de barra requerido *offset* se pasa en notación de series temporales: 0 significa la barra derecha más reciente (normalmente aún no completada), y números más altos significan barras más antiguas. Como en el caso de las funciones Copiar, se pueden utilizar NULL y 0 para que el símbolo y el periodo sean iguales a las propiedades del gráfico actual.

Dado que las funciones *i* equivalen a llamar a las funciones *Copy*, todas las características de solicitud de series temporales desde distintos tipos de programas, descritas en la sección Visión general de las funciones Copy para obtener arrays de cotizaciones les son aplicables.

Función	Descripción
iTime	Hora de apertura de la barra
iOpen	Precio de apertura de la barra
iHigh	Precio alto de la barra
iLow	Precio bajo de la barra
iClose	Precio de cierre de la barra
iTickVolume	Volumen de ticks de la barra (similar a iVolume)
iVolume	Volumen de ticks de la barra (similar a iTickVolume)
iRealVolume	Volumen real de trading de la barra
iSpread	Diferencial mínimo de la barra (en pips)

Las funciones devuelven el valor solicitado o 0 en caso de error (lamentablemente, 0 puede ser un valor real en algunos casos). Para obtener más información sobre el error, llame a la función *GetLastError*.

Las funciones no almacenan en caché los resultados. En cada llamada, devuelven datos reales de la serie temporal para el símbolo/período especificado. Esto significa que en ausencia de datos preparados (en la primera llamada, o tras una pérdida de sincronización), la función puede tardar algún tiempo en preparar el resultado.

A modo de ejemplo, vamos a intentar obtener una estimación más o menos realista del tamaño del diferencial de cada barra. El valor mínimo del diferencial se almacena en las cotizaciones, lo que puede causar expectativas excesivamente altas a la hora de diseñar estrategias de negociación. Para obtener valores absolutamente precisos del diferencial medio, mediano o máximo por barra, sería necesario analizar ticks reales, pero aún no hemos aprendido a trabajar con ellos. Y además, sería un proceso que consumiría muchos recursos. Un enfoque más racional consiste en analizar los diferenciales en el marco temporal inferior M1: para las barras de marcos temporales superiores, basta con buscar el diferencial máximo en las barras interiores de M1. Por supuesto, estrictamente hablando, no será el máximo, sino el máximo de los valores mínimos, pero dada la transitoriedad de las lecturas de minutos, podemos esperar detectar expansiones de diferencial característicos al menos en algunas barras M1, y esto es suficiente para obtener una relación aceptable de precisión y velocidad de análisis.

Una de las versiones del algoritmo está implementada en el script *SeriesSpread.mq5*. En las variables de entrada, puede establecer el símbolo, el marco temporal y el número de barras para el análisis. De manera predeterminada, se procesa el símbolo del gráfico actual y su periodo (debería ser mayor que M1).

```
input string WorkSymbol = NULL; // Symbol (leave empty for current)
input ENUM_TIMEFRAMES TimeFrame = PERIOD_CURRENT;
input int BarCount = 100;
```

Dado que para cada barra sólo es importante la información sobre su hora y diferencial, se ha descrito una estructura especial con dos campos. Podríamos utilizar la estructura estándar *MqlRates* y añadir diferenciales «máximos» a algún campo no utilizado (por ejemplo, *real_volume* para los símbolos de Forex), pero entonces se copiarían los datos de la mayoría de los campos y se desperdiciaría memoria.

```
struct SpreadPerBar
{
   datetime time;
   int spread;
};
```

Utilizando el nuevo tipo de estructura, preparamos el array *peaks* para calcular los datos del número especificado de barras.

```
void OnStart()
{
   SpreadPerBar peaks[];
   ArrayResize(peaks, BarCount);
   ZeroMemory(peaks);
   ...
```

Más adelante, la parte principal del algoritmo se ejecuta en el bucle de barras. Para cada barra, utilizamos la función *iTime* para determinar dos marcas de tiempo que definen los límites de la barra. De hecho, se trata de la hora de apertura de la barra i-ésima y de la barra vecina (i+1)-ésima. Dados los principios de indexación, podemos decir que la (i+1)ª barra es la barra anterior (más antigua, véase la variable *prev*) y la i-ésima es la siguiente (más nueva, véase la variable *next*). La hora de apertura de la barra pertenece a una sola barra, es decir, la etiqueta *prev* está contenida en la (i+1)-ésima barra, y la etiqueta *next* está en la i-ésima. Así, al procesar cada barra, su borde derecho debe excluirse del intervalo [*prev;next*).

Nos interesan los diferenciales en un marco temporal de un minuto, por lo que utilizaremos la función *CopySpread* para PERIOD_M1. En este caso, el intervalo de media apertura se consigue ajustando los parámetros *start/stop* al valor exacto *prev* y el valor *next* reducido en 1 segundo. La información de diferencial se copia en el array dinámico *spreads* (la memoria para ello la asigna la propia función).

```
   for(int i = 0; i < BarCount; ++i)
   {
      int spreads[]; // receiving array for M1 spreads inside the i-th bar
      const datetime next = iTime(WorkSymbol, TimeFrame, i);
      const datetime prev = iTime(WorkSymbol, TimeFrame, i + 1);
      const int n = CopySpread(WorkSymbol, PERIOD_M1, prev, next - 1, spreads);
      const int m = ArrayMaximum(spreads);
      if(m > -1)
      {
         peaks[i].spread = spreads[m];
         peaks[i].time = prev;
      }
   }
```

A continuación, encontramos el valor máximo en este array y lo guardamos en la estructura apropiada *SpreadPerBar* junto con el tiempo de barra. Tenga en cuenta que la barra incompleta cero no se incluye en el análisis (puede completar el algoritmo si es necesario).

Una vez completado el bucle, enviamos un array de estructuras al diario.

```
   PrintFormat("Maximal speeds per intraday bar\nProcessed %d bars on %s %s",
      BarCount, StringLen(WorkSymbol) > 0 ? WorkSymbol : _Symbol,
      EnumToString(TimeFrame == PERIOD_CURRENT ? _Period : TimeFrame));
   ArrayPrintM(peaks);
```

Ejecutando el script en el gráfico EURUSD,H1, obtendremos estadísticas de diferencial dentro de barras horarias (abreviadas):

```
Maximal speeds per intraday bar
Processed 100 bars on EURUSD PERIOD_H1
[ 0] 2021.10.12 14:00        1
[ 1] 2021.10.12 13:00        1
[ 2] 2021.10.12 12:00        1
[ 3] 2021.10.12 11:00        1
[ 4] 2021.10.12 10:00        0
[ 5] 2021.10.12 09:00        1
[ 6] 2021.10.12 08:00        2
[ 7] 2021.10.12 07:00        2
[ 8] 2021.10.12 06:00        1
[ 9] 2021.10.12 05:00        1
[10] 2021.10.12 04:00        1
[11] 2021.10.12 03:00        1
[12] 2021.10.12 02:00        4
[13] 2021.10.12 01:00       16
[14] 2021.10.12 00:00       65
[15] 2021.10.11 23:00       15
[16] 2021.10.11 22:00        2
[17] 2021.10.11 21:00        1
[18] 2021.10.11 20:00        1
[19] 2021.10.11 19:00        2
[20] 2021.10.11 18:00        1
[21] 2021.10.11 17:00        1
[22] 2021.10.11 16:00        1
[23] 2021.10.11 15:00        2
[24] 2021.10.11 14:00        1
```

Hay un aumento evidente de los diferenciales por la noche: por ejemplo, cerca de medianoche, las cotizaciones contienen diferenciales de 7-15 puntos, y en nuestras mediciones, son de 15-65. No obstante, también se encuentran valores distintos de cero en otros periodos, aunque las métricas de las barras horarias suelen contener ceros.

5.3.10 Encontrar los valores máximo y mínimo de una serie temporal

Entre el grupo de funciones para trabajar con series temporales de cotizaciones, hay dos que proporcionan el tratamiento agregado más sencillo: la búsqueda de los valores máximo y mínimo de la serie en un intervalo determinado, respectivamente *iHighest* y *iLowest*.

int iHighest(const string symbol, ENUM_TIMEFRAMES timeframe, ENUM_SERIESMODE type, int count = WHOLE_ARRAY, int offset = 0)
int iLowest(const string symbol, ENUM_TIMEFRAMES timeframe, ENUM_SERIESMODE type, int count = WHOLE_ARRAY, int offset = 0)

Las funciones devuelven el índice del valor mayor o menor para un tipo específico de serie temporal, que se especifica mediante un par de parámetros *symbol/timeframe*, así como el elemento de enumeración ENUM_SERIESMODE (describe los campos de cotización que ya nos son familiares).

Identificador	Descripción
MODE_OPEN	Precio de apertura
MODE_LOW	Precio alto
MODE_HIGH	Precio bajo
MODE_CLOSE	Precio de cierre
MODE_VOLUME	Volumen de ticks
MODE_REAL_VOLUME	Volumen real
MODE_SPREAD	Diferencial

El parámetro *offset* especifica el índice en el que se inicia la búsqueda. La numeración se realiza como en una serie temporal, es decir, el aumento de *offset* da lugar a un desplazamiento hacia el pasado, y el índice 0-ésimo indica la barra actual (éste es el valor por defecto). El número de barras analizadas se especifica en el parámetro *count* (WHOLE_ARRAY por defecto).

En caso de error, las funciones devuelven -1. Utilice *GetLastError* para buscar el código de error.

Para demostrar cómo funciona una de estas funciones (*iHighest*) vamos a modificar el ejemplo del apartado anterior sobre la estimación de los tamaños reales de los diferenciales por barras y a comparar los resultados. Por supuesto, deben coincidir. La nueva versión del script se adjunta en el archivo *SeriesSpreadHighest.mq5*.

Los cambios han afectado a la estructura *SpreadPerBar* y al ciclo de trabajo dentro de *OnStart*.

Se han añadido campos a la estructura que permiten comprender cómo funciona la nueva función. Debido a la naturaleza del algoritmo, no son obligatorios.

```
struct SpreadPerBar
{
   datetime time;
   int spread;
   int max; // through index of the M1 bar with a spread, the value of which is maxim
            // among all M1-bars within the current bar of the higher timeframe
   int num; // number of M1 bars in the current bar of the higher timeframe
   int pos; // initial index of the M1 bar within the current bar of the higher timef
};
```

Las principales transformaciones han afectado a *OnStart*, pero están localizadas dentro del bucle (todos los demás fragmentos de código han permanecido inalterados).

```
for(int i = 0; i < BarCount; ++i)
{
   const datetime next = iTime(WorkSymbol, TimeFrame, i);
   const datetime prev = iTime(WorkSymbol, TimeFrame, i + 1);
   ...
```

Los bordes de la barra actual, *prev* y *next*, se definen como antes. No obstante, en lugar de copiar los elementos de la serie temporal entre estas etiquetas en su propio array *spreads*, y la posterior llamada de *ArrayMaximum* para ello, determinamos los índices y el número de barras M1 que forman la barra actual del marco temporal superior. Esto se hace de la siguiente manera.

La función *iBarShift* permite conocer el desplazamiento (variable *p*) en el histórico de M1, donde se encuentra el borde derecho de la barra con el tiempo *next - 1*. La función *bars* calcula el número de barras M1 (variable *n*) comprendidas entre *prev* y *next - 1*. Estos dos valores se convierten en parámetros en la llamada a la función *iHighest* realizada para buscar el valor máximo de tipo MODE_SPREAD, entre las barras *n* M1, a partir del índice *p*. Si se encuentra el máximo sin problemas (*m > -1*), nos queda tomar el valor correspondiente mediante *iSpread* y colocarlo en una estructura.

```
   const int p = iBarShift(WorkSymbol, PERIOD_M1, next - 1);
   const int n = Bars(WorkSymbol, PERIOD_M1, prev, next - 1);
   const int m = iHighest(WorkSymbol, PERIOD_M1, MODE_SPREAD, n, p);
   if(m > -1)
   {
      peaks[i].spread = iSpread(WorkSymbol, PERIOD_M1, m);
      peaks[i].time = prev;
      peaks[i].max = m;
      peaks[i].num = n;
      peaks[i].pos = p;
   }
}
```

Al enviar el array con los resultados al registro, ahora veremos adicionalmente los índices de las barras M1, donde «comienza» la barra del marco temporal superior y donde se encontró en ella el diferencial máximo. La palabra «comienza» está entre comillas porque, a medida que lleguen nuevas barras M1, estos índices aumentarán, y el «comienzo» virtual de cada una se desplazará, aunque las horas de apertura de las barras históricas, por supuesto, permanecen constantes.

```
Maximal speeds per intraday bar
Processed 100 bars on EURUSD PERIOD_H1
               [time] [spread] [max] [num] [pos]
[ 0] 2021.10.12 15:00        0     7    60     7
[ 1] 2021.10.12 14:00        1    89    60    67
[ 2] 2021.10.12 13:00        1   181    60   127
[ 3] 2021.10.12 12:00        1   213    60   187
[ 4] 2021.10.12 11:00        1   248    60   247
[ 5] 2021.10.12 10:00        0   307    60   307
[ 6] 2021.10.12 09:00        1   385    60   367
[ 7] 2021.10.12 08:00        2   469    60   427
[ 8] 2021.10.12 07:00        2   497    60   487
[ 9] 2021.10.12 06:00        1   550    60   547
[10] 2021.10.12 05:00        1   616    60   607
[11] 2021.10.12 04:00        1   678    60   667
[12] 2021.10.12 03:00        1   727    60   727
[13] 2021.10.12 02:00        4   820    60   787
[14] 2021.10.12 01:00       16   906    60   847
[15] 2021.10.12 00:00       65   956    60   907
[16] 2021.10.11 23:00       15   967    60   967
[17] 2021.10.11 22:00        2  1039    60  1027
[18] 2021.10.11 21:00        1  1090    60  1087
[19] 2021.10.11 20:00        1  1148    60  1147
[20] 2021.10.11 19:00        2  1210    60  1207
[21] 2021.10.11 18:00        1  1313    60  1267
[22] 2021.10.11 17:00        1  1345    60  1327
[23] 2021.10.11 16:00        1  1411    60  1387
[24] 2021.10.11 15:00        2  1461    60  1447
[25] 2021.10.11 14:00        1  1526    60  1507
...
```

Por ejemplo, en el momento de lanzar el script, la barra con la etiqueta 2021.10.12 14:00 partía de la barra 67 M1 (es decir, se abrió hace 67 minutos), y la barra M1 con el diferencial máximo dentro de esta barra H1 se encontró bajo el índice 89. Obviamente, este índice debe ser menor que el número de la barra M1 donde comenzó la barra H1 anterior: 2021.10.12 13:00 - se marcó hace 127 minutos. En esta barra H1, a su vez, se encontró el máximo diferencial para el índice 181. Y esto es menor que el índice 187 para una barra aún más antigua 2021.10.12 12:00.

Los índices de las columnas *pos* y *max* aumentan constantemente porque recorremos las barras en orden del presente al pasado. La columna *num* casi tendrá 60, ya que la mayoría de las barras H1 están formadas por 60 barras M1, pero no siempre es así. Por ejemplo, a continuación se muestran barras horarias incompletas, compuestas por menos minutos: esto puede ser consecuencia de un cierre más temprano del mercado debido al calendario de vacaciones, o de lagunas reales en la actividad comercial (falta de liquidez).

```
...
[38] 2021.10.11 01:00       20  2346    60  2287
[39] 2021.10.11 00:00       85  2404    58  2347
[40] 2021.10.08 23:00       15  2406    55  2405
[41] 2021.10.08 22:00        2  2463    60  2460
...
```

5.3.11 Trabajar con arrays de ticks reales en estructuras MqlTick

MetaTrader 5 ofrece la posibilidad de trabajar no sólo con la historia de las cotizaciones (barras), sino también con la historia de los ticks reales. Desde la interfaz de usuario, todos los datos históricos están disponibles en el cuadro de diálogo *Symbols*, que tiene tres pestañas: *Specification*, *Bars* y *Ticks*. Cuando se selecciona un elemento específico en la lista de símbolos en forma de árbol de la primera pestaña, al cambiar a las pestañas *Bars* y *Ticks* se pueden solicitar cotizaciones en forma de barras o ticks, respectivamente.

Desde los programas MQL, el historial de ticks reales también está disponible utilizando las funciones *CopyTicks* y *CopyTicksRange*.

```
int CopyTicks(const string symbol, MqlTick &ticks[], uint flags = COPY_TICKS_ALL, ulong from = 0,
uint count = 0)
int CopyTicksRange(const string symbol, MqlTick &ticks[], uint flags = COPY_TICKS_ALL, ulong from
= 0, ulong to = 0)
```

Ambas funciones solicitan ticks para el instrumento *symbol* especificado en el array *ticks* pasado por referencia. La estructura *MqlTick* contiene toda la información sobre un tick y se describe en MQL5 de la siguiente manera:

```
struct MqlTick
{
   datetime time;         // time of this price update
   double   bid;          // current Bid price
   double   ask;          // current Ask price
   double   last;         // Last trade price
   ulong    volume;       // volume for Last price
   long     time_msc;     // time of this price update in milliseconds
   uint     flags;        // flags (which fields of the structure have changed)
   double   volume_real; // volume for the Last price with increased accuracy
};
```

El campo *flags* está destinado a almacenar una máscara de bits de signos, cuyos campos de la estructura de ticks contienen valores modificados.

Constante	Valor	Descripción
TICK_FLAG_BID	2	Precio de oferta modificado
TICK_FLAG_ASK	4	Precio de venta modificado
TICK_FLAG_LAST	8	Último precio modificado
TICK_FLAG_VOLUME	16	Volumen modificado
TICK_FLAG_BUY	32	El tick se generó como resultado de una operación de compra
TICK_FLAG_SELL	64	El tick se generó como resultado de una operación de venta

Esto era necesario porque cada tick siempre rellena todos los campos, independientemente de si los datos han cambiado en comparación con el tick anterior. Esto le permite disponer siempre del estado actual de los precios en cualquier momento, sin necesidad de buscar valores anteriores en el historial

de ticks. Por ejemplo, sólo el precio de oferta podría cambiar con un tick, pero además del nuevo precio, se indicarán otros parámetros en la estructura: anteriores *Ask*, *Last*, volumen, etc.

Al mismo tiempo, debe tener en cuenta que, dependiendo del tipo de instrumento, algunos campos de los ticks pueden ser siempre cero (y los bits de máscara correspondientes nunca se activan para ellos). En particular, para los instrumentos Forex, por regla general, los campos *last*, *volume* y *volume_real* permanecen vacíos.

El array receptor *ticks* puede ser de tamaño fijo o dinámico. Las funciones no copiarán en un array fijo más ticks que los correspondientes al tamaño del array, independientemente del número real de ticks en el intervalo de tiempo solicitado (especificado por los parámetros *from/to* en la función *CopyTicksRange*) o en el parámetro *count* de la función *CopyTicks*. En el array *ticks*, los ticks más antiguos se colocan en primer lugar y los más recientes, en último lugar.

En los parámetros de ambas funciones, las lecturas de tiempo se especifican en milisegundos desde el 01.01.1970 00:00:00. En la función *CopyTicks*, el rango de ticks solicitados se establece mediante el valor inicial *from* y el número de ticks *count*, y en *CopyTicksRange* se establece mediante *from* y *to* (ambos valores están incluidos).

En otras palabras: *CopyTicksRange* está diseñado para recibir ticks en un intervalo específico, y su número no se conoce de antemano. *CopyTicks* garantiza un máximo de *count* ticks, pero no permite determinar de antemano qué intervalo de tiempo cubrirán estos ticks.

El orden cronológico de los valores *from* y *to* en *CopyTicksRange* no es importante: la función dará ticks en cualquier caso, empezando por el mínimo de los dos valores y terminando por el máximo.

La función *CopyTicks* evalúa el parámetro *from* como el borde izquierdo con el tiempo mínimo y cuenta desde él *count* ticks hacia el futuro. No obstante, hay una excepción importante: *from = 0* (por defecto) se trata como el momento actual en el tiempo, y los ticks se cuentan desde él hacia el pasado. Esto permite obtener siempre el número especificado de últimos ticks. Cuando *count = 0* (por defecto), la función no copia más de 2000 ticks.

Ambas funciones devuelven el número de ticks copiados, o -1 en caso de error. En concreto, *GetLastError* puede devolver los siguientes códigos de error:

- ERR_HISTORY_TIMEOUT: el tiempo de espera de sincronización de ticks ha expirado; la función ha devuelto todo lo que tenía.
- ERR_HISTORY_SMALL_BUFFER: el búfer estático es demasiado pequeño, por lo que da tanto como cabe en el array.
- ERR_NOT_ENOUGH_MEMORY: no se ha podido asignar la cantidad de memoria necesaria para obtener el historial de ticks del rango especificado en un array dinámico.

El parámetro *flags* define el tipo de ticks solicitados.

Constante	Valor	Descripción
COPY_TICKS_INFO	1	Ticks causados por cambios de *Bid* y/o *Ask* (TICK_FLAG_BID, TICK_FLAG_ASK)
COPY_TICKS_TRADE	2	Ticks con cambios de *Last* y *Volume* (TICK_FLAG_LAST, TICK_FLAG_VOLUME, TICK_FLAG_BUY, TICK_FLAG_SELL)
COPY_TICKS_ALL	3	Todos los ticks

Para cualquier tipo de solicitud, los campos restantes de la estructura *MqlTick*, que no coincidan con las banderas, contendrán los valores reales anteriores. Por ejemplo, si sólo se han solicitado ticks de información (COPY_TICKS_INFO), los campos restantes seguirán rellenándose en ellos. Significa que si sólo ha cambiado el precio de *Bid*, se escribirán los últimos valores conocidos en los campos *ask* y *volume*. Para averiguar qué ha cambiado en el tick, analice su campo *flags* (estará el valor TICK_FLAG_BID o TICK_FLAG_ASK, o una combinación de ambos). Si un tick tiene valores cero de los precios *Bid* y *Ask*, y las banderas indican que estos precios han cambiado (*flags == TICK_FLAG_BID | TICK_FLAG_ASK*), entonces ello indica el vaciado del libro de órdenes.

Del mismo modo, si se solicitaron ticks de trading (COPY_TICKS_TRADE), se registrarán los últimos valores de precio conocidos en sus campos *bid* y *ask*. En este caso, el campo *flags* puede tener una combinación de TICK_FLAG_LAST, TICK_FLAG_VOLUME, TICK_FLAG_BUY, TICK_FLAG_SELL.

Cuando se solicita COPY_TICKS_ALL, se devuelven todos los ticks.

Al llamar a cualquiera de las funciones de *CopyTicks/CopyTicksRange* se comprueba la sincronización de la base de ticks almacenada en el disco duro para el símbolo dado. Si no hay suficientes ticks en la base de datos local, los que falten se descargarán automáticamente del servidor de operaciones. En este caso, los ticks se sincronizarán teniendo en cuenta la fecha más antigua de los parámetros de consulta y hasta el momento actual. Después, todos los ticks entrantes para este símbolo irán a la base de datos de ticks y la mantendrán actualizada en un estado sincronizado.

> Los datos por ticks son mucho mayores que las cotizaciones por minutos. Cuando solicite por primera vez un historial de ticks o inicie las pruebas por tick reales, descargarlos puede llevar mucho tiempo. El historial de datos de tick se almacena en archivos con formato interno TKC en el directorio *{terminal_dir}/bases/{server_name}/ticks/{symbol_name}*. Cada archivo contiene información correspondiente a un mes.

En los indicadores, las funciones devuelven el resultado inmediatamente, es decir, copian los ticks disponibles por símbolo e inician el proceso en segundo plano de sincronización de la base de ticks si no hay datos suficientes. Todos los indicadores de un símbolo funcionan en un hilo común, por lo que no tienen derecho a esperar a que se complete la sincronización. Una vez finalizada la sincronización, la siguiente llamada a la función devolverá todos los ticks solicitados.

En los Asesores Expertos y scripts, las funciones pueden esperar hasta 45 segundos para obtener un resultado: a diferencia de un indicador, cada Asesor Experto y script se ejecuta en su propio hilo y, por lo tanto, puede esperar a que la sincronización se complete dentro de un tiempo de espera. Si durante este tiempo todavía no se sincronizan los ticks en la cantidad requerida, entonces sólo se devolverán los ticks disponibles, y la sincronización continuará en segundo plano.

Recordemos que los ticks en tiempo real se transmiten a los gráficos como eventos: los indicadores reciben notificaciones de nuevos ticks en el gráfico *OnCalculate*, mientras que los Asesores Expertos los

reciben en el manejador *OnTick*. Hay que tener en cuenta que el sistema no garantiza la entrega de todos los eventos. Si llegan nuevos ticks al terminal mientras el programa está procesando el evento *OnCalculate/OnTick* actual, es posible que no se añadan a su cola nuevos eventos para este programa «ocupado» (véase la sección Visión general de las funciones de gestión de eventos). Además, pueden llegar varios ticks al mismo tiempo, pero sólo se generará un evento para cada programa MQL: el evento de estado actual del mercado. En este caso, puede utilizar la función *CopyTicks* para solicitar todos los ticks que se han producido desde el procesamiento anterior del evento. He aquí el aspecto de este algoritmo en pseudocódigo:

```
void processAllTicks()
{
   static ulong prev = 0;
   if(!prev)
   {
      MqlTick ticks[];
      const int n = CopyTicks(_Symbol, ticks, COPY_TICKS_ALL, prev + 1, 1000000);
      if(n > 0)
      {
         prev = ticks[n - 1].time_msc;
         ... // processing all missed ticks
      }
   }
   else
   {
      MqlTick tick;
      SymbolInfoTick(_Symbol, tick);
      prev = tick.time_msc;
      ... // processing the first tick
   }
}
```

La función *SymbolInfoTick* utilizada aquí rellena una única estructura *MqlTick* pasada por referencia con los datos del último tick. Lo estudiaremos en una secciónaparte.

Tenga en cuenta que al llamar a *CopyTicks* se añade un milisegundo a la antigua marca de tiempo *prev*. De este modo se garantiza que no se vuelva a procesar el tick anterior. No obstante, si hubiera varios ticks dentro de un milisegundo correspondiente a *prev*, este algoritmo los omitirá. Si desea cubrir absolutamente todos los ticks, debe recordar el número de ticks disponibles con el tiempo *prev* mientras actualiza la variable *prev*. En la siguiente llamada a *CopyTicks*, consulte los ticks desde el momento *prev* y omita (ignore en el array) el número de ticks «antiguos».

Sin embargo, tenga en cuenta que el algoritmo anterior no es necesario para todos los programas MQL. La mayoría de ellos no analizan cada tick, mientras que el estado actual del precio correspondiente al último tick conocido se difunde rápidamente a los gráficos del modelo de eventos y está disponible por medio de las propiedades de símbolo y gráfico.

Para demostrar las funciones, veamos dos ejemplos, uno para cada función. Para ambos ejemplos se ha desarrollado un archivo de encabezado común *TickEnum.mqh*, en el que las constantes anteriores para las banderas de tic solicitadas y las banderas de estado de tic se resumen en dos enumeraciones.

```
enum COPY_TICKS
{
   ALL_TICKS = /* -1 */ COPY_TICKS_ALL,    // all ticks
   INFO_TICKS = /* 1 */ COPY_TICKS_INFO,   // info ticks
   TRADE_TICKS = /* 2 */ COPY_TICKS_TRADE, // trade ticks
};

enum TICK_FLAGS
{
   TF_BID = /* 2 */ TICK_FLAG_BID,
   TF_ASK = /* 4 */ TICK_FLAG_ASK,
   TF_BID_ASK = TICK_FLAG_BID | TICK_FLAG_ASK,

   TF_LAST = /* 8 */ TICK_FLAG_LAST,
   TF_BID_LAST = TICK_FLAG_BID | TICK_FLAG_LAST,
   TF_ASK_LAST = TICK_FLAG_ASK | TICK_FLAG_LAST,
   TF_BID_ASK_LAST = TF_BID_ASK | TICK_FLAG_LAST,

   TF_VOLUME = /* 16 */ TICK_FLAG_VOLUME,
   TF_LAST_VOLUME = TICK_FLAG_LAST | TICK_FLAG_VOLUME,
   TF_BID_VOLUME = TICK_FLAG_BID | TICK_FLAG_VOLUME,
   TF_BID_ASK_VOLUME = TF_BID_ASK | TICK_FLAG_VOLUME,
   TF_BID_ASK_LAST_VOLUME = TF_BID_ASK | TF_LAST_VOLUME,

   TF_BUY = /* 32 */ TICK_FLAG_BUY,
   TF_SELL = /* 64 */ TICK_FLAG_SELL,
   TF_BUY_SELL = TICK_FLAG_BUY | TICK_FLAG_SELL,
   TF_LAST_VOLUME_BUY = TF_LAST_VOLUME | TICK_FLAG_BUY,
   TF_LAST_VOLUME_SELL = TF_LAST_VOLUME | TICK_FLAG_SELL,
   TF_LAST_VOLUME_BUY_SELL = TF_BUY_SELL | TF_LAST_VOLUME,
   ...
};
```

El uso de enumeraciones hace que la comprobación de tipos en el código fuente sea más rigurosa, y también facilita la visualización del significado de los valores como cadenas con *EnumToString*. Además, se han añadido las combinaciones de banderas más populares a la enumeración TICK_FLAGS para optimizar la visualización o el filtrado de los ticks. No es posible dar a los elementos de enumeración los mismos nombres que a las constantes integradas, ya que se produce un conflicto de nombres.

El primer script *SeriesTicksStats.mq5* utiliza la función *CopyTicks* para contar el número de ticks con diferentes banderas establecidas a una profundidad de historia dada.

En los parámetros de entrada, puede establecer el símbolo de trabajo (símbolo del gráfico por defecto), el número de ticks analizados y el modo de solicitud de COPY_TICKS.

```
input string WorkSymbol = NULL; // Symbol (leave empty for current)
input int TickCount = 10000;
input COPY_TICKS TickType = ALL_TICKS;
```

Las estadísticas de la aparición de cada bandera (cada bit de la máscara de bits) en las propiedades del tick se recogen en la estructura *TickFlagStats*.

```
struct TickFlagStats
{
   TICK_FLAGS flag; // mask with bit (one or more)
   int count;       // number of ticks with this bit in the flags field
   string legend;   // bit description
};
```

La función *OnStart* describe un array de estructuras *TickFlagStats* con un tamaño de 8 elementos: 6 de ellos (del 1 al 6 inclusive) se utilizan para los bits TICK_FLAG correspondientes, y los otros dos se utilizan para combinaciones de bits (véase más abajo). Utilizando un bucle simple se rellenan los elementos para los bits/banderas estándar individuales en el array, y después del bucle se rellenan dos máscaras combinadas (en el elemento 0º, los ticks se contarán con un cambio simultáneo de *Bid* y *Ask*, y en el elemento 7º contamos los ticks con transacciones simultáneas de *Buy* y *Sell*).

```
void OnStart()
{
   TickFlagStats stats[8] = {};
   for(int k = 1; k < 7; ++k)
   {
      stats[k].flag = (TICK_FLAGS)(1 << k);
      stats[k].legend = EnumToString(stats[k].flag);
   }
   stats[0].flag = TF_BID_ASK;  // combination of BID AND ASK
   stats[7].flag = TF_BUY_SELL; // combination of BUY AND SELL
   stats[0].legend = "TF_BID_ASK (COMBO)";
   stats[7].legend = "TF_BUY_SELL (COMBO)";
   ...
```

Confiaremos todo el trabajo principal a la función auxiliar *CalcTickStats*, pasándole parámetros de entrada y un array preparado para recopilar estadísticas. Después, queda mostrar los números contados en el diario.

```
   const int count = CalcTickStats(TickType, 0, TickCount, stats);
   PrintFormat("%s stats requested: %d (got: %d) on %s",
      EnumToString(TickType),
      TickCount, count, StringLen(WorkSymbol) > 0 ? WorkSymbol : _Symbol);
   ArrayPrint(stats);
}
```

La propia función *CalcTickStats* es muy interesante.

```
int CalcTickStats(const string symbol, const COPY_TICKS type,
   const datetime start, const int count,
   TickFlagStats &stats[])
{
   MqlTick ticks[];
   ResetLastError();
   const int nf = ArraySize(stats);
   const int nt = CopyTicks(symbol, ticks, type, start * 1000, count);
   if(nt > -1 && _LastError == 0)
   {
      PrintFormat("Ticks range: %s'%03d - %s'%03d",
         TimeToString(ticks[0].time, TIME_DATE | TIME_SECONDS),
         ticks[0].time_msc % 1000,
         TimeToString(ticks[nt - 1].time, TIME_DATE | TIME_SECONDS),
         ticks[nt - 1].time_msc % 1000);

      // loop through ticks
      for(int j = 0; j < nt; ++j)
      {
         // loop through TICK_FLAGs (2 4 8 16 32 64) and combinations
         for(int k = 0; k < nf; ++k)
         {
            if((ticks[j].flags & stats[k].flag) == stats[k].flag)
            {
               stats[k].count++;
            }
         }
      }
   }
   return nt;
}
```

Utiliza *CopyTicks* para solicitar ticks del *symbol* especificado, de un *type* específico, a partir de la fecha *start*, en la cantidad de elementos *count*. El parámetro *start* es del tipo *datetime*, y debe convertirse a milisegundos cuando se pase a *CopyTicks*. Recuerde que si *start = 0* (que es el caso aquí, en la función *OnStart*), el sistema devolverá los últimos ticks, contando desde la hora actual. Por lo tanto, cada vez que se llame al script, lo más probable es que las estadísticas se actualicen debido a la llegada de nuevos ticks. Las únicas excepciones posibles son las solicitudes en fin de semana o las de instrumentos de poca liquidez.

Si *CopyTicks* se ejecuta sin errores, nuestro código registra el intervalo de tiempo cubierto por los ticks recibidos.

Por último, en el bucle, recorremos todos los ticks y contamos el número de coincidencias bit a bit en las banderas de ticks y las máscaras de elementos del array de estructuras estadísticas *TickFlagStats* preparado de antemano.

Es aconsejable ejecutar el script en instrumentos en los que haya información sobre volúmenes y operaciones reales para probar todos los modos de la enumeración COPY_TICKS (recuerde que corresponden a las constantes del parámetro *flags* en *CopyTicks*: COPY_TICKS_INFO, COPY_TICKS_TRADE y COPY_TICKS_ALL).

He aquí un ejemplo de entradas de registro al solicitar estadísticas para 100000 ticks de todo tipo (*TickType* = *ALL_TICKS*):

```
Ticks range: 2021.10.11 07:39:53'278 - 2021.10.13 11:51:29'428
ALL_TICKS stats requested: 100000 (got: 100000) on YNDX.MM
    [flag] [count]              [legend]
[0]      6   11323 "TF_BID_ASK (COMBO)"
[1]      2   26700 "TF_BID"
[2]      4   33541 "TF_ASK"
[3]      8   51082 "TF_LAST"
[4]     16   51082 "TF_VOLUME"
[5]     32   25654 "TF_BUY"
[6]     64   28802 "TF_SELL"
[7]     96    3374 "TF_BUY_SELL (COMBO)"
```

Esto es lo que se obtiene al solicitar sólo ticks de información (*TickType* = *INFO_TICKS*).

```
Ticks range: 2021.10.07 07:08:24'692 - 2021.10.13 11:54:01'297
INFO_TICKS stats requested: 100000 (got: 100000) on YNDX.MM
    [flag] [count]              [legend]
[0]      6   23115 "TF_BID_ASK (COMBO)"
[1]      2   60860 "TF_BID"
[2]      4   62255 "TF_ASK"
[3]      8       0 "TF_LAST"
[4]     16       0 "TF_VOLUME"
[5]     32       0 "TF_BUY"
[6]     64       0 "TF_SELL"
[7]     96       0 "TF_BUY_SELL (COMBO)"
```

Aquí puede comprobar la exactitud de los cálculos: la suma de los números de TF_BID y TF_ASK menos las coincidencias TF_BID_ASK (COMBO) da exactamente 100000 (número total de ticks). Los ticks con volúmenes y precios *Last* no han entrado en el resultado, como era de esperar.

Ahora vamos a ejecutar el script de nuevo, exclusivamente para ticks de trading (*TickType* = *TRADE_TICKS*).

```
Ticks range: 2021.10.06 20:43:40'024 - 2021.10.13 11:52:40'044
TRADE_TICKS stats requested: 100000 (got: 100000) on YNDX.MM
    [flag] [count]              [legend]
[0]      6       0 "TF_BID_ASK (COMBO)"
[1]      2       0 "TF_BID"
[2]      4       0 "TF_ASK"
[3]      8  100000 "TF_LAST"
[4]     16  100000 "TF_VOLUME"
[5]     32   51674 "TF_BUY"
[6]     64   55634 "TF_SELL"
[7]     96    7308 "TF_BUY_SELL (COMBO)"
```

Todos los ticks tenían las banderas TF_LAST y TF_VOLUME, y la mezcla de direcciones de trading se ha producido 7308 veces. De nuevo, la suma de TF_BUY y TF_SELL menos su combinación coincide con el número total de ticks.

El segundo script *SeriesTicksDeltaVolume.mq5* utiliza la función *CopyTicksRange* para calcular los deltas de volumen en cada barra. Como sabe, las cotizaciones de MetaTrader 5 contienen sólo volúmenes

impersonales, en los que las compras y las ventas se combinan en un valor para cada barra. Sin embargo, la presencia de un historial de ticks reales permite calcular por separado las sumas de los volúmenes de compra y venta, así como su diferencia. Estas características son factores adicionales importantes para tomar decisiones de trading.

Los parámetros de entrada contienen ajustes similares a los del primer script, en concreto, el nombre del símbolo para el análisis y el modo de solicitud de ticks. Es cierto que, en este caso, deberá especificar además un marco temporal, ya que los deltas de volumen deben calcularse barra por barra. Por defecto se utilizará el marco temporal del gráfico actual. El parámetro *BarCount* permite especificar el número de barras calculadas.

```
input string WorkSymbol = NULL; // Symbol (leave empty for current)
input ENUM_TIMEFRAMES TimeFrame = PERIOD_CURRENT;
input int BarCount = 100;
input COPY_TICKS TickType = INFO_TICKS;
```

Las estadísticas de cada barra se almacenan en la estructura *DeltaVolumePerBar*.

```
struct DeltaVolumePerBar
{
   datetime time; // bar time
   ulong buy;     // net volume of buy operations
   ulong sell;    // net sell operations
   long delta;    // volume difference
};
```

La función *OnStart* describe un array de estructuras de este tipo, mientras que su tamaño se asigna para el número especificado de barras.

```
void OnStart()
{
   DeltaVolumePerBar deltas[];
   ArrayResize(deltas, BarCount);
   ZeroMemory(deltas);
   ...
```

Y aquí está el algoritmo principal.

```
   for(int i = 0; i < BarCount; ++i)
   {
      MqlTick ticks[];
      const datetime next = iTime(WorkSymbol, TimeFrame, i);
      const datetime prev = iTime(WorkSymbol, TimeFrame, i + 1);
      ResetLastError();
      const int n = CopyTicksRange(WorkSymbol, ticks, COPY_TICKS_ALL,
         prev * 1000, next * 1000 - 1);
      if(n > -1 && _LastError == 0)
      {
         ...
      }
   }
```

En el bucle a través de las barras obtenemos el intervalo de tiempo para cada barra: *prev* y *next* (la 0ª barra incompleta no se procesa). Cuando llame a *CopyTicksRange* para este intervalo, recuerde traducir

datetime a milisegundos y restar 1 milisegundo desde el borde derecho, ya que este tiempo pertenece a la barra siguiente. En ausencia de errores, procesamos el array de ticks recibidos en un bucle.

```
deltas[i].time = prev; // remember the bar time
for(int j = 0; j < n; ++j)
{
   // when real volumes can be available, take them from ticks
   if(TickType == TRADE_TICKS)
   {
      // separately accumulate volumes for buy and sell deals
      if((ticks[j].flags & TICK_FLAG_BUY) != 0)
      {
         deltas[i].buy += ticks[j].volume;
      }
      if((ticks[j].flags & TICK_FLAG_SELL) != 0)
      {
         deltas[i].sell += ticks[j].volume;
      }
   }
   // when there are no real volumes, we evaluate them by the price movement
   else
   if(TickType == INFO_TICKS && j > 0)
   {
      if((ticks[j].flags & (TICK_FLAG_ASK | TICK_FLAG_BID)) != 0)
      {
         const long d = (long)(((ticks[j].ask + ticks[j].bid)
                     - (ticks[j - 1].ask + ticks[j - 1].bid)) / _Point);
         if(d > 0) deltas[i].buy += d;
         else deltas[i].sell += -d;
      }
   }
}
deltas[i].delta = (long)(deltas[i].buy - deltas[i].sell);
```

Si en la configuración del script se solicitó el análisis por ticks de trading (TRADE_TICKS), compruebe la presencia de las banderas TICK_FLAG_BUY y TICK_FLAG_SELL, y si al menos una de ellos está activada, tenga en cuenta el volumen del campo *volume* en la variable correspondiente de la estructura *DeltaVolumePerBar*. Este modo sólo es adecuado para instrumentos bursátiles. En el caso de los instrumentos de Forex, las banderas de volúmenes y dirección de las operaciones de trading no se rellenan, por lo que debe utilizarse un enfoque diferente.

Si en los ajustes se especifican los ticks de información (INFO_TICKS) disponibles para todos los instrumentos, el algoritmo se basa en las siguientes reglas empíricas. Como sabe, la compra hace subir el precio y la venta lo hace bajar. Por lo tanto, podemos suponer que si el precio medio *Ask+Bid* ha subido en un nuevo tick con respecto al anterior, se ha ejecutado una operación de compra sobre el mismo, y si el precio ha bajado, se ha producido una operación de venta. El volumen puede estimarse aproximadamente como el número de puntos pasados (*_Point*).

Los resultados de los cálculos se muestran simplemente como un array de estructuras con las estadísticas recopiladas.

```
   PrintFormat("Delta volumes per intraday bar\nProcessed %d bars on %s %s %s",
      BarCount, StringLen(WorkSymbol) > 0 ? WorkSymbol : _Symbol,
      EnumToString(TimeFrame == PERIOD_CURRENT ? _Period : TimeFrame),
      EnumToString(TickType));
   ArrayPrint(deltas);
}
```

A continuación se muestran algunos registros de los modos TRADE_TICKS e INFO_TICKS.

```
Delta volumes per intraday bar
Processed 100 bars on YNDX.MM PERIOD_H1 TRADE_TICKS
                  [time] [buy] [sell] [delta]
[ 0] 2021.10.13 11:00:00  7912  14169   -6257
[ 1] 2021.10.13 10:00:00  8470  11467   -2997
[ 2] 2021.10.13 09:00:00 10830  13047   -2217
[ 3] 2021.10.13 08:00:00 23682  19478    4204
[ 4] 2021.10.13 07:00:00 14538  11600    2938
[ 5] 2021.10.12 20:00:00  2132   4786   -2654
[ 6] 2021.10.12 19:00:00  9173  13775   -4602
[ 7] 2021.10.12 18:00:00  1297   1719    -422
[ 8] 2021.10.12 17:00:00  3803   2995     808
[ 9] 2021.10.12 16:00:00  6743   7045    -302
[10] 2021.10.12 15:00:00 17286  37286  -20000
[11] 2021.10.12 14:00:00 33263  54157  -20894
[12] 2021.10.12 13:00:00 56060  52659    3401
[13] 2021.10.12 12:00:00 12832  10489    2343
[14] 2021.10.12 11:00:00  7530   6092    1438
[15] 2021.10.12 10:00:00  6268  25201  -18933
...
```

Los valores, por supuesto, son significativamente diferentes, pero la cuestión no está en los valores absolutos: en ausencia de volúmenes de intercambio, incluso tal emulación de la dinámica delta y de división nos permite observar el comportamiento del mercado desde un ángulo diferente.

```
Delta volumes per intraday bar
Processed 100 bars on YNDX.MM PERIOD_H1 INFO_TICKS
                  [time]  [buy] [sell] [delta]
[ 0] 2021.10.13 11:00:00   1939   2548    -609
[ 1] 2021.10.13 10:00:00   2222   2400    -178
[ 2] 2021.10.13 09:00:00   2903   2909      -6
[ 3] 2021.10.13 08:00:00   4489   4060     429
[ 4] 2021.10.13 07:00:00   4999   4285     714
[ 5] 2021.10.12 20:00:00   1444   1556    -112
[ 6] 2021.10.12 19:00:00   5464   5867    -403
[ 7] 2021.10.12 18:00:00   2522   2653    -131
[ 8] 2021.10.12 17:00:00   2111   2017      94
[ 9] 2021.10.12 16:00:00   4617   6096   -1479
[10] 2021.10.12 15:00:00   5716   5411     305
[11] 2021.10.12 14:00:00  10044  10866    -822
[12] 2021.10.12 13:00:00  10893  11178    -285
[13] 2021.10.12 12:00:00   2822   2783      39
[14] 2021.10.12 11:00:00   2070   1936     134
[15] 2021.10.12 10:00:00   2053   2303    -250
...
```

Cuando aprendamos a crear indicadores podremos incrustar este algoritmo en uno de ellos (véase *IndDeltaVolume.mq5* en la sección Esperar datos y gestionar la visibilidad) para visualizar los deltas directamente en el gráfico.

5.4 Creación de indicadores personalizados

Los indicadores son uno de los tipos de programas MQL más populares. Se trata de herramientas sencillas pero potentes para el análisis técnico. El principal mecanismo de su utilización es el tratamiento de los datos de precios iniciales mediante fórmulas para crear series temporales derivadas. Esto permite evaluar y visualizar las características específicas de los procesos de mercado. Cualquier serie temporal, incluidas las obtenidas como resultado de los cálculos de un indicador, puede introducirse en otro indicador, y así sucesivamente. Las fórmulas de muchos indicadores conocidos (por ejemplo, MACD) consisten en realidad en llamadas a varios indicadores interrelacionados.

Los usuarios de terminales están sin duda familiarizados con muchos indicadores integrados, y también saben que la lista de indicadores disponibles puede ampliarse utilizando el lenguaje MQL5. Desde el punto de vista del usuario, los indicadores integrados y personalizados implementados en MQL5 funcionan exactamente de la misma manera.

Por regla general, los indicadores muestran los resultados de sus operaciones en forma de líneas, histogramas y otras construcciones gráficas en la ventana del gráfico de precios. Cada uno de estos gráficos se visualiza sobre la base de series temporales calculadas, que se almacenan dentro de los indicadores en arrays especiales denominados búferes de indicadores: están disponibles para su visualización en el terminal *Data Window* junto con los precios OHLC. No obstante, los indicadores pueden ofrecer funciones adicionales además de los búferes, o no tener ningún búfer. Por ejemplo, los indicadores se utilizan a menudo para resolver problemas en los que es necesario crear objetos gráficos, gestionar el gráfico y sus propiedades, e interactuar con el usuario (véase *OnChartEvent*).

En este capítulo estudiaremos los principios básicos de la creación de indicadores en MQL5. Estos indicadores suelen denominarse «personalizados» porque el usuario puede escribirlos desde cero o

compilarlos a partir de códigos fuente ya hechos. En el próximo capítulo abordaremos las cuestiones de la gestión programática de los indicadores personalizados e integrados, lo que nos permitirá construir indicadores más complejos y allanar el camino para las señales de trading basadas en indicadores y los filtros para Asesores Expertos.

Un poco más adelante dominaremos la tecnología de introducir indicadores en programas MQL ejecutables en forma de recursos.

5.4.1 Principales características de los indicadores

El indicador implementa un determinado algoritmo de cálculo aplicado por barras a una serie temporal inicial dada o a varias series temporales. Todas estas series temporales son los arrays *terminal's own* (véase la función *ArrayIsSeries*): el terminal les asigna memoria y añade nuevos elementos cada vez que se forman nuevas barras. Naturalmente, entre estos arrays, los arrays con cotizaciones de símbolos en diferentes marcos temporales desempeñan un papel fundamental, ya que son rellenadas por el terminal. Sin embargo, los indicadores lanzados pueden ampliar considerablemente el conjunto de series temporales disponibles para el análisis.

El indicador suele guardar los resultados de sus operaciones en arrays dinámicos, que se registran como búferes de indicadores mediante una función especial (*SetIndexBuffer*) y también se convierten en los arrays propios del terminal. Además de asignarles memoria, el terminal proporciona acceso público a estos arrays como a nuevas series temporales, sobre las que se pueden calcular otros indicadores.

El punto de entrada a la parte calculada del indicador es la función *OnCalculate*: un manejador de eventos del mismo nombre. En Visión general de las funciones de gestión de eventosmencionamos ya esta función: su sola presencia en el código fuente basta para que el programa MQL sea percibido por el terminal como un indicador. La función *OnCalculate* se describirá detalladamente en la sección siguiente. En concreto, la principal característica de *OnCalculate* es la presencia de dos formas diferentes. El programador debe seleccionar la opción al principio del diseño del indicador, ya que esto determina la finalidad y los posibles casos de uso.

La función *OnCalculate* no es la única característica distintiva del indicador. Además, un grupo de directivas *#property* especiales del preprocesador está destinado exclusivamente a los indicadores: los analizaremos paso a paso en varias secciones relevantes de este capítulo. Anteriormente ya hemos visto algunas Propiedades generales del programa, y estas directivas, por supuesto, también se aplican a los indicadores.

Como saben los usuarios de MetaTrader 5, cada indicador tiene una forma de mostrar sus construcciones gráficas (series temporales): en la ventana principal que muestra los precios de los símbolos o en una subventana separada. Esta subventana se crea en la parte inferior de la ventana cuando se añade un indicador específico (o un grupo de indicadores) al gráfico si está diseñado para trabajar en una subventana. Por ejemplo, el indicador estándar de Media Móvil (MA) se dibuja en el gráfico de precios, mientras que el Williams Percent Range (WPR) se dibuja en una subventana aparte.

Desde el punto de vista del desarrollador, esto significa que usted debe determinar inicialmente si el indicador se mostrará en la ventana principal o en una subventana, ya que estos dos modos no pueden combinarse. Además, esta característica, así como el número de búferes de indicador, sólo puede fijarse una vez mediante las directivas *#property* (véase Dos tipos de indicadores y Ajuste del número de búferes y trazados), y entonces no será posible cambiarlos utilizando llamadas a funciones de la API de MQL5, ya que dichas funciones simplemente no se proporcionan. A diferencia de estos atributos inmutables, la mayoría de las demás propiedades de los indicadores pueden ajustarse dinámicamente

mediante funciones especiales. Así, a medida que estudiemos los aspectos técnicos de la programación de indicadores, podremos establecer correspondencias entre las propiedades de *#property* y las funciones MQL5.

Además, los indicadores suelen implementar los manejadores *OnInit* y *OnDeinit* (véase Eventos de referencia de indicadores y Asesores Expertos). *OnInit* es especialmente importante para asignar arrays que actuarán como búferes de indicadores, es decir, para acumular los resultados de los cálculos intermedios y finales, visibles para el usuario y disponibles para otros programas, como los Asesores Expertos.

El indicador es uno de los programas MQL interactivos que puede, si es necesario, trabajar con eventos de temporizador (*OnTimer*) y cambios en los gráficos (*OnChartEvent*) producidos por el usuario u otros programas. Estas características técnicas son opcionales para los indicadores y se basan en la cola de eventos gráficos. Los trataremos por separado en el capítulo sobre gráficos.

5.4.2 Evento indicador principal: OnCalculate

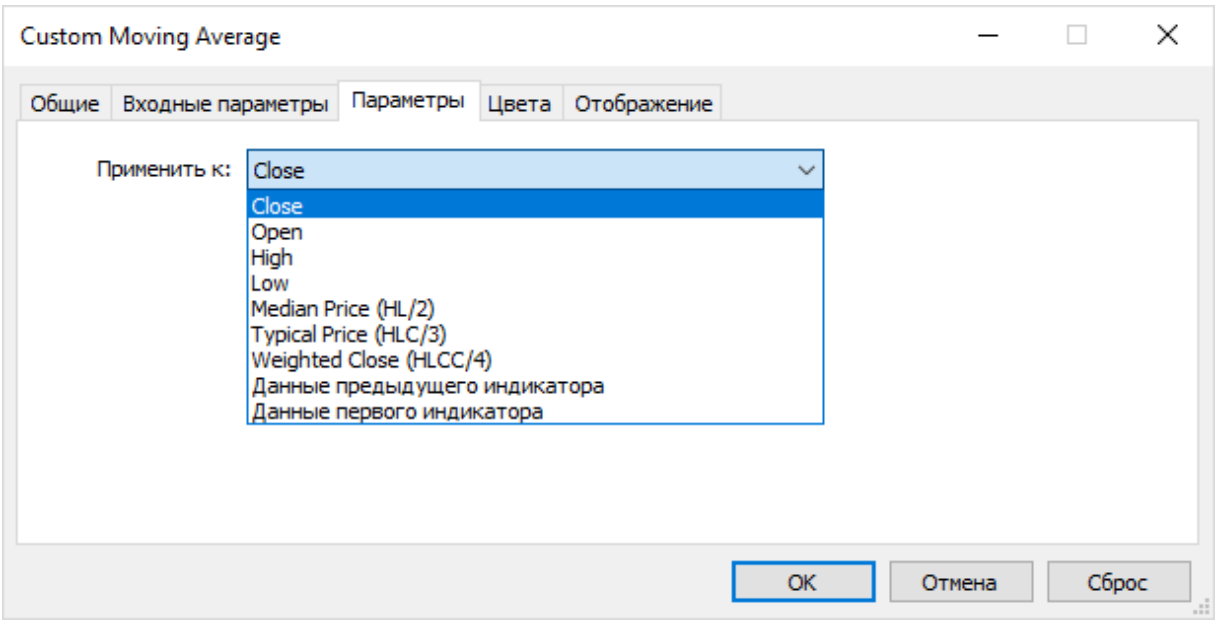
La función *OnCalculate* es el principal punto de entrada al código de indicador MQL5. Se llama cada vez que se produce el evento OnCalculate, que se genera cuando cambian los datos del precio. Por ejemplo, puede ocurrir cuando llega un nuevo tick para un símbolo o cuando cambian los precios antiguos (rellenando un hueco en el historial o descargando los datos que faltan del servidor).

Existen dos variantes de la función, que difieren en el material de partida para los cálculos:

- Completa: proporciona un conjunto de series temporales de precios estándar en los parámetros (precios OHLC, volúmenes, diferenciales).
- Reducida: para una serie temporal arbitraria (no necesariamente estándar).

Un indicador debe utilizar sólo una de las dos opciones, mientras que es imposible combinarlas en un indicador.

En el caso de utilizar la forma reducida de *OnCalculate*, al colocar un indicador en un gráfico, aparece una pestaña adicional en su cuadro de diálogo de propiedades. Proporciona una lista desplegable *Apply to*, en la que debe seleccionar la serie temporal inicial en base a la cual se calculará el indicador. Por defecto, si no se selecciona ninguna serie temporal, el cálculo se basa en los valores de precios de *Close*.



Selección de la serie temporal inicial para el indicador con la forma abreviada OnCalculate

La lista siempre ofrece tipos estándar de precios, pero si hay otros indicadores en el gráfico, esta configuración le permite seleccionar uno de ellos como fuente de datos para otro indicador, construyendo así una cadena de procesamiento a partir de indicadores. Intentaremos construir un indicador a partir de otro en la sección Omitir dibujo en barras iniciales. Cuando se utiliza el formulario completo, esta opción no está disponible.

> Está prohibido aplicar indicadores a los siguientes indicadores integrados: Fractales, Gator, Ichimoku y SAR Parabólico.

La forma abreviada de *OnCalculate* tiene el siguiente prototipo.

```
int OnCalculate(const int rates_total, const int prev_calculated, const int begin,
 const double &data[])
```

El array *data* contiene los datos iniciales para el cálculo. Puede ser una de las series temporales de precios o un búfer calculado de otro indicador. El parámetro *rates_total* especifica el tamaño del array *data*. *ArraySize(data)* o las llamadas a *iBars(NULL, 0)* deberían dar el mismo valor que *rates_total*.

El parámetro *prev_calculated* está diseñado para recalcular efectivamente el indicador en un pequeño número de barras nuevas (normalmente en una, la última), en lugar de un cálculo completo en todas las barras. El valor *prev_calculated* es igual al resultado de la función *OnCalculate* devuelto al tiempo de ejecución desde una llamada de función anterior. Por ejemplo, si al recibir el siguiente tick, el indicador ha calculado la fórmula para todas las barras, debería devolver el valor de $rates\_total_A$ desde *OnCalculate* (aquí el índice A significa el momento inicial). A continuación, en el siguiente tick, al recibir el evento *OnCalculate*, el terminal fijará *prev_calculated* al valor anterior $rates\_total_A$. No obstante, el número de barras durante este tiempo puede haber cambiado ya, y el nuevo valor *rates_total* aumentará; llamémoslo $rates\_total_B$. Por lo tanto, sólo las barras de *prev_calculated* (alias $rates\_total_A$) hasta $rates\_total_B$ se calculará.

No obstante, la situación más común es cuando los nuevos ticks encajan en la barra cero actual, es decir, *rates_total* no cambia, y por lo tanto en la mayoría de las llamadas a *OnCalculate*, tenemos la igualdad *prev_calculated* == *rates_total*. ¿Tenemos que recalcular algo en este caso? Depende de la naturaleza de los cálculos. Por ejemplo, si el indicador se calcula a partir de los precios de apertura de

la barra, que no cambian, no tiene sentido recalcular nada. Sin embargo, si el indicador utiliza el precio de cierre (de hecho, el precio del último tick conocido) o cualquier otro precio de resumen que dependa de *Close*, entonces siempre deberá recalcularse la última barra.

La primera vez que se llama a la función *OnCalculate*, el valor de *prev_calculated* es igual a 0.

Si desde la última llamada a la función *OnCalculate*, los datos de precios han cambiado (por ejemplo, se ha cargado un historial más profundo o se han rellenado huecos), entonces el valor del parámetro *prev_calculated* también será puesto a 0 por el terminal. Así, el indicador recibirá una señal para un recálculo completo sobre todo el historial disponible.

Si la función *OnCalculate* devuelve un valor nulo, el indicador no se dibuja, y los nombres y valores de sus búferes en *Data window* se ocultarán.

> Tenga en cuenta que la devolución del número completo de barras *rates_total* es la única forma estándar de indicar al terminal y a otros programas MQL que utilizarán el indicador que sus datos están listos. Aunque un indicador esté diseñado para calcular y mostrar sólo una cantidad limitada de datos, debería devolver *rates_total*.

La dirección de indexación del array *data* puede seleccionarse llamando a *ArraySetAsSeries* (el valor por defecto es *false*, que puede verificarse llamando a *ArrayGetAsSeries*). Al mismo tiempo, si aplicamos la función *ArrayIsSeries* al array, devolverá *true*. Esto significa que este array es un array interno, gestionado por el terminal. El indicador no puede modificarlo de ninguna manera, sino sólo leerlo, sobre todo porque hay un modificador *const* en la descripción del parámetro.

El parámetro *begin* informa del número de valores iniciales del array *data* que deben excluirse del cálculo. El parámetro lo establece el sistema cuando nuestro indicador es configurado por el usuario de forma que recibe *data* de otro indicador (ver imagen superior). Por ejemplo, si el indicador de fuente de datos seleccionado calcula un período de media móvil *N*, entonces las primeras barras *N - 1*, por definición, no contienen datos fuente, ya que allí es imposible calcular la media sobre las barras *N*. Si el desarrollador ha establecido una propiedad especial en este indicador fuente, se nos pasará correctamente en el parámetro *begin*. Pronto comprobaremos este aspecto en la práctica (véase la sección Omitir dibujo en barras iniciales).

Vamos a intentar crear un indicador vacío con una forma abreviada de *OnCalculate*. Aún no podrá hacer nada, pero servirá de preparación para futuros experimentos. El archivo original *IndStub.mq5* se encuentra en la carpeta *MQL5/Indicators/MQL5Book/p5/*. Para asegurarnos de que el indicador funciona, vamos a añadir lo siguiente a *OnCalculate*: la posibilidad de mostrar los valores *prev_calculated* y *rates_total* en el registro y de contar el número de llamadas a la función.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   static int count = 0;
   ++count;
   // compare the number of bars on the previous call and the current one
   if(prev_calculated != rates_total)
   {
      // signal only if there is a difference
      PrintFormat("calculated=%d rates=%d; %d ticks",
         prev_calculated, rates_total, count);
   }
   return rates_total; // return the number of processed bars
}
```

La condición para la desigualdad de *prev_calculated* y *rates_total* garantiza que el mensaje sólo aparecerá la primera vez que se coloque el indicador en el gráfico, y también cuando aparezcan nuevas barras. Todos los ticks que lleguen durante la formación de la barra actual no cambiarán el número de barras, y por lo tanto *prev_calculated* y *rates_total* serán iguales. Sin embargo, contaremos el número total de ticks en la variable count.

El resto de parámetros siguen sin funcionar, pero poco a poco iremos utilizando todas las posibilidades.

Este código fuente se compila correctamente, pero genera dos advertencias.

```
no indicator window property is defined, indicator_chart_window is applied
no indicator plot defined for indicator
```

Indican la ausencia de algunas directivas *#property* que, aunque no son obligatorias, establecen las propiedades básicas del indicador. En concreto, la primera advertencia dice que no se ha seleccionado ningún método de vinculación para el indicador: la ventana principal o la subventana, y por lo tanto se utilizará por defecto la ventana principal del gráfico. La segunda advertencia está relacionada con el hecho de que no hemos establecido el número de gráficos que se van a mostrar. Como ya se ha mencionado, algunos indicadores están diseñados a propósito sin búferes porque se han diseñado para realizar otras acciones, pero en nuestro caso se trata de un recordatorio para añadir una parte visual más adelante.

Nos ocuparemos de la eliminación de las advertencias en un par de párrafos, pero por ahora, lanzaremos el indicador en el gráfico EURUSD,M1. Utilizamos el marco temporal M1 porque así podemos ver rápidamente la formación de nuevas barras y la aparición de mensajes en el registro.

```
calculated=0 rates=10002; 1 ticks
calculated=10002 rates=10003; 30 ticks
calculated=10003 rates=10004; 90 ticks
calculated=10004 rates=10005; 167 ticks
calculated=10005 rates=10006; 240 ticks
```

Así, vemos que se llama al manejador *OnCalculate* como se esperaba, y se pueden realizar cálculos en él, tanto en cada tick como en barras. El indicador puede eliminarse del gráfico llamando al cuadro de diálogo *Indicator List* desde el menú contextual del gráfico: seleccione el indicador deseado y pulse *Delete*.

Ahora volvamos a otro prototipo de la función *OnCalculate*. Hemos visto probar en la práctica una versión reducida, pero podríamos aplicar exactamente el mismo espacio en blanco para el formulario completo.

El formulario completo está diseñado para el cálculo basado en series temporales de precios estándar y tiene el siguiente prototipo:

```
int OnCalculate(const int rates_total, const int prev_calculated, const datetime &time[],
 const double &open[], const double &high[], const double &low[], const double &close[],
 const long &tick_volume[], const long &volume[], const int &spread[])
```

Los parámetros *rates_total* y *prev_calculated* tienen el mismo significado que en el formulario simple de *OnCalculate*: *rates_total* establece el tamaño de la serie temporal transmitida (todos los arrays tienen la misma longitud, ya que este es el número total de barras del gráfico), y *prev_calculated* contiene el número de barras procesadas en la llamada anterior (es decir, el valor que la función *OnCalculate* devolvió anteriormente al terminal mediante la sentencia return).

Los arrays *open*, *high*, *low* y *close* contienen los precios relevantes para las barras del gráfico actual: las series temporales del símbolo de trabajo y el marco temporal. El array *time* contiene la hora de apertura de cada barra, y *tick_volume* y *volume* contienen los volúmenes de trading (tick y exchange) por barra.

En el capítulo anterior estudiamos las series temporales con los tipos de precio y volumen estándar proporcionados por el terminal para los programas MQL mediante un conjunto de funciones. Por lo tanto, para la conveniencia de calcular el indicador, estas series temporales se pasan al manejador *OnCalculate* directamente por referencia como arrays. Esto elimina la necesidad de llamar a estas funciones y copiar (duplicar) cotizaciones en arrays internos. Por supuesto, esta técnica sólo es adecuada para aquellos indicadores que se calculan en una combinación de un símbolo de trabajo y un marco temporal que coincide con el gráfico actual. Sin embargo, MQL5 permite crear indicadores multidivisa y de marco temporal múltiple, así como indicadores para símbolos y periodos distintos a los del gráfico actual. En todos estos casos ya es imposible prescindir de las funciones de acceso a las series temporales. Un poco más adelante veremos cómo se hace.

Si comprobamos para todos los arrays pasados si pertenecen al terminal utilizando *ArrayIsSeries*, esta función devolverá *true*. Todos los arrays son de sólo lectura. El modificador *const* en la descripción del parámetro también lo subraya.

Elija entre la forma completa y la reducida en función de los datos que necesite el algoritmo de cálculo. Por ejemplo, para suavizar un array utilizando el algoritmo de media móvil, sólo se requiere un array de entrada y, por lo tanto, el indicador puede construirse para cualquier tipo de precio que el usuario elija. Sin embargo, los indicadores conocidos *ParabolicSAR* o *ZigZag* exigen los precios *High* y *Low*, y por lo tanto, deben utilizar la versión completa de *OnCalculate*. En las siguientes secciones veremos ejemplos de indicadores tanto para la versión simple de *OnCalculate*, como para la versión completa.

5.4.3 Dos tipos de indicadores: para la ventana principal y para la subventana

Como usted sabe, los indicadores en MetaTrader 5 pueden mostrar sus líneas en dos lugares: en la ventana del gráfico principal en la parte superior de las cotizaciones o en una ventana separada creada por debajo del gráfico de precios. Estos dos modos se excluyen mutuamente: cada indicador está diseñado para la ventana principal o para una subventana, pero no puede combinar ambos métodos.

Existen varias soluciones alternativas para los casos en los que se requiere que el programa visualice los datos en ambas ventanas. Por ejemplo, un proyecto puede implementarse en forma de dos indicadores

que interactúan (el aspecto técnico de la interacción queda abierto: pueden ser recursos, archivos, sistema de gestión de bases datos (DBMS) o memoria compartida a la que se accede a través de una DLL). Otro enfoque consiste en utilizar los búferes de indicadores en una de las ventanas, por ejemplo, en el panel inferior, y realizar la visualización en el gráfico principal utilizando objetos gráficos.

Se pueden aplicar varios indicadores tanto en la ventana principal como en la subventana. Si el indicador está diseñado para trabajar en una ventana independiente, al arrastrarlo con el ratón desde el Navegador a la ventana principal se creará automáticamente una nueva ventana para este indicador. Sin embargo, si la ventana ya tiene una subventana con otro indicador, entonces el nuevo puede arrastrarse al mismo lugar, alineando así dos o más indicadores. En este caso, son posibles varios modos de escalar indicadores en una ventana. Por defecto, las construcciones de cada indicador se escalan automática e independientemente unas de otras a la altura total del panel, pero esto puede cambiarse (véase el ejemplo *SubScaler.mq5* en la sección sobre eventos de teclado).

La ventana de visualización del indicador se selecciona mediante una de las dos directivas de compilación.

```
#property indicator_chart_window     // display the indicator in the chart window
#property indicator_separate_window  // display the indicator in a separate window
```

El desarrollador del indicador debería insertar uno de ellos al principio del código fuente. Si ninguna de las directivas está presente, la opción por defecto lo mostrará en la ventana principal, pero el compilador generará una advertencia. Ya lo vimos en la sección anterior. En los siguientes ejemplos nos aseguraremos de indicar *#property indicator_chart_window* o *#property indicator_separate_window*.

La segunda advertencia de compilación *IndStub.mq5* se refería a la falta de configuración de búferes y gráficos. Nos ocuparemos de ellos en la próxima sección.

> La acción de la lista desplegable *Apply to* en la configuración del indicador depende de la ventana para la que se diseñó.
>
> Un indicador para una ventana individual puede ser *Applied* al indicador de la subventana, pero no al indicador de la ventana principal.
>
> Sin embargo, el indicador de la ventana principal puede ser *Applied to* cualquier indicador, tanto al de la ventana principal como al de la subventana.

5.4.4 Ajuste del número de búferes y trazados

Para que el indicador muestre los resultados de sus cálculos en el gráfico, debe definir uno o varios arrays y declararlos como búferes del indicador. El número de búferes se establece mediante la directiva:

```
#property indicator_buffers N
```

Aquí N es un número entero de 1 a 512. Esta directiva establece el número de búferes que estarán disponibles en el código para calcular el indicador.

N debe ser una constante de tipo entero (literal) o una definición de macro equivalente. Dado que se trata de una directiva de preprocesador, aún no existe ninguna variable (ni siquiera con el modificador *const*) en la fase de preprocesamiento del código fuente.

No obstante, los búferes no bastan para visualizar los datos calculados. En MQL5, el sistema de visualización es de dos niveles. El primer nivel está formado por los búferes de indicadores, que son

arrays dinámicos que almacenan datos para su visualización. El segundo nivel es para gestionar cómo se mostrarán estos datos. Se construye sobre la base de nuevas entidades denominadas construcciones gráficas (o diagramas, o trazados). La cuestión es que las distintas formas de mostrar los datos pueden requerir distintos números de búferes de indicadores. Por ejemplo, la media móvil tiene exactamente un valor por barra y, por lo tanto, un búfer de indicador es suficiente para un gráfico lineal de este tipo. Sin embargo, para visualizar un gráfico de velas, se necesitan 4 valores por barra (precios OHLC). Así, un trazado gráfico de este tipo requiere 4 búferes de indicadores.

El número de gráficos (P) también debe definirse en el código fuente mediante una directiva especial.

```
#property indicator_plots   P
```

En el caso más sencillo, el número de búferes y diagramas es el mismo, pero pronto analizaremos ejemplos en los que se necesitan más búferes que construcciones gráficas. Además de las situaciones en las que la construcción gráfica de un tipo concreto requiere un número predeterminado de búferes, a veces tenemos que hacer frente a la necesidad de asignar uno o varios arrays para cálculos intermedios. Dichos arrays no intervienen directamente en el renderizado, pero contienen datos para construir búferes de renderizado. Por supuesto, puede utilizar arrays dinámicos simples para tales fines sin declararlos como búferes, pero entonces tendríamos que controlarlos y redimensionarlos de forma independiente: es mucho más práctico convertirlos en búferes y así ordenar al terminal que asigne memoria.

> El número de búferes y trazados gráficos sólo se puede establecer mediante directivas de preprocesador; estas propiedades no se pueden cambiar dinámicamente utilizando funciones MQL5.

Una vez determinado el número de buffers y gráficos, se deben describir en el código fuente los propios arrays, que se convertirán en búferes de indicadores.

Empecemos por desarrollar un nuevo indicador de ejemplo *IndReplica1.mq5* para demostrar las partes necesarias en el código fuente. La esencia del indicador será simple: en su único búfer, mostraremos los valores del array de parámetros *data* recibidos. Como dijimos anteriormente, el usuario selecciona una serie temporal específica para transferir al array *data* en el momento en que el indicador se aplica al gráfico; por defecto, se ofrecerá una serie temporal con precios de cierre de barra.

Vamos a añadir directivas que describan un búfer y un gráfico.

```
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots 1
```

Las directivas no asignan el búfer en sí, sino que sólo establecen las propiedades del indicador y preparan el sistema de ejecución para que el programa determine y configure el número especificado de arrays. A continuación, vamos a ver cómo registrar un array como búfer.

5.4.5 Asignación de un array como búfer: SetIndexBuffer

La función de los búferes de indicadores puede ser desempeñada por cualquier array dinámico del tipo *double* a lo largo de la vida útil desde el inicio del programa hasta su finalización. La forma más habitual de definir un array de este tipo es a nivel global, pero en algunos casos es más conveniente establecer arrays como miembros de clases y luego crear objetos globales con arrays. Examinaremos ejemplos de este tipo de enfoque al aplicar un indicador multidivisa (véase el ejemplo *IndUnityPercent.mq5* en la sección Indicadores multidivisa y de marco temporal múltiple) e indicador de volumen delta (véase *IndDeltaVolume.mq5* en la sección Esperar datos y gestionar la visibilidad).

Así pues, vamos a describir un array dinámico *buffer* a nivel global (sin dimensionamiento).

```
double buffer[];
```

Puede registrarse como búfer mediante la función especial *SetIndexBuffer* del terminal. Por regla general, se llama en el manejador *OnInit*, como muchas otras funciones para configurar el indicador, que comentaremos más adelante.

```
bool SetIndexBuffer(int index, double buffer[],
 ENUM_INDEXBUFFER_TYPE mode = INDICATOR_DATA)
```

La función enlaza el búfer de indicador especificado por *index* con el array dinámico *buffer*. El valor de *index* debe estar comprendido entre 0 y N - 1, donde N es el número de búferes especificado por la directiva *#property indicator_buffers*.

Inmediatamente después de la vinculación, el array aún no está listo para trabajar con datos y ni siquiera cambia su tamaño, por lo que la inicialización y todos los cálculos deben realizarse en la función *OnCalculate*. No se puede modificar el tamaño de un array dinámico después de haberlo asignado como un búfer de indicador. En el caso de los búferes de indicadores, todas las operaciones de redimensionamiento las realiza el propio terminal.

La dirección de indexación tras enlazar un array con un búfer de indicador se establece por defecto como en los arrays ordinarios. Si es necesario, puede modificarse mediante la función *ArraySetAsSeries*.

La función *SetIndexBuffer* devuelve *true* en caso de éxito y *false* en caso de error.

El parámetro opcional *mode* indica al sistema cómo se utilizará el búfer. Los valores posibles se proporcionan en el enum ENUM_INDEXBUFFER_TYPE.

Identificador	Descripción
INDICATOR_DATA	Datos para procesar
INDICATOR_COLOR_INDEX	Colores de renderizado
INDICATOR_CALCULATIONS	Resultados internos de los cálculos intermedios

Por defecto, el búfer de indicador está destinado a los datos de dibujo (INDICATOR_DATA). Este valor tiene otro efecto además de mostrar el array en el gráfico: el valor de cada búfer para la barra bajo el cursor del ratón se muestra en la *Data window*. Sin embargo, este comportamiento puede modificarse mediante algunos ajustes del indicador (véase la propiedad PLOT_SHOW_DATA en la sección Ajuste de trazado gráfico). La mayoría de los ejemplos de este capítulo se refieren al modo INDICATOR_DATA.

Si el cálculo del indicador requiere almacenar resultados intermedios para cada barra, se puede asignar un búfer auxiliar no visualizado (INDICATOR_CALCULATIONS) para ellos. Esto es más práctico que utilizar un array ordinario para el mismo propósito, ya que entonces el programador debe controlar independientemente su tamaño. En este capítulo se presentan dos ejemplos con INDICATOR_CALCULATIONS: *IndTripleEMA.mq5* (véase Omitir dibujo en barras iniciales) y *IndSubChartSimple.mq5* (véase Indicadores multidivisa y de marco temporal múltiple).

Algunas construcciones permiten establecer el color de visualización de cada barra. Los búferes de color (INDICATOR_COLOR_INDEX) se utilizan para almacenar información sobre el color. El color se representa mediante el tipo entero *color*, pero todos los búferes de indicadores deben tener el tipo

double, y en este caso almacenan el número de color de una paleta especial establecida por el desarrollador (véase la sección Coloreado de diagramas elemento por elemento e indicador de ejemplo *IndColorWPR.mq5* en él).

Los valores del color y del búfer auxiliar no se muestran en *Data window*, y no pueden obtenerse utilizando la función *CopyBuffer* que exploraremos más adelante en el capítulo sobre Uso de indicadores integrados y personalizados desde MQL5.

El búfer de indicador no se inicializa con ningún valor. Si algunos de sus elementos no se calculan por una razón u otra (por ejemplo, en la configuración del indicador hay un límite en el número máximo de barras o la propia construcción gráfica implica elementos significativos raros entre los que debería haber huecos, como entre los vértices de zigzag), entonces deben rellenarse explícitamente con un valor especial «vacío». El valor vacío no aparece en el gráfico ni en *Data Window*. Por defecto, existe una constante EMPTY_VALUE (DBL_MAX) para ello, pero si es necesario se puede sustituir por cualquier otra, por ejemplo, por 0. Para ello se utiliza la función *PlotIndexSetDouble*.

Dado el nuevo conocimiento sobre la función *SetIndexBuffer*, vamos a completar nuestro siguiente ejemplo *IndReplica1.mq5*, que empezamos en la sección anterior. En concreto, necesitamos el manejador *OnInit*.

```
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots 1

#include <MQL5Book/PRTF.mqh>

double buffer[]; // global dynamic array

int OnInit()
{
   // register an array as an indicator buffer
   PRTF(SetIndexBuffer(0, buffer)); // true / ok
   // the second incorrect call is made here intentionally to show an error
   PRTF(SetIndexBuffer(1, buffer)); // false / BUFFERS_WRONG_INDEX(4602)
   // check size: still 0
   PRTF(ArraySize(buffer)); // 0
   return INIT_SUCCEEDED;
}
```

El número de búferes está definido por la directiva igual a 1, por lo que la asignación de arrays para un solo búfer utiliza el índice 0 (el primer parámetro *SetIndexBuffer*). La segunda llamada a la función es errónea y sólo se añade para demostrar el problema: dado que el índice 1 implica dos búferes declarados, genera un error BUFFERS_WRONG_INDEX (4602).

Al principio de la función *OnCalculate*, vamos a imprimir de nuevo el tamaño del array. En este lugar, se distribuirá ya de acuerdo con el número de barras.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   // after starting, check that the platform automatically manages the size of the a
   if(prev_calculated == 0)
   {
      PRTF(ArraySize(buffer)); // 10189 - actual number of bars
   }
   ...
```

Pasemos ahora a la cuestión de qué calculará nuestro indicador. Como ya se ha mencionado, no pondremos fórmulas complejas en él todavía, sino que simplemente intentaremos copiar las series temporales pasadas desde el parámetro *data* al búfer. Esto se refleja en el nombre del indicador.

```
   ...
   // on each new bar or set of bars (including the first calculation)
   if(prev_calculated != rates_total)
   {
      // fill in all new bars
      ArrayCopy(buffer, data, prev_calculated, prev_calculated);
   }
   else // ticks on the current bar
   {
      // update the last bar
      buffer[rates_total - 1] = data[rates_total - 1];
   }

   // we report the number of processed bars to ourselves in the future
   return rates_total;
}
```

Ahora el indicador se compila sin advertencias. Podemos ejecutarlo en el gráfico, y con la configuración por defecto, debería duplicar los valores de los precios de cierre de las barras en el búfer. Esto se debe a la forma abreviada de *OnCalculate*; hemos analizado este aspecto en la sección Evento indicador principal: OnCalculate.

Sin embargo, hay algo extraño: nuestro búfer se muestra en *Data window* y contiene los valores correctos, pero no hay ninguna línea en el gráfico. Esto es consecuencia del hecho de que las construcciones gráficas, y no los búferes, son responsables de la visualización. En la versión actual del indicador hemos configurado sólo el búfer. En la siguiente sección, crearemos una nueva versión *IndReplica2.mq5* y la completaremos con las instrucciones necesarias.

Al mismo tiempo, el efecto descrito puede ser útil para crear indicadores «ocultos» que no muestren sus líneas en el gráfico pero que estén disponibles para su lectura programática desde otros programas MQL. Si lo desea, el desarrollador podrá ocultar incluso la mención de los búferes de indicadores de *Data windows* (véase PLOT_SHOW_DATA en la sección siguiente).

Abordaremos cómo gestionar indicadores desde el código MQL5 en el capítulo siguiente.

5.4.6 Configuración de trazado: PlotIndexSetInteger

La API de MQL5 proporciona las siguientes funciones para configurar los trazados: *PlotIndexSetInteger*, *PlotIndexSetDouble* y *PlotIndexSetString*. Las propiedades de tipo entero también pueden leerse a través de *PlotIndexGetInteger*. Nos interesan sobre todo las propiedades de tipo entero.

La función *PlotIndexSetInteger* tiene dos formas. Veremos sus diferencias un poco más adelante.

```
bool PlotIndexSetInteger(int index, ENUM_PLOT_PROPERTY_INTEGER property, int value)
bool PlotIndexSetInteger(int index, ENUM_PLOT_PROPERTY_INTEGER property, int modifier,
 int value)
```

La función establece el valor de la propiedad de un trazado gráfico en la dirección *index* especificada. El valor de *index* debe estar comprendido entre 0 y P - 1, donde P es el número de trazados especificado por la directiva *#property indicator_plots*. La propiedad en sí se identifica mediante el parámetro *property*: los valores permitidos deben tomarse de la enumeración ENUM_PLOT_PROPERTY_INTEGER (véase más abajo). El valor de la propiedad se pasa en el parámetro *value*.

La segunda forma de la función se utiliza para las propiedades que se aplican a varios componentes (aunque pertenezcan a la misma propiedad). En particular, para algunos tipos de diagramas, es posible asignar un conjunto de colores en lugar de un color. En este caso, puede utilizar el parámetro *modifier* para cambiar cualquier color de este conjunto.

En caso de éxito, la función devuelve *true*; en caso contrario, devuelve *false*.

En la siguiente tabla se muestran las propiedades ENUM_PLOT_PROPERTY_INTEGER disponibles.

Identificador	Descripción	Tipo de propiedad
PLOT_ARROW	Código de flecha de la fuente Wingdings para gráficos DRAW_ARROW	uchar
PLOT_ARROW_SHIFT	Desplazamiento vertical de la flecha para gráficos DRAW_ARROW	int
PLOT_DRAW_BEGIN	Índice de la primera barra (de izquierda a derecha) donde comienzan los datos	int
PLOT_DRAW_TYPE	Tipo de trazado (gráfico)	ENUM_DRAW_TYPE
PLOT_SHOW_DATA	Bandera para mostrar los valores del trazado en *Data window* (*true* - visible, *false* - no visible)	bool
PLOT_SHIFT	Desplazamiento de los gráficos del indicador a lo largo del eje temporal en barras (desplazamientos positivos a la derecha, negativos a la izquierda).	int
PLOT_LINE_STYLE	Estilo de dibujo lineal	ENUM_LINE_STYLE
PLOT_LINE_WIDTH	Grosor de la línea en píxeles (1 - 5)	int
PLOT_COLOR_INDEXES	Número de colores (1 - 64)	int
PLOT_LINE_COLOR	Color de representación	color (modificador - número de color)

Poco a poco iremos aprendiendo todas las propiedades, pero por ahora nos centraremos en las tres principales: PLOT_DRAW_TYPE, PLOT_LINE_STYLE y PLOT_LINE_COLOR.

Los indicadores en MetaTrader 5 admiten varios tipos de trazado predefinidos. Determinan la representación visual y la estructura necesaria de los búferes con los datos iniciales para la visualización.

Hay 10 de estos trazados básicos en total, y en el nivel MQL5, que se describen mediante identificadores en la enumeración ENUM_DRAW_TYPE. Es a la propiedad PLOT_DRAW_TYPE a la que debe asignarse uno de los valores ENUM_DRAW_TYPE.

Tipo de visualización, ***ejemplos***	**Descripción**	**Número de búferes**
DRAW_NONE *IndDeltaVolume.mq5*	No se muestra nada en el gráfico, pero los valores del búfer correspondiente están disponibles en la ventana de datos	1
DRAW_LINE *IndLabelHighLowClose.mq5* *,IndWPR.mq5,* *IndUnityPercent.mq5*	Línea curva por valores de búfer (los elementos «vacíos» forman un hueco en la línea)	1
DRAW_SECTION	Segmentos rectos que forman una polilínea entre elementos de búfer «no vacíos» (si no hay huecos, similar a DRAW_LINE)	1
DRAW_ARROW *IndReplica3.mq5* *,IndFractals.mq5*	Caracteres (etiquetas)	1
DRAW_HISTOGRAM *IndDeltaVolume.mq5*	Histograma de la línea cero a los valores del búfer	1
DRAW_HISTOGRAM2 *IndLabelHighLowClose.mq5*	Histograma entre los valores de elementos emparejados de dos búferes de indicadores	2
DRAW_ZIGZAG *IndFractalsZigZag.mq5*	Segmentos rectos que forman una polilínea entre elementos sucesivos «no vacíos» de dos búferes (similar a DRAW_SECTION, pero a diferencia de éste permite segmentos verticales en una barra)	2
DRAW_FILLING	Relleno de color del canal entre dos líneas mediante valores emparejados en dos búferes	2
DRAW_BARS *IndSubChartSimple.mq5*	Visualización en barras: se muestran cuatro precios por barra en cuatro búferes adyacentes, en el orden OHLC	4
DRAW_CANDLES *IndSubChartSimple.mq5*	Visualización de velas: se muestran cuatro precios por barra en cuatro búferes adyacentes, en el orden OHLC	4

Esta tabla no enumera todos los elementos ENUM_DRAW_TYPE. Hay análogos de los mismos trazados con soporte para colorear elementos individuales (barras). Las presentaremos en una sección aparte Coloreado de diagramas elemento por elemento. La documentación de MQL5 proporciona ejemplos para todos los tipos, y dentro del ámbito de este libro, hay algunas excepciones: la presencia de indicadores de demostración se indica junto a los nombres de los tipos.

En todos los casos, incluido DRAW_NONE, los datos del búfer están disponibles en otros programas a través de la función *CopyBuffer*.

Una característica adicional del tipo DRAW_NONE es que los valores de dicho búfer no participan en la escala automática del gráfico, que está activada por defecto para los indicadores mostrados en subventanas.

El estilo de las líneas viene determinado por la propiedad PLOT_LINE_STYLE, que también tiene una enumeración con valores ENUM_LINE_STYLE válidos.

Identificador	Descripción
STYLE_SOLID	Línea continua
STYLE_DASH	Línea discontinua
STYLE_DOT	Línea de puntos
STYLE_DASHDOT	Línea de puntos
STYLE_DASHDOTDOT	Guion-dos puntos

Por último, el color de la línea se establece mediante la propiedad PLOT_LINE_COLOR. En el caso más sencillo, esta propiedad contiene un único color para todo el gráfico. Para algunos tipos de gráficos, en particular DRAW_CANDLES, puede especificar varios colores utilizando un parámetro modificador. Hablaremos de ello más adelante (véase el ejemplo *IndSubChartSimple.mq5* en la sección Indicadores multidivisa y de marco temporal múltiple).

Las tres propiedades anteriores bastan para demostrar el indicador *IndReplica2.mq5*. Vamos a añadir dos parámetros de entrada *DrawType* y *LineStyle* de los tipos ENUM_DRAW_TYPE y ENUM_LINE_STYLE respectivamente, y luego llamaremos a la función *PlotIndexSetInteger* varias veces en *OnInit* para establecer las propiedades de renderización del indicador.

```
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots 1

input ENUM_DRAW_TYPE DrawType = DRAW_LINE;
input ENUM_LINE_STYLE LineStyle = STYLE_SOLID;

double buffer[];

int OnInit()
{
   // register an array as an indicator buffer
   SetIndexBuffer(0, buffer);

   // set the properties of the chart numbered 0
   PlotIndexSetInteger(0, PLOT_DRAW_TYPE, DrawType);
   PlotIndexSetInteger(0, PLOT_LINE_STYLE, LineStyle);
   PlotIndexSetInteger(0, PLOT_LINE_COLOR, clrBlue);

   return INIT_SUCCEEDED;
}
```

Para la propiedad PLOT_LINE_COLOR, no creamos una variable de entrada, ya que ésta y algunas otras propiedades están directamente disponibles desde el cuadro de diálogo de propiedades de cualquier indicador, en la pestaña *Colors*. Por defecto, es decir, inmediatamente después del lanzamiento del indicador, el color de la línea será azul. Pero el color, así como el grosor y el estilo de la línea, pueden cambiarse en el cuadro de diálogo (en la pestaña especificada). Nuestro parámetro

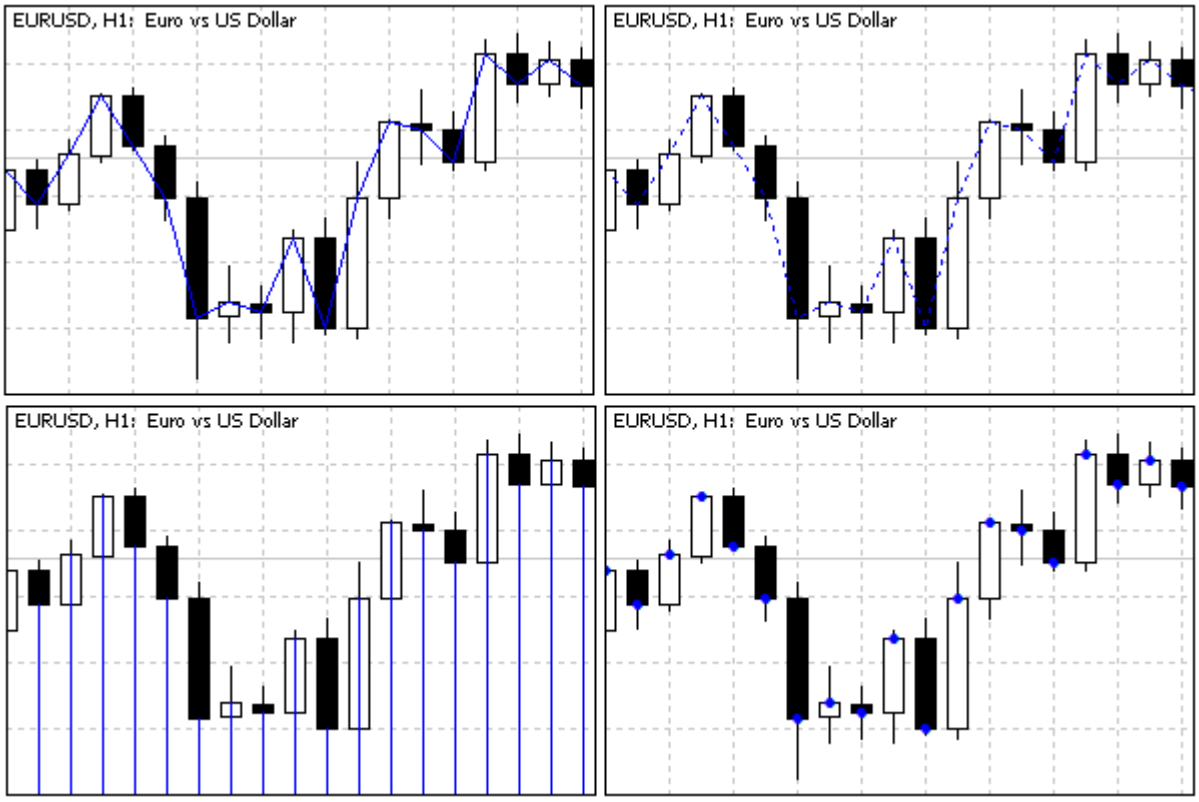
LineStyle duplica parcialmente la celda *Style* correspondiente de la tabla *Colors*. Sin embargo, ofrece ventajas adicionales. Los controles estándar del cuadro de diálogo no permiten seleccionar un estilo cuando el ancho de línea es superior a 1. Al utilizar la variable de entrada *LineStyle*, podemos obtener, por ejemplo, una línea discontinua con una anchura dada de 3 píxeles.

El rellenado del búfer con datos en *OnCalculate* permanece sin cambios en comparación con *IndReplica1.mq5*.

Tras compilar y lanzar el indicador en el gráfico, obtenemos la imagen esperada: una línea azul a los precios de cierre en el gráfico, y los correspondientes precios de cierre de las barras en *Data window*.

Cambiando el parámetro de entrada *DrawType*, podemos cambiar cómo se muestran los datos del búfer. En este caso, sólo debe seleccionar tipos que requieran un único búfer. Cualquier otro tipo de gráfico (DRAW_HISTOGRAM2, DRAW_ZIGZAG, DRAW_FILLING, DRAW_BARS, DRAW_CANDLES) simplemente no puede funcionar en un solo búfer y no mostrará nada. Tampoco tiene sentido elegir los tipos de construcciones con colores (que comienzan con la palabra «Color»), ya que requieren un búfer adicional con números de color en cada barra (como ya se ha mencionado, nos familiarizaremos con esta posibilidad en la sección Coloreado de diagramas elemento por elemento).

A continuación se muestran las opciones de visualización DRAW_LINE, DRAW_SECTION, DRAW_HISTOGRAM y DRAW_ARROW.



Tipos de gráficos de un búfer

Si no fuera por los diferentes estilos especialmente elegidos, STYLE_SOLID para DRAW_LINE y STYLE_DOT para DRAW_SECTION, estos tipos de dibujo serían iguales, porque todos los elementos de nuestro búfer tienen valores «no vacíos». Por defecto, el valor «vacío» significa la constante especial EMPTY_VALUE, que no utilizamos. Las secciones (segmentos) en DRAW_SECTION se dibujan saltándose los elementos «vacíos», y esto sólo se nota si hay alguno. Hablaremos de la instalación de elementos «vacíos» en la sección Visualización de lagunas de datos.

El histograma de la línea cero DRAW_HISTOGRAM se suele utilizar en indicadores con ventana propia, pero aquí se muestra a efectos de demostración. Crearemos un indicador en una subventana con este tipo de renderización en la sección Esperar datos y gestionar la visibilidad (véase el ejemplo *IndDeltaVolume.mq5*).

Para el tipo DRAW_ARROW, el sistema utiliza por defecto el carácter de círculo relleno (código 159), pero puede cambiarlo por otro llamando a *PlotIndexSetInteger(index, PLOT_ARROW, code)*.

Los códigos y apariencia de símbolos de fuentes Wingdings se pueden encontrar en la Ayuda MQL5.

En otra modificación del indicador *IndReplica3.mq5*, añadimos parámetros de entrada para seleccionar el símbolo «flecha» (*ArrowCode*), así como para desplazar estas etiquetas en el gráfico verticalmente (*Arrow padding*) y horizontalmente (*TimeShift*).

```
input uchar ArrowCode = 159;
input int ArrowPadding = 0;
input int TimeShift = 0;
```

El desplazamiento vertical a lo largo de la escala de precios se especifica en píxeles (los valores positivos significan desplazamiento hacia abajo; los negativos, hacia arriba). El desplazamiento horizontal a lo largo de la escala temporal se establece en barras (los valores positivos son un desplazamiento hacia la derecha, hacia el futuro, y los negativos, hacia la izquierda, hacia el pasado). Las nuevas variables de entrada se pasan a las llamadas de *PlotIndexSetInteger* en *OnInit*.

```
int OnInit()
{
   ...
   PlotIndexSetInteger(0, PLOT_DRAW_TYPE, DRAW_ARROW);
   PlotIndexSetInteger(0, PLOT_ARROW, ArrowCode);
   PlotIndexSetInteger(0, PLOT_ARROW_SHIFT, ArrowPadding);
   PlotIndexSetInteger(0, PLOT_SHIFT, TimeShift);
   ...
}
```

La siguiente captura de pantalla muestra un ejemplo de *IndReplica3.mq5* en un gráfico con los ajustes 117 (diamante), -50 (50 puntos hacia arriba), 3 (3 barras a la derecha/adelante).

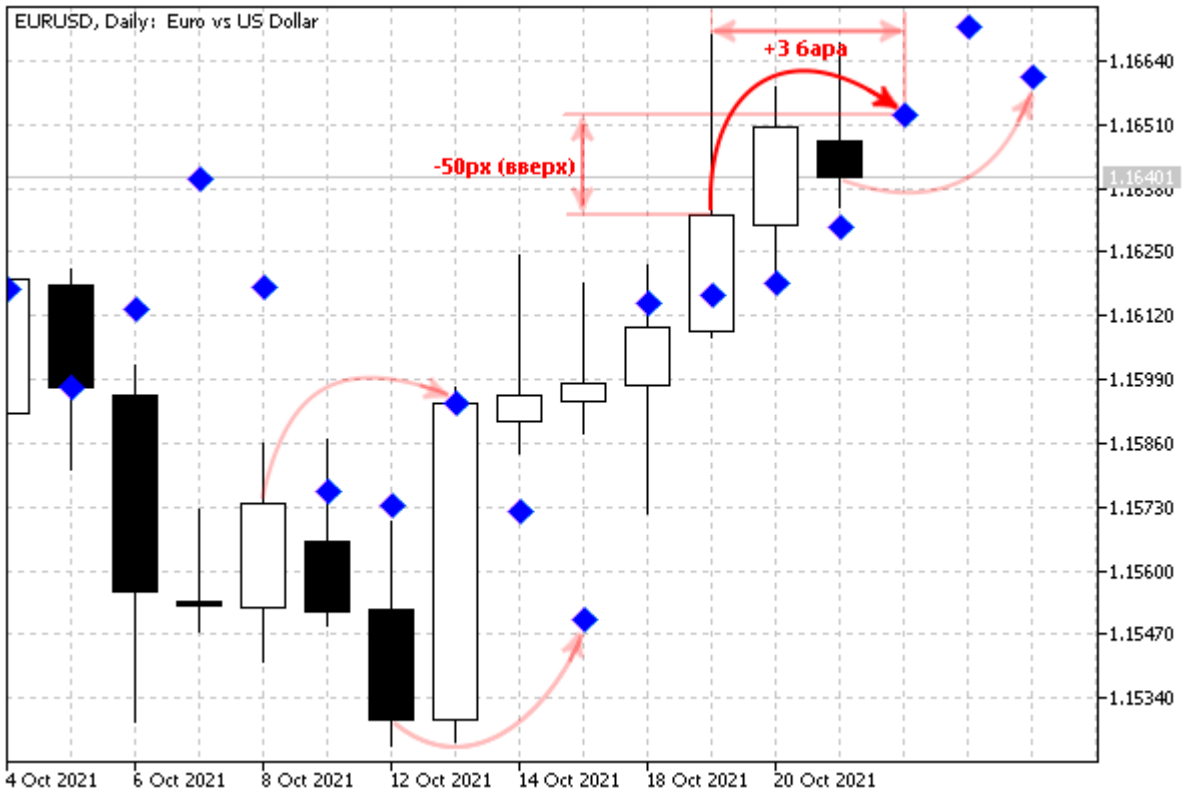


Diagrama de dispersión con desplazamientos verticales y horizontales de las etiquetas

Nuestro indicador por defecto se basa en el tipo de precio *Close* (aunque el usuario puede cambiarlo en el cuadro de diálogo de propiedades, en la lista desplegable *Apply to*). Si es necesario, puede asignar una configuración inicial diferente utilizando la directiva:

```
#property indicator_applied_price PRICE_TYPE
```

Aquí, en lugar de PRICE_TYPE, debe especificar cualquier constante de la lista ENUM_APPLIED_PRICE. También incluye PRICE_CLOSE, que corresponde al valor por defecto. Por ejemplo, la siguiente directiva añadida al código fuente hará que el indicador se base por defecto en el precio típico.

```
#property indicator_applied_price PRICE_TYPICAL
```

Una vez más, observamos que esta configuración sólo especifica el valor por defecto. La variable integrada *_AppliedTo* permite conocer el tipo de precio real sobre el que se construye el indicador. Si el indicador se construye de acuerdo con el descriptor de otro indicador, entonces sólo será posible averiguar este hecho, pero no el nombre de un indicador específico que proporcione los datos.

Para averiguar el estado actual de las propiedades de la enumeración ENUM_PLOT_PROPERTY_INTEGER en el código fuente, utilice la función *PlotIndexGetInteger*.

```
int PlotIndexGetInteger(int index, ENUM_PLOT_PROPERTY_INTEGER property)
int PlotIndexGetInteger(int index, ENUM_PLOT_PROPERTY_INTEGER property, int modifier)
```

La función se utiliza a menudo junto con *PlotIndexSetInteger* para copiar propiedades de dibujo de una línea a otra, o para leer propiedades de los códigos de archivos mqh universales incluidos en el código fuente de varios indicadores.

Por desgracia, no se ofrecen funciones similares en *PlotIndexGetDouble* y *PlotIndexGetString*.

5.4.7 Reglas de asignación de búferes y gráficos

Cuando se registran diagramas utilizando *PlotIndexSetInteger(i, PLOT_DRAW_TYPE, type)*, cada llamada asigna secuencialmente un cierto número de búferes al i-ésimo diagrama de acuerdo con su número requerido para la renderización *type* (véase la tabla ENUM_DRAW_TYPE en la sección anterior). Por lo tanto, este número de búferes no se tiene en cuenta al vincular los búferes a los diagramas siguientes (durante las próximas llamadas a *PlotIndexSetInteger*).

Por ejemplo, si el primer trazado (bajo el índice 0) es DRAW_CANDLES, que requiere 4 búferes de indicadores, se le asociará exactamente este número. Por lo tanto, los búferes indexados del 0 al 3 inclusive se vincularán, y el siguiente búfer libre que se vinculará será el búfer indexado 4.

Si a continuación se registra un gráfico lineal simple DRAW_LINE (su índice en la secuencia de gráficos es 1), sólo ocupará 1 búfer, justo en el índice 4.

Si además se configura un gráfico DRAW_ZIGZAG (el siguiente índice del gráfico es 2), entonces, como utiliza dos búferes, los búferes con índices 5 y 6 irán a él.

Por supuesto, el número de búferes debe ser suficiente para todos los trazados registrados. El ejemplo anterior se ilustra en la siguiente tabla. Sólo tiene 7 búferes y 3 trazados (diagramas).

Índice del búfer en SetIndexBuffer	0	1	2	3	4	5	6
Índice gráfico in PlotIndexSetInteger	0				1	2	
Tipo de renderizado	DRAW_CANDLES				DRAW__LINE__	DRAW_ZIGZAG	

La indexación del búfer y del gráfico es independiente, es decir, el índice del búfer no tiene por qué ser el mismo que el del gráfico. Al mismo tiempo, a medida que aumentan los índices de los gráficos, aumentan los índices de los búferes vinculados a ellos, y la discrepancia en la indexación puede hacerse cada vez mayor si se utilizan tipos de renderización que toman más de un búfer para sí mismos.

Aunque es habitual llamar a las funciones *SetIndexBuffer* antes que a *PlotIndexSetInteger*, esto no es obligatorio. Lo único importante es la correspondencia correcta de los índices de los búferes y los índices de los diagramas. Cuando se utilizan directivas (véase la sección siguiente), que son una alternativa a la llamada a *PlotIndexSetInteger*, las directivas se ejecutan en cualquier caso antes que el manejador *OnInit*.

Para demostrar la diferencia entre el búfer y la indexación de gráficos, considere un ejemplo sencillo de *IndHighLowClose.mq5*. En este archivo, dibujaremos el rango de cada vela entre *High* y *Low* en forma de histograma del tipo DRAW_HISTOGRAM2 y subrayaremos el precio de *Close* con una línea simple DRAW_LINE. Para acceder a series temporales de precios de distintos tipos, también tenemos que cambiar la forma *OnCalculate* de simplificada a completa.

Como el histograma requiere 2 búferes, entonces, junto con el búfer para la línea *Close*, deberíamos describir tres búferes.

```
#property indicator_chart_window
#property indicator_buffers 3
#property indicator_plots 2

double highs[];
double lows[];
double closes[];
```

Regístrelos en *OnInit* por orden de prioridad.

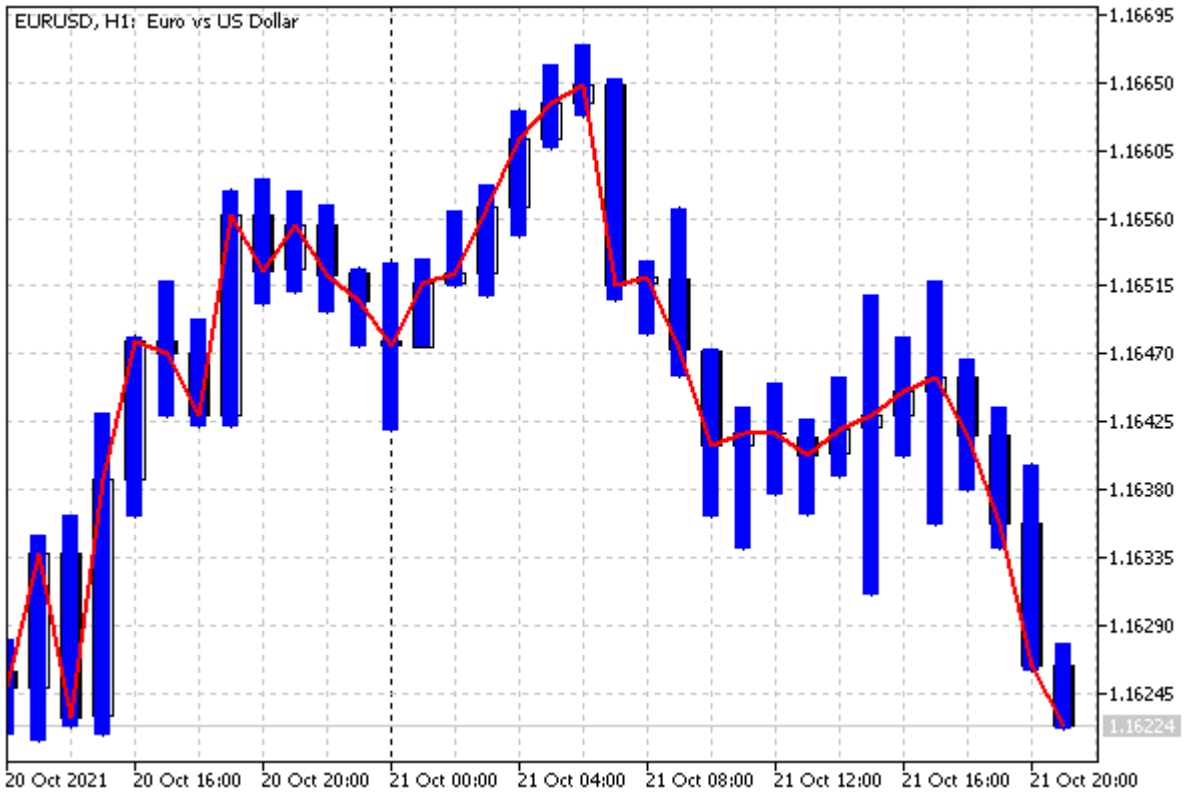
```
int OnInit()
{
   // arrays for buffers for 3 price types
   SetIndexBuffer(0, highs);
   SetIndexBuffer(1, lows);
   SetIndexBuffer(2, closes);

   // drawing a histogram between the High and Low candles under index 0
   PlotIndexSetInteger(0, PLOT_DRAW_TYPE, DRAW_HISTOGRAM2);
   PlotIndexSetInteger(0, PLOT_LINE_WIDTH, 5);
   PlotIndexSetInteger(0, PLOT_LINE_COLOR, clrBlue);

   // drawing the line Close at index 1
   PlotIndexSetInteger(1, PLOT_DRAW_TYPE, DRAW_LINE);
   PlotIndexSetInteger(1, PLOT_LINE_WIDTH, 2);
   PlotIndexSetInteger(1, PLOT_LINE_COLOR, clrRed);

   return INIT_SUCCEEDED;
}
```

Por el camino, la anchura del histograma se fija en 5 píxeles, y la anchura de la línea en 2. Los estilos no se asignan explícitamente, y por defecto son STYLE_SOLID.

Veamos ahora la función *OnCalculate*.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const datetime &time[],
                const double &open[],
                const double &high[],
                const double &low[],
                const double &close[],
                const long &tick_volume[],
                const long &volume[],
                const int &spread[])
{
   // on each new bar or set of bars (including the first calculation)
   if(prev_calculated != rates_total)
   {
      // fill in all new bars
      ArrayCopy(highs, high, prev_calculated, prev_calculated);
      ArrayCopy(lows, low, prev_calculated, prev_calculated);
      ArrayCopy(closes, close, prev_calculated, prev_calculated);
   }
   else // ticks on the current bar
   {
      // update the last bar
      highs[rates_total - 1] = high[rates_total - 1];
      lows[rates_total - 1] = low[rates_total - 1];
      closes[rates_total - 1] = close[rates_total - 1];
   }
   // return the number of processed bars for the next call
   return rates_total;
}
```

El resultado de este indicador se muestra en la siguiente imagen:



Histograma alto-bajo y línea de cierre

Preste atención a un punto importante: los diagramas se representan en el gráfico en el orden correspondiente a sus índices, por lo que algunos son visualmente más altos que otros (se superponen). En este caso, primero se dibuja un histograma con índice 0 y, a continuación, una línea con índice 1. A veces tiene sentido cambiar el orden de registro de los gráficos para proporcionar una mejor visibilidad de las construcciones gráficas más pequeñas, que pueden quedar cubiertas por trazados más grandes (más anchos).

El establecimiento de estas prioridades a lo largo del eje Z imaginario, adentrándose en la pantalla (perpendicular a la pantalla) se denomina orden Z. Volveremos a encontrarnos con esta técnica al estudiar objetos gráficos.

Además, recuerde que, por defecto, los indicadores se muestran encima del gráfico de precios, pero este comportamiento se puede cambiar en la configuración: cuadro de diálogo *Chart Properties*, pestaña *Common*, opción *Chart on foreground*. Existe una opción similar en la interfaz del software (*ChartSetInteger(CHART_FOREGROUND)*, véase la sección Modos de visualización de gráficos).

5.4.8 Aplicación de directivas para personalizar trazados gráficos

Hasta ahora hemos estado personalizando los trazados gráficos mediante llamadas a funciones de *PlotIndexSetInteger*. MQL5 le permite hacer lo mismo utilizando directivas del preprocesador *#property*. La principal diferencia entre estos dos métodos es que las directivas se procesan en tiempo de compilación y las propiedades descritas con ellas se leen del archivo ejecutable durante la carga, incluso antes de que se ejecute el manejador *OnInit* (si existe). Es decir, las directivas proporcionan algunos valores por defecto que pueden utilizarse tal cual si no es necesario cambiarlos.

Por otro lado, la llamada a la función *PlotIndexSetInteger* permite cambiar las propiedades sobre la marcha, durante la ejecución del programa. Cambiar las propiedades dinámicamente mediante

funciones permite crear escenarios más flexibles para utilizar el indicador. Las directivas y las correspondientes llamadas a la función *PlotIndexSetInteger* se muestran en la siguiente tabla:

Directivas	Función	Descripción
indicator_colorN	PlotIndexSetInteger(N-1, PLOT_LINE_COLOR, color)	Color de línea para el trazado
indicator_styleN	PlotIndexSetInteger(N-1, PLOT_LINE_STYLE, type)	Estilo de dibujo de la enumeración ENUM_LINE_STYLE
indicator_typeN	PlotIndexSetInteger(N-1, PLOT_DRAW_TYPE, type)	Tipo de dibujo de la enumeración ENUM_DRAW_TYPE
indicator_widthN	PlotIndexSetInteger(N-1, PLOT_LINE_WIDTH, width)	Grosor de la línea en píxeles (1 - 5)

Tenga en cuenta que la numeración de los trazados en las directivas empieza por 1, mientras que en las funciones empieza por 0. Por ejemplo, la directiva *#property indicator_type1 DRAW_ZIGZAG* equivale a llamar a *PlotIndexSetInteger(0, PLOT_DRAW_TYPE, DRAW_ZIGZAG)*.

También vale la pena señalar que mediante el uso de la función puede establecer muchas más propiedades que a través de directivas: la enumeración ENUM_PLOT_PROPERTY_INTEGER proporciona diez elementos.

> Las propiedades descritas por las directivas están disponibles (visibles y editables por el usuario) en el cuadro de diálogo de configuración del indicador, incluso cuando se coloca en el gráfico por primera vez. En concreto, esto incluye el grosor, el color y el estilo de las líneas (pestaña *Colors*), el número y la colocación de los niveles (pestaña *Levels*). Las mismas propiedades establecidas por las funciones (y si no tienen valores por defecto en las directivas) aparecen en el cuadro de diálogo sólo la segunda vez y siguientes.

Vamos a ajustar el indicador *IndHighLowClose.mq5* para que utilice directivas. La nueva versión se encuentra en el archivo *IndPropHighLowClose.mq5*. El uso de directivas simplifica el manejador *OnInit*; *OnCalculate* no cambia.

```
#property indicator_chart_window
#property indicator_buffers 3
#property indicator_plots 2

// High-Low histogram rendering settings (change index 0 to 1 in the directive)
#property indicator_type1   DRAW_HISTOGRAM2
#property indicator_style1  STYLE_SOLID // by default, can be omitted
#property indicator_color1  clrBlue
#property indicator_width1  5

// close line drawing settings (change index 1 to 2 in the directive)
#property indicator_type2   DRAW_LINE
#property indicator_style2  STYLE_SOLID // by default, can be omitted
#property indicator_color2  clrRed
#property indicator_width2  2

double highs[];
double lows[];
double closes[];

int OnInit()
{
   // arrays for buffers for 3 price types
   SetIndexBuffer(0, highs);
   SetIndexBuffer(1, lows);
   SetIndexBuffer(2, closes);

   return INIT_SUCCEEDED;
}
```

El nuevo indicador es absolutamente igual al anterior.

5.4.9 Configuración de nombres de trazados

En los ejemplos anteriores de este capítulo, los búferes de indicadores de la ventana de datos se designaban por el nombre del propio indicador. Esto no es informativo. La API de MQL5 ofrece la posibilidad de establecer un nombre personalizado para cada búfer. Esto puede hacerse de dos maneras que ya conocemos: utilizando la directiva *#property* y llamando a la función especial *PlotIndexSetString*.

bool PlotIndexSetString(int index, ENUM_PLOT_PROPERTY_STRING property, string value)

El prototipo de la función es similar a *PlotIndexSetInteger* excepto en que el tipo de propiedades (parámetro *value*) es *string*. La función sólo admite una propiedad PLOT_LABEL (es la constante de enumeración ENUM_PLOT_PROPERTY_STRING). El índice de gráficos personalizados del parámetro *index* debe estar comprendido entre 0 y N-1, donde N es el número total de trazados especificado en *#property indicator_plots N*.

Al utilizar la directiva, el índice del gráfico debe ajustarse en 1, ya que la numeración de los trazados en las directivas empieza por uno, mientras que en los parámetros de función empieza por cero.

Directiva	Función	Descripción
#property indicator_labelN	PlotIndexSetString(N-1, PLOT_LABEL, string)	Especifica una etiqueta de texto que se mostrará en *Data window* y en la información sobre herramientas.

Para las series gráficas que requieren varios búferes de indicadores (por ejemplo, DRAW_CANDLES, DRAW_FILLING, etc.), los nombres de las etiquetas se especifican con el separador ';'.

Las etiquetas también se muestran en la información sobre herramientas que aparece al pasar el ratón por encima de un gráfico.

En el ejemplo de *IndLabelHighLowClose.mq5*, añadimos dos directivas (a diferencia de *IndPropHighLowClose.mq5*).

```
#property indicator_label1  "High;Low"
#property indicator_label2  "Close"
```

Ahora es mucho más fácil entender los valores que aparecen al visualizar el indicador en la página *Data Window*.

5.4.10 Visualizar lagunas de datos (elementos vacíos)

En muchos casos, las lecturas de los indicadores deben mostrarse sólo en algunas barras, dejando el resto de las barras intactas (visualmente, sin líneas ni etiquetas adicionales). Por ejemplo, muchos indicadores de señales muestran flechas hacia arriba o hacia abajo en aquellas barras en las que aparece una recomendación de compra o de venta. Pero las señales son raras.

Un valor vacío que no se muestra ni en el gráfico ni en *Data Window* se establece mediante la función *PlotIndexSetDouble*.

bool PlotIndexSetDouble(int index, ENUM_PLOT_PROPERTY_DOUBLE property, double value)

La función establece las propiedades de *double* para el trazado en el *index* especificada. El conjunto de estas propiedades se resume en la enumeración ENUM_PLOT_PROPERTY_DOUBLE, pero de momento sólo tiene un elemento: PLOT_EMPTY_VALUE. También establece el valor vacío. El valor en sí se pasa en el último parámetro *value*.

Como ejemplo de indicador con valores poco frecuentes, consideraremos un detector fractal. Marca en el gráfico los precios altos (*High*) que son superiores a N barras vecinas y los precios bajos (*Low*) que son inferiores a N barras vecinas, en ambas direcciones. El archivo del indicador se llama *IndFractals.mq5*.

El indicador tendrá dos búferes y dos trazados del tipo DRAW_ARROW.

```
#property indicator_chart_window
#property indicator_buffers 2
#property indicator_plots   2

// rendering settings
#property indicator_type1   DRAW_ARROW
#property indicator_type2   DRAW_ARROW
#property indicator_color1  clrBlue
#property indicator_color2  clrRed
#property indicator_label1  "Fractal Up"
#property indicator_label2  "Fractal Down"

// indicator buffers
double UpBuffer[];
double DownBuffer[];
```

La variable de entrada *FractalOrder* le permitirá establecer el número de barras vecinas, mediante las cuales se determina el extremo superior o inferior.

```
input int FractalOrder = 3;
```

Para los símbolos de flecha, proporcionaremos una sangría de 10 píxeles desde los extremos para una mejor visibilidad.

```
const int ArrowShift = 10;
```

En la función *OnInit*, declare arrays como búferes y enlácelos a trazados gráficos.

```
int OnInit()
{
   // binding buffers
   SetIndexBuffer(0, UpBuffer, INDICATOR_DATA);
   SetIndexBuffer(1, DownBuffer, INDICATOR_DATA);

   // up and down arrow character codes
   PlotIndexSetInteger(0, PLOT_ARROW, 217);
   PlotIndexSetInteger(1, PLOT_ARROW, 218);

   // padding for arrows
   PlotIndexSetInteger(0, PLOT_ARROW_SHIFT, -ArrowShift);
   PlotIndexSetInteger(1, PLOT_ARROW_SHIFT, +ArrowShift);

   // setting an empty value (can be omitted, since EMPTY_VALUE is the default)
   PlotIndexSetDouble(0, PLOT_EMPTY_VALUE, EMPTY_VALUE);
   PlotIndexSetDouble(1, PLOT_EMPTY_VALUE, EMPTY_VALUE);

   return FractalOrder > 0 ? INIT_SUCCEEDED : INIT_PARAMETERS_INCORRECT;
}
```

Tenga en cuenta que el valor vacío por defecto es la constante especial EMPTY_VALUE, por lo que las llamadas a *PlotIndexSetDouble* anteriores son opcionales.

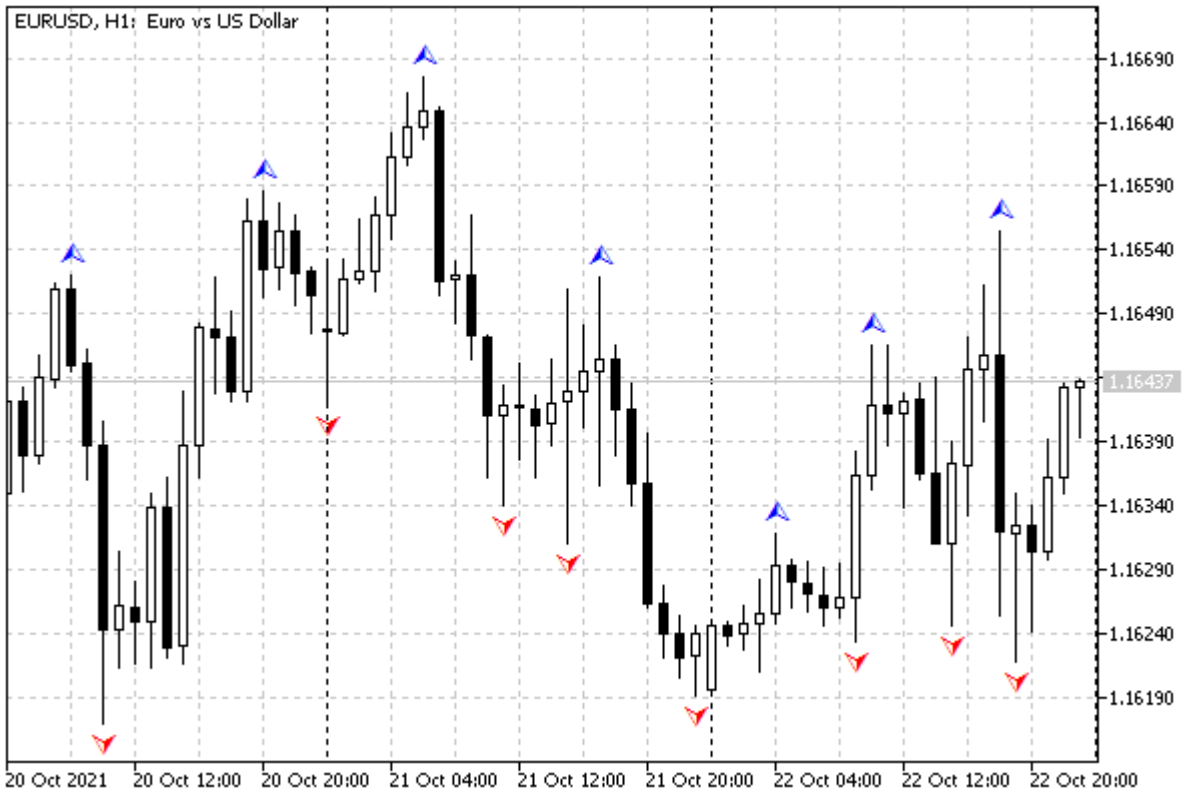
En el manejador *OnCalculate*, en el momento de la primera llamada, inicializamos ambos arrays con EMPTY_VALUE, y luego lo asignamos a nuevos elementos a medida que se forman las barras. El relleno es necesario porque la memoria asignada al búfer puede contener datos arbitrarios (basura).

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const datetime &time[],
                const double &open[],
                const double &high[],
                const double &low[],
                const double &close[],
                const long &tick_volume[],
                const long &volume[],
                const int &spread[])
{
   if(prev_calculated == 0)
   {
      // at the start, fill the arrays entirely
      ArrayInitialize(UpBuffer, EMPTY_VALUE);
      ArrayInitialize(DownBuffer, EMPTY_VALUE);
   }
   else
   {
      // on new bars we also clean the elements
      for(int i = prev_calculated; i < rates_total; ++i)
      {
         UpBuffer[i] = EMPTY_VALUE;
         DownBuffer[i] = EMPTY_VALUE;
      }
   }
   ...
```

En el bucle principal se comparan los precios *high* y *low* barra por barra con los mismos tipos de precios de las barras vecinas y se establecen marcas donde se encuentre un extremo entre las barras *FractalOrder* de cada lado.

```
   // view all or new bars that have bars in the FractalOrder environment
   for(int i = fmax(prev_calculated - FractalOrder - 1, FractalOrder);
       i < rates_total - FractalOrder; ++i)
   {
      // check if the upper price is higher than neighboring bars
      UpBuffer[i] = high[i];
      for(int j = 1; j <= FractalOrder; ++j)
      {
         if(high[i] <= high[i + j] || high[i] <= high[i - j])
         {
            UpBuffer[i] = EMPTY_VALUE;
            break;
         }
      }

      // check if the lower price is lower than neighboring bars
      DownBuffer[i] = low[i];
      for(int j = 1; j <= FractalOrder; ++j)
      {
         if(low[i] >= low[i + j] || low[i] >= low[i - j])
         {
            DownBuffer[i] = EMPTY_VALUE;
            break;
         }
      }
   }

   return rates_total;
}
```

Veamos cómo se ve este indicador en el gráfico.



Indicador fractal

Ahora cambiemos el tipo de dibujo de DRAW_ARROW a DRAW_ZIGZAG y comparemos el efecto de los valores vacíos para ambas opciones. El resultado debería ser un zigzag en fractales. La versión modificada del indicador se adjunta en el archivo *IndFractalsZigZag.mq5*.

Uno de los principales cambios se refiere al número de diagramas: ahora es uno, ya que DRAW_ZIGZAG «consume» ambos búferes.

```
#property indicator_chart_window
#property indicator_buffers 2
#property indicator_plots   1

// rendering settings
#property indicator_type1   DRAW_ZIGZAG
#property indicator_color1  clrMediumOrchid
#property indicator_width1  2
#property indicator_label1  "ZigZag Up;ZigZag Down"
...
```

Todas las llamadas a funciones relacionadas con el ajuste de las flechas se eliminan de *OnInit*.

```
int OnInit()
{
   SetIndexBuffer(0, UpBuffer, INDICATOR_DATA);
   SetIndexBuffer(1, DownBuffer, INDICATOR_DATA);

   PlotIndexSetDouble(0, PLOT_EMPTY_VALUE, EMPTY_VALUE);

   return FractalOrder > 0 ? INIT_SUCCEEDED : INIT_PARAMETERS_INCORRECT;
}
```

El resto del código fuente no se modifica.

En la siguiente imagen se muestra un gráfico en el que se aplica un zigzag además de los fractales: de este modo, se pueden comparar visualmente sus resultados. Ambos indicadores funcionan de forma totalmente independiente, pero debido al mismo algoritmo, los extremos encontrados son los mismos.

Indicador zigzag por fractales

Es importante tener en cuenta que, si se producen extremos del mismo tipo en una fila, el zigzag utiliza el primero de ellos. Esto es consecuencia del hecho de que los fractales se utilizan como extremos. Por supuesto, esto no puede ocurrir en un zigzag estándar. Si es necesario, quienes lo deseen pueden mejorar el algoritmo adelgazando primero las secuencias de fractales.

También debe tenerse en cuenta que para la representación de DRAW_ZIGZAG (así como de DRAW_SECTION), los segmentos visibles conectan elementos no vacíos y, por lo tanto, estrictamente hablando, se dibuja algún fragmento del segmento en cada barra, incluidos aquellos que tienen el valor EMPTY_VALUE (u otro designado en su lugar). Sin embargo, puede ver en *Data Window* que los elementos vacíos están realmente vacíos: no se muestra ningún valor para ellos.

5.4.11 Indicadores de subventanas independientes: tamaños y niveles

Hasta ahora nos hemos limitado a los indicadores que funcionan en la ventana principal del gráfico, es decir, que tienen la directiva *#property indicator_chart_window*. Ha llegado el momento de estudiar los indicadores colocados en una subventana separada debajo del gráfico de precios. Recuerde que deben declararse con la directiva *#property indicator_separate_window*.

Todo lo que aprendimos anteriormente se aplica a los indicadores en una subventana, incluyendo la descripción y el anclaje de los búferes, la configuración de los tipos y estilos de dibujo y el uso de las formas completa y abreviada de *OnCalculate*, a elegir. Sin embargo, también tienen algunas características y ajustes adicionales.

Dado que la subventana tiene su escala de valores, MQL5 permite establecer los valores máximo y mínimo para ella (los usuarios pueden establecer restricciones similares en el cuadro de diálogo de configuración del indicador, en la pestaña *Scale*). Esto se hace mediante programación utilizando la función *IndicatorSetDouble* con el siguiente prototipo.

```
bool IndicatorSetDouble(ENUM_CUSTOMIND_PROPERTY_DOUBLE property, double value)
bool IndicatorSetDouble(ENUM_CUSTOMIND_PROPERTY_DOUBLE property, int modifier,
 double value)
```

La función establece un valor de propiedad *double* para un indicador. Se necesitaron dos formularios porque algunas propiedades pueden ser múltiples; en concreto, los niveles horizontales (se abordarán un poco más adelante). Las propiedades disponibles se recopilan en la enum ENUM_CUSTOMIND_PROPERTY_DOUBLE.

Identificador	Descripción
INDICATOR_MINIMUM	Mínimo en el eje vertical
INDICATOR_MAXIMUM	Máximo en el eje vertical
INDICATOR_LEVELVALUE	Valor del nivel horizontal (el número se establece en el parámetro modificador)

En muchos indicadores oscilatorios, como WPR, se utiliza un rango de escala fijo. Veremos un ejemplo con él en el que se abarcan todas las funciones (propiedades) de esta sección.

La función devuelve *true* en caso de éxito y *false* en caso contrario.

Además de controlar la escala, los indicadores de la subventana pueden, como ya hemos entendido, tener niveles horizontales. Para establecer su número y atributos, se utiliza otra función, *IndicatorSetInteger*. El usuario puede realizar acciones similares en el cuadro de diálogo de configuración del indicador en la pestaña *Levels* tab/

```
bool IndicatorSetInteger(ENUM_CUSTOMIND_PROPERTY_INTEGER property, int value)
bool IndicatorSetInteger(ENUM_CUSTOMIND_PROPERTY_INTEGER property, int modifier,
 int value)
```

La función también tiene dos formas y permite establecer el valor de la propiedad de tipo para el indicador *int* o equivalente (por ejemplo, *color* o listado). Las propiedades disponibles se recopilan en la enumeración ENUM_CUSTOMIND_PROPERTY_INTEGER. Además de las propiedades asociadas a los

niveles, contiene la propiedad INDICATOR_DIGITS, que es común para los indicadores de cualquier tipo: la estudiaremos en la sección siguiente.

Identificador	Descripción
INDICATOR_DIGITS	Precisión de visualización de los valores de los indicadores (dígitos después del punto decimal)
INDICATOR_HEIGHT	Altura fija de la propia ventana del indicador en píxeles (comando del preprocesador *#property indicator_height*)
INDICATOR_LEVELS	Número de niveles horizontales en la ventana del indicador
INDICATOR_LEVELCOLOR	Color de la línea de nivel (tiene el tipo *color*, el parámetro *modifier* establece el número de nivel)
INDICATOR_LEVELSTYLE	Estilo de línea de nivel (tiene un tipo ENUM_LINE_STYLE, el parámetro *modifier* establece el número de nivel)
INDICATOR_LEVELWIDTH	Grosor de la línea de nivel (1-5) (el parámetro *modifier* establece el número de nivel)

Los niveles pueden tener etiquetas de texto. Para asignarlas, utilice la función *IndicatorSetString*.

```
bool IndicatorSetString(ENUM_CUSTOMIND_PROPERTY_STRING property, string value)
bool IndicatorSetString(ENUM_CUSTOMIND_PROPERTY_STRING property, int modifier,
 string value)
```

ENUM_CUSTOMIND_PROPERTY_STRING contiene la lista de parámetros del indicador de cadena. Preste atención a la propiedad INDICATOR_SHORTNAME que no está relacionada con los niveles: también es común a todos los indicadores y se abordará en la sección siguiente.

Identificador	Descripción
INDICATOR_SHORTNAME	Indicador título público
INDICATOR_LEVELTEXT	Descripción del nivel (el número se indica en el modificador)

Todas las funciones mencionadas para los tipos numéricos *int* y *double* están duplicadas por directivas especiales (a continuación se muestra una tabla resumen).

Directivas para propiedades de nivel	Funciones analógicas	Tipo de propiedad	Descripción
indicator_levelN	IndicatorSetDouble(INDICATOR_LEVELVALUE, N-1, valor)	double	Valor del número de nivel horizontal N en el eje vertical
indicator_levelcolor	IndicatorSetInteger(INDICATOR_LEVELCOLOR, N-1, color)	color	Color de los niveles horizontales (sólo se pueden establecer diferentes colores mediante números utilizando la función)
indicator_levelwidth	IndicatorSetInteger(INDICATOR_LEVELWIDTH, N-1, anchura)	int	Grosor de línea de los niveles horizontales en píxeles (sólo se pueden establecer diferentes grosores mediante números utilizando la función)
indicator_levelstyle	IndicatorSetInteger(INDICATOR_LEVELSTYLE, N-1, estilo)	ENUM __LINE__ _STYLE	Estilos de línea de los niveles horizontales (sólo se pueden establecer diferentes estilos por número utilizando la función)
indicator_minimum	IndicatorSetDouble(INDICATOR_MINIMUM, mínimo)	double	Valor mínimo fijo, límite inferior de la escala en el eje vertical
indicator_maximum	IndicatorSetDouble(INDICATOR_MAXIMUM, máximo)	double	Valor máximo fijo, límite superior de la escala en el eje vertical

Tenga en cuenta que la numeración de las instancias de propiedades (modificadores) cuando se utilizan directivas *#property* empieza por 1 (uno), mientras que las funciones utilizan una numeración a partir de 0 (cero).

Un lector atento observará que no hay directivas para algunas propiedades. Entre ellas se incluyen INDICATOR_LEVELTEXT, INDICATOR_SHORTNAME, INDICATOR_DIGITS. Se da por sentado que estas propiedades deben rellenarse dinámicamente desde el código MQL, dependiendo de las variables de entrada y del gráfico en el que se coloque el indicador. INDICATOR_LEVELS se establece indirectamente especificando varias directivas para los niveles.

Por último, la característica distintiva de los indicadores en una subventana es que un programa puede «congelar» el tamaño vertical de su ventana.

Directiva para tamaño de la subventana	Función analógica	Descripción
indicator_height	IndicatorSetInteger(INDICATOR_HEIGHT, altura)	Altura fija de la subventana del indicador en píxeles (el usuario no podrá cambiar la altura).

Una altura de subventana fija se utiliza normalmente sólo para paneles de control con controles (botones, banderas, campos de entrada) implementados utilizando objetos gráficos.

Para las funciones de configuración de propiedades, desgraciadamente, no hay inversas (*IndicatorGetInteger*, *IndicatorGetDouble*, *IndicatorGetString*). Entre otras cosas, esto no permite, por ejemplo, buscar el número y los valores de los niveles horizontales si han sido modificados por el usuario.

Como ejemplo de trabajo con una escala y niveles fijos, considere el indicador *IndWPR.mq5*. En él, utilizaremos el algoritmo WPR estándar: en un número determinado de barras pasadas (periodo WPR), encontraremos los máximos H y los mínimos L del precio (es decir, su rango). A continuación, calculamos la relación entre la diferencia entre el precio actual C y el mínimo L, C - L (o la diferencia - (H - C), con signo negativo) y todo el rango, y lo llevamos todo al rango de 0 a -100. He aquí la fórmula canónica para calcular WPR:

```
R% = (-(H - C) / (H - L)) * 100
```

Vamos a añadir algunas directivas al principio del código fuente. Además de la propiedad de ubicación del indicador en su propia ventana, vamos a establecer la escala de valores de 0 a -100.

```
#property indicator_separate_window
#property indicator_maximum    0.0
#property indicator_minimum    -100.0
```

Un búfer y un gráfico de líneas son suficientes para almacenar los valores y mostrar el indicador.

```
#property indicator_buffers    1
#property indicator_plots      1
#property indicator_type1      DRAW_LINE
#property indicator_color1     clrDodgerBlue
```

En el indicador WPR es habitual distinguir dos niveles: -20 y -80, como límites de las zonas de sobrecompra y sobreventa, respectivamente. Vamos a crear un par de líneas horizontales para ellos.

```
#property indicator_level1     -20.0
#property indicator_level2     -80.0
#property indicator_levelstyle STYLE_DOT
#property indicator_levelcolor clrSilver
#property indicator_levelwidth 1
```

La única variable de entrada permite establecer el periodo de cálculo para WPR.

```
input int WPRPeriod = 14; // Period
```

El array para el búfer se declara a nivel global y se registra con *OnInit*.

```
double WPRBuffer[];

void OnInit()
{
   // check for correct input
   if(WPRPeriod < 1)
   {
      Alert(StringFormat("Incorrect Period value (%d). Should be 1 or larger",
         WPRPeriod));
   }

   // binding array as buffer
   SetIndexBuffer(0, WPRBuffer);
}
```

El manejador *OnInit* se describe con el tipo *void*, que conlleva de forma implícita una inicialización exitosa. No obstante, si el periodo es inferior a 1, no permitirá realizar el cálculo y se mostrará un aviso al usuario.

Para simplificar el encabezado de la función *OnCalculate* se ha preparado el archivo de encabezado *IndCommon.mqh* para indicadores, con dos macros que describen las listas de parámetros estándar de ambas formas del manejador de eventos.

```
#define ON_CALCULATE_STD_FULL_PARAM_LIST \
const int rates_total,     \
const int prev_calculated, \
const datetime &time[],    \
const double &open[],      \
const double &high[],      \
const double &low[],       \
const double &close[],     \
const long &tick_volume[], \
const long &volume[],      \
const int &spread[]

#define ON_CALCULATE_STD_SHORT_PARAM_LIST \
const int rates_total,     \
const int prev_calculated, \
const int begin,           \
const double &data[]
```

Ahora podemos utilizar la definición concisa de *OnCalculate* en este y otros indicadores (siempre que estemos satisfechos con los nombres de los parámetros propuestos en las macros).

```
#include <MQL5Book/IndCommon.mqh>

int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   if(rates_total < WPRPeriod || WPRPeriod < 1) return 0;
   ...
   return rates_total;
}
```

Al principio de *OnCalculate* comprobamos si es posible calcular utilizando los valores actuales de *WPRPeriod* y *rates_total*. Si no hay datos suficientes o el periodo es demasiado corto, devuelve 0, lo que dejará vacía la ventana del indicador.

A continuación, rellenamos con un valor vacío las primeras barras para las que es imposible calcular el WPR de un periodo determinado.

```
int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   ...
   if(prev_calculated == 0)
   {
      ArrayFill(WPRBuffer, 0, WPRPeriod - 1, EMPTY_VALUE);
   }
   ...
}
```

Por último, realizamos los cálculos de WPR y almacenamos los resultados. Observe que la última barra se actualiza en cada tick: esto se consigue iniciando el bucle con *prev_calculated - 1*.

```
int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   ...
   for(int i = fmax(prev_calculated - 1, WPRPeriod - 1);
      i < rates_total && !IsStopped(); i++)
   {
      double max_high = high[fmax(ArrayMaximum(high, i - WPRPeriod + 1, WPRPeriod), 0
      double min_low = low[fmax(ArrayMinimum(low, i - WPRPeriod + 1, WPRPeriod), 0)];
      if(max_high != min_low)
      {
         WPRBuffer[i] = -(max_high - close[i]) * 100 / (max_high - min_low);
      }
      else
      {
         WPRBuffer[i] = WPRBuffer[i - 1];
      }
   }
   return rates_total;
}
```

ArrayMaximum y *ArrayMinimum* permiten buscar los índices del *high* más alto y del *low* más bajo.

El indicador aparece en una ventana independiente de la siguiente manera.

Indicador WPR

En las siguientes secciones seguiremos mejorando este indicador, añadiendo gradualmente otras propiedades de uso común.

5.4.12 Propiedades generales de los indicadores: título y precisión de los valores

Para todos los indicadores se admiten un par de propiedades importantes que no están relacionadas con los cálculos, pero que mejoran la experiencia del usuario. Su correcta configuración en el manejador *OnInit* pasó a formar parte del estándar de desarrollo de indicadores.

La propiedad de tipo entero INDICATOR_DIGITS se establece mediante la función *IndicatorSetInteger* comentada anteriormente y afecta a la precisión de la representación de los números reales en el gráfico y en *Data Window*. Por defecto, el terminal emite 6 dígitos después del punto decimal. Si las lecturas del indicador están relacionadas con el precio del instrumento actual, entonces tiene sentido establecer esta propiedad igual a la precisión de la representación del precio: *IndicatorSetInteger(INDICATOR_DIGITS,_Digits)*.

En el caso de WPR, los valores son análogos a porcentajes, por lo que tiene sentido limitar los valores mostrados a dos decimales.

```
IndicatorSetInteger(INDICATOR_DIGITS, 2);
```

La segunda propiedad de uso común es la cadena INDICATOR_SHORTNAME: utiliza la función *IndicatorSetString*. Este es el título del indicador que aparece en la información sobre herramientas y también en la esquina superior izquierda de la subventana si el indicador tiene su propia ventana. Si no se especifica explícitamente, se utiliza el nombre del archivo indicador. En concreto, en la captura de pantalla de la sección anterior, vemos el título IndWPR.

Es habitual mostrar las principales variables de entrada y los modos de funcionamiento (si hay varios) en el encabezado del indicador.

Por ejemplo, para WPR, por regla general, el periodo seleccionado por el usuario se incluye en el título.

Además, el título permite acortar el nombre. Esto es importante porque el título está limitado a 63 caracteres.

Para la versión actualizada de WPR, utilizaremos la siguiente configuración:

```
IndicatorSetString(INDICATOR_SHORTNAME, "%R" + "(" + (string)WPRPeriod + ")");
```

Comprobaremos los resultados de estas mejoras en la siguiente sección, después de asignar diferentes colores a las zonas de sobrecompra y sobreventa (véase el ejemplo *IndColorWPR.mq5*).

5.4.13 Coloreado de gráficos por elementos

Además de los tipos de dibujo estándar enumerados anteriormente en ENUM_DRAW_TYPE, la plataforma proporciona a sus variantes la posibilidad de colorear individualmente los valores de cada barra. Para ello se utiliza un búfer de indicador adicional en el que se almacenan los números de color. Los números hacen referencia a elementos de un array especial que contiene un conjunto de colores definidos por el programador. El número máximo de colores es 64.

En la siguiente tabla se enumeran los elementos ENUM_DRAW_TYPE con soporte de color y el número de búferes necesarios para dibujarlos, incluyendo 1 búfer con índices de color.

Tipo de visualización	Descripción	Número de búferes
DRAW_COLOR_LINE	Línea multicolor	1+1
DRAW_COLOR_SECTION	Segmentos multicolores	1+1
DRAW_COLOR_ARROW	Flechas multicolores	1+1
DRAW_COLOR_HISTOGRAM	Histograma multicolor a partir de la línea cero	1+1
DRAW_COLOR_HISTOGRAM2	Histograma multicolor entre valores emparejados de dos búferes de indicadores	2+1
DRAW_COLOR_ZIGZAG	Zigzag multicolor	2+1
DRAW_COLOR_BARS	Barras multicolores	4+1
DRAW_COLOR_CANDLES	Velas multicolores	4+1

Al vincular los búferes a los gráficos, tenga en cuenta que debe especificarse un búfer de color adicional en el primer parámetro *SetIndexBuffer* bajo el número que va inmediatamente después de los búferes de datos. Por ejemplo, para colorear una línea utilizando un búfer de datos y un búfer de color, los datos reciben el número 0 y sus colores, el número 1:

```
double ColorLineData[];
double ColorLineColors[];

void OnInit()
{
   SetIndexBuffer(0, ColorLineData, INDICATOR_DATA);
   SetIndexBuffer(1, ColorLineColors, INDICATOR_COLOR_INDEX);
   PlotIndexSetInteger(0, PLOT_DRAW_TYPE, DRAW_COLOR_LINE);
   ...
}
```

El conjunto inicial de colores de la paleta para el diagrama N puede especificarse mediante la directiva *#property indicator_colorN*, que especifica los colores necesarios separados por comas como constantes con nombre o literales de color. Por ejemplo, la siguiente entrada en el indicador seleccionará 6 colores estándar para colorear el gráfico 0º (la numeración empieza por 1 en las directivas):

```
#property indicator_color1   clrRed,clrBlue,clrGreen,clrYellow,clrMagenta,clrCyan
```

Más adelante en el programa, debe especificar no el color en sí, que mostrará la construcción gráfica, sino sólo su índice. La numeración en la paleta se realiza como en un array normal, empezando por 0. Por lo tanto, si necesita establecer un color verde para la i-ésima barra, entonces es suficiente con establecer el índice del color verde de la paleta en el búfer de color, es decir, 2 en este caso.

```
ColorLineColors[i]=2;// reference to element with color clrGreen
```

El conjunto de colores para colorear no se establece de una vez por todas, sino que puede cambiarse dinámicamente mediante la función *PlotIndexSetInteger(index, PLOT_LINE_COLOR, color)*.

Por ejemplo, para sustituir el color *clrGreen* de la paleta anterior por *clrGray*, utilice la siguiente llamada:

```
   PlotIndexSetInteger(0, PLOT_LINE_COLOR, clrGray);
```

Vamos a aplicar la coloración en nuestro indicador WPR. El nuevo archivo es *IndColorWPR.mq5*. Los cambios afectan a los siguientes ámbitos.

El número de búferes se ha incrementado en 1. Tres colores en lugar de uno.

```
#property indicator_buffers    2
#property indicator_plots      1
#property indicator_type1      DRAW_COLOR_LINE
#property indicator_color1     clrDodgerBlue,clrGreen,clrRed
```

Se ha añadido un nuevo array bajo el búfer de color y su registro en *OnInit*.

```
double WPRColors[];

void OnInit()
{
   ...
   SetIndexBuffer(1, WPRColors, INDICATOR_COLOR_INDEX);
   ...
```

Si no establece el tipo de búfer INDICATOR_COLOR_INDEX (es decir, con una llamada *SetIndexBuffer(1, WPRColors)* se trataría por defecto como INDICATOR_DATA), se hará visible en la página *Data Window*.

En la función *OnCalculate* dentro del ciclo de trabajo, vamos a añadir coloración basada en el análisis del valor de la i-ésima barra. Por defecto, utilizamos el color con índice 0, es decir, el antiguo *clrDodgerBlue*. Si las lecturas del indicador se mueven hacia la zona superior, se resaltan en color 2 (*clrRed*), y si entran en la zona inferior, se colorean en 1 (*clrGreen*).

```
int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   ...
   for(int i = fmax(prev_calculated - 1, WPRPeriod - 1);
      i < rates_total && !IsStopped(); i++)
   {
      ...
      WPRColors[i] = 0;
      if(WPRBuffer[i] > -20) WPRColors[i] = 2;
      else if(WPRBuffer[i] < -80) WPRColors[i] = 1;
   }
   return rates_total;
}
```

Este es su aspecto en la pantalla:

Indicador WPR con zonas de sobrecompra y sobreventa coloreadas

Tenga en cuenta que el fragmento de línea se pinta en un color alternativo si su punto final (barra) se encuentra en la zona superior o inferior. En este caso, la lectura anterior puede estar dentro de la zona central, lo que puede dar la impresión de que el color es incorrecto. Sin embargo, se trata de un comportamiento correcto, coherente con la implementación actual y con el modo en que la plataforma utiliza el color.

El color del segmento del gráfico de líneas DRAW_COLOR_LINE entre dos barras adyacentes viene determinado por el color de la barra de la derecha (más reciente).

Si desea resaltar con color sólo los fragmentos en los que ambas barras adyacentes se encuentran en la misma zona, modifique el código como sigue:

```
WPRColors[i] = 0;
if(WPRBuffer[i] > -20 && WPRBuffer[i - 1] > -20) WPRColors[i] = 2;
else if(WPRBuffer[i] < -80 && WPRBuffer[i - 1] < -80) WPRColors[i] = 1;
```

Además, recuerde que hemos añadido al código fuente la configuración del título y la precisión de la representación de los valores (2 caracteres). Comparar la nueva imagen con la antigua le permitirá darse cuenta de estas diferencias visuales. En concreto, el título tiene ahora el aspecto de «%R(14)», y la escala vertical de valores es mucho más compacta.

El último aspecto que cambiaremos en el indicador *IndColorWPR.mq5* es que omitimos el dibujo en las barras iniciales.

5.4.14 Omitir dibujo en barras iniciales

En muchos casos, según las condiciones del algoritmo, el cálculo de los valores del indicador no puede iniciarse desde la primera barra (la más a la izquierda disponible), ya que se requiere garantizar el

número mínimo especificado de barras anteriores en el historial. Por ejemplo, muchos tipos de suavizado implican que el valor actual se calcula utilizando un array de precios para las N barras anteriores.

En tales casos, puede que no sea posible calcular los valores de los indicadores en las primeras barras, o que estos valores no estén destinados a mostrarse en el gráfico y sólo sean auxiliares para calcular los valores posteriores.

Para desactivar la visualización del indicador en las primeras N-1 barras del historial, establezca la propiedad PLOT_DRAW_BEGIN en N para el índice del trazado correspondiente: *PlotIndexSetInteger(index, PLOT_DRAW_BEGIN, N)*. Por defecto, esta propiedad es 0, lo que significa que los datos se muestran desde el principio.

Podemos desactivar la visualización de líneas en las barras necesarias estableciéndolas en un valor vacío (EMPTY_VALUE por defecto). No obstante, la llamada a la función *PlotIndexSetInteger* hace algo más con la propiedad PLOT_DRAW_BEGIN. De este modo, indicamos a los programas externos el número de primeros valores insignificantes de nuestro búfer de indicadores. En concreto, otros indicadores que potencialmente puedan construirse basándose en las series temporales de nuestro indicador recibirán el valor de la propiedad PLOT_DRAW_BEGIN en el parámetro *begin* de su manejador *OnCalculate*. Así, tendrán la oportunidad de saltarse barras.

En el ejemplo del indicador *IndColorWPR.mq5*, añadamos una configuración similar a la función *OnInit*.

```
input int WPRPeriod = 14; // Period

void OnInit()
{
   ...
   PlotIndexSetInteger(0, PLOT_DRAW_BEGIN, WPRPeriod - 1);
   ...
}
```

Ahora, en la función *OnCalculate*, sería posible eliminar el borrado forzado de las primeras barras, ya que siempre estarán ocultas.

```
   if(prev_calculated == 0)
   {
      ArrayFill(WPRBuffer, 0, WPRPeriod - 1, EMPTY_VALUE);
   }
```

Pero esto funcionará correctamente sólo cuando el usuario haya seleccionado manualmente nuestro indicador como fuente de series temporales para otro indicador. Si algún programador decide utilizar nuestro indicador en sus desarrollos, entonces existe un mecanismo diferente para obtener los datos (hablaremos de ello en el próximo capítulo), y no le permitirá averiguar la propiedad PLOT_DRAW_BEGIN. Por lo tanto, es mejor utilizar una inicialización explícita del búfer.

Para demostrar cómo se puede utilizar esta propiedad en otro indicador calculado utilizando los datos de nuestro indicador vamos a preparar otro indicador: se trata del conocido algoritmo de la Media Móvil Exponencial Triple integrado en el indicador *IndTripleEMA.mq5*. Cuando esté listo, será fácil aplicarlo tanto a series temporales de precios como a indicadores arbitrarios, como el indicador *IndColorWPR.mq5* anterior.

Además, nos familiarizaremos con la posibilidad técnica de describir búferes auxiliares para cálculos (INDICATOR_CALCULATIONS).

La triple fórmula EMA consta de varios pasos computacionales. El suavizado exponencial simple del período P para la serie temporal inicial T se expresa del siguiente modo:

```
K = 2.0 / (P + 1)
A[i] = T[i] * K + A[i - 1] * (1 - K)
```

donde K es el factor de ponderación para tener en cuenta los elementos de la serie original, calculado después de un período determinado P; (1 - K) es el coeficiente de inercia aplicado a los elementos de la serie suavizada A. Para obtener el elemento i-ésimo de la serie A, sumamos la parte K-ésima del elemento i-ésimo de la serie original T[i] y la parte (1 - K)-ésima del elemento anterior A[i - 1].

Si denotamos el suavizado según las fórmulas indicadas como el operador E, entonces la EMA triple incluye, como su nombre indica, la aplicación de E tres veces, tras lo cual las tres filas suavizadas resultantes se combinan de un modo especial.

```
EMA1 = E(A, P), for all i
EMA2 = E(EMA1, P), for all i
EMA3 = E(EMA2, P), for all i
TEMA = 3 * EMA1 - 3 * EMA2 + EMA3, for all i
```

La EMA triple proporciona un retraso menor respecto a la serie original en comparación con la EMA normal del mismo periodo. Sin embargo, se caracteriza por una mayor capacidad de respuesta, lo que puede provocar irregularidades en la línea resultante y dar señales falsas.

> El suavizado EMA permite obtener una estimación aproximada de la media, ya a partir del segundo elemento de la serie, y para ello no es necesario cambiar el algoritmo. Esto distingue a la EMA de otros métodos de suavizado que requieren P elementos previos o un algoritmo modificado para las muestras iniciales si se dispone de menos de P elementos. Algunos desarrolladores prefieren invalidar los primeros elementos P-1 de una fila suavizada incluso cuando se utiliza la EMA. Sin embargo, debe tenerse en cuenta que la influencia de los elementos anteriores de la serie en la fórmula EMA no se limita a P elementos, y se vuelve despreciable sólo cuando el número de elementos tiende a infinito (en otros algoritmos MA bien conocidos, exactamente P elementos anteriores tienen una influencia).
>
> A los efectos de este libro, para investigar el impacto de omitir los datos iniciales, no desactivaremos la salida de los valores iniciales de la EMA.

Para calcular los tres niveles de EMA necesitamos búferes auxiliares y uno más para la serie final: se mostrará como un gráfico de líneas.

```
#property indicator_chart_window
#property indicator_buffers 4
#property indicator_plots   1

#property indicator_type1   DRAW_LINE
#property indicator_color1  Orange
#property indicator_width1  1
#property indicator_label1  "EMA³"

double TemaBuffer[];
double Ema[];
double EmaOfEma[];
double EmaOfEmaOfEma[];

void OnInit()
{
   ...
   SetIndexBuffer(0, TemaBuffer, INDICATOR_DATA);
   SetIndexBuffer(1, Ema, INDICATOR_CALCULATIONS);
   SetIndexBuffer(2, EmaOfEma, INDICATOR_CALCULATIONS);
   SetIndexBuffer(3, EmaOfEmaOfEma, INDICATOR_CALCULATIONS);
   ...
}
```

Una variable de entrada *InpPeriodEMA* permite establecer el periodo de suavizado. La segunda variable, *InpHandleBegin*, es un conmutador de modo con el que podemos explorar cómo reacciona el indicador al tener en cuenta, o ignorar, el parámetro *begin* en el manejador *OnCalculate*. Los modos disponibles se resumen en la enumeración BEGIN_POLICY y significan lo siguiente (en el orden en que están dispuestos):

- turno estricto según *begin*
- validación personalizada de los datos iniciales, sin tener en cuenta *begin*
- sin manipulación, es decir, ignorando *begin* y calculando directamente todos los datos

```
enum BEGIN_POLICY
{
   STRICT, // strict
   CUSTOM, // custom
   NONE,   // no
};

input int InpPeriodEMA = 14;                // EMA period:
input BEGIN_POLICY InpHandleBegin = STRICT; // Handle 'begin' parameter:
```

El segundo modo CUSTOM (personalizado) se basa en una comparación preliminar de cada elemento fuente con EMPTY_VALUE y su sustitución por un valor adecuado para el algoritmo. Esto funcionará correctamente sólo con aquellos indicadores que inicialicen honestamente el comienzo no utilizado de los búferes sin dejar basura allí. Nuestro indicador *IndColorWPR* rellena el búfer según sea necesario, por lo que puede esperar resultados casi idénticos con los modos STRICT (estricto) y CUSTOM.

Se prepara la constante K para calcular la EMA a partir de *InpPeriodEMA*.

```
const double K = 2.0 / (InpPeriodEMA + 1);
```

La función EMA en sí es bastante sencilla (aquí se omite el fragmento de protección para la variante CUSTOM con comprobaciones EMPTY_VALUE).

```
void EMA(const double &source[], double &result[], const int pos, const int begin = 0
{
   ...
   if(pos <= begin)
   {
      result[pos] = source[pos];
   }
   else
   {
      result[pos] = source[pos] * K + result[pos - 1] * (1 - K);
   }
}
```

Y aquí está el cálculo completo del triple suavizado en *OnCalculate*.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
   const int _begin = InpHandleBegin == STRICT ? begin : 0;
   // fresh start, or history update
   if(prev_calculated == 0)
   {
      Print("begin=", begin, " ", EnumToString(InpHandleBegin));

      // we can change chart settings dynamically
      PlotIndexSetInteger(0, PLOT_DRAW_BEGIN, _begin);

      // preparing arrays
      ArrayInitialize(Ema, EMPTY_VALUE);
      ArrayInitialize(EmaOfEma, EMPTY_VALUE);
      ArrayInitialize(EmaOfEmaOfEma, EMPTY_VALUE);
      ArrayInitialize(TemaBuffer, EMPTY_VALUE);
      Ema[_begin] = EmaOfEma[_begin] = EmaOfEmaOfEma[_begin] = price[_begin];
   }

   // main loop, taking into account the start from _begin
   for(int i = fmax(prev_calculated - 1, _begin);
      i < rates_total && !IsStopped(); i++)
   {
      EMA(price, Ema, i, _begin);
      EMA(Ema, EmaOfEma, i, _begin);
      EMA(EmaOfEma, EmaOfEmaOfEma, i, _begin);

      if(InpHandleBegin == CUSTOM) // protection from empty elements at the beginning
      {
         if(Ema[i] == EMPTY_VALUE
         || EmaOfEma[i] == EMPTY_VALUE
         || EmaOfEmaOfEma[i] == EMPTY_VALUE)
            continue;
      }

      TemaBuffer[i] = 3 * Ema[i] - 3 * EmaOfEma[i] + EmaOfEmaOfEma[i];
   }
   return rates_total;
}
```

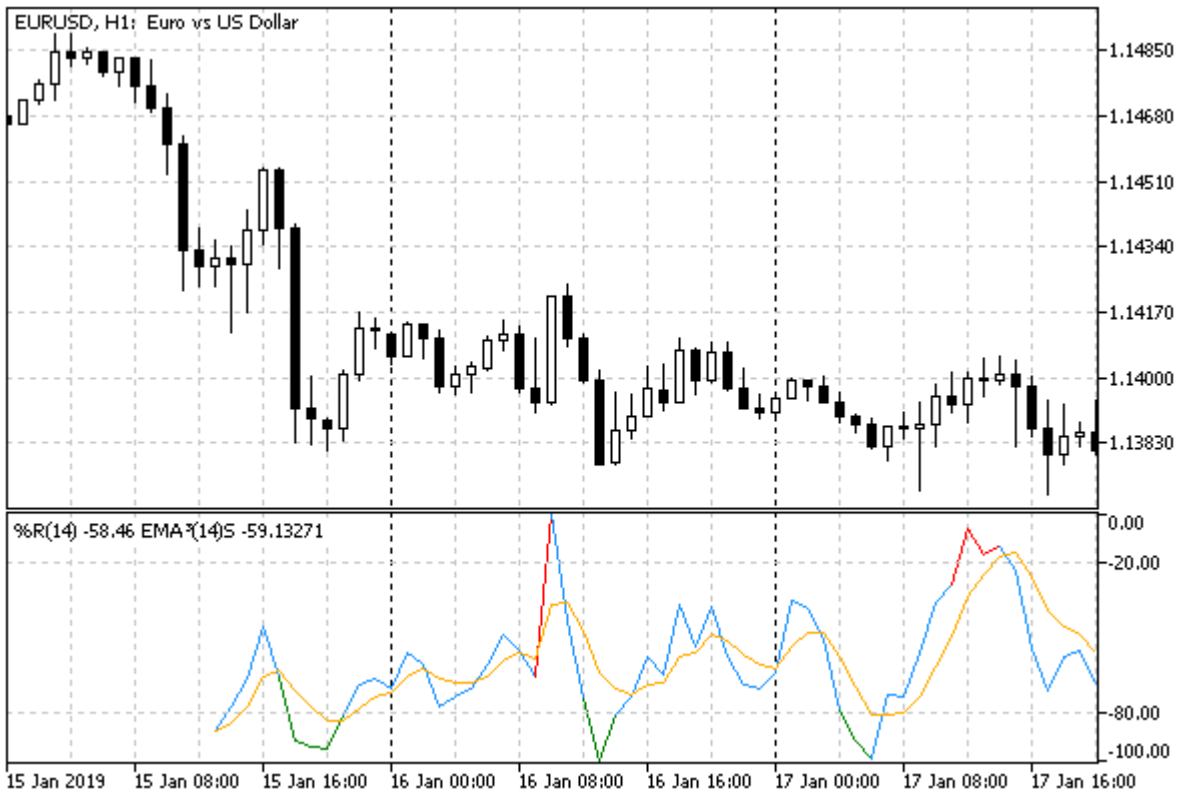
Durante el primer arranque o cuando se actualiza el historial, el valor recibido del parámetro *begin* junto con el modo de procesamiento seleccionado por el usuario se escriben en el registro.

Tras la correcta compilación, todo está listo para los experimentos.

En primer lugar, vamos a ejecutar el indicador *IndColorWPR* (por defecto, su período es 14, lo que significa, de acuerdo con los códigos fuente, establecer la propiedad PLOT_DRAW_BEGIN a 1 menos, ya que la indexación comienza a partir de 0 y la 13ª barra será la primera para la que aparecerá un valor). A continuación, arrastre el indicador *IndTripleEMA* a la subventana que muestra WPR. En el

cuadro de diálogo de configuración de propiedades que se abre, en la pestaña *Options*, seleccione *Previous indicator data* en la lista desplegable *Apply to*. Deje los valores por defecto en la pestaña *Inputs*.

En la siguiente imagen se muestra el inicio del gráfico. El registro tendrá la siguiente entrada: *begin=13 STRICT*.



Indicador EMA triple aplicado al WPR dado el inicio de los datos

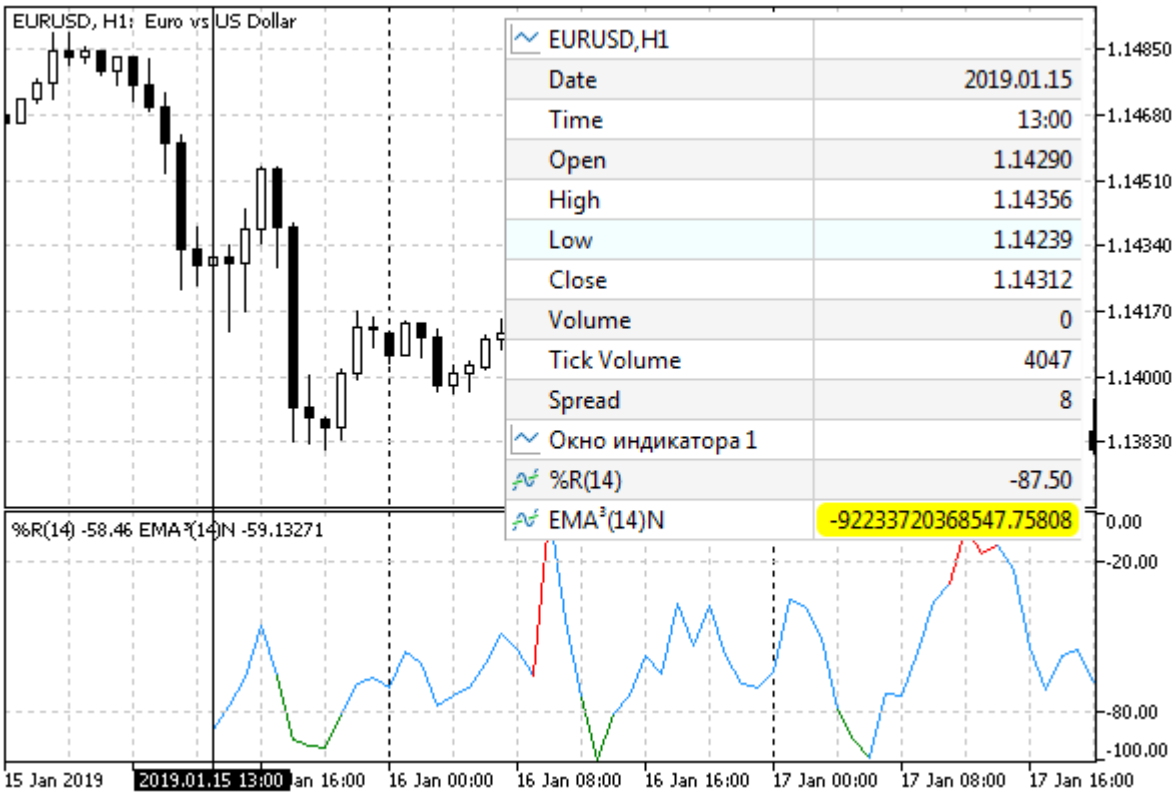
Tenga en cuenta que la línea promediada comienza a una distancia del inicio, así como WPR.

> ¡Atención! El número de barras disponibles para el cálculo del indicador *rates_total* (o *iBars(_Symbol,_Period)*) puede superar el número máximo de barras permitido en el gráfico desde la configuración del terminal si existe un historial local de cotizaciones más largo. En este caso, los elementos vacíos al principio de la línea WPR (o cualquier otro indicador que se salte los primeros elementos, como MA) se volverán invisibles: se ocultarán tras el borde izquierdo del gráfico. Para reproducir la situación de ausencia de líneas en las barras iniciales, deberá aumentar el número de barras del gráfico o cerrar el terminal y borrar el historial local de un símbolo determinado.

Ahora vamos a cambiar al modo CUSTOM en los ajustes del indicador *IndTripleEMA* (el registro mostrará *begin=0 CUSTOM*). No debería haber ningún cambio importante en las lecturas de los indicadores.

Por último, activamos el modo NONE (ninguno). El registro mostrará *begin=0 NONE*.

En este caso, la situación en el gráfico parecerá extraña, ya que la línea de hecho desaparecerá. En la ventana de datos puede ver que los valores de los elementos son muy grandes.



Indicador EMA triple aplicado a WPR sin inicio de datos

Esto se debe a que los valores de EMPTY_VALUE son iguales al número real máximo DBL_MAX. Por lo tanto, sin tener en cuenta el parámetro *begin*, los cálculos con tales valores también generan números muy grandes. Dependiendo de las particularidades del cálculo, el desbordamiento puede hacer que recibamos un NaN especial (Not A Number, véase Comprobación de la normalidad de los números reales). Uno de ellos, *-nan(ind)*, está resaltado en la imagen (*Data Window* ya sabe cómo mostrar algunos tipos de NaN, como por ejemplo, «inf» e «-inf», pero esto todavía no se aplica a «-nan(ind)»). Como sabemos, estos valores NaN son peligrosos, ya que los cálculos que los incluyan también seguirán dando NaN. Si no se genera ningún NaN, entonces, a medida que se desplace hacia la derecha a través de las barras, el «proceso transitorio» en el cálculo de números grandes se desvanece (debido al factor de reducción (1 - K) en la fórmula EMA), y el resultado se estabiliza, volviéndose razonable. Si se desplaza por el gráfico hasta el momento actual, verá una EMA triple normal.

Tener en cuenta el parámetro *begin* es una buena práctica, pero no garantiza que el proveedor de datos (si se trata de un indicador de terceros) haya rellenado correctamente esta propiedad. Por lo tanto, es deseable proporcionar cierta protección en su código. En esta implementación de *IndTripleEMA*, se implementa en el nivel inicial.

Si ejecutamos el indicador *IndTripleEMA* en el gráfico de precios, siempre recibirá *begin = 0*, ya que las series temporales de precios se rellenan con datos reales desde el principio, incluso en las barras más antiguas.

5.4.15 Esperar datos y gestionar la visibilidad (DRAW_NONE)

En el capítulo anterior, en la sección Trabajar con arrays de ticks reales en estructuras *MqlTick*, trabajamos con el script *SeriesTicksDeltaVolume.mq5*, que calcula el volumen delta en cada barra. En ese momento mostrábamos los resultados en un registro, pero una forma mucho más cómoda y lógica

de analizar esa información técnica es un indicador. En esta sección crearemos un indicador de este tipo: *IndDeltaVolume.mq5*.

Aquí tendremos que abordar dos factores con los que nos encontramos a menudo a la hora de desarrollar indicadores, pero que no se han tratado en los ejemplos anteriores.

El primero de ellos es que los datos de tick no se refieren a series temporales de precios estándar, que el terminal envía al indicador en los parámetros *OnCalculate*. Esto significa que el propio indicador debe solicitarlos y esperar antes de que sea posible mostrar algo en la ventana.

El segundo factor está relacionado con el hecho de que los volúmenes de compra y venta, por regla general, son mucho mayores que su delta, y cuando se muestran en una ventana, será difícil distinguir entre estos últimos. No obstante, es el delta un valor indicativo, que suele analizarse junto con el movimiento del precio. Por ejemplo, hay 4 combinaciones más obvias de configuraciones de barra y volumen delta:

- Barra alcista y delta positivo = confirmación de una tendencia alcista
- Barra bajista y delta negativo = confirmación de la tendencia bajista
- Barra alcista y delta negativo = posible inversión de tendencia a la baja
- Barra bajista y delta positivo = es posible una inversión de tendencia al alza

Para ver el histograma de deltas necesitamos proporcionar un modo para desactivar los histogramas «grandes» (compras y ventas), para lo cual utilizaremos el tipo DRAW_NONE. Desactiva el dibujo de un trazado específico e impide su influencia en la escala de ventana seleccionada automáticamente (pero deja el búfer en *Data Window*). Así, al eliminar los trazados grandes de la consideración, conseguiremos una autoescala mayor para el diagrama delta restante. Otra forma de ocultar los búferes marcándolos como auxiliares (modo INDICATOR_CALCULATIONS) se abordará en la siguiente sección.

La idea del delta de volumen es calcular por separado los volúmenes de compra y venta en ticks, después de lo cual podemos encontrar la diferencia entre estos volúmenes. En consecuencia, obtenemos tres series temporales con volúmenes de compra, volúmenes de venta y las diferencias entre ellos. Como esta información no cabe en la escala de precios, el indicador debe mostrarse en su propia ventana, y elegiremos histogramas desde cero (DRAW_HISTOGRAM) como forma de mostrar tres series temporales.

De acuerdo con esto, vamos a describir las propiedades de los indicadores en las directivas: ubicación, número de búferes y trazados, así como sus tipos.

```
#property indicator_separate_window
#property indicator_buffers 3
#property indicator_plots   3
#property indicator_type1   DRAW_HISTOGRAM
#property indicator_color1  clrBlue
#property indicator_width1  1
#property indicator_label1  "Buy"
#property indicator_type2   DRAW_HISTOGRAM
#property indicator_color2  clrRed
#property indicator_width2  1
#property indicator_label2  "Sell"
#property indicator_type3   DRAW_HISTOGRAM
#property indicator_color3  clrMagenta
#property indicator_width3  3
#property indicator_label3  "Delta"
```

Vamos a utilizar las variables de entrada del script anterior. Dado que los ticks representan datos bastante cuantiosos, limitaremos el número de barras para el cálculo en la historia (*BarCount*). Además, dependiendo de la presencia o ausencia de volúmenes reales en ticks de un determinado instrumento financiero, podemos calcular el delta de dos formas diferentes, para lo cual utilizaremos el parámetro *tick type* (la enumeración COPY_TICKS está definida en el archivo de encabezado *TickEnum.mqh*, que ya hemos utilizado en el script).

```
#include <MQL5Book/TickEnum.mqh>

input int BarCount = 100;
input COPY_TICKS TickType = INFO_TICKS;
input bool ShowBuySell = true;
```

En el manejador *OnInit*, cambiamos el modo de operar de los dos primeros histogramas entre DRAW_HISTOGRAM y DRAW_NONE, dependiendo del parámetro *ShowBuySell* seleccionado por el usuario (el valor por defecto *true* significa mostrar los tres histogramas). Tenga en cuenta que la configuración dinámica a través de *PlotIndexSetInteger* sobrescribe la configuración estática (en este caso, sólo algunos de los ajustes) incrustado en el archivo ejecutable utilizando las directivas *#property*.

```
int OnInit()
{
   PlotIndexSetInteger(0, PLOT_DRAW_TYPE, ShowBuySell ? DRAW_HISTOGRAM : DRAW_NONE);
   PlotIndexSetInteger(1, PLOT_DRAW_TYPE, ShowBuySell ? DRAW_HISTOGRAM : DRAW_NONE);

   return INIT_SUCCEEDED;
}
```

Pero, ¿dónde está el registro de los búferes de indicadores? Volveremos sobre ello en un par de párrafos. Ahora vamos a empezar a preparar la función *OnCalculate*.

```
int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   if(prev_calculated == 0)
   {
      // TODO(1): initialization, padding with zeros
   }

   // on each new bar or set of new bars on first run
   if(prev_calculated != rates_total)
   {
      // process all or new bars
      for(int i = fmax(prev_calculated, fmax(1, rates_total - BarCount));
         i < rates_total && !IsStopped(); ++i)
      {
         // TODO(2): try to get the data and calculate the i-th bar,
         // if it doesn't work, do something!
      }
   }
   else // ticks on the current bar
   {
      // TODO(3): updating the current bar
   }

   return rates_total;
}
```

El principal problema técnico se encuentra en el bloque denominado TODO(2). El algoritmo de solicitud de ticks, que se utilizó en el script y se transferirá al indicador con cambios mínimos, los solicita utilizando la función *CopyTicksRange*. Una llamada de este tipo devuelve los datos disponibles en la base de datos de ticks. Pero si aún no está disponible para la barra histórica dada, la solicitud hace que los datos de ticks se descarguen y sincronicen de forma asíncrona (en segundo plano). En este caso, el código de llamada recibe 0 ticks. En este sentido, tras recibir una respuesta «vacía» de este tipo, el indicador debería interrumpir los cálculos con una señal de fallo (pero no de error) y volver a solicitar ticks al cabo de un tiempo. En una situación de mercado abierto normal recibimos regularmente ticks, por lo que la función *OnCalculate* probablemente debería ser llamada pronto y recalculada con la base de ticks actualizada. Pero, ¿qué hacer los fines de semana cuando no hay ticks?

Para el correcto manejo de tal situación, MQL5 proporciona un temporizador. Lo estudiaremos en uno de los capítulos siguientes, pero por ahora lo utilizaremos como «caja negra». La función especial *EventSetTimer* «solicita» al kernel que llame a nuestro programa MQL después de un número especificado de segundos. El punto de entrada para una llamada de este tipo es un manejador *OnTimer* reservado, que hemos visto en la tabla general de la sección Visión general de las funciones de gestión de eventos. Por lo tanto, si hay un retraso en la recepción de datos de tick, debe iniciar el temporizador utilizando *EventSetTimer* (un período mínimo de 1 segundo es suficiente) y devolver cero desde *OnCalculate*.

```
int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
      ...
      for(int i = fmax(prev_calculated, fmax(1, rates_total - BarCount));
         i < rates_total && !IsStopped(); ++i)
      {
         // TODO(2): try to get the data and calculate the i-th bar,
         if(/*if no data*/)
         {
            Print("No data on bar ", i, ", at ", TimeToString(time[i]),
               ". Setting up timer for refresh...");
            EventSetTimer(1); // please call us in 1 second
            return 0; // don't show anything in the window yet
         }
      }
      ...
}
```

En el manejador *OnTimer*, utilizamos la función *EventKillTimer* para detener el temporizador (si no se hace esto, el sistema seguirá llamando a nuestro manejador cada segundo). Además, necesitamos iniciar de alguna manera el recálculo del indicador. Para ello, aplicaremos otra función que aún tenemos que descubrir en el capítulo sobre gráficos: *ChartSetSymbolPeriod* (véase la sección Cambiar símbolo y marco temporal). Permite establecer una nueva combinación de un símbolo y un marco temporal para un gráfico con un identificador determinado (0 significa el gráfico actual). No obstante, si no se modifican pasando *_Symbol* y *_Period* (véase Variables predefinidas), entonces el gráfico simplemente se actualizará (los indicadores se recalculan).

```
void OnTimer()
{
   EventKillTimer();
   ChartSetSymbolPeriod(0, _Symbol, _Period); // auto-updating of the chart
}
```

Otro punto a tener en cuenta aquí es que, en el mercado abierto, el evento del temporizador y la actualización automática del gráfico pueden ser redundantes si el siguiente tick aparece antes de la llamada a *OnTimer*. Por lo tanto, crearemos una variable global (*calcDone*) para cambiar la bandera de la preparación de los cálculos. Al principio de *OnCalculate*, lo restableceremos en *false;* ; a la finalización normal del cálculo, lo estableceremos en *true*.

```
bool calcDone = false;

int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   calcDone = false;
   ...
         if(/*if no data*/)
         {
            ...
            return 0; // exit with calcDone = false
         }
   ...
   calcDone = true;
   return rates_total;
}
```

A continuación, en *OnTimer*, podemos iniciar la actualización automática del gráfico sólo cuando *calcDone* sea igual a *false*.

```
void OnTimer()
{
   EventKillTimer();
   if(!calcDone)
   {
      ChartSetSymbolPeriod(0, _Symbol, _Period);
   }
}
```

Pasemos ahora a los comentarios de *TODO(1,2,3)*, donde debemos realizar los cálculos y rellenar los búferes de indicadores. Combinemos todas estas operaciones en una clase *CalcDeltaVolume*. Así, se asignará un método distinto para cada acción, mientras que mantendremos el manejador *OnCalculate* simple (aparecerán llamadas a métodos en lugar de comentarios).

En la clase, proporcionaremos variables de miembro que aceptarán las configuraciones de usuario para el número de barras históricas procesadas y el método de cálculo del delta, así como tres arrays para los búferes de indicadores. Vamos a inicializarlos en el constructor.

```
class CalcDeltaVolume
{
   const int limit;
   const COPY_TICKS tickType;

   double buy[];
   double sell[];
   double delta[];

public:
   CalcDeltaVolume(
      const int bars,
      const COPY_TICKS type)
      : limit(bars), tickType(type), lasttime(0), lastcount(0)
   {
      // register internal arrays as indicator buffers
      SetIndexBuffer(0, buy);
      SetIndexBuffer(1, sell);
      SetIndexBuffer(2, delta);
   }
```

Podemos asignar arrays de miembros como búferes porque vamos a crear un objeto global de esta clase a continuación. Para que los datos se muestren correctamente, sólo tenemos que asegurarnos de que los arrays adjuntos a los gráficos existen en el momento de dibujarlos. Es posible cambiar los enlaces de búfer de forma dinámica (véase el ejemplo de *IndSubChartSimple.mq5* en la siguiente sección).

Tenga en cuenta que los búferes de indicadores deben ser del tipo *double*, mientras que los volúmenes son del tipo *ulong*. Por lo tanto, para valores muy grandes (por ejemplo, en plazos muy largos), hipotéticamente puede haber una pérdida de precisión.

Se ha creado el método *reset* para inicializar los búferes. La mayoría de los elementos del array se rellenan con el valor vacío EMPTY_VALUE, y las últimas barras *limit* se rellenan con cero porque allí sumaremos los volúmenes de compras y ventas por separado.

```
   void reset()
   {
      // fill in the buys array and copy the rest from it
      // empty value in all elements except the last limit bars with 0
      ArrayInitialize(buy, EMPTY_VALUE);
      ArrayFill(buy, ArraySize(buy) - limit, limit, 0);

      // duplicate the initial state into other arrays
      ArrayCopy(sell, buy);
      ArrayCopy(delta, buy);
   }
```

El cálculo en la i-ésima barra histórica se realiza mediante el método *createDeltaBar*. Su entrada recibe el número de barras y un enlace al array con las marcas de tiempo de las barras (lo recibimos como el parámetro *OnCalculate*). Los elementos del array i-ésima se inicializan a cero.

```
int createDeltaBar(const int i, const datetime &time[])
{
   delta[i] = buy[i] = sell[i] = 0;
   ...
```

A continuación, necesitamos los límites de tiempo de la i-ésima barra: *prev* y *next*, donde *next* se cuenta a la derecha de *prev* sumando el valor de la función *PeriodSeconds*, que es nueva para nosotros. Devuelve el número de segundos en el marco temporal actual. Sumando esta cantidad, encontramos el comienzo teórico de la siguiente barra. En el historial, cuando *i* no es igual al número de la última barra, podríamos sustituir la búsqueda de la siguiente marca de tiempo por *time[i + 1]*. Sin embargo, el indicador también debería funcionar en la última barra que todavía está en proceso de formación y que no tiene una barra siguiente. Por lo tanto, en general, el uso de *time[i + 1]* está prohibido.

```
   ...
   const datetime prev = time[i];
   const datetime next = prev + PeriodSeconds();
```

Cuando hicimos un cálculo similar en el script, no tuvimos que utilizar la función *PeriodSeconds*, porque no contamos la última barra actual y podíamos permitirnos buscar *next* y *prev*, como *iTime(WorkSymbol, TimeFrame, i)* y *iTime(WorkSymbol, TimeFrame, i + 1)*, respectivamente.

Además, en el método *createDeltaBar*, solicitamos ticks dentro de las marcas de tiempo encontradas (restamos 1 milisegundo desde la derecha para no tocar la siguiente barra). Los ticks llegan al array *ticks*, que es procesada por el método de ayuda *calc*. Contiene el algoritmo de script casi sin cambios. Nos hemos visto obligados a separarlo en un método designado porque el cálculo se realizará en dos situaciones diferentes: utilizando barras históricas (recuerde el comentario *TODO(2)*) y utilizando ticks en la barra actual (comentario *TODO(3)*). Vamos a analizar a continuación la segunda situación.

```
   ResetLastError();
   MqlTick ticks[];
   const int n = CopyTicksRange(_Symbol, ticks, COPY_TICKS_ALL,
      prev * 1000, next * 1000 - 1);
   if(n > -1 && _LastError == 0)
   {
      calc(i, ticks);
   }
   else
   {
      return -_LastError;
   }
   return n;
}
```

En caso de solicitud correcta, el método devuelve el número de ticks procesados, y en caso de error, devuelve un código de error con un signo menos. Tenga en cuenta que si todavía no hay ticks para la barra en la base de datos (lo cual no es un error, estrictamente hablando, pero no permite continuar con el funcionamiento visual del indicador), el método devolverá 0 (el signo de 0 no cambia su valor). Por lo tanto, en la función *OnCalculate*, tenemos que comprobar que el resultado del método para «menor o igual que» 0.

El método *calc* consiste prácticamente en líneas de trabajo del script *SeriesTicksDeltaVolume.mq5*, por lo que no lo presentaremos aquí. Quienes deseen refrescarse la memoria, pueden hacerlo consultando *IndDeltaVolume.mq5*.

Para calcular el delta en una última barra constantemente actualizada necesitamos fijar la marca de tiempo del último tick procesado con una precisión de milisegundos. A continuación, en la siguiente llamada de *OnCalculate*, podremos consultar todos los ticks posteriores a esta etiqueta.

Tenga en cuenta que no hay garantía de que el sistema tenga tiempo de llamar a nuestro manejador *OnCalculate* en cada tick en tiempo real. Si realizamos cálculos pesados, o si algún otro programa MQL carga el terminal con cálculos, o si los ticks vienen muy rápido (por ejemplo, después de importantes comunicados de prensa), los eventos pueden no llegar a la cola del indicador (no más de un evento de cada tipo se almacena en la cola, incluyendo no más de una notificación de tick). Por lo tanto, si el programa quiere obtener todos los ticks, debe solicitarlos utilizando *CopyTicksRange* o *CopyTicks*.

No obstante, la marca de tiempo del último tick procesado por sí sola no es suficiente. Los ticks pueden tener el mismo tiempo incluso teniendo en cuenta los milisegundos. Por lo tanto, no podemos añadir 1 milisegundo a la etiqueta para excluir el tick «antiguo»: los «nuevos» ticks con la misma etiqueta pueden detrás.

A este respecto, debe recordar no sólo la etiqueta, sino también el número de los últimos ticks con esta etiqueta. Entonces, la próxima vez que solicitemos ticks, podremos hacerlo empezando desde el momento recordado (es decir, incluyendo los ticks «antiguos»), pero saltándonos exactamente tantos de ellos como ya se procesaron la última vez.

Para implementar este algoritmo, se declaran dos variables en la clase *last time* y *last count*.

```
ulong last time; // millisecond marker of the last processed online tick
int last count;  // number of ticks with this label at that moment
```

A partir del array de ticks recibido del sistema, hallamos los valores para estas variables utilizando el método auxiliar *updateLastTime*.

```
void updateLastTime(const int n, const MqlTick &ticks[])
{
   lasttime = ticks[n - 1].time_msc;
   lastcount = 0;
   for(int k = n - 1; k >= 0; --k)
   {
      if(ticks[k].time_msc == ticks[n - 1].time_msc) ++lastcount;
   }
}
```

Ahora podemos refinar el método *createDeltaBar*: al procesar la última barra, llamamos por primera vez a *updateLastTime*.

```
int createDeltaBar(const int i, const datetime &time[])
{
   ...
   const int size = ArraySize(time);
   const int n = CopyTicksRange(_Symbol, ticks, COPY_TICKS_ALL,
      prev * 1000, next * 1000 - 1);
   if(n > -1 && _LastError == 0)
   {
      if(i == size - 1) // last bar
      {
         updateLastTime(n, ticks);
      }
      calc(i, ticks);
   }
   ...
}
```

Teniendo valores actualizados para *last time* y *last count*, podemos implementar un método para calcular deltas en la actual barra en línea.

```
int updateLastDelta(const int total)
{
   MqlTick ticks[];
   ResetLastError();
   const int n = CopyTicksRange(_Symbol, ticks, COPY_TICKS_ALL, lasttime);
   if(n > -1 && _LastError == 0)
   {
      const int skip = lastcount;
      updateLastTime(n, ticks);
      calc(total - 1, ticks, skip);
      return n - skip;
   }
   return -_LastError;
}
```

Para implementar este modo, hemos introducido un parámetro opcional adicional *skip* en el método *calc*. Permite omitir el cálculo en un número determinado de ticks «antiguos».

```
void calc(const int i, const MqlTick &ticks[], const int skip = 0)
{
   const int n = ArraySize(ticks);
   for(int j = skip; j < n; ++j)
   ...
}
```

La clase para el cálculo está lista; ahora sólo tenemos que insertar llamadas a tres métodos públicos en *OnCalculate*.

```
int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   if(prev_calculated == 0)
   {
      deltas.reset(); // initialization, padding with zeros
   }

   calcDone = false;

   // on each new bar or set of new bars on first run
   if(prev_calculated != rates_total)
   {
      // process all or new bars
      for(int i = fmax(prev_calculated, fmax(1, rates_total - BarCount));
         i < rates_total && !IsStopped(); ++i)
      {
         // try to get data and calculate the i-th bar,
         if((deltas.createDeltaBar(i, time)) <= 0)
         {
            Print("No data on bar ", i, ", at ", TimeToString(time[i]),
               ". Setting up timer for refresh...");
            EventSetTimer(1); // call us in 1 second
            return 0; // don't show anything in the window yet
         }
      }
   }
   else // ticks on the current bar
   {
      if((deltas.updateLastDelta(rates_total)) <= 0)
      {
         return 0; // error
      }
   }

   calcDone = true;
   return rates_total;
}
```

Vamos a compilar y ejecutar el indicador. Para empezar, es aconsejable elegir un marco temporal no superior a H1 y dejar el número de barras de *BarCount* establecido en 100 por defecto. Después de esperar un poco a que se construya el indicador, el resultado debería ser algo parecido a esto:

Indicador de volumen de delta con todos los histogramas, incluidas compras y ventas

Ahora, compárelo con lo que ocurrirá al establecer el parámetro *ShowBuySell* en *false*.

Indicador de volumen con un histograma de deltas (se ocultan las compras y ventas por separado)

Así, en este indicador, implementamos la espera de la carga de los datos de ticks para el instrumento de trabajo utilizando un temporizador, ya que los ticks pueden requerir recursos significativos. En la próxima sección analizaremos los indicadores multidivisa que trabajan a nivel de cotización, y por lo tanto una petición asíncrona simplificada para actualizar el gráfico usando *ChartSetSymbolPeriod* será suficiente para ellos. Más adelante tendremos que implementar otro tipo de espera para asegurarnos de que las series temporales de otro indicador están listas.

5.4.16 Indicadores multidivisa y de marco temporal múltiple

Hasta ahora, hemos considerado indicadores que funcionan con cotizaciones o ticks del símbolo del gráfico actual. Sin embargo, a veces es necesario analizar varios instrumentos financieros o un instrumento diferente del actual. En tales casos, como vimos en el caso del análisis de ticks, las series temporales estándar pasadas al indicador a través de los parámetros de *OnCalculate* no son suficientes. Es necesario solicitar de algún modo cotizaciones «extranjeras», esperar a que se construyan y sólo entonces calcular el indicador basándose en ellas.

La solicitud y creación de cotizaciones para un marco temporal distinto del marco temporal del gráfico actual no difiere de los mecanismos para trabajar con otros símbolos. Por lo tanto, en esta sección, consideraremos la creación de indicadores multidivisa, mientras que los indicadores de marco temporal múltiple pueden organizarse según un principio similar.

Uno de los problemas que tendremos que resolver es la sincronización de las barras en el tiempo. En concreto, para los distintos símbolos puede haber diferentes horarios de trading, fines de semana y, en general, la numeración de las barras en el gráfico principal y en las cotizaciones del símbolo «extranjero» puede ser diferente.

Para empezar, vamos a simplificar la tarea y a limitarnos a un símbolo arbitrario, que puede diferir del actual. Muy a menudo, el operador de trading necesita ver varios gráficos de diferentes símbolos al mismo tiempo (por ejemplo, el líder y el seguidor en un par correlacionado). Vamos a crear el indicador *IndSubChartSimple.mq5* para mostrar la cotización de un símbolo seleccionado por el usuario en una subventana.

IndSubChartSimple

Para repetir la apariencia del gráfico principal, proporcionaremos en los parámetros de entrada no sólo una indicación del símbolo, sino también el modo de dibujo: DRAW_CANDLES, DRAW_BARS, DRAW_LINE. Los dos primeros requieren cuatro búferes y dan salida a los cuatro precios: *Open*, *High*, *Low* y *Close* (velas japonesas o barras), y este último utiliza un único búfer para mostrar la línea en el precio *Close*. Para admitir todos los modos, utilizaremos el número máximo necesario de búferes.

```
#property indicator_separate_window
#property indicator_buffers 4
#property indicator_plots   1
#property indicator_type1   DRAW_CANDLES
#property indicator_color1  clrBlue,clrGreen,clrRed // border,bullish,bearish
```

Los arrays de búferes se describen mediante nombres de tipo de precio.

```
double open[];
double high[];
double low[];
double close[];
```

La visualización de velas japonesas está activada por defecto. En este modo, MQL5 le permite especificar no sólo un color, sino varios. En la directiva *#property indicator_colorN*, están separados por comas. Si hay dos colores, el primero determina el color de los contornos de la vela, y el segundo determina el relleno. Si hay tres colores, como en nuestro caso, el primero determina el color de los contornos, mientras que el segundo y el tercero determinan el cuerpo de las velas alcistas y bajistas, respectivamente.

En el capítulo dedicado a gráficosnos familiarizaremos con la enumeración ENUM_CHART_MODE, que describe tres modos de gráficos disponibles.

Elementos ENUM_CHART_MODE	Elementos ENUM_DRAW_TYPE
CHART_CANDLES	DRAW_CANDLES
CHART_BARS	DRAW_BARS
CHART_LINE	DRAW_LINE

Se corresponden con los modos de dibujo que hemos elegido, ya que hemos elegido deliberadamente los métodos de dibujo que repiten los estándares. Es conveniente usar aquí ENUM_CHART_MODE porque sólo contiene los 3 elementos que necesitamos, a diferencia de ENUM_DRAW_TYPE, que tiene muchos otros métodos de dibujo.

Así, las variables de entrada tienen las siguientes definiciones:

```
input string SubSymbol = ""; // Symbol
input ENUM_CHART_MODE Mode = CHART_CANDLES;
```

Se implementa una función sencilla para traducir ENUM_CHART_MODE a ENUM_DRAW_TYPE.

```
ENUM_DRAW_TYPE Mode2Style(const ENUM_CHART_MODE m)
{
   switch(m)
   {
      case CHART_CANDLES: return DRAW_CANDLES;
      case CHART_BARS: return DRAW_BARS;
      case CHART_LINE: return DRAW_LINE;
   }
   return DRAW_NONE;
}
```

La cadena vacía en el parámetro de entrada *SubSymbol* significa el símbolo de gráfico actual. Sin embargo, como MQL5 no permite editar variables de entrada, tendremos que añadir una variable global para almacenar el símbolo de trabajo real y asignarlo en el manejador *OnInit*.

```
string symbol;
...
int OnInit()
{
   symbol = SubSymbol;
   if(symbol == "") symbol = _Symbol;
   else
   {
      // making sure the symbol exists and is selected in the Market Watch
      if(!SymbolSelect(symbol, true))
      {
         return INIT_PARAMETERS_INCORRECT;
      }
   }
   ...
}
```

También necesitamos comprobar si el símbolo introducido por el usuario existe y añadirlo a *Market Watch*: esto se hace mediante la función *SymbolSelect*, que estudiaremos en el capítulo sobre símbolos.

Para generalizar la configuración de búferes y gráficos, el código fuente dispone de varias funciones de ayuda:

- InitBuffer: configuración de un búfer
- InitBuffers: configuración de todo el conjunto de buffers
- InitPlot: configuración de un gráfico

Las funciones separadas combinan varias acciones que se repiten al registrar entidades idénticas. También abren la vía a un mayor desarrollo de este indicador en el capítulo sobre gráficos: apoyaremos el cambio interactivo de los ajustes de dibujo en respuesta a las manipulaciones del usuario con el gráfico (véase la versión completa del indicador *IndSubChart.mq5* en el capítulo Modos de visualización de gráficos).

```
void InitBuffer(const int index, double &buffer[],
   const ENUM_INDEXBUFFER_TYPE style = INDICATOR_DATA,
   const bool asSeries = false)
{
   SetIndexBuffer(index, buffer, style);
   ArraySetAsSeries(buffer, asSeries);
}

string InitBuffers(const ENUM_CHART_MODE m)
{
   string title;
   if(m == CHART_LINE)
   {
      InitBuffer(0, close, INDICATOR_DATA, true);
      // hiding all buffers not used for the line chart
      InitBuffer(1, high, INDICATOR_CALCULATIONS, true);
      InitBuffer(2, low, INDICATOR_CALCULATIONS, true);
      InitBuffer(3, open, INDICATOR_CALCULATIONS, true);
      title = symbol + " Close";
   }
   else
   {
      InitBuffer(0, open, INDICATOR_DATA, true);
      InitBuffer(1, high, INDICATOR_DATA, true);
      InitBuffer(2, low, INDICATOR_DATA, true);
      InitBuffer(3, close, INDICATOR_DATA, true);
      title = "# Open;# High;# Low;# Close";
      StringReplace(title, "#", symbol);
   }
   return title;
}
```

Tenga en cuenta que, cuando activa el modo de gráfico de líneas, sólo se utiliza el array *close*. Se le asigna el índice 0. Los tres arrays restantes están completamente ocultos al usuario debido a la propiedad INDICATOR_CALCULATIONS. Los cuatro arrays se utilizan en los modos vela y barra, y su numeración cumple con el estándar OHLC, tal y como requieren los tipos de dibujo DRAW_CANDLES y DRAW_BARS. A todos los arrays se les asigna la propiedad «serie», es decir, se indexan de derecha a izquierda.

La función *InitBuffers* devuelve el encabezado de los búferes en *Data Window*.

Todos los atributos de trazado necesarios se establecen en la función *InitPlot*.

```
void InitPlot(const int index, const string name, const int style,
   const int width = -1, const int colorx = -1,
   const double empty = EMPTY_VALUE)
{
  PlotIndexSetInteger(index, PLOT_DRAW_TYPE, style);
  PlotIndexSetString(index, PLOT_LABEL, name);
  PlotIndexSetDouble(index, PLOT_EMPTY_VALUE, empty);
  if(width != -1) PlotIndexSetInteger(index, PLOT_LINE_WIDTH, width);
  if(colorx != -1) PlotIndexSetInteger(index, PLOT_LINE_COLOR, colorx);
}
```

La configuración inicial de un único gráfico (con índice 0) se realiza mediante nuevas funciones en el manejador *OnInit*.

```
int OnInit()
{
   ...
   InitPlot(0, InitBuffers(Mode), Mode2Style(Mode));
   IndicatorSetString(INDICATOR_SHORTNAME, "SubChart (" + symbol + ")");
   IndicatorSetInteger(INDICATOR_DIGITS, (int)SymbolInfoInteger(symbol, SYMBOL_DIGITS

   return INIT_SUCCEEDED;
}
```

Aunque la configuración se realiza una sola vez en esta versión del indicador, se hace de forma dinámica, teniendo en cuenta el parámetro de entrada *mode*, a diferencia de la configuración estática proporcionadas por las directivas *#property*. En el futuro, en la versión completa del indicador, podremos llamar a *InitPlot* muchas veces, cambiando la representación externa del indicador «sobre la marcha».

Los búferes se rellenan en *OnCalculate*. En el caso más sencillo, cuando el símbolo dado coincide con el gráfico, podemos utilizar simplemente la siguiente implementación.

```
int OnCalculate(const int rates_total, const int prev_calculated,
   const datetime &time[],
   const double &op[], const double &hi[], const double &lo[], const double &cl[],
   const long &[], const long &[], const int &[]) // unused
{
   if(prev_calculated ==0) // needs clarification (see further)
   {
      ArrayInitialize(open, EMPTY_VALUE);
      ArrayInitialize(high, EMPTY_VALUE);
      ArrayInitialize(low, EMPTY_VALUE);
      ArrayInitialize(close, EMPTY_VALUE);
   }

   if(_Symbol != symbol)
   {
      // being developed
      ...
   }
   else
   {
      ArraySetAsSeries(op, true);
      ArraySetAsSeries(hi, true);
      ArraySetAsSeries(lo, true);
      ArraySetAsSeries(cl, true);
      for(int i = 0; i < MathMax(rates_total - prev_calculated, 1); ++i)
      {
         open[i] = op[i];
         high[i] = hi[i];
         low[i] = lo[i];
         close[i] = cl[i];
      }
   }

   return rates_total;
}
```

No obstante, al procesar un símbolo arbitrario, los parámetros del array no contienen las cotizaciones necesarias, y el número total de barras disponibles es probablemente diferente. Además, cuando se coloca un indicador en un gráfico por primera vez, es posible que las cotizaciones de un símbolo «extranjero» no estén listas en absoluto si no se ha abierto previamente otro gráfico cercano para él. Además, las cotizaciones de un símbolo ajeno se cargarán de forma asíncrona, por lo que puede «llegar» un nuevo lote de barras en cualquier momento, lo que requerirá un recálculo completo.

Por lo tanto, vamos a crear variables que controlan el número de barras en el otro símbolo (*lastAvailable*), un «clon» editable de un argumento constante *prev_calculated*, así como una bandera de cotizaciones preparadas.

```
static bool initialized; // symbol quotes readiness flag
static int lastAvailable; // number of bars for a symbol (and the current timefram
int _prev_calculated = prev_calculated; // editable copy of prev_calculated
```

Al principio de *OnCalculate* vamos a añadir una comprobación para la aparición simultánea de más de una barra: utilizamos la variable *lastAvailable* que rellenamos en función del valor de *iBars(symbol,*

_Period) antes de la salida regular anterior de la función, es decir, en caso de que el cálculo se realice con éxito. Si se carga un historial adicional, debemos restablecer *_prev_calculated* y el número de barras a 0, así como eliminar la bandera de preparación para volver a calcular el indicador.

```
int OnCalculate(const int rates_total, const int prev_calculated,
   const datetime &time[],
   const double &op[], const double &hi[], const double &lo[], const double &cl[],
   const long &[], const long &[], const int &[]) // unused
{
   ...
   if(iBars(symbol, _Period) - lastAvailable > 1)
   {
      // loading additional history or first start
      _prev_calculated = 0;
      initialized = false;
      lastAvailable = 0;
   }

   // then everywhere we use a copy of _prev_calculated
   if(_prev_calculated == 0)
   {
      ArrayInitialize(open, EMPTY_VALUE);
      ArrayInitialize(high, EMPTY_VALUE);
      ArrayInitialize(low, EMPTY_VALUE);
      ArrayInitialize(close, EMPTY_VALUE);
   }

   if(_Symbol != symbol)
   {
      // request quotes and "wait" till they are ready
      ...
      // main calculation (filling buffers)
      ...
   }
   else
   {
      ... // as is
   }
   lastAvailable = iBars(symbol, _Period);
   return rates_total;
}
```

La palabra «wait» (esperar) del comentario no está entrecomillada accidentalmente. Como recordamos, no podemos esperar realmente en los indicadores (para no ralentizar el hilo de interfaz del terminal). En lugar de ello, si no hay suficientes datos, simplemente debemos salir de la función. Así, «wait» significa esperar a que se calcule el siguiente evento: a la llegada de un tick o en respuesta a una solicitud de actualización del gráfico.

El siguiente código comprobará si las cotizaciones están listas.

```
int OnCalculate(const int rates_total, const int prev_calculated,
   const datetime &time[],
   const double &op[], const double &hi[], const double &lo[], const double &cl[],
   const long &[], const long &[], const int &[]) // unused
{
   ...
   if(_Symbol != symbol)
   {
      if(!initialized)
      {
         Print("Host ", _Symbol, " ", rates_total, " bars up to ", (string)time[0]);
         Print("Updating ", symbol, " ", lastAvailable, " -> ", iBars(symbol, _Period
            (iBars(symbol, _Period) > 0 ?
               (string)iTime(symbol, _Period, iBars(symbol, _Period) - 1) : "n/a"),
            "... Please wait");
         if(QuoteRefresh(symbol, _Period, time[0]))
         {
            Print("Done");
            initialized = true;
         }
         else
         {
            // asynchronous request to update the chart
            ChartSetSymbolPeriod(0, _Symbol, _Period);
            return 0; // nothing to show yet
         }
      }
      ...
```

El trabajo principal lo realiza la función especial *QuoteRefresh*. Recibe como argumentos el símbolo deseado, el marco temporal y la hora de la primera barra (la más antigua) del gráfico actual: no nos interesan fechas anteriores, pero el símbolo solicitado puede no tener un historial para toda esta profundidad. Por eso es conveniente ocultar todas las complejidades de las comprobaciones en una función independiente.

La función devolverá *true* en cuanto los datos se hayan descargado y sincronizado en la medida de lo posible. Analizaremos su estructura interna dentro de un minuto.

Una vez realizada la sincronización, utilizamos la función *iBarShift* para buscar barras síncronas y copiar sus valores OHLC (funciones *iOpen*, *iHigh*, *iLow*, *iClose*).

```
   ArraySetAsSeries(time, true); // go from present to past
   for(int i = 0; i < MathMax(rates_total - _prev_calculated, 1); ++i)
   {
      int x = iBarShift(symbol, _Period, time[i], true);
      if(x != -1)
      {
         open[i] = iOpen(symbol, _Period, x);
         high[i] = iHigh(symbol, _Period, x);
         low[i] = iLow(symbol, _Period, x);
         close[i] = iClose(symbol, _Period, x);
      }
      else
      {
         open[i] = high[i] = low[i] = close[i] = EMPTY_VALUE;
      }
   }
```

Una forma alternativa y, a primera vista, más eficiente de copiar arrays de precios enteros utilizando las funciones Copiar no es adecuada en este caso, porque las barras con índices iguales pueden corresponder a diferentes marcas de tiempo en diferentes símbolos. Por lo tanto, después de copiar, habría que analizar las fechas y mover los elementos dentro de los búferes, ajustándolos a la hora del gráfico actual.

Puesto que en la función *iBarShift* *true* se pasa como último parámetro, la función buscará una coincidencia exacta de la hora de las barras. Si no hay ninguna barra en otro símbolo, obtendremos -1 y mostraremos un espacio vacío (EMPTY_VALUE) en el gráfico.

Tras un cálculo completo satisfactorio, las nuevas barras se calcularán en modo económico, es decir, teniendo en cuenta *_prev_calculated* y *rates_total*.

Pasemos ahora a la función *QuoteRefresh*. Se trata de una función universal y útil, por lo que se incluye en el archivo de encabezado *QuoteRefresh.mqh*.

Al principio, comprobamos si las series temporales del símbolo actual y el marco temporal actual se solicitan a un programa MQL de tipo indicador. Tales solicitudes están prohibidas, ya que la serie temporal «nativa» sobre la que se ejecuta el indicador ya está siendo construida por el terminal o está lista: solicitarla de nuevo puede provocar bucles o bloqueos. Por lo tanto, simplemente devolvemos el indicador de sincronización (SERIES_SYNCHRONIZED) y, si aún no está listo, el indicador deberá comprobar los datos más tarde (en los próximos ticks, por temporizador, o cualquier otra cosa).

```
bool QuoteRefresh(const string asset, const ENUM_TIMEFRAMES period,
   const datetime start)
{
   if(MQL5InfoInteger(MQL5_PROGRAM_TYPE) == PROGRAM_INDICATOR
      && _Symbol == asset && _Period == period)
   {
      return (bool)SeriesInfoInteger(asset, period, SERIES_SYNCHRONIZED);
   }
   ...
```

La segunda comprobación se refiere al número de barras: si ya es igual al máximo permitido en los gráficos, no tiene sentido seguir descargando nada.

```
   if(Bars(asset, period) >= TerminalInfoInteger(TERMINAL_MAXBARS))
   {
      return (bool)SeriesInfoInteger(asset, period, SERIES_SYNCHRONIZED);
   }
   ...
```

La siguiente parte de código solicita secuencialmente al terminal las fechas de inicio de las cotizaciones disponibles:

- en un plazo determinado (SERIES_FIRSTDATE);
- sin enlace a un marco temporal (SERIES_TERMINAL_FIRSTDATE) en la base de datos local del terminal;
- sin un enlace a un marco temporal (SERIES_SERVER_FIRSTDATE) en el servidor.

Si en algún momento la fecha solicitada ya se encuentra en el área de datos disponibles, obtenemos *true* como señal de que está lista. En caso contrario, se solicitan datos a la base de datos local del terminal o al servidor, seguidos de la construcción de una serie temporal (todo ello se realiza de forma asíncrona y automática en respuesta a nuestras llamadas a *CopyTime*; pueden utilizarse otras funciones de *Copy*).

```
      datetime times[1];
      datetime first = 0, server = 0;
      if(PRTF(SeriesInfoInteger(asset, period, SERIES_FIRSTDATE, first)))
      {
         if(first > 0 && first <= start)
         {
            // application data exists, it is already ready or is being prepared
            return (bool)SeriesInfoInteger(asset, period, SERIES_SYNCHRONIZED);
         }
         else
         if(PRTF(SeriesInfoInteger(asset, period, SERIES_TERMINAL_FIRSTDATE, first)))
         {
            if(first > 0 && first <= start)
            {
               // technical data exists in the terminal database,
               // initiate the construction of a timeseries or immediately get the desir
               return PRTF(CopyTime(asset, period, first, 1, times)) == 1;
            }
            else
            {
               if(PRTF(SeriesInfoInteger(asset, period, SERIES_SERVER_FIRSTDATE, server)
               {
                  // technical data exists on the server, let's request it
                  if(first > 0 && first < server)
                     PrintFormat(
                       "Warning: %s first date %s on server is less than on terminal ",
                        asset, TimeToString(server), TimeToString(first));
                  // you can't ask for more than the server has - so fmax
                  return PRTF(CopyTime(asset, period, fmax(start, server), 1, times)) ==
               }
            }
         }
      }

      return false;
   }
```

El indicador está listo. Vamos a compilarlo y ejecutarlo, por ejemplo, en el gráfico EURUSD, H1, especificando USDRUB como símbolo adicional. El registro mostrará algo como esto:

```
Host EURUSD 20001 bars up to 2018.08.09 13:00:00
Updating USDRUB 0 -> 14123 / 2014.12.22 11:00:00... Please wait
SeriesInfoInteger(symbol,period,SERIES_FIRSTDATE,first)=false / HISTORY_NOT_FOUND(440
Host EURUSD 20001 bars up to 2018.08.09 13:00:00
Updating USDRUB 0 -> 14123 / 2014.12.22 11:00:00... Please wait
SeriesInfoInteger(symbol,period,SERIES_FIRSTDATE,first)=true / ok
Done
```

Una vez finalizado el proceso (mensaje «Done» (hecho)), la subventana mostrará las velas del otro gráfico.

Indicador IndSubChartSimple - DRAW_CANDLES con cotizaciones de un símbolo de terceros

Es importante señalar que, debido a la brevedad de la sesión de trading, las barras significativas para USDRUB ocupan sólo la parte diaria de cada intervalo diario.

IndUnityPercent

El segundo indicador que crearemos en esta sección es un indicador multidivisa (multiactivo) real *IndUnityPercent.mq5*. Su idea es mostrar la fuerza relativa de todas las divisas independientes (activos) incluidas en los instrumentos financieros dados. Por ejemplo, si negociamos una cesta de dos tickers EURUSD y XAUUSD, entonces en el valor de la cesta se tienen en cuenta el dólar, el euro y el oro: cada uno de estos activos tiene un valor relativo en comparación con los demás.

En cada momento existen precios corrientes, que se describen mediante las siguientes fórmulas:

```
EUR / USD = EURUSD
XAU / USD = XAUUSD
```

donde las variables EUR, USD, XAU son algunos «valores» independientes de los activos, y EURUSD y XAUUSD son constantes (cotizaciones conocidas).

Para encontrar las variables vamos a añadir otra ecuación al sistema, limitando la suma de los cuadrados de las variables a uno (de ahí la primera palabra del nombre del indicador, Unidad):

```
EUR * EUR + USD * USD + XAU * XAU = 1
```

Puede haber muchas más variables, y es lógico designarlas como x_i. Tenga en cuenta que x_0 es la divisa principal, necesaria y común a todos los instrumentos.

Entonces, en términos generales, las fórmulas para calcular variables se escribirán de la siguiente manera (omitiremos el proceso de su derivación):

```
x0 = sqrt(1 / (1 + sum(C(xi, x0)2))), i = 1..n
xi = C(xi, x0) * x0, i = 1..n
```

donde n es el número de variables, $C(x_i,x_0)$ es la cotización del par i-ésimo. Tenga en cuenta que el número de variables es mayor que el número de instrumentos en 1.

Dado que las cotizaciones que intervienen en el cálculo suelen ser muy diferentes (por ejemplo, como en el caso de EURUSD y XAUUSD) y se expresan sólo entre sí (es decir, sin referencia a ninguna base estable), tiene sentido pasar de los valores absolutos a los porcentajes de variación. Así, al escribir algoritmos según las fórmulas anteriores, en lugar de la cotización $C(x_i,x_0)$ tomaremos la proporción $C(x_i,x_0)[0] / C(x_i,x_0)[1]$, donde los índices entre corchetes significan la barra actual [0] y la anterior [1]. Además, para acelerar el cálculo, puede prescindir de elevar al cuadrado y sacar la raíz cuadrada.

Para visualizar las líneas, proporcionaremos un cierto número máximo admisible de divisas y búferes de indicadores. Por supuesto, es posible utilizar sólo algunos de ellos en los cálculos si el usuario introduce menos símbolos. Pero no puede aumentar el límite dinámicamente: tendrá que cambiar las directivas y recompilar el indicador.

```
#define BUF_NUM 15
#property indicator_separate_window
#property indicator_buffers BUF_NUM
#property indicator_plots BUF_NUM
```

Al aplicar este indicador, resolveremos un problema desagradable por el camino. Dado que habrá muchos búferes del mismo tipo, el enfoque estándar es codificarlos ampliamente por «multiplicación» (el indeseable estilo de programación «copiar y pegar»).

```
double buffer1[];
...
double buffer15[];

void OnInit()
{
   SetIndexBuffer(0, buffer1);
   ...
   SetIndexBuffer(14, buffer15);
}
```

Esto es incómodo, ineficaz y propenso a errores. En lugar de ello, vamos a aplicar la POO. Crearemos una clase que almacenará un array para el buffer del indicador y será responsable de su configuración uniforme, ya que nuestros búferes deberían ser iguales (excepto por los colores y, posiblemente, un mayor grosor para aquellas divisas que conforman el símbolo del gráfico actual, pero esto se afina más tarde, después de que el usuario introduzca los parámetros).

Con una clase de este tipo, podemos simplemente distribuir un array de sus objetos, y los búferes de indicadores se conectarán y configurarán automáticamente en la cantidad requerida. Esquemáticamente, este enfoque se ilustra con el siguiente pseudocódigo:

```
// "engine" code supporting an array of unified indicator buffers
class Buffer
{
   static int count; // global buffer counter
   double array[];   // array for this buffer
   int cursor;       // pointer of assigned element
public:
   // constructor sets up and connects the array
   Buffer()
   {
      SetIndexBuffer(count++, array);
      ArraySetAsSeries(array, ...);
   }
   // overload to set the number of the element of interest
   Buffer *operator[](int index)
   {
      cursor = index;
      return &this;
   }
   // overload to write value to selected element
   double operator=(double x)
   {
      buffer[cursor] = x;
      return x;
   }
   ...
};

static int Buffer::count;
```

Con las sobrecargas de operadores, podemos ceñirnos a la sintaxis familiar para asignar valores a los elementos de un objeto de búfer: *buffer[i] = value*.

En el código del indicador, en lugar de muchas líneas con descripciones de arrays individuales, bastará con definir un «array de arrays».

```
// indicator code
// construct 15 buffer objects with auto-registration and configuration
Buffer buffers[15];
...
```

La versión completa de las clases que implementan este mecanismo está disponible en el archivo *IndBufArray.mqh*. Tenga en cuenta que sólo admite búferes, no diagramas. Lo ideal sería ampliar el conjunto de clases con otras nuevas, lo que permitiría crear objetos de diagrama ya hechos que ocuparían el número necesario de búferes en el array de búferes según el tipo de diagrama concreto. Le sugerimos que estudie y complete usted mismo el expediente. En concreto, el código contiene una clase que gestiona un array de búferes de indicador *BufferArray* para crear «arrays de arrays» con los mismos valores de propiedad, como el tipo ENUM_INDEXBUFFER_TYPE, dirección de indexación, valor vacío. Lo utilizamos en el nuevo indicador del siguiente modo:

```
BufferArray buffers(BUF_NUM, true);
```

Aquí, el número requerido de búferes se pasa en el primer parámetro del constructor, y el indicador de indexación como en una serie temporal se pasa en el segundo parámetro (encontrará más información al respecto más adelante).

Después de esta definición, podemos utilizar una práctica notación en cualquier parte del código para establecer el valor de la barra j-ésima del búfer i-ésimo (utiliza una sobrecarga doble del operador [] en el objeto de búfer y también en el array de búferes):

```
buffers[i][j] = value;
```

En las variables de entrada del indicador, permitiremos que el usuario especifique una lista de símbolos separados por comas y limite el número de barras para el cálculo sobre el historial, a fin de controlar la carga y sincronización de un conjunto potencialmente grande de instrumentos. Si decide mostrar todo el historial disponible, deberá identificar y aplicar el menor número de barras disponibles para los distintos instrumentos y controlar la carga de historial adicional desde el servidor.

```
input string Instruments = "EURUSD,GBPUSD,USDCHF,USDJPY,AUDUSD,USDCAD,NZDUSD";
input int BarLimit = 500;
```

Al iniciar el programa, analice la lista de símbolos y forme un array *Symbols* independiente de tamaño *SymbolCount*.

```
string Symbols[];
int direction[]; // direct(+1)/reverse(-1) rate to the common currency
int SymbolCount;
```

Todos los símbolos deben tener la misma divisa común (normalmente USD) para revelar correlaciones mutuas. Dependiendo de si esta divisa común en un símbolo concreto es la base (en el primer lugar del par, si hablamos de Forex) o la divisa de cotización (en el segundo lugar del par Forex), el cálculo utiliza sus cotizaciones directas o inversas (1.0 / tasa). Esta dirección se almacenará en el array *Direction*.

Veamos la función *InitSymbols* que realiza las acciones descritas. Si la lista se analiza correctamente, devuelve el nombre de la divisa común. La función *SymbolInfoString* integrada permite obtener la divisa base y la divisa de cotización de cualquier instrumento financiero: la estudiaremos en el capítulo sobre instrumentos financieros.

```
string InitSymbols()
{
   SymbolCount = fmin(StringSplit(Instruments, ',', Symbols), BUF_NUM - 1);
   ArrayResize(Symbols, SymbolCount);
   ArrayResize(Direction, SymbolCount);
   ArrayInitialize(Direction, 0);

   string common = NULL; // common currency

   for(int i = 0; i < SymbolCount; i++)
   {
      // guarantee the presence of the symbol in the Market Review
      if(!SymbolSelect(Symbols[i], true))
      {
         Print("Can't select ", Symbols[i]);
         return NULL;
      }

      // get the currencies that make up the symbol
      string first, second;
      first = SymbolInfoString(Symbols[i], SYMBOL_CURRENCY_BASE);
      second = SymbolInfoString(Symbols[i], SYMBOL_CURRENCY_PROFIT);

      // count the number of inclusions of each currency
      if(first != second)
      {
         workCurrencies.inc(first);
         workCurrencies.inc(second);
      }
      else
      {
         workCurrencies.inc(Symbols[i]);
      }
   }
   ...
```

El bucle realiza un seguimiento de la aparición de cada divisa en todos los instrumentos utilizando una clase de plantilla auxiliar *MapArray*. Dicho objeto se describe en el indicador a nivel global y requiere la conexión del archivo de encabezado *MapArray.mqh*.

```
#include <MQL5Book/MapArray.mqh>
...
// array of pairs [name; number]
// to calculate currency usage statistics
MapArray<string,int> workCurrencies;
...
string InitSymbols()
{
   ...
}
```

Dado que esta clase desempeña un papel secundario, no se describe en detalle aquí. Puede consultar el código fuente para obtener más detalles. La conclusión es que cuando se llama a su método *inc* para un nuevo nombre de divisa, se añade al array interno con el valor inicial del contador igual a 1, y si el nombre ya se ha encontrado, el contador se incrementa en 1.

Posteriormente, encontramos la divisa común como aquella cuyo contador es mayor que 1. Con los ajustes correctos, el resto de divisas deberían encontrarse exactamente una vez. He aquí la continuación de la función *InitSymbols*.

```
      ...
      // find the common currency based on currency usage statistics
      for(int i = 0; i < workCurrencies.getSize(); i++)
      {
         if(workCurrencies[i] > 1) // counter greater than 1
         {
            if(common == NULL)
            {
               common = workCurrencies.getKey(i); // get the name of the i-th currency
            }
            else
            {
               Print("Collision: multiple common symbols");
               return NULL;
            }
         }
      }

      if(common == NULL) common = workCurrencies.getKey(0);

      // knowing the common currency, determine the "direction" of each symbol
      for(int i = 0; i < SymbolCount; i++)
      {
         if(SymbolInfoString(Symbols[i], SYMBOL_CURRENCY_PROFIT) == common)
            Direction[i] = +1;
         else if(SymbolInfoString(Symbols[i], SYMBOL_CURRENCY_BASE) == common)
            Direction[i] = -1;
         else
         {
            Print("Ambiguous symbol direction ", Symbols[i], ", defaults used");
            Direction[i] = +1;
         }
      }

      return common;
   }
```

Teniendo lista la función *InitSymbols*, podemos escribir *OnInit* (con simplificaciones).

```
int OnInit()
{
   const string common = InitSymbols();
   if(common == NULL) return INIT_PARAMETERS_INCORRECT;

   string base = SymbolInfoString(_Symbol, SYMBOL_CURRENCY_BASE);
   string profit = SymbolInfoString(_Symbol, SYMBOL_CURRENCY_PROFIT);

   // setting up lines by the number of currencies (number of symbols + 1)
   for(int i = 0; i <= SymbolCount; i++)
   {
      string name = workCurrencies.getKey(i);
      PlotIndexSetString(i, PLOT_LABEL, name);
      PlotIndexSetInteger(i, PLOT_DRAW_TYPE, DRAW_LINE);
      PlotIndexSetInteger(i, PLOT_SHOW_DATA, true);
      PlotIndexSetInteger(i, PLOT_LINE_WIDTH, 1 + (name == base || name == profit));
   }

   // hide extra buffers in the Data Window
   for(int i = SymbolCount + 1; i < BUF_NUM; i++)
   {
      PlotIndexSetInteger(i, PLOT_SHOW_DATA, false);
   }

   // single level at 1.0
   IndicatorSetInteger(INDICATOR_LEVELS, 1);
   IndicatorSetDouble(INDICATOR_LEVELVALUE, 0, 1.0);

   // Name with parameters
   IndicatorSetString(INDICATOR_SHORTNAME,
      "Unity [" + (string)workCurrencies.getSize() + "]");

   // accuracy
   IndicatorSetInteger(INDICATOR_DIGITS, 5);

   return INIT_SUCCEEDED;
}
```

Ahora vamos a familiarizarnos con el manejador del evento principal *OnCalculate*.

Es importante señalar que el orden de iteración sobre las barras en el bucle principal se invierte, como en una serie temporal, del presente al pasado. Este enfoque es más conveniente para los indicadores multidivisa, porque la profundidad del historial de los distintos símbolos puede ser diferente, y tiene sentido calcular las barras desde el momento actual hacia atrás, hasta el primer momento en que no hay datos para ninguno de los símbolos. En este caso, la finalización anticipada del bucle no debe tratarse como un error: debemos devolver *rates_total* para mostrar en el gráfico los valores de las barras más relevantes que ya se han calculado.

No obstante, en esta versión simplificada de *IndUnityPercent*, no hacemos esto y utilizamos un enfoque más sencillo y rígido: el usuario debe definir la profundidad incondicional de la consulta del historial utilizando el parámetro *BarLimit*. En otras palabras: para todos los símbolos, debe haber datos hasta la

marca de tiempo de la barra con el número *BarLimit* en el símbolo del gráfico. De lo contrario, el indicador intentará descargar los datos que faltan.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double& price[])
{
   if(prev_calculated == 0)
   {
      buffers.empty(); // delegate total cleanup to the BufferArray class
   }

   // main loop in the direction "as in a timeseries" from the present to the past
   const int limit = MathMin(rates_total - prev_calculated + 1, BarLimit);
   for(int i = 0; i < limit; i++)
   {
      if(!calculate(i))
      {
         EventSetTimer(1); // give 1 more second to upload and prepare data
         return 0; // let's try to recalculate on the next call
      }
   }

   return rates_total;
}
```

La función *Calculate* (véase más abajo) calcula los valores de todos los búferes de la barra i-ésima. En caso de que falten datos, devolverá *false*, e iniciaremos un temporizador para dar tiempo a construir series temporales para todos los instrumentos requeridos. En el manejador del temporizador, enviaremos una petición al terminal para que actualice el gráfico de la forma habitual.

```
void OnTimer()
{
   EventKillTimer();
   ChartSetSymbolPeriod(0, _Symbol, _Period);
}
```

En la función *Calculate*, primero determinamos el intervalo de fechas de la barra actual y la anterior, sobre el que se calcularán los cambios.

```
bool Calculate(const int bar)
{
   const datetime time0 = iTime(_Symbol, _Period, bar);
   const datetime time1 = iTime(_Symbol, _Period, bar + 1);
   ...
```

Se han necesitado dos fechas para llamar a la siguiente función *CopyClose* en su versión, donde se indica el intervalo de fechas. En este indicador no podemos utilizar la opción con el número de barras, ya que cualquier símbolo puede tener huecos arbitrarios en las barras, diferentes de los huecos en otros símbolos. Por ejemplo, si hay barras en un símbolo *t* (actual) y *t-1* (anterior), entonces es posible calcular correctamente el cambio *Close[t]/Close[t-1]*. No obstante, en otro símbolo, la barra *t* puede estar ausente, y una petición de dos barras devolverá las barras «más cercanas» (en el pasado) a la

izquierda, y este pasado puede estar bastante alejado del «presente» (por ejemplo, corresponder a la sesión de trading del día anterior si el símbolo no se negocia las 24 horas del día).

Para evitar que esto ocurra, el indicador solicita cotizaciones estrictamente en el intervalo, y si éste resulta estar vacío para un símbolo concreto, significa que no hay cambios.

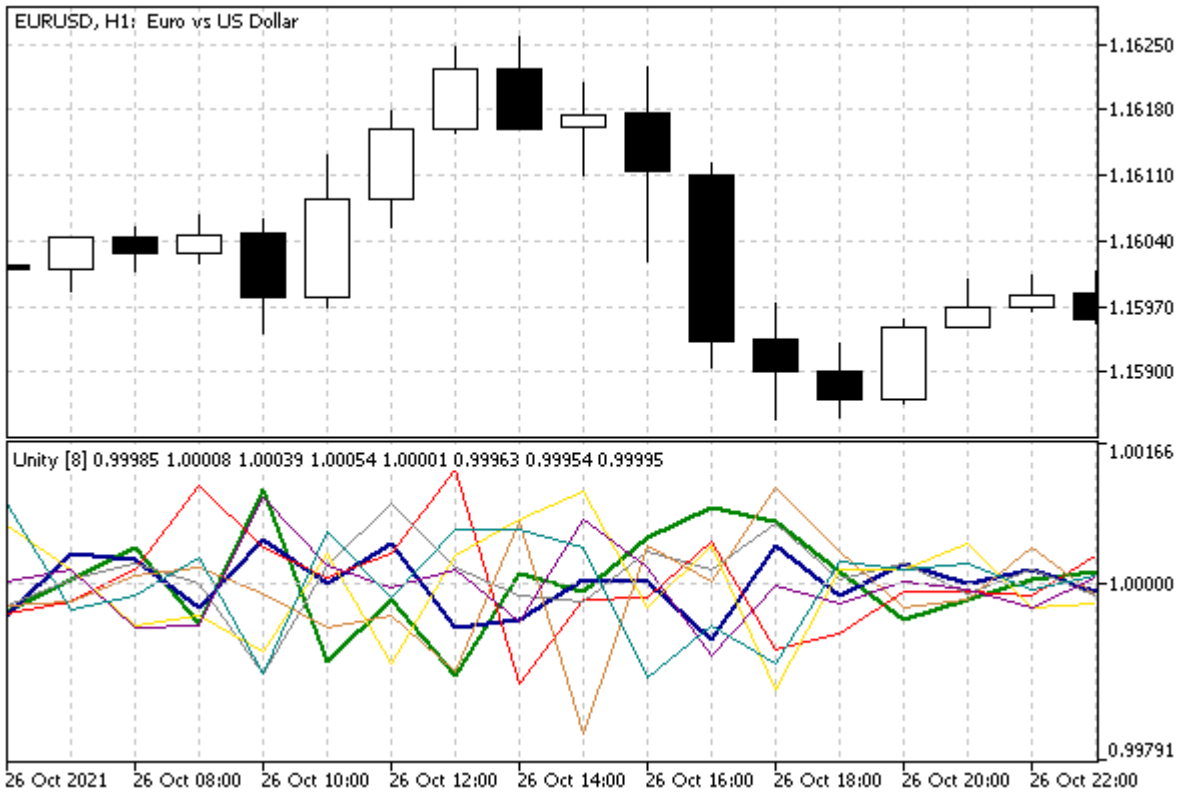
Al mismo tiempo, se pueden dar situaciones en las que una consulta de este tipo devuelva más de dos barras, y en este caso, las dos últimas (a la derecha) se toman siempre como las más relevantes. Por ejemplo, cuando se coloca en el gráfico USDRUB,H1, el indicador «verá» que después de la barra de las 17:00 de cada día hábil, hay una barra a las 10:00 del siguiente día hábil. Sin embargo, para los principales pares de divisas Forex, como EURUSD, habrá 16 barras H1 vespertinas, nocturnas y matutinas entre ellas.

```
bool Calculate(const int bar)
{
   ...
   double w[]; // receiving array of quotes (by bar)
   double v[]; // character changes
   ArrayResize(v, SymbolCount);

   // find quote changes for each symbol
   for(int j = 0; j < SymbolCount; j++)
   {
      // try to get at least 2 bars for the j-th symbol,
      // corresponding to two bars of the symbol of the current chart
      int x = CopyClose(Symbols[j], _Period, time0, time1, w);
      if(x < 2)
      {
         // if there are no bars, try to get the previous bar from the past
         if(CopyClose(Symbols[j], _Period, time0, 1, w) != 1)
         {
            return false;
         }
         // then duplicate it as no change indication
         // (in principle, it was possible to write any constant 2 times)
         x = 2;
         ArrayResize(w, 2);
         w[1] = w[0];
      }

      // find the reverse course when needed
      if(Direction[j] == -1)
      {
         w[x - 1] = 1.0 / w[x - 1];
         w[x - 2] = 1.0 / w[x - 2];
      }

      // calculating changes as a ratio of two values
      v[j] = w[x - 1] / w[x - 2]; // last / previous
   }
   ...
```

Cuando se reciben los cambios, el algoritmo trabaja según las fórmulas dadas anteriormente y escribe los valores en los búferes de indicadores.

```
      double sum = 1.0;
      for(int j = 0; j < SymbolCount; j++)
      {
         sum += v[j];
      }

      const double base_0 = (1.0 / sum);
      buffers[0][bar] = base_0 * (SymbolCount + 1);
      for(int j = 1; j <= SymbolCount; j++)
      {
         buffers[j][bar] = base_0 * v[j - 1] * (SymbolCount + 1);
      }

      return true;
   }
```

Vamos a ver cómo funciona el indicador con la configuración por defecto en un conjunto de instrumentos básicos de Forex (en la primera colocación, puede llevar un tiempo notable recibir series temporales si los gráficos no se abrieron para los instrumentos).



Indicador multidivisa IndUnityPercent con las principales divisas Forex

La distancia entre las líneas de dos divisas en la ventana del indicador es igual al cambio de la cotización correspondiente en porcentaje (entre dos precios consecutivos *Close*). De ahí la segunda palabra del nombre del indicador: porcentaje.

En el próximo capítulo sobre el uso programático de los indicadores presentaremos una versión avanzada de *IndUnityPercentPro.mq5*, en la que las funciones de *Copy* se sustituirán por la llamada a indicadores integrados *iMA*, lo que nos permitirá aplicar el suavizado y el cálculo para un tipo arbitrario de precios sin ningún esfuerzo adicional.

5.4.17 Seguimiento de formación de barras

El indicador *IndUnityPercent.mq5* comentado en la sección anterior se recalcula en la última barra de cada tick, ya que utiliza los precios de *Close*. Algunos indicadores y Asesores Expertos se han desarrollado especialmente con un estilo más económico, con un único cálculo en cada barra. Por ejemplo, podríamos calcular la fórmula de Unity a precios de apertura, y entonces tiene sentido saltarse los ticks. Hay varias formas de detectar una nueva barra:

- Recordar la hora de la barra 0 actual (a través del parámetro *time* de la función *OnCalculate* - *time[0]* o, en general, *iTime(symbol, period, 0)*) y esperar a que cambie.
- Memorizar el número de barras *rates_total* (o *iBars(symbol, period)*) y responder a un aumento de 1 (un cambio a una cantidad diferente en un sentido u otro es sospechoso y puede indicar una modificación del historial).
- Esperar una barra con un volumen de ticks igual a 1 (el primer tick de la barra).

Sin embargo, con la naturaleza multidivisa del indicador, el propio concepto de formación de una nueva barra no resulta tan unívoco.

En cada símbolo, la barra siguiente aparece a la llegada de sus propios ticks, y estos suelen tener tiempos de llegada diferentes. En este caso, el desarrollador del indicador debe determinar cómo actuar: si esperar a la aparición de barras con la misma hora en todos los símbolos o recalcular el indicador en las últimas barras varias veces tras la aparición de una nueva barra en cualquiera de los símbolos.

En esta sección introduciremos una clase simple *MultiSymbolMonitor* (véase el archivo *MultiSymbolMonitor.mqh*) para seguir la formación de nuevas barras según una lista dada de símbolos.

El marco temporal requerido puede pasarse al constructor de la clase. Por defecto, sigue el marco temporal del gráfico actual, en el que se está ejecutando el programa.

```
class MultiSymbolMonitor
{
protected:
   ENUM_TIMEFRAMES period;

public:
   MultiSymbolMonitor(): period(_Period) {}
   MultiSymbolMonitor(const ENUM_TIMEFRAMES p): period(p) {}
   ...
```

Para almacenar la lista de símbolos rastreados utilizaremos una clase auxiliar *MapArray* de la sección anterior. En este array escribiremos los pares [nombre del símbolo;marca de tiempo de la última barra], es decir, los tipos de plantilla *<string,datetime>*. El método *attach* rellena el array.

```
protected:
   MapArray<string,datetime> lastTime;
...
public:
   void attach(const string symbol)
   {
      lastTime.put(symbol, NULL);
   }
```

Para un array dado, la clase puede actualizar y comprobar las marcas de tiempo en el método *check* llamando a la función *iTime* en un bucle sobre símbolos.

```
   ulong check(const bool refresh = false)
   {
      ulong flags = 0;
      for(int i = 0; i < lastTime.getSize(); i++)
      {
         const string symbol = lastTime.getKey(i);
         const datetime dt = iTime(symbol, period, 0);

         if(dt != lastTime[symbol]) // are there any changes?
         {
            flags |= 1 << i;
         }

         if(refresh) // update timestamp
         {
            lastTime.put(symbol, dt);
         }
      }
      return flags;
   }
```

El código de llamada debe llamar a *check* a su propia discreción, que suele ser a la llegada de ticks, o en un temporizador. Estrictamente hablando, ambas opciones no proporcionan una reacción instantánea a la aparición de ticks (y nuevas barras) en otros instrumentos, ya que el evento *OnCalculate* sólo aparece en los ticks del símbolo de trabajo del gráfico, y si entre ellos hubiera un tick de algún otro símbolo, no lo sabríamos hasta el siguiente tick «propio».

Abordaremos el seguimiento en tiempo real de los ticks de varios instrumentos en el capítulo sobre eventos gráficos interactivos (véase el indicador espía *EventTickSpy.mq5* en la sección Generación de eventos personalizados).

Por ahora, comprobaremos las barras con la precisión disponible. Así pues, procedamos con el método *check*.

Cada punto en el tiempo se caracteriza por su propio estado de las marcas de tiempo establecidas para todos los símbolos del array. Por ejemplo, una nueva barra puede formarse a las 12:00 sólo para el instrumento más líquido, y para varios otros instrumentos, los ticks aparecerán en unos pocos milisegundos o incluso segundos. Durante este intervalo, se actualizará un elemento del array y el resto serán antiguos. Luego, gradualmente, todos los símbolos tendrán barras de 12:00.

Para todos los símbolos cuya hora de apertura de la última barra no sea igual a la guardada, el método fija el bit con el número de símbolo, formando así una máscara de bits con cambios. La lista no debe contener más de 64 símbolos.

Si el valor de retorno es cero, no se ha registrado ningún cambio.

El parámetro *refresh* especifica si el método *check* sólo registrará los cambios (*false*), o si actualizará el estado según la situación actual del mercado (*true*).

El método *describe* permite obtener una lista de símbolos modificados por una máscara de bits.

```
string describe(ulong flags = 0)
{
   string message = "";
   if(flags == 0) flags = check();
   for(int i = 0; i < lastTime.getSize(); i++)
   {
      if((flags & (1 << i)) != 0)
      {
         message += lastTime.getKey(i) + "\t";
      }
   }
   return message;
}
```

A continuación, utilizaremos *inSync* para determinar si todos los símbolos del array tienen la misma hora de la última barra. Tiene sentido utilizarlo sólo para un conjunto de divisas con las mismas sesiones de trading.

```
bool inSync() const
{
   if(lastTime.getSize() == 0) return false;
   const datetime first = lastTime[0];
   for(int i = 1; i < lastTime.getSize(); i++)
   {
      if(first != lastTime[i]) return false;
   }
   return true;
}
```

Utilizando la clase descrita, implementamos un sencillo indicador multidivisa *IndMultiSymbolMonitor.mq5*, cuya única tarea será detectar nuevas barras para una lista de símbolos.

Dado que no se proporciona ningún dibujo para el indicador, el número de búferes y gráficos es 0.

```
#property indicator_chart_window
#property indicator_buffers 0
#property indicator_plots   0
```

La lista de instrumentos se especifica en la variable de entrada correspondiente y, a continuación, se convierte en un array registrado en el objeto *monitor*.

```
input string Instruments = "EURUSD,GBPUSD,USDCHF,USDJPY,AUDUSD,USDCAD,NZDUSD";

#include <MQL5Book/MultiSymbolMonitor.mqh>

MultiSymbolMonitor monitor;

void OnInit()
{
   string symbols[];
   const int n = StringSplit(Instruments, ',', symbols);
   for(int i = 0; i < n; ++i)
   {
      monitor.attach(symbols[i]);
   }
}
```

El manejador *OnCalculate* llama al monitor cuando se producen ticks y envía los cambios de estado al registro.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
   const ulong changes = monitor.check(true);
   if(changes != 0)
   {
      Print("New bar(s) on: ", monitor.describe(changes),
         ", in-sync:", monitor.inSync());
   }
   return rates_total;
}
```

Para comprobar este indicador tendríamos que pasar mucho tiempo en línea en el terminal. Sin embargo, MetaTrader 5 le permite hacer esto mucho más fácilmente: con la ayuda de un probador. Lo haremos en la próxima sección.

5.4.18 Comprobación de indicadores

El probador incorporado de MetaTrader 5 admite dos tipos de programas MQL: Asesores Expertos e indicadores. Los indicadores se comprueban siempre en la ventana visual, pero esto sólo se aplica a la comprobación de un indicador aislado. Si el indicador se crea y se llama desde un Asesor Experto mediante programación, entonces este Asesor Experto junto con el indicador o indicadores puede probarse sin visualización, a discreción del usuario. Estudiaremos la tecnología del uso de indicadores desde el código MQL en el próximo capítulo. La misma tecnología se utilizará para la integración con los Asesores Expertos.

Al mismo tiempo, el desarrollador del indicador debe prestar atención al hecho de que, sin visualización, el probador utiliza un método de cálculo acelerado para los indicadores que se llaman desde Asesores Expertos. Los datos no se calculan en cada tick, sino sólo cuando se solicitan los datos pertinentes desde los búferes de indicadores (véase la función *CopyBuffer*).

Si el indicador aún no se ha calculado en el tick actual, se calcula una vez en el primer acceso a sus datos. Si se generan otras solicitudes durante el mismo tick, los datos calculados se devuelven en el formulario preparado. Si no se leen los búferes del indicador en el tick actual, este no se calcula. El cálculo de indicadores a la carta proporciona un importante impulso a las pruebas y la optimización.

Si un determinado indicador requiere cálculos precisos y no puede saltarse ticks, MQL5 puede dar instrucciones al probador para que habilite el recálculo del indicador en cada tick. Esto se hace con la siguiente directiva:

```
#property tester_everytick_calculate
```

La palabra *everytick* de la directiva se refiere específicamente al cálculo del indicador y no afecta al modo de generación de ticks. En otras palabras: los ticks significan cambios de precios generados por el probador, ya sea para cada tick, para precios OHLC M1, o para aperturas de barras, y esta configuración del probador permanece en efecto.

Para los indicadores que hemos considerado en este capítulo, esta propiedad no es crítica. También debe tenerse en cuenta que sólo se aplica a las operaciones en el probador de estrategias. En el terminal, los indicadores siempre reciben eventos de *OnCalculate* en cada tick entrante (lo que ofrece la posibilidad de saltarse ticks si sus cálculos en *OnCalculate* tardan demasiado y no se completan antes de que llegue un nuevo tick).

En cuanto al comprobador, los indicadores se calculan en cada tick en cualquiera de las siguientes condiciones:

- En modo visual;
- Si existe la directiva *tester_everytick_calculate*;
- Si tienen la llamada *EventChartCustom* o las funciones *OnChartEvent* u *OnTimer*.

Vamos a intentar probar el indicador *IndMultiSymbolMonitor.mq5* de la sección anterior.

Seleccionamos el símbolo principal y el marco temporal de EURUSD, gráfico H1. El método de generación de ticks está «basado en ticks reales».

Tras iniciar la prueba, deberíamos ver las siguientes entradas en el registro de la ventana de modo visual:

```
2021.10.20 00:00:00 New bar(s) on: EURUSD USDCHF USDJPY , in-sync:false
2021.10.20 00:00:00 New bar(s) on: AUDUSD , in-sync:false
2021.10.20 00:00:00 New bar(s) on: GBPUSD , in-sync:false
2021.10.20 00:00:02 New bar(s) on: USDCAD , in-sync:false
2021.10.20 00:00:11 New bar(s) on: NZDUSD , in-sync:true
2021.10.20 01:00:04 New bar(s) on: EURUSD GBPUSD USDCHF USDJPY AUDUSD USDCAD NZDUSD ,
2021.10.20 02:00:00 New bar(s) on: EURUSD USDJPY NZDUSD , in-sync:false
2021.10.20 02:00:00 New bar(s) on: USDCHF , in-sync:false
2021.10.20 02:00:01 New bar(s) on: AUDUSD , in-sync:false
2021.10.20 02:00:15 New bar(s) on: GBPUSD USDCAD , in-sync:true
2021.10.20 03:00:00 New bar(s) on: EURUSD AUDUSD NZDUSD , in-sync:false
2021.10.20 03:00:00 New bar(s) on: GBPUSD USDJPY USDCAD , in-sync:false
2021.10.20 03:00:12 New bar(s) on: USDCHF , in-sync:true
```

Como puede ver, aparecen nuevas barras en diferentes símbolos de forma gradual. Normalmente, se producen varios eventos antes de que aparezca la bandera «in-sync» en *true*.

También puede realizar pruebas para otros indicadores de este capítulo. Tenga en cuenta que, si un programa MQL consulta el historial de ticks, seleccione el método de generación «basado en ticks reales» en el comprobador.

La prueba «por precios de apertura» sólo se puede utilizar para los indicadores y Asesores Expertos que se desarrollan con ayuda para este modo; por ejemplo, calculan sólo por *Open* precios o analizan barras completadas a partir de la 1ª.

> ¡Atención! Al probar indicadores en el probador, el evento *OnDeinit* no funciona. Además, no se realizan otras finalizaciones; por ejemplo, no se llama a los destructores de objetos globales.

5.4.19 Limitaciones y ventajas de los indicadores

Todas las funciones especializadas que se tratan en este capítulo sólo están disponibles en los códigos fuente de los indicadores. No tiene sentido utilizarlos en otros tipos de programas MQL: devolverán un error.

Hay otras funciones que están prohibidas en los indicadores:

- *OrderCalcMargin*
- *OrderCalcProfit*
- *OrderCheck*
- *OrderSend*
- *SendFTP*
- *WebRequest*
- *Socket****
- *Sleep*
- *MessageBox*
- *ExpertRemove*

Algunos de ellos (con el prefijo *Order*-) se refieren a cálculos de negociación y sólo están permitidos en Asesores Expertos y scripts. Otros están pensados para ejecutar peticiones que bloquean la ejecución del hilo hasta que se devuelve el resultado, mientras que esto no está permitido para los indicadores porque se ejecutan en el hilo de interfaz del terminal. Por una razón similar, las funciones *Sleep* y *MessageBox* están prohibidas.

Los indicadores se encargan principalmente de visualizar los datos y, curiosamente, no sirven para realizar cálculos masivos. En concreto, si decide crear un indicador que entrene una red neuronal o un árbol de decisión en el proceso, lo más probable es que esto afecte negativamente al funcionamiento normal del terminal.

El efecto de un cálculo largo se demuestra con el indicador *IndBarIndex.mq5*, que en el modo normal está diseñado para mostrar números de barra en los elementos de su búfer. No obstante, utilizando el parámetro de entrada *SimulateCalculation*, que debe ajustarse a *true*, puede iniciar un bucle infinito en un temporizador.

```
// Setting to true will freeze the drawing of indicators
// on charts of the same working symbol
// Attention! Don't forget to remove the indicator after the experiment!
input bool SimulateCalculation = false;

void OnInit()
{
   ...
   if(SimulateCalculation)
   {
      EventSetTimer(1);
   }
}
...
void OnTimer()
{
   Comment("Calculation started at ", TimeLocal());
   while(!IsStopped())
   {
   // infinite loop to emulate calculations
   }
   Comment("");
}
```

En este modo, el indicador, como era de esperar, empieza a ocupar por completo 1 núcleo del procesador, pero también aparece otro efecto secundario: cualquier indicador en el mismo símbolo donde se coloca *IndBarIndex* deja de actualizarse. Por ejemplo, podemos ejecutar *IndBarIndex* en EURUSD (cualquier marco temporal), y luego en cualquier otro gráfico EURUSD, puede intentar aplicar una media móvil regular: no se mostrará hasta que elimine *IndBarIndex* del primer gráfico.

A este respecto, todos los cálculos largos deben colocarse en hilos separados, es decir, scripts o Asesores Expertos no comerciales, y sólo sus resultados deben ser utilizados en los indicadores. La API de MQL5 permite crear nuevos gráficos u objetos con gráficos en los que es posible aplicar plantillas tpl con el Asesor Experto o script requerido.

5.4.20 Crear un borrador de indicador en el Asistente MQL

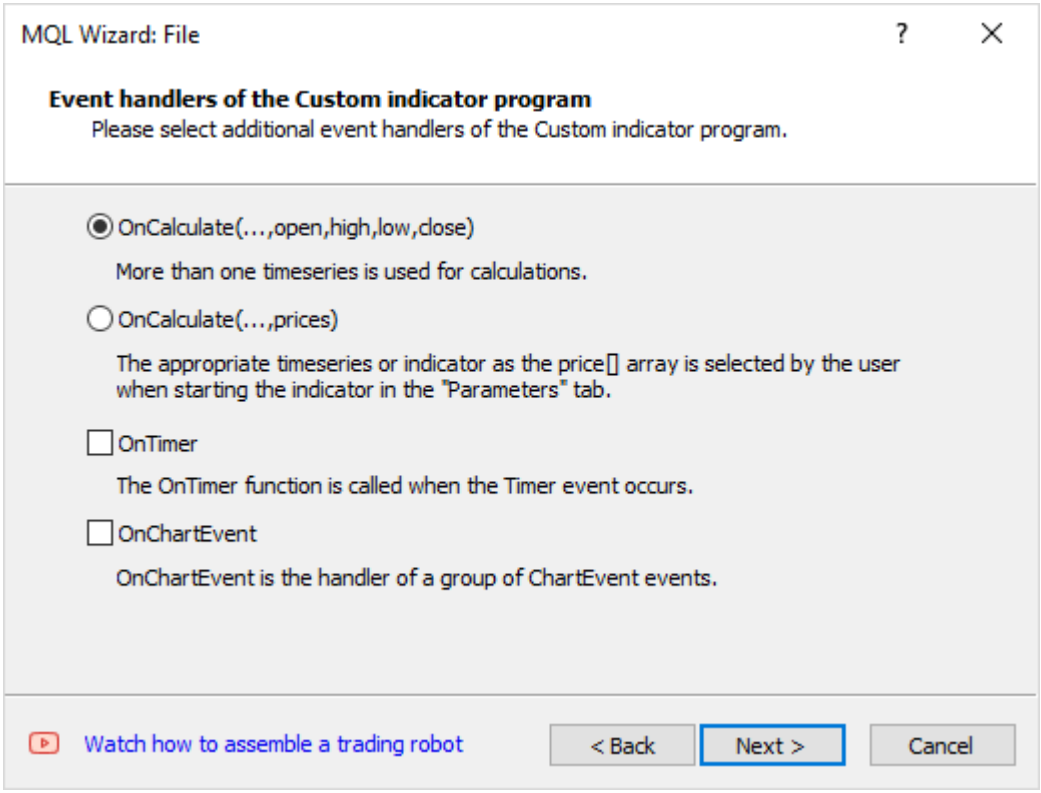
Así pues, hemos considerado la estructura interna de los indicadores y podemos entender cómo determinadas construcciones sintácticas del código fuente afectan a la representación externa y al cálculo del indicador. Con este nivel de formación, puede empezar a trabajar con el código de otra persona y modificarlo para adaptarlo a sus necesidades. O puede intentar crear algo por su cuenta. Para no empezar desde cero, puede utilizar el Asistente MQL. En concreto, también puede utilizarse para crear un borrador de indicador.

Para iniciar el Asistente, abra el menú contextual en MetaEditor *Navegador* para la rama *Indicators* y ejecute el comando *Nuevo archivo* (Ctrl + N). En la primera parte del libro, en la sección Asistente MQL y borrador del programa hemos creado el primer script utilizando el Asistente y hemos visto cómo es este paso.

En este caso (cuando se inicia desde el menú contextual), el primer paso del Asistente seleccionará automáticamente el elemento *Custom indicator*.

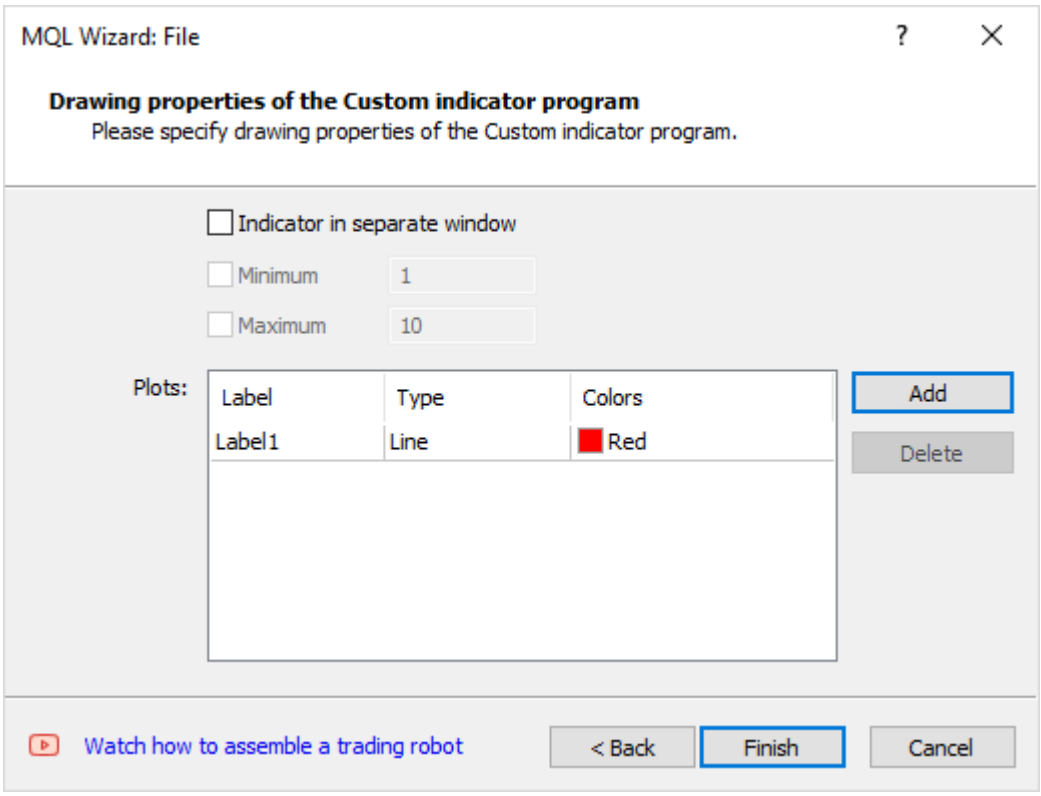
Haga clic en *Siguiente* para ir al segundo paso, donde debe especificar el nombre del archivo. Aquí puede *Add* parámetros de entrada del indicador. Este paso no difiere de lo que ha ocurrido con los scripts.

En el tercer paso, el Asistente ofrece elegir uno de los formularios manejadores *OnCalculate* y otros manejadores de eventos opcionales.



Asistente MQL: selección de manejadores de eventos al crear un indicador

El último paso le permite definir la parte del gráfico en la que se mostrarán las líneas: puede ser la ventana principal (por defecto) o una subventana independiente debajo del gráfico (si activa la bandera *Indicator in a separate window*).



Asistente MQL: selección de ventanas y lista de gráficos al crear un indicador

Con el botón *Añadir* puede listar varias construcciones gráficas y establecer sus propiedades básicas.

Todos estos términos ya nos son familiares «desde dentro», y puede elegir una u otra opción conscientemente.

Intente generar varias versiones de indicadores con diferentes opciones activadas y evalúe su impacto en el texto resultante del programa.

Por supuesto, al haber recibido un borrador del código fuente, el desarrollador es libre de realizar cambios arbitrarios, modificando cualquiera de los aspectos establecidos en el Asistente. Esto es tanto más relevante cuanto que la gama de ajustes del Asistente es mínima. En particular, la lista de tipos de parámetros de entrada se limita a los tipos estándar MQL5, no hay niveles, paletas de colores, etc. En cuanto a los manejadores de eventos adicionales, el Asistente sólo ofrece *OnTimer* y *OnChartEvent* dejando entre bastidores *OnBookEvent* y *OnDeinit*. Pero de acuerdo con el material de este capítulo, puede ir completando el borrador con todo lo que necesite.

5.5 Utilización de indicadores preconfeccionados de programas MQL

En el capítulo anterior aprendimos a desarrollar indicadores personalizados. Los usuarios pueden colocarlos en gráficos y realizar con ellos análisis técnicos manuales. Pero ésta no es la única forma de utilizar los indicadores: MQL5 le permite crear instancias de indicadores y solicitar sus datos calculados mediante programación. Esto puede hacerse tanto a partir de otros indicadores, combinando varios sencillos en otros más complejos, como a partir de Asesores Expertos que implementan el trading automático o semiautomático en señales de indicadores.

Basta con conocer los parámetros del indicador, así como la ubicación y el significado de los datos calculados en sus búferes públicos, a fin de organizar la construcción de estas series temporales recién aplicadas y acceder a ellas.

En este capítulo estudiaremos las funciones para la creación y eliminación de indicadores, así como la lectura de sus búferes. Esto se aplica no sólo a los indicadores personalizados escritos en MQL5, sino también a un amplio conjunto de indicadores integrados.

Los principios generales de la interacción programática con los indicadores incluyen varios pasos:

- Creación del indicador **descriptor**, que es un número de identificación único emitido por el sistema en respuesta a una determinada llamada a una función (*iCustom* o *IndicatorCreate*) y a través del cual el código MQL informa del nombre y los parámetros del indicador requerido.
- Lectura de datos de los búferes de indicadores especificados por el descriptor mediante la función *CopyBuffer*.
- Liberación del manejador (*IndicatorRelease*) si el indicador ya no es necesario.

La creación y liberación del descriptor suelen realizarse durante la inicialización y desinicialización del programa, respectivamente, y los búferes se leen y analizan repetidamente, según sea necesario; por ejemplo, cuando llegan los ticks.

En todos los casos, excepto en los exóticos, cuando se requiera cambiar dinámicamente la configuración de los indicadores durante la ejecución del programa, se recomienda obtener descriptores de indicadores una vez en *OnInit* o en el constructor de la clase de objeto global.

Todas las funciones de creación de indicadores tienen al menos 2 parámetros: símbolo y marco temporal. En lugar de un símbolo, puede pasar NULL, que significa el instrumento actual. Además, el valor 0 corresponde al marco temporal actual. Opcionalmente, puede utilizar las variables integradas *_Symbol* y *_Period*. Si es necesario, puede establecer un símbolo y un marco temporal arbitrarios que no estén relacionados con el gráfico. Así, en particular, es posible aplicar indicadores multiactivos y de marco temporal múltiple.

No se puede acceder a los datos del indicador inmediatamente después de crear su instancia porque el cálculo de los búferes lleva cierto tiempo. Antes de leer los datos, debe comprobar su disponibilidad mediante la función *BarsCalculated* (también recibe un argumento descriptor y devuelve el número de barras calculadas). De lo contrario, se recibirá un error en lugar de datos. Aunque no es crítico, ya que no provoca que el programa se detenga y se descargue, la ausencia de datos hará que el programa sea inútil.

Más adelante en este capítulo, por brevedad, nos referiremos a la creación de instancias de indicadores y a la obtención de sus descriptores simplemente como «creación de indicadores». Debe distinguirse del término similar «creación de indicadores personalizados», con el que nos referíamos a escribir el código fuente de los indicadores en el capítulo anterior.

5.5.1 Manejadores y contadores de propietarios de indicadores

El trabajo programático con indicadores requiere operar con manejadores. Se puede establecer aquí un paralelismo con los descriptores de archivo (véase la sección Abrir y cerrar archivos): allí utilizamos la función *File Open* para informar al sistema sobre el nombre del archivo y los modos de apertura, tras lo cual el descriptor sirvió de «pase» para todas las demás funciones de archivo.

El sistema de descriptores de indicadores tiene varias finalidades.

Permite indicar de antemano al terminal qué indicador debe lanzar y qué series temporales debe calcular. Dado que se requiere cierto tiempo para descargar los datos históricos iniciales y calcular el indicador (al menos durante la solicitud inicial), junto con la asignación de recursos (memoria, gráficos), los puntos en los que se crea el indicador y en los que está listo son diferentes. El descriptor es un vínculo entre ellos. Se trata de una especie de vínculo al objeto terminal interno que almacena el conjunto de propiedades que establecimos al crear el indicador y su estado actual.

Por supuesto, para trabajar con descriptores, el terminal necesita mantener una tabla determinada de todos los indicadores solicitados y sus propiedades. Sin embargo, el terminal no proporciona información sobre el número real en la tabla general: en lugar de ello, cada programa forma su propia lista privada de indicadores que se le han solicitado. Las entradas de esta lista se refieren a los elementos de la tabla general, y el descriptor es sólo un número en la lista.

Por lo tanto, puede haber indicadores completamente distintos detrás de los mismos descriptores en programas diferentes. Así, no tiene sentido transferir los valores de los descriptores entre programas.

Los descriptores forman parte del sistema de gestión de recursos de los terminales, ya que excluyen la duplicación de instancias de indicador con las mismas características, siempre que sea posible. En otras palabras: todos los indicadores integrados y personalizados creados mediante programación, manualmente o a partir de plantillas tpl se almacenan en caché.

Antes de crear una nueva instancia de indicador, el terminal comprueba si existe un indicador idéntico entre los de la caché. Los siguientes criterios se aplican a la hora de buscar una copia:

- Carácter y periodo que coincidan
- Parámetros que coincidan

Para los indicadores personalizados, además debe coincidir lo siguiente:

- La ruta en disco (como cadena, sin normalización a forma absoluta)
- El gráfico en el que se está ejecutando el indicador (cuando se crea un indicador a partir de un programa MQL, el indicador que se está creando hereda el gráfico del programa que lo crea).

Los indicadores integrados se almacenan en caché por símbolo y, por lo tanto, sus instancias pueden asignarse para su uso por separado en distintos gráficos (con el mismo símbolo/marco temporal).

Tenga en cuenta que no se pueden crear manualmente dos indicadores idénticos en el mismo gráfico. Distintas instancias del programa pueden solicitar el mismo indicador, en cuyo caso sólo se creará una copia del mismo y se proporcionará a ambos programas.

Para cada combinación única de condiciones, el terminal mantiene un contador: tras la primera solicitud de creación de un indicador específico, su contador es igual a 1, y en las siguientes, aumenta en 1 (no se crea una copia del indicador). Cuando se libera un indicador, su contador disminuye en 1. El indicador sólo se descarga cuando se pone a cero el contador, es decir, cuando todos sus propietarios se niegan explícitamente a utilizarlo.

Debe tenerse en cuenta que múltiples llamadas a la función de construcción de indicadores con los mismos parámetros (incluyendo símbolo/marco temporal) dentro del mismo programa MQL no conducen a múltiples incrementos en el contador de referencia: el contador se incrementará sólo una vez. En consecuencia, para cada valor del manejador, basta con una llamada a la función de liberación (*IndicatorRelease*). Todas las demás llamadas son superfluas y devuelven un error porque no tienen nada que liberar.

Además de crear indicadores utilizando *iCustom* e *IndicatorCreate* en MQL5, es posible obtener el control de un indicador de terceros (ya existente). Esto puede hacerse utilizando la función

ChartIndicatorGet, que estudiaremos en el capítulo sobre gráficos. Es importante señalar aquí que la adquisición de un manejador de esta manera también aumentará su recuento de referencias e impedirá la descarga a menos que el manejador se libere a continuación.

Si el programa creó indicadores subordinados, sus manejadores serán liberados automáticamente (el contador disminuye en 1) cuando este programa se descargue, incluso si no se llama a la función *IndicatorRelease*.

5.5.2 Una forma sencilla de crear instancias de indicadores: iCustom

MQL5 proporciona dos funciones para crear instancias de indicadores a partir de programas: *iCustom* y *IndicatorCreate*. La primera función consiste en pasar una lista de parámetros, que deben conocerse en el momento de compilar el programa. La segunda permite formar dinámicamente un array con los parámetros del indicador llamado durante la ejecución del programa. Este modo avanzado se abordará en la sección Forma avanzada de crear indicadores:*IndicatorCreate*.

int iCustom(const string symbol, ENUM_TIMEFRAMES timeframe, const string pathname, ...)

La función crea un indicador para el símbolo y el marco temporal especificados. NULL en el parámetro *symbol* puede utilizarse para indicar el símbolo del gráfico actual, mientras que 0 en el parámetro *timeframe* establece el periodo actual.

En el parámetro *pathname*, especifique el nombre del indicador (el nombre del archivo ex5 sin extensión) y, opcionalmente, la ruta. A continuación se ofrecen más detalles sobre la ruta.

El indicador referenciado por *pathname* debe compilarse.

La función devuelve un manejador de indicador o INVALID_HANDLE en caso de error. El manejador será necesario para llamar a otras funciones descritas en este capítulo e incluidas en el grupo de control del programa indicador. El manejador es un número entero que describe de forma única la instancia del indicador creado dentro del programa de llamada.

La elipsis en el prototipo de función *iCustom* indica una lista de parámetros reales para el indicador. Sus tipos y orden deben corresponder a los parámetros formales (en el código del indicador). No obstante, se permite omitir valores a partir del final de la lista de parámetros. Para tales parámetros no especificados en el código de llamada, el indicador creado utilizará los valores por defecto de los correspondientes *input*s.

Por ejemplo, si el indicador toma dos variables de entrada: periodo (*input int WorkPeriod = 14*) y tipo de precio (*input ENUM_APPLIED_PRICE WorkPrice = PRICE_CLOSE*), entonces puede llamar a *iCustom* con distintos grados de detalle:

- iCustom(_Symbol, _Period, 21, PRICE_TYPICAL): establecer valores para toda la lista de parámetros.
- iCustom(_Symbol, _Period, 21): al establecer el primer parámetro, el segundo se omite y recibirá el valor PRICE_CLOSE.
- iCustom(_Symbol, _Period): se omiten ambos parámetros y se obtendrán los valores 14 y PRICE_CLOSE.

No puede omitir un parámetro al principio o en medio de la lista de parámetros.

Si el indicador que se está creando tiene una forma abreviada de *OnCalculate*, entonces el último parámetro adicional (además de la lista de variables de entrada descritas dentro del indicador) puede ser el tipo de precio utilizado para construir el indicador. Es como una lista desplegable *Apply to* en el

cuadro de diálogo de propiedades del indicador. Además, en este parámetro adicional, puede pasar un manejador a otro indicador creado previamente (véase un ejemplo más abajo). En este caso, el indicador recién creado se calculará utilizando el primer búfer de indicador con el manejador especificado. En otras palabras: el programador puede establecer el cálculo de un indicador a partir de otro.

> MQL5 no proporciona medios programáticos para averiguar si un indicador de terceros específico se implementa utilizando la forma corta o la forma larga de *OnCalculate*, es decir, si se permite pasar un manejador adicional al crearlo a través de *iCustom*. Además, MQL5 no permite seleccionar el número de búfer si el indicador identificado por el manejador adicional tiene varios búferes.

Volvamos al parámetro *pathname*.

Una ruta es una cadena que contiene al menos una barra invertida ('\') o una barra diagonal ('/'), que es un carácter especial utilizado en el sistema de archivos como separador en la jerarquía de carpetas y archivos. Puede utilizar una barra diagonal o una barra invertida, pero esta última requiere «escape», lo que significa que debe escribirse dos veces. Esto se debe a que la barra invertida es un carácter de control que forma muchos códigos de servicio, como tabulación ('\t'), nueva línea ('\n'), etc. (véase la sección Tipos de caracteres).

Si la ruta comienza con una barra, se llama absoluta, y su carpeta raíz es el directorio de todos los códigos fuente MQL5. Por ejemplo, si se especifica la cadena «/MiIndicador» en el parámetro *pathname* se buscará el archivo *MQL5/MyIndicator.ex5*, y la ruta más larga con el directorio «/Ejercicio/MiIndicador» hará referencia a *MQL5/Exercise/MyIndicator.ex5*.

Si el parámetro *pathname* contiene una o varias barras pero no empieza por una, la ruta se denomina relativa porque entonces se considera relativa a una de las dos ubicaciones predefinidas. En primer lugar, el archivo del indicador se busca en relación con la carpeta donde se encuentra el programa MQL de llamada. Si no se encuentra allí, la búsqueda continúa dentro de la carpeta común de indicadores *MQL5/Indicators*.

En una línea con barras, el fragmento situado a la derecha de la barra más a la derecha se trata como el nombre del archivo, y todos los anteriores describen la jerarquía de carpetas. Por ejemplo, la ruta «Carpeta/Subcarpeta/NombreDeArchivo» coincide con dos subcarpetas: *SubFolder* dentro de *Folder*, y el archivo *Filename* dentro de *SubFolder*.

El caso más sencillo es cuando *pathname* no contiene barras. De este modo, sólo se especifica el nombre del archivo. También se considera en el contexto de los dos puntos de partida de la búsqueda mencionados anteriormente.

Por ejemplo, el Asesor Experto *MyExpert.ex5* se encuentra en la carpeta *MQL5/Experts/Examples*, y contiene la llamada de *iCustom(_Symbol,_Period, "MyIndicator")*. Aquí la ruta relativa está degenerada (vacía) y sólo aparece el nombre del archivo. Así, la búsqueda del indicador parte de la carpeta *MQL5/Experts/Examples/* y el nombre *MyIndicator*, lo que da *MQL5/Experts/Examples/MyIndicator.ex5*. Si no se encuentra un indicador de este tipo en este directorio, la búsqueda continuará en la carpeta raíz de los indicadores, es decir, por la ruta conectada y el nombre *MQL5/Indicators/MyIndicator.ex5*.

Si el indicador no se encuentra en ambos lugares, la función devolverá INVALID_HANDLE y establecerá el código de error 4802 (ERR_INDICATOR_CANNOT_CREATE) en *_LastError*.

Un caso más difícil es si *pathname* contiene no sólo el nombre, sino también el directorio, por ejemplo «SeñalesTrading/MiIndicador». La ruta especificada se añade a la carpeta del programa de llamada, lo que da como resultado el siguiente objetivo de búsqueda:*MQL5/Experts/Examples/TradeSignals/MyIndicator.ex5*. A continuación, en caso de fallo, se

añade la misma ruta a *MQL5/Indicators*, es decir, se busca en el archivo *MQL5/Indicators/TradeSignals/MyIndicator.ex5*. Tenga en cuenta que si utiliza una barra invertida como separador, no debe olvidar escribirla dos veces, por ejemplo, *iCustom(_Symbol, _Period, "TradeSignals\\MyIndicator")*.

Para liberar la memoria del ordenador de un indicador que ya no se utiliza, utilice la función *IndicatorRelease* pasándole el manejador de este indicador.

Debe prestarse especial atención a la comprobación de un programa que utilice indicadores. Si el parámetro *pathname* de la llamada a *iCustom* se especifica como una cadena constante, el compilador detecta automáticamente el indicador requerido correspondiente y lo pasa al comprobador junto con el programa que se está probando. De lo contrario, si el parámetro se calcula en una expresión o se obtiene del exterior (por ejemplo, a través de *input* desde el usuario), deberá especificar la propiedad en el código fuente *#property tester_indicator*:

```
#property tester_indicator "indicator_name.ex5"
```

Esto significa que sólo los indicadores personalizados previamente conocidos pueden probarse en los programas.

Consideremos el ejemplo de un nuevo indicador *UseWPR1.mq5*, que, dentro de su manejador *OnInit*, estará creando un manejador del indicador *IndWPR* que discutimos en el capítulo anterior (no olvide compilar *IndWPR* porque *iCustom* descarga archivos ex5). El manejador recibido en *UseWPR1* no se utiliza de ninguna manera todavía, ya que sólo estudiaremos la posibilidad en sí y comprobaremos la indicación de éxito. Por lo tanto, no necesitamos búferes en el nuevo indicador.

```
#property indicator_separate_window
#property indicator_buffers 0
#property indicator_plots   0
```

El indicador creará una subventana vacía pero aún no mostrará nada en ella. Este es un comportamiento normal.

Vamos a comprobar varias opciones para obtener un descriptor, con distintos valores de *pathname*:

1. Una ruta absoluta que comienza con una barra y por lo tanto incluye toda la jerarquía de carpetas (a partir de MQL5) con ejemplos de indicadores del capítulo 5, es decir, «/Indicadores/MQL5Book/p5/IndWPR»
2. Sólo el nombre «IndWPR» para buscar en la misma carpeta donde se encuentra el indicador de llamada *UseWPR1.mq5* (ambos indicadores se proporcionan en la misma carpeta).
3. Ruta con jerarquía de carpetas de ejemplos de indicadores relativa al directorio estándar *MQL5/Indicators*, es decir, «MQL5Book/p5/IndWPR» (tenga en cuenta que no hay barra al principio).
4. Sólo el nombre como en el punto 2 pero para el indicador inexistente «IndWPR NonExistent».
5. Ruta absoluta como en el punto 1 pero con barras invertidas sin evitarlas, es decir, «\Indicadores\MQL5Book\p5\IndWPR»
6. Copia íntegra del punto 2.

```
int OnInit()
{
   int handle1 = PRTF(iCustom(_Symbol, _Period, "/Indicators/MQL5Book/p5/IndWPR"));
   int handle2 = PRTF(iCustom(_Symbol, _Period, "IndWPR"));
   int handle3 = PRTF(iCustom(_Symbol, _Period, "MQL5Book/p5/IndWPR"));
   int handle4 = PRTF(iCustom(_Symbol, _Period, "IndWPR NonExistent"));
   int handle5 = PRTF(iCustom(_Symbol, _Period, "\Indicators\MQL5Book\p5\IndWPR"));
   int handle6 = PRTF(iCustom(_Symbol, _Period, "IndWPR"));
   return INIT_SUCCEEDED;
}
```

Dado que las variables del manejador no se utilizan, se declaran locales. Vamos a explicar de forma específica que, aunque las variables locales *handle* se borran al salir de *OnInit*, esto no afecta a los manejadores, que siguen existiendo mientras se ejecute el indicador «padre» *UseWPR*. Simplemente perdemos los valores de estos manejadores en nuestro código, lo que sin embargo no es ningún problema, porque no se utilizan en ninguna parte aquí. En los ejemplos de indicadores reales que consideraremos más adelante, los manejadores, por supuesto, se almacenan (normalmente en variables globales) y se utilizan.

No se preocupe tampoco por las fugas de recursos: al eliminar el indicador *UseWPR* del gráfico, todos los manejadores creados por él serán borrados automáticamente por el terminal. Los principios y la necesidad de una liberación explícita de los manejadores se describirán con más detalle en la sección sobre borrar instancias de indicadores: utilizando *IndicatorRelease*.

El código *OnInit* anterior genera las siguientes entradas de registro:

```
iCustom(_Symbol,_Period,/Indicators/MQL5Book/p5/IndWPR)=10 / ok
iCustom(_Symbol,_Period,IndWPR)=11 / ok
iCustom(_Symbol,_Period,MQL5Book/p5/IndWPR)=12 / ok
cannot load custom indicator 'IndWPR NonExistent' [4802]
iCustom(_Symbol,_Period,IndWPR NonExistent)=-1 / INDICATOR_CANNOT_CREATE(4802)
iCustom(_Symbol,_Period,\Indicators\MQL5Book\p5\IndWPR)=13 / ok
iCustom(_Symbol,_Period,IndWPR)=11 / ok
```

Como podemos ver, los manejadores significativos 10, 11, 12 y 13 se reciben en todos los casos excepto en el 4º, con un indicador de llamada inexistente. El valor del manejador es -1 (INVALID_HANDLE).

Observe también que la 5ª línea genera varias advertencias de «secuencia de escape de caracteres no reconocida» al compilar. Esto es consecuencia del hecho de que no escapamos la barra invertida. Y también tuvimos suerte de que la instrucción se ejecutara correctamente, porque si el nombre de cualquier carpeta o archivo empezara por una de las letras de las secuencias de escape admitidas, la interpretación de la secuencia violaría la lectura esperada del nombre. Por ejemplo, si tuviéramos un indicador llamado «test» en la misma carpeta e intentáramos crearlo a través de la ruta «MQL5Book\p5\test», obtendríamos INVALID_HANDLE y el error 4802. Esto se debe a que '\t' es un carácter de tabulación, por lo que el terminal buscaría «MQL5Book\p5<nbsp> est». La entrada correcta debe ser «MQL5Book\p5\\test». Por lo tanto, es más fácil utilizar una barra diagonal.

También es importante tener en cuenta que, aunque todas las variaciones de éxito se refieren al mismo indicador *MQL5/Indicators/MQL5Book/p5/IndWPR.ex5*, y de hecho las rutas 1, 2, 3 y 5 son equivalentes, el terminal las trata como cadenas diferentes, que es la razón por la que obtenemos valores de descriptor distintos. Y sólo la opción 6, que duplica completamente la opción 2, devuelve un descriptor idéntico: 11.

¿Por qué la numeración de los manejadores empieza por 10? Los valores más pequeños se reservan para el sistema. Como se mencionó anteriormente, para los indicadores con una forma abreviada de *OnCalculate*, el último parámetro se puede utilizar para pasar el tipo de precio o un manejador de otro indicador, cuyo búfer se utilizará para calcular la nueva instancia creada. Dado que los elementos de la enumeración ENUM_APPLIED_PRICE tienen sus propios valores constantes, ocupan el área inferior a 10. Para más información, consulte Definir la fuente de datos de un indicador.

En el siguiente ejemplo de *UseWPR2.mq5* implementaremos un indicador que creará una instancia de *IndWPR* y comprobará el progreso de su cálculo utilizando el manejador. Pero para ello es necesario familiarizarse con la nueva función *BarsCalculated*.

5.5.3 Comprobación del número de barras calculadas: BarsCalculated

Cuando creamos un indicador de terceros llamando a *iCustom* o a otras funciones que veremos más adelante en este capítulo, se requiere algún tiempo para hacer el cálculo. Como sabemos, la principal medida de la disponibilidad de datos del indicador es el número de barras calculadas, que devuelve desde su función *OnCalculate*. Podemos averiguar este número dado que tenemos el manejador del indicador.

int BarsCalculated(int handle)

La función devuelve el número de barras cuyos datos se calculan en el indicador especificado por *handle*. En caso de error, obtenemos -1.

Mientras no se hayan calculado los datos, el resultado es 0. Posteriormente, este número debe compararse con el tamaño de la serie temporal (por ejemplo, con *rates_total* si el indicador que llama comprueba *BarsCalculated* en el contexto de su propia función *OnCalculate*) para analizar el procesamiento de nuevas barras por parte del indicador.

En el indicador *UseWPR2.mq5*, intentaremos crear *IndWPR* mientras cambiamos el periodo WPR en el argumento de entrada.

```
input int WPRPeriod = 0;
```

Su valor por defecto es 0, que es un valor no válido. Se propone de forma intencionada para demostrar una situación anormal. Recordemos que en el código fuente *IndWPR.mq5* hay comprobaciones en *OnInit* y en *OnCalculate*.

```
// IndWPR.mq5
void OnInit()
{
   if(WPRPeriod < 1)
   {
      Alert(StringFormat("Incorrect Period value (%d). Should be 1 or larger",
         WPRPeriod));
   }
   ...
}

int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   if(rates_total < WPRPeriod || WPRPeriod < 1) return 0;
   ...
}
```

Así, en el periodo cero, deberíamos recibir un mensaje de error, y *BarsCalculated* debería devolver siempre 0. Una vez que hemos introducido un valor positivo para el periodo, el indicador auxiliar debería empezar a calcular normalmente (y dada la facilidad de cálculo de WPR, casi inmediatamente), y *BarsCalculated* debería devolver el número total de barras.

Ahora vamos a presentar el código fuente para la creación de un manejador en *UseWPR2.mq5*.

```
// UseWPR2.mq5
int handle; // handle to global variable

int OnInit()
{
   // passing name and parameter
   handle = PRTF(iCustom(_Symbol, _Period, "IndWPR", WPRPeriod));
   // next check is useless here because you have to wait,
   // when the indicator is loaded, run and calculate
   // (here it is for demonstration purposes only)
   PRTF(BarsCalculated(handle));
   // successful initialization depends on the descriptor
   return handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}
```

En *OnCalculate* sólo registramos los valores *BarsCalculated* y *rates_total*.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   // wait until the slave indicator is calculated on all bars
   if(PRTF(BarsCalculated(handle)) != PRTF(rates_total))
   {
      return prev_calculated;
   }

   // ... here is usually further work using handle

   return rates_total;
}
```

Compile y ejecute *UseWPR2*, primero con el parámetro 0, y después con algún valor válido, por ejemplo, 21. Aquí están las entradas de registro para el período cero.

```
iCustom(_Symbol,_Period,IndWPR,WPRPeriod)=10 / ok
BarsCalculated(handle)=-1 / INDICATOR_DATA_NOT_FOUND(4806)
Alert: Incorrect Period value (0). Should be 1 or larger
BarsCalculated(handle)=0 / ok
rates_total=20000 / ok
...
```

Inmediatamente después de la creación del manejador, los datos aún no están disponibles, por lo que se muestra el error INDICATOR_DATA_NOT_FOUND(4806), y el resultado *BarsCalculated* es igual a -1. A continuación aparece una notificación sobre un parámetro de entrada incorrecto que confirma la carga y el inicio correctos del indicador *IndWPR*. En el siguiente segmento obtenemos el valor *BarsCalculated* igual a 0.

Para que se calcule el indicador, introduciremos el parámetro de entrada correcto. En este caso, *BarsCalculated* es igual a *rates_total*.

```
iCustom(_Symbol,_Period,IndWPR,WPRPeriod)=10 / ok
BarsCalculated(handle)=-1 / INDICATOR_DATA_NOT_FOUND(4806)
BarsCalculated(handle)=20000 / ok
rates_total=20000 / ok
...
```

Después de haber dominado la comprobación de la disponibilidad de un indicador esclavo, podemos empezar a leer sus datos. Hagámoslo en el siguiente ejemplo *UseWPR3.mq5*, donde nos familiarizaremos con la función *CopyBuffer*.

5.5.4 Obtención de datos de series temporales a partir de un indicador: CopyBuffer

Un programa MQL puede leer datos de los búferes públicos del indicador por su manejador. Recordemos que en los indicadores personalizados, dichos búferes son arrays especificados en el código fuente en llamadas a la función *SetIndexBuffer*.

La API de MQL5 proporciona la función *CopyBuffer* para la lectura de búferes; la función tiene 3 formas.

int CopyBuffer(int handle, int buffer, int offset, int count, double &array[])
int CopyBuffer(int handle, int buffer, datetime start, int count, double &array[])
int CopyBuffer(int handle, int buffer, datetime start, datetime stop, double &array[])

El parámetro *handle* especifica el manejador recibido de la llamada *iCustom* u otras funciones (para obtener más detalles, consulte las secciones sobre *IndicatorCreate* e indicadores integrados). El parámetro *buffer* establece el índice del búfer del indicador del que se van a solicitar los datos. La numeración se realiza empezando por 0.

Los elementos recibidos de la serie temporal solicitada se introducen en el conjunto *array* por referencia.

Las tres variantes de la función difieren en la forma de especificar el rango de marcas de tiempo (*start/stop*) o números (*offset*) y la cantidad (*count*) de barras para los que se obtienen los datos. Los fundamentos del trabajo con estos parámetros coinciden plenamente con lo que estudiamos en Visión general de las funciones Copy para obtener arrays de cotizaciones. En particular, los elementos de datos copiados en *offset* y *count* se cuentan del presente al pasado, es decir, la posición inicial igual a 0 significa la barra actual. Los elementos del *array* receptor se ordenan físicamente del pasado al presente (sin embargo, este direccionamiento puede invertirse a nivel lógico utilizando *ArraySetAsSeries*).

CopyBuffer es un análogo de funciones para leer series temporales integradas del tipo *Copy Open*, *CopyClose*, entre otros. La principal diferencia es que las series temporales con cotizaciones son generadas por el propio terminal, mientras que las series temporales de los búferes de los indicadores son calculadas por indicadores personalizados o integrados. Además, en el caso de los indicadores, establecemos un par específico de símbolo y marco temporal que definen e identifican una serie temporal de antemano, en la función de creación de manejadores como *iCustom*, y en *CopyBuffer* esta información se transmite indirectamente a través de *handle*.

Cuando se copia una cantidad desconocida de datos como array de destino, es conveniente utilizar un array dinámico. En este caso, la función *CopyBuffer* distribuirá el tamaño del array receptor en función del tamaño de los datos copiados. Si es necesario copiar repetidamente una cantidad conocida de datos, es mejor hacerlo en un búfer asignado estáticamente (local con el modificador de *static* o de tamaño fijo en el contexto global) para evitar repetidas asignaciones de memoria.

Si el array receptor es un búfer de indicadores (un array previamente registrado en el sistema por la función *SetIndexBufer*), entonces la indexación en la serie temporal y en el búfer receptor son las mismas (sujeto a una petición para el mismo par símbolo/marco temporal). En este caso, es fácil implementar el rellenado parcial del receptor (en concreto, se utiliza para actualizar las últimas barras, véase un ejemplo más abajo). Si el símbolo o el marco temporal de la serie temporal solicitada no coincide con el símbolo y/o el marco temporal del gráfico actual, la función no devolverá más elementos que el número mínimo de barras en estos dos: origen y destino.

Si un array ordinario (no un búfer) se pasa como el argumento *array*, la función lo rellenará empezando por los primeros elementos, por entero (en el caso de dinámico) o parcialmente (en el caso de estático, con exceso de tamaño). Por lo tanto, si es necesario copiar parcialmente los valores de indicadores en una ubicación arbitraria en otro array, entonces para estos propósitos es necesario utilizar un array intermedio, en el cual se copia el número requerido de elementos, y desde allí se transfieren al destino final.

La función devuelve el número de elementos copiados o -1 en caso de error, incluida la ausencia temporal de datos preparados.

Dado que, por regla general, los indicadores dependen directa o indirectamente de las series temporales de precios, su cálculo no comienza antes de que se sincronicen las cotizaciones. A este respecto, hay que tener en cuenta los aspectos técnicos de la organización y el almacenamiento de series temporales en el terminal y prepararse para que los datos solicitados no aparezcan inmediatamente. En concreto, podemos recibir 0 o una cantidad inferior a la solicitada. Todos estos casos deben tratarse en función de las circunstancias, como esperar a una compilación o informar de un problema al usuario.

Si las series temporales solicitadas aún no se han construido, o necesitan descargarse del servidor, entonces la función se comporta de forma diferente dependiendo del tipo de programa MQL desde el que se llame.

Cuando se solicitan datos que aún no están listos del indicador, la función devolverá inmediatamente -1, pero se iniciará el proceso de carga y construcción de series temporales.

Al solicitar datos a un Asesor Experto o a un script, se iniciará la descarga desde el servidor y/o la construcción de las series temporales requeridas si los datos pueden construirse a partir del historial local. La función devolverá la cantidad de datos que estarán listos en el tiempo de espera (45 segundos) asignado para la ejecución sincrónica de la función (el código de llamada está esperando a que la función finalice).

Tenga en cuenta que la función *CopyBuffer* puede leer datos de los búferes independientemente de su modo de funcionamiento, INDICATOR_DATA, INDICATOR_COLOR_INDEX, INDICATOR_CALCULATIONS, mientras que los dos últimos están ocultos para el usuario.

También es importante tener en cuenta que el desplazamiento de la serie temporal puede establecerse en el indicador llamado mediante la propiedad PLOT_SHIFT y afecta al desplazamiento de los datos leídos con *CopyBuffer*. Por ejemplo, si las líneas del indicador se desplazan hacia el futuro en N barras, en los parámetros *CopyBuffer* (primera forma) hay que dar *offset* igual a (- N), es decir, con un menos, ya que la barra de la serie temporal actual tiene un índice de 0, y los índices de las barras futuras con desplazamiento disminuyen en uno en cada barra. En concreto, tal situación surge con el indicador *Gator*, ya que su gráfico nulo está desplazado hacia delante por el valor del parámetro *TeethShift*, y el primer diagrama está desplazado por el valor del parámetro *LipsShift*. La corrección debe hacerse en función de la más alta de ellas. Veremos un ejemplo en la sección Leer datos de gráficos que tienen un desplazamiento.

MQL5 no proporciona herramientas programáticas para encontrar la propiedad PLOT_SHIFT de un indicador de terceros. Por lo tanto, si es necesario, tendrá que solicitar esta información al usuario a través de una variable de entrada.

Trabajaremos con *CopyBuffer* desde el código del Asesor Experto en el capítulo sobre Asesores Expertos, pero por ahora nos limitaremos a los indicadores.

Vamos a continuar desarrollando un ejemplo con un indicador auxiliar *IndWPR*. Esta vez, en la versión *UseWPR3.mq5* proporcionaremos un búfer de indicador y lo rellenaremos con datos de *IndWPR* utilizando *CopyBuffer*. Para ello, aplicaremos las directivas con el número de búferes y los ajustes de renderización.

```
#property indicator_separate_window
#property indicator_buffers 1
#property indicator_plots   1

#property indicator_type1   DRAW_LINE
#property indicator_color1  clrBlue
#property indicator_width1  1
#property indicator_label1  "WPR"
```

En el contexto global, describimos el parámetro de entrada con el periodo WPR, un array para el búfer y una variable con un descriptor.

```
input int WPRPeriod = 14;

double WPRBuffer[];

int handle;
```

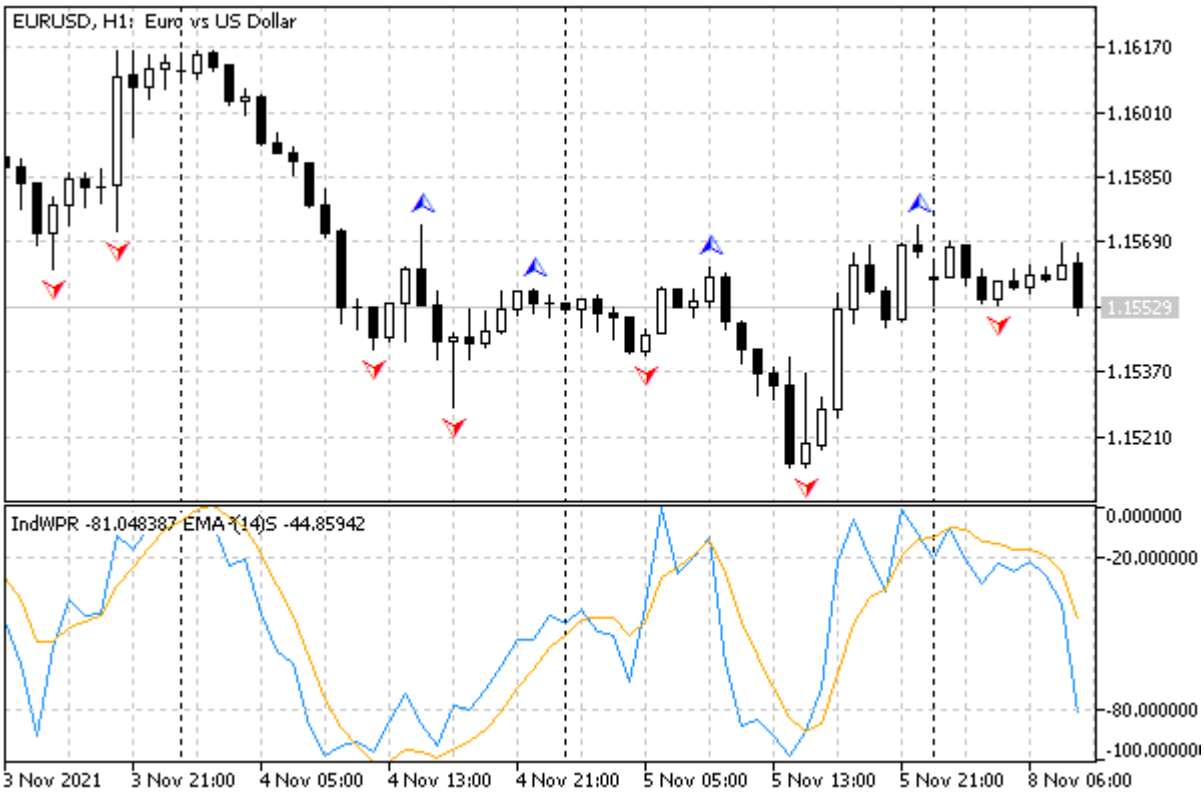
El manejador *OnInit* prácticamente no cambia: sólo se ha añadido la llamada *SetIndexBuffer*.

```
int OnInit()
{
   SetIndexBuffer(0, WPRBuffer);
   handle = iCustom(_Symbol, _Period, "IndWPR", WPRPeriod);
   return handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}
```

En *OnCalculate* copiaremos los datos sin transformaciones.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   // waiting for the calculation to be ready for all bars
   if(BarsCalculated(Handle) != rates_total)
   {
      return prev_calculated;
   }

   // copy the entire timeseries of the subordinate indicator or on new bars to our b
   const int n = CopyBuffer(handle, 0, 0, rates_total - prev_calculated + 1, WPRBuffe
   // if there are no errors, our data is ready for all bars rates_total
   return n > -1 ? rates_total : 0;
}
```

Al compilar y ejecutar *UseWPR3*, obtendremos de hecho una copia del WPR original, con la excepción del ajuste de los niveles, la exactitud de los números y el título. Esto es suficiente para probar el mecanismo, pero normalmente los nuevos indicadores basados en uno o más indicadores auxiliares ofrecen alguna idea y transformación de datos propia. Por lo tanto, desarrollaremos otro indicador que genere señales de compra y venta (desde la posición del trading, no deben considerarse como un modelo, ya que se trata únicamente de una tarea de programación). La idea del indicador se muestra en la siguiente imagen:



Indicadores IndWPR, IndTripleEMA, IndFractals

Utilizamos la salida del WPR de las zonas de sobrecompra y sobreventa como recomendación, respectivamente, para vender y comprar. Para que las señales no reaccionen a fluctuaciones aleatorias, aplicaremos una media móvil triple a WPR y comprobaremos si su valor cruza los límites de las zonas superior e inferior.

Como filtro para estas señales, comprobaremos qué fractal fue el último antes de este momento: un fractal superior significa un retroceso del precio a la baja y confirma una venta, y un fractal inferior significa un retroceso al alza y, por tanto, apoya una compra. Los fractales aparecen con un desfase de un número de barras igual al orden de los fractales.

El nuevo indicador está disponible en el archivo *UseWPRFractals.mq5*.

Necesitamos tres búferes: dos de señal y uno más para el filtro. Podríamos emitir este último en el modo INDICATOR_CALCULATIONS. En su lugar, hagámoslo el INDICATOR_DATA estándar, pero con el estilo DRAW_NONE; de esta forma no interferirá en el gráfico, pero sus valores serán visibles en la ventana de datos.

Las señales se mostrarán en el gráfico principal (en los precios *Close* por defecto), por lo que utilizaremos la directiva *indicator_chart_window*. Todavía podemos llamar a los indicadores del tipo WPR que se dibujan en una ventana separada, ya que todos los indicadores subordinados se pueden calcular sin visualización. Si es necesario, podemos trazarlos, pero hablaremos de ello en el capítulo sobre gráficos (véase *ChartIndicatorAdd*).

```
#property indicator_chart_window
#property indicator_buffers 3
#property indicator_plots   3
// buffer drawing settings
#property indicator_type1   DRAW_ARROW
#property indicator_color1  clrRed
#property indicator_width1  1
#property indicator_label1  "Sell"
#property indicator_type2   DRAW_ARROW
#property indicator_color2  clrBlue
#property indicator_width2  1
#property indicator_label2  "Buy"
#property indicator_type3   DRAW_NONE
#property indicator_color3  clrGreen
#property indicator_width3  1
#property indicator_label3  "Filter"
```

En las variables de entrada proporcionaremos la posibilidad de especificar el periodo WPR, el periodo de promediación (suavizado) y el orden fractal. Estos son los parámetros de los indicadores subordinados. Además, introducimos la variable *offset* con el número de la barra en la que se analizarán las señales. El valor 0 (por defecto) significa la barra actual y el análisis en modo tick (nota: las señales de la última barra pueden ser redibujadas; a algunos operadores de trading eso no les gusta). Si hacemos *offset* igual a 1, analizaremos las barras ya formadas, y tales señales no cambian.

```
input int PeriodWPR = 11;
input int PeriodEMA = 5;
input int FractalOrder = 1;
input int Offset = 0;
input double Threshold = 0.2;
```

La variable *Threshold* define el tamaño de las zonas de sobrecompra y sobreventa como una fracción de ±1.0 (en cada dirección). Por ejemplo, si sigue la configuración clásica de WPR con niveles -20 y -80 en una escala de 0 a -100, entonces *Threshold* debería ser igual a 0.4.

Se proporcionan los siguientes arrays para los búferes de indicadores.

```
double UpBuffer[];   // upper signal means overbought, i.e. selling
double DownBuffer[]; // lower signal means oversold, i.e. buy
double filter[];     // fractal filter direction +1 (up/buy), -1 (down/sell)
```

Los manejadores de indicadores se guardarán en variables globales.

```
int handleWPR, handleEMA3, handleFractals;
```

Realizaremos todos los ajustes, como de costumbre, en *OnInit*. Dado que la función *CopyBuffer* utiliza la indexación del presente al pasado, para la uniformidad de la lectura de datos fijamos la bandera «serie» (*ArraySetAsSeries*) para todos los arrays.

```
int OnInit()
{
   // binding buffers
   SetIndexBuffer(0, UpBuffer);
   SetIndexBuffer(1, DownBuffer);
   SetIndexBuffer(2, Filter, INDICATOR_DATA); // version: INDICATOR_CALCULATIONS
   ArraySetAsSeries(UpBuffer, true);
   ArraySetAsSeries(DownBuffer, true);
   ArraySetAsSeries(Filter, true);

   // arrow signals
   PlotIndexSetInteger(0, PLOT_ARROW, 234);
   PlotIndexSetInteger(1, PLOT_ARROW, 233);

   // subordinate indicators
   handleWPR = iCustom(_Symbol, _Period, "IndWPR", PeriodWPR);
   handleEMA3 = iCustom(_Symbol, _Period, "IndTripleEMA", PeriodEMA, 0, handleWPR);
   handleFractals = iCustom(_Symbol, _Period, "IndFractals", FractalOrder);
   if(handleWPR == INVALID_HANDLE
   || handleEMA3 == INVALID_HANDLE
   || handleFractals == INVALID_HANDLE)
   {
      return INIT_FAILED;
   }

   return INIT_SUCCEEDED;
}
```

En las llamadas de *iCustom* debe prestarse atención a cómo se crea *handleEMA3*. Dado que esta media debe calcularse a partir del WPR, pasamos *handleWPR* (obtenido en la llamada anterior a *iCustom*) como último parámetro, después de los parámetros reales del indicador *IndTripleEMA*. Al hacerlo, debemos especificar la lista completa de parámetros de entrada de *IndTripleEMA* (los parámetros que contiene son *int InpPeriodEMA* y *BEGIN_POLICY InpHandleBegin*; utilizamos el segundo parámetro para estudiar la omisión de las barras iniciales y no lo necesitamos ahora, pero debemos pasarlo, así que simplemente lo ponemos a 0). Si omitiéramos el segundo parámetro en la llamada por considerarlo irrelevante en el contexto actual de la aplicación, entonces el manejador *handleWPR* pasado se interpretaría en el indicador llamado como *InpHandleBegin*. En consecuencia, *IndTripleEMA* se aplicaría al precio *Close* normal.

Cuando no necesitamos pasar un manejador extra, la sintaxis de la llamada *iCustom* le permite omitir un número arbitrario de últimos parámetros, mientras que éstos recibirán los valores por defecto del código fuente.

En el manejador *OnCalculate* esperamos a que los indicadores WPR y los fractales estén preparados, y luego calculamos las señales para todo el historial o la última barra utilizando la función auxiliar *MarkSignals*.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   if(BarsCalculated(handleEMA3) != rates_total
   || BarsCalculated(handleFractals) != rates_total)
   {
      return prev_calculated;
   }

   ArraySetAsSeries(data, true);

   if(prev_calculated == 0) // first launch
   {
      ArrayInitialize(UpBuffer, EMPTY_VALUE);
      ArrayInitialize(DownBuffer, EMPTY_VALUE);
      ArrayInitialize(Filter, 0);

      // look for signals throughout history
      for(int i = rates_total - FractalOrder - 1; i >= 0; --i)
      {
         MarkSignals(i, Offset, data);
      }
   }
   else // online
   {
      for(int i = 0; i < rates_total - prev_calculated; ++i)
      {
         UpBuffer[i] = EMPTY_VALUE;
         DownBuffer[i] = EMPTY_VALUE;
         Filter[i] = 0;
      }

      // looking for signals on a new bar or each tick (if Offset == 0)
      if(rates_total != prev_calculated
      || Offset == 0)
      {
         MarkSignals(0, Offset, data);
      }
   }

   return rates_total;
}
```

Nos interesa sobre todo trabajar con la función *CopyBuffer* oculta en *MarkSignals*. Los valores del WPR suavizado se leerán en el array *wpr[2]*, y los fractales se leerán en *peaks[1]* y *hollows[1]*.

```
int MarkSignals(const int bar, const int offset, const double &data[])
{
   double wpr[2];
   double peaks[1], hollows[1];
   ...
```

A continuación, rellenamos los arrays locales mediante tres llamadas a *CopyBuffer*. Tenga en cuenta que no necesitamos lecturas directas de *IndWPR*, porque se utiliza en los cálculos de *IndTripleEMA*. Leemos los datos en el array *wpr* a través del manejador *handleEMA3*. También es importante que hay 2 búferes en el indicador fractal, y por lo tanto la función *CopyBuffer* llamada dos veces con diferentes índices 0 y 1 para los arrays *peaks* y *hollows*, respectivamente. Los arrays de fractales se leen con una sangría de *FractalOrder*, porque un fractal sólo puede formarse en una barra que tenga un cierto número de barras a la izquierda y a la derecha.

```
   if(CopyBuffer(handleEMA3, 0, bar + offset, 2, wpr) == 2
   && CopyBuffer(handleFractals, 0, bar + offset + FractalOrder, 1, peaks) == 1
   && CopyBuffer(handleFractals, 1, bar + offset + FractalOrder, 1, hollows) == 1)
   {
      ...
```

A continuación, tomamos de la barra anterior del búfer *Filter* la dirección anterior del filtro (al principio del historial es 0, pero cuando aparece un fractal alcista o bajista, escribimos ahí +1 o -1, esto se puede ver en el código fuente justo debajo) y la cambiamos en consecuencia cuando se detecta algún fractal nuevo.

```
      int filterdirection = (int)Filter[bar + 1];

      // the last fractal sets the reversal movement
      if(peaks[0] != EMPTY_VALUE)
      {
         filterdirection = -1; // sell
      }
      if(hollows[0] != EMPTY_VALUE)
      {
         filterdirection = +1; // buy
      }

      Filter[bar] = filterdirection; // remember the current direction
```

Por último, analizamos la transición del indicador WPR suavizado de la zona superior o inferior a la zona media, teniendo en cuenta la anchura de las zonas especificadas en *Threshold*.

```
      // translate 2 WPR values into the range [-1,+1]
      const double old = (wpr[0] + 50) / 50;     // +1.0 -1.0
      const double last = (wpr[1] + 50) / 50;    // +1.0 -1.0

      // bounce from the top down
      if(filterdirection == -1
      && old >= 1.0 - Threshold && last <= 1.0 - Threshold)
      {
         UpBuffer[bar] = data[bar];
         return -1; // sale
      }

      // bounce from the bottom up
      if(filterdirection == +1
      && old <= -1.0 + Threshold && last >= -1.0 + Threshold)
      {
         DownBuffer[bar] = data[bar];
         return +1; // purchase
      }
   }
   return 0; // no signal
}
```

A continuación se muestra una captura de pantalla del indicador resultante en el gráfico.

Indicador de señal UseWPRFractals basado en WPR, EMA3 y fractales

5.5.5 Soporte para múltiples símbolos y marcos temporales

Hasta ahora, en todos los ejemplos de indicadores hemos creado descriptores para el mismo símbolo y marco temporal que en el gráfico actual. Sin embargo, no existe tal limitación. Podemos crear indicadores auxiliares en cualquier símbolo y marco temporal. Por supuesto, en este caso, es necesario esperar a la disposición de las series temporales de terceros, como hicimos antes, por ejemplo, por temporizador.

Implementemos el indicador WPR de marco temporal múltiple (véase el archivo *UseWPRMTF.mq5*), al que también se le puede asignar un cálculo sobre un símbolo arbitrario (distinto del gráfico).

Mostraremos los valores WPR de un periodo determinado para todos los marcos temporales estándar de la enumeración ENUM_TIMEFRAMES. El número de marcos temporales es 21, por lo que el indicador siempre se mostrará en las últimas 21 barras. La barra cero más a la derecha contendrá WPR para M1, la siguiente contendrá WPR para M2, y así sucesivamente hasta la vigésima barra con WPR para el marco temporal mensual. Para facilitar la lectura, colorearemos los trazados con diferentes colores: los marcos temporales de minutos serán rojos, los de horas, verdes, y los diarios y más antiguos, azules.

Dado que será posible establecer un símbolo de trabajo en el indicador y crear varias copias para diferentes símbolos en el mismo gráfico, seleccionaremos el estilo de dibujo DRAW_ARROW y proporcionaremos un parámetro de entrada para asignar un símbolo. De este modo, será posible distinguir las indicaciones para los distintos símbolos. El coloreado requiere un búfer adicional.

```
#property indicator_separate_window
#property indicator_buffers 2
#property indicator_plots   1

#property indicator_type1   DRAW_COLOR_ARROW
#property indicator_color1  clrRed,clrGreen,clrBlue
#property indicator_width1  3
#property indicator_label1  "WPR"
```

Los valores WPR se convierten al rango [-1,+1]. Vamos a escoger la escala de la subventana con cierto margen del rango. Los niveles con valores de ±0.6 corresponden a las normas -20 y -80 antes de la conversión WPR.

```
#property indicator_maximum    +1.2
#property indicator_minimum    -1.2

#property indicator_level1     +0.6
#property indicator_level2     -0.6
#property indicator_levelstyle STYLE_DOT
#property indicator_levelcolor clrSilver
#property indicator_levelwidth 1
```

En variables de entrada: periodo WPR, símbolo de trabajo y código de la flecha mostrada. Cuando el símbolo se deja en blanco, se utiliza el símbolo del gráfico actual.

```
input int WPRPeriod = 14;
input string WorkSymbol = ""; // Symbol
input int Mark = 0;

const string _WorkSymbol = (WorkSymbol == "" ? _Symbol : WorkSymbol);
```

Para facilitar la codificación, el conjunto de marcos temporales figura en el array *TF*.

```
#define TFS 21

ENUM_TIMEFRAMES TF[TFS] =
{
   PERIOD_M1,
   PERIOD_M2,
   PERIOD_M3,
   ...
   PERIOD_D1,
   PERIOD_W1,
   PERIOD_MN1,
};
```

Los descriptores de los indicadores de cada marco temporal se almacenan en el array *Handle*.

```
int Handle[TFS];
```

Configuraremos búferes de indicadores y obtendremos manejadores en *OnInit*.

```
double WPRBuffer[];
double Colors[];

int OnInit()
{
   SetIndexBuffer(0, WPRBuffer);
   SetIndexBuffer(1, Colors, INDICATOR_COLOR_INDEX);
   ArraySetAsSeries(WPRBuffer, true);
   ArraySetAsSeries(Colors, true);
   PlotIndexSetString(0, PLOT_LABEL, _WorkSymbol + " WPR");

   if(Mark != 0)
   {
      PlotIndexSetInteger(0, PLOT_ARROW, Mark);
   }

   for(int i = 0; i < TFS; ++i)
   {
      Handle[i] = iCustom(_WorkSymbol, TF[i], "IndWPR", WPRPeriod);
      if(Handle[i] == INVALID_HANDLE) return INIT_FAILED;
   }

   IndicatorSetInteger(INDICATOR_DIGITS, 2);
   IndicatorSetString(INDICATOR_SHORTNAME,
      "%Rmtf" + "(" + _WorkSymbol + "/" + (string)WPRPeriod + ")");

   return INIT_SUCCEEDED;
}
```

El cálculo en *OnCalculate* sigue el esquema habitual: espera a que los datos estén listos, inicialización, rellenado en nuevas barras. Las funciones auxiliares *IsDataReady* y *FillData* realizan un trabajo directo con los descriptores (véase más adelante).

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   // waiting for slave indicators to be ready
   if(!IsDataReady())
   {
      EventSetTimer(1); // if not ready, postpone the calculation
      return prev_calculated;
   }
   if(prev_calculated == 0) // initialization
   {
      ArrayInitialize(WPRBuffer, EMPTY_VALUE);
      ArrayInitialize(Colors, EMPTY_VALUE);
      // constant colors for the latest TFS bars
      for(int i = 0; i < TFS; ++i)
      {
         Colors[i] = i < 11 ? 0 : (i < 18 ? 1 : 2);
      }
   }
   else // preparing a new bar
   {
      for(int i = prev_calculated; i < rates_total; ++i)
      {
         WPRBuffer[i] = EMPTY_VALUE;
         Colors[i] = 0;
      }
   }

   if(prev_calculated != rates_total) // new bar
   {
      // clear the label on the oldest bar that moved to the left beyond TFS bars
      WPRBuffer[TFS] = EMPTY_VALUE;
      // update bar coloring
      for(int i = 0; i < TFS; ++i)
      {
         Colors[i] = i < 11 ? 0 : (i < 18 ? 1 : 2);
      }
   }

   // copy the data from the subordinate indicators to our buffer
   FillData();
   return rates_total;
}
```

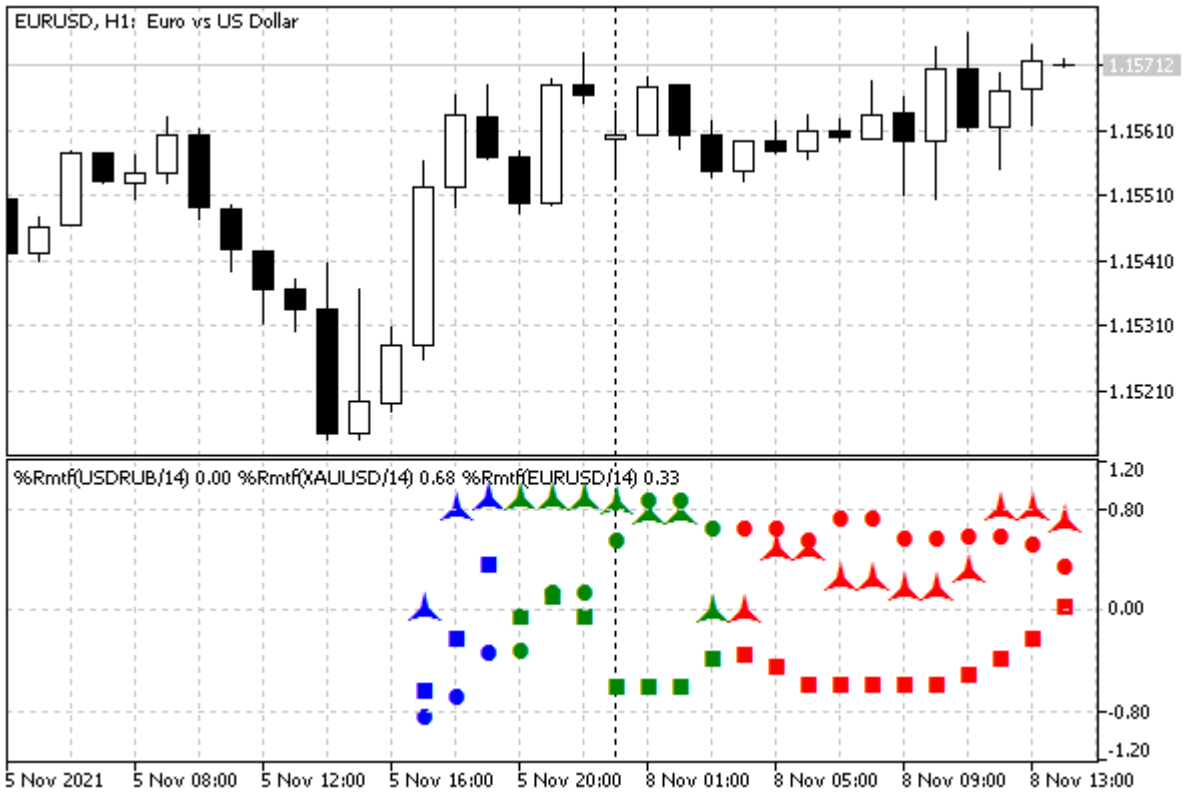
Si es necesario, iniciamos el recálculo por temporizador.

```
void OnTimer()
{
   ChartSetSymbolPeriod(0, _Symbol, _Period);
   EventKillTimer();
}
```

Y aquí están las funciones *IsDataReady* y *FillData*.

```
bool IsDataReady()
{
   for(int i = 0; i < TFS; ++i)
   {
      if(BarsCalculated(Handle[i]) != iBars(_WorkSymbol, TF[i]))
      {
         Print("Waiting for ", _WorkSymbol, " ", EnumToString(TF[i]));
         return false;
      }
   }
   return true;
}

void FillData()
{
   for(int i = 0; i < TFS; ++i)
   {
      double data[1];
      // taking the last actual value (buffer 0, index 0)
      if(CopyBuffer(Handle[i], 0, 0, 1, data) == 1)
      {
         WPRBuffer[i] = (data[0] + 50) / 50;
      }
   }
}
```

Vamos a compilar el indicador y ver cómo se ve en el gráfico. Por ejemplo, vamos a crear tres copias para EURUSD, USDRUB y XAUUSD.

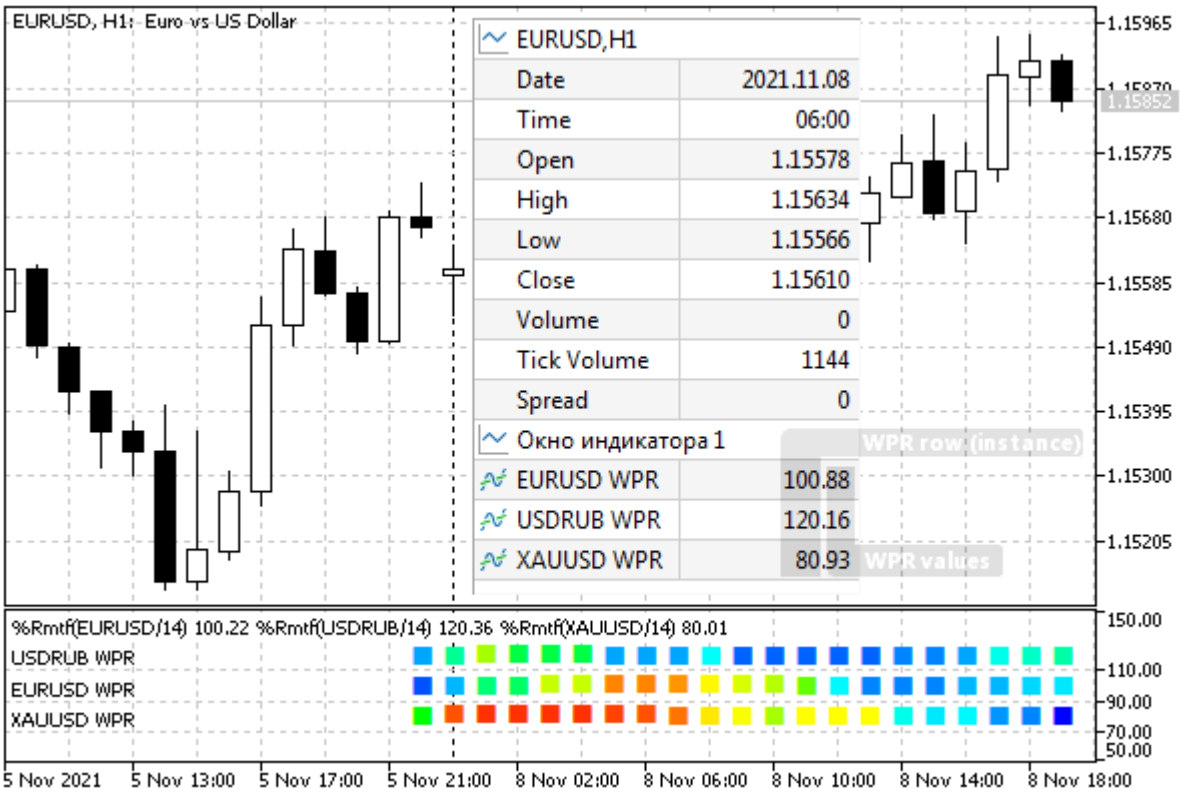


Tres instancias de WPR de marco temporal múltiple para distintos símbolos de trabajo

Durante el primer cálculo, el indicador puede requerir una cantidad de tiempo considerable para preparar las series temporales de todos los marcos temporales.

En cuanto a la parte calculada, exactamente el mismo indicador *UseWPRMTFDashboard.mq5* está diseñado en forma de un panel informativo popular entre los operadores de trading. Para cada símbolo establecemos sangrías verticales individuales en el parámetro *Level* del indicador. Aquí es donde los valores WPR de todos los marcos temporales se muestran como una línea de marcadores, y los valores están codificados por colores. En esta versión, los valores WPR están normalizados en el intervalo [0..1], por lo que el uso de reglas en niveles separados por varias decenas (por ejemplo, 20, como en la captura de pantalla siguiente) permite colocar varias instancias del indicador en la subventana sin que se solapen (80 , 100, 120, etc.). Cada copia se utiliza para su propio símbolo de trabajo. Además, debido a que *Level* es mayor que 1.0, y los valores de WPR son menores, son visibles en los valores de *Data window* por separado: a la izquierda y a la derecha del punto decimal.

Las etiquetas para las reglas de etiquetas se proporcionan mediante niveles añadidos dinámicamente en *OnInit*.



Panel de tres líneas WPR de marco temporal múltiple para distintos símbolos de trabajo

Puede explorar el código fuente de *UseWPRMTFDashboard.mq5* y compararlo con *UseWPRMTF.mq5*. Para generar una paleta de tonos de color, utilizamos el archivo *ColorMix.mqh*.

Cuando terminemos de estudiar los indicadores integrados, incluido *iWPR*, podemos sustituir el *IndWPR* personalizado por el *iWPR* integrado.

Eficacia e intensidad de recursos de los indicadores compuestos

El enfoque mostrado anteriormente, con la generación de muchos indicadores auxiliares, no es eficiente en términos de velocidad y consumo de recursos. Se trata principalmente de un ejemplo de integración de programas MQL y de intercambio de datos entre ellos. Pero, como cualquier tecnología, debe utilizarse adecuadamente.

Cada uno de los dos indicadores creados calcula WPR en todas las barras de la serie temporal, y luego sólo el último valor se toma en el indicador de llamada. Desperdiciamos tanto memoria como tiempo de procesador.

Si se dispone del código fuente de los indicadores auxiliares o se conoce el concepto de su funcionamiento, lo más óptimo es ubicar el algoritmo de cálculo dentro del indicador principal (o Asesor Experto) y aplicarlo para un historial limitado e inmediato de la profundidad mínima requerida.

En algunos casos, puede prescindir de referirse a marcos temporales superiores realizando cálculos equivalentes en el marco temporal actual: por ejemplo, en lugar de un rango de precios en 14 barras diarias (lo que requiere construir una serie temporal D1 completa), puede tomar un rango en 14 * 24 barras H1, sujeto a una operación de trading de 24 horas y lanzar el indicador en el gráfico H1.

Al mismo tiempo, cuando se utiliza un indicador comercial en un sistema de trading (sin código fuente), sólo se pueden obtener datos de él a través de interfaces de programación abiertas. En este caso, crear un manejador y luego leer los datos del búfer del indicador a través de *CopyBuffer* es la única opción disponible, pero al mismo tiempo es la forma práctica y universal. Sólo debe tener siempre en cuenta que llamar a funciones de la API es una operación más «cara» que manipular su propio array dentro de un programa MQL y llamar a funciones locales. Si necesita mantener abiertos muchos terminales, probablemente cada uno con un conjunto de programas MQL no optimizados de este tipo, y si tiene recursos limitados, es probable que el rendimiento disminuya.

5.5.6 Visión general de los indicadores integrados

El terminal proporciona un amplio conjunto de indicadores populares, que también están disponibles a través de la API. Por lo tanto, no es necesario implementar sus algoritmos en MQL5. Estos indicadores se crean utilizando funciones integradas similares a *iCustom*. Por ejemplo, anteriormente creamos nuestras propias versiones de los indicadores WPR y Media Móvil Triple EMA con fines educativos. Sin embargo, los indicadores correspondientes pueden utilizarse directamente a través de las funciones *iWPR* y *iTEMA*. Todos los indicadores disponibles figuran en la tabla de más abajo.

Todos los indicadores integrados toman una cadena con un símbolo de trabajo y un marco temporal como los primeros dos parámetros, y también devuelven un número entero que es el descriptor del indicador. En general, el prototipo de todas las funciones tiene este aspecto:

```
int iFunction(const string symbol, ENUM_TIMEFRAMES timeframe, ...)
```

En lugar de una elipsis, siguen los parámetros específicos de un indicador concreto. Su número y tipos difieren. Algunos indicadores no tienen parámetros.

Por ejemplo, WPR tiene un parámetro, como en nuestra versión casera, un periodo: *int iWPR(const string symbol, ENUM_TIMEFRAMES timeframe, int period)*. Y el indicador fractal integrado, a diferencia de nuestra versión, no tiene parámetros especiales: *int iFractals(const string symbol, ENUM_TIMEFRAMES period)*. En este caso, el orden de los fractales está codificado y es igual a 2, es decir, antes del extremo (superior o inferior) y después de él, debe haber al menos dos barras con precios *high* y *low* menos pronunciados, respectivamente.

Se permite establecer el valor NULL en lugar de un símbolo. NULL significa el símbolo de trabajo del gráfico actual, y el valor 0 del parámetro *timeframe* corresponde al marco temporal actual del gráfico, ya que también es el valor PERIOD_CURRENT de la enumeración ENUM_TIMEFRAMES (véase la sección Símbolos y marcos temporales).

También hay que tener en cuenta que los distintos tipos de indicadores tienen distintos números de búferes. Por ejemplo, una media móvil o WPR sólo tiene un búfer, mientras que los fractales tienen dos. El número de búferes también se indica en la tabla en una columna aparte.

Función	Nombre del indicador	Opciones	Búferes
iAC	Oscilador Acelerador	—	1*
iAD	Acumulación / Distribución	ENUM_APPLIED_VOLUME volume	1*
iADX	Índice direccional medio	int period	3*
iADXWilder	Índice direccional medio de Welles Wilder	int period	3*

Función	Nombre del indicador	Opciones	Búferes
iAlligator	Alligator	int jawPeriod, int jawShift, int teethPeriod,int teethShift, int lipsPeriod, int lipsShift, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price	3
iAMA	Media móvil adaptativa	int period, int fast, int slow, int shift, ENUM_APPLIED_PRICE price	1
iAO	Oscilador Impresionante	—	1*
iATR	Rango medio verdadero	int period	1*
iBands	Bandas de Bollinger	int period, int shift, double deviation, ENUM_APPLIED_PRICE price	3
iBearsPower	Bears Power	int period	1*
iBullsPower	Bulls Power	int period	1*
iBWMFI	Índice de Facilitación del Mercado de Bill Williams	ENUM_APPLIED_VOLUME volume	1*
iCCI	Índice del Canal de Materias Primas	int period, ENUM_APPLIED_PRICE price	1*
iChaikin	Oscilador Chaikin	int fast, int slow, ENUM_MA_METHOD method, ENUM_APPLIED_VOLUME volume	1*
iDEMA	Media móvil exponencial doble	int period, int shift, ENUM_APPLIED_PRICE price	1
iDeMarker	DeMarker	int period	1*
iEnvelopes	Envelopes	int period, int shift, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price, double deviation	2
iForce	Índice de Fuerza	int period, ENUM_MA_METHOD method, ENUM_APPLIED_VOLUME volume	1*
iFractals	Fractales	—	2
iFrAMA	Media móvil adaptativa fractal	int period, int shift, ENUM_APPLIED_PRICE price	1
iGator	Oscilador Gator	int jawPeriod, int jawShift, int teethPeriod, int teethShift, int lipsPeriod, int lipsShift, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price	4*

Función	Nombre del indicador	Opciones	Búferes
iIchimoku	Ichimoku Kinko Hyo	int tenkan, int kijun, int senkou	5
iMomentum	Momentum	int period, ENUM_APPLIED_PRICE price	1*
iMFI	Índice de flujo de dinero	int period, ENUM_APPLIED_VOLUME volume	1*
iMA	Media móvil	int period, int shift, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price	1
iMACD	Convergencia/Divergencia de Medias Móviles	int fast, int slow, int signal, ENUM_APPLIED_PRICE price	2*
iOBV	Volumen de balance	ENUM_APPLIED_VOLUME volume	1*
iOsMA	Media móvil del oscilador (histograma MACD)	int fast, int slow, int signal, ENUM_APPLIED_PRICE price	1*
iRSI	Índice de Fuerza Relativa	int period, ENUM_APPLIED_PRICE price	1*
iRVI	Índice de Vigor Relativo	int period	1*
iSAR	Sistema parabólico de parada y reversión	doble paso, doble máximo	1
iStdDev	Desviación típica	int period, int shift, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price	1*
iStochastic	Oscilador estocástico	int Kperiod, int Dperiod, int slowing, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price	2*
iTEMA	Media móvil exponencial triple	int period, int shift, ENUM_APPLIED_PRICE price	1
iTriX	Oscilador de medias móviles exponenciales triples	int period, ENUM_APPLIED_PRICE price	1*
iVIDyA	Media Móvil con Periodo de Promediación Dinámico	int momentum, int smooth, int shift, ENUM_APPLIED_PRICE price	1
iVolumes	Volúmenes	ENUM_APPLIED_VOLUME volume	1*
iWPR	Rango Porcentual de Williams	int period	1*

En la columna de la derecha se indican con un asterisco los indicadores con ventana propia * (aparecen debajo del gráfico principal).

Los parámetros más utilizados son los que definen los periodos de los indicadores (*period*, *fast*, *slow* y otras variaciones), así como la línea *shift*: cuando es positiva, los trazados se desplazan hacia la derecha; cuando es negativa se desplazan hacia la izquierda un número determinado de barras.

Muchos parámetros tienen tipos de enumeración de aplicación: ENUM_APPLIED_PRICE, ENUM_APPLIED_VOLUME, ENUM_MA_METHOD. Ya nos familiarizamos con ENUM_APPLIED_PRICE en la sección Enumeraciones. Todos los tipos disponibles se presentan a continuación en tablas con descripciones.

Identificador	Descripción	Valor
PRICE_CLOSE	Precio de cierre de la barra	1
PRICE_OPEN	Precio de apertura de la barra	2
PRICE_HIGH	Precio alto de la barra	3
PRICE_LOW	Precio bajo de la barra	4
PRICE_MEDIAN	Precio medio, (alto+bajo)/2	5
PRICE_TYPICAL	Precio típico, (alto+bajo+cierre)/3	6
PRICE_WEIGHTED	Precio medio ponderado, (alto+bajo+cierre+cierre)/4	7

Los indicadores que trabajan con volúmenes pueden operar con volúmenes de ticks (de hecho, se trata de un contador de ticks) o con volúmenes reales (suelen estar disponibles sólo para instrumentos bursátiles). Ambos tipos se resumen en el enum ENUM_APPLIED_VOLUME.

Identificador	Descripción	Valor
VOLUMEN_TICK	Volumen de ticks	0
VOLUME_REAL	Volumen de trading	1

Muchos indicadores técnicos suavizan (o promedian) las series temporales. El terminal admite los cuatro métodos de suavizado más comunes, que se especifican en MQL5 utilizando los elementos de la enumeración ENUM_MA_METHOD.

Identificador	Descripción	Valor
MODE_SMA	Promedio simple	0
MODE_EMA	Promedio exponencial	1
MODE_SMMA	Promedio suavizado	2
MODE_LWMA	Promedio ponderado lineal	3

Para el indicador Estocástico, cuyo ejemplo analizaremos en la siguiente sección, existen dos opciones de cálculo: por precios *Close* o por precios *High/Low*. Estos valores se proporcionan en la enumeración especial ENUM_STO_PRICE.

Identificador	Descripción	Valor
STO_LOWHIGH	Cálculo por precios bajos/altos	0
STO_CLOSECLOSE	Cálculo por precios de cierre/cierre	1

El propósito y la numeración de los búferes para aquellos indicadores que tienen más de un búfer se muestran en la siguiente tabla.

Indicadores	Constantes	Descripciones	Valor
ADX, ADXW			
	MAIN_LINE	Línea principal	0
	PLUSDI_LINE	Línea +DI	1
	MINUSDI_LINE	Línea -DI	2
iAlligator			
	GATORJAW_LINE	Línea de mandíbula	0
	GATORTEETH_LINE	Línea de dientes	1
	GATORLIPS_LINE	Línea de labios	2
iBands			
	BASE_LINE	Línea principal	0
	UPPER_BAND	Banda superior	1
	LOWER_BAND	Banda inferior	2
iEnvelopes, iFractals			
	UPPER_LINE	Línea superior	0
	LOWER_LINE	Línea inferior	1
iGator			
	UPPER_HISTOGRAM	Histograma superior	0
	LOWER_HISTOGRAM	Histograma inferior	2
iIchimoku			
	TENKANSEN_LINE	Línea Tenkan-sen	0
	KIJUNSEN_LINE	Línea Kijun-sen	1
	SENKOUSPANA_LINE	Línea Senkou Span A	2
	SENKOUSPANB_LINE	Línea Senkou Span B	3
	CHIKOUSPAN_LINE	Línea Chikou span	4
iMACD, iRVI, iStochastic			
	MAIN_LINE	Línea principal	0
	SIGNAL_LINE	Línea de señalización	1

Las fórmulas para calcular todos los indicadores figuran en Documentación de MetaTrader 5 .

Encontrará información técnica completa sobre la llamada a funciones indicadoras, incluidos ejemplos de códigos fuente, en Documentación MQL5. Más adelante veremos algunos ejemplos en este libro.

5.5.7 Utilización de los indicadores integrados

Como sencillo ejemplo introductorio del uso del indicador integrado, vamos a utilizar una llamada a *iStochastic*. El prototipo de esta función indicadora es el siguiente:

```
int iStochastic(const string symbol, ENUM_TIMEFRAMES timeframe,
 int Kperiod, int Dperiod, int slowing,
 ENUM_MA_METHOD method, ENUM_STO_PRICE price)
```

Como vemos, además de los parámetros estándar *symbol* y *time frame*, el estocástico tiene varios parámetros específicos:

- *Kperiod* : número de barras para calcular la línea %K
- *Dperiod* : periodo de suavizado primario para la línea %D
- *slowing* : periodo de suavizado secundario (deceleración)
- *method* : método de promediado (suavizado)
- *price* : método de cálculo estocástico

Vamos a intentar crear nuestro propio indicador *UseStochastic.mq5*, que copiará los valores del estocástico en sus búferes. Como hay dos búferes en el estocástico, reservaremos también dos: son las líneas «principal» y «señal».

```
#property indicator_separate_window
#property indicator_buffers 2
#property indicator_plots   2

#property indicator_type1   DRAW_LINE
#property indicator_color1  clrBlue
#property indicator_width1  1
#property indicator_label1  "St'Main"

#property indicator_type2   DRAW_LINE
#property indicator_color2  clrChocolate
#property indicator_width2  1
#property indicator_label2  "St'Signal"
#property indicator_style2  STYLE_DOT
```

En las variables de entrada proporcionamos todos los parámetros necesarios.

```
input int KPeriod = 5;
input int DPeriod = 3;
input int Slowing = 3;
input ENUM_MA_METHOD Method = MODE_SMA;
input ENUM_STO_PRICE StochasticPrice = STO_LOWHIGH;
```

A continuación, describimos arrays para los búferes de indicadores y una variable global para el descriptor.

```
double MainBuffer[];
double SignalBuffer[];

int Handle;
```

Inicializaremos en *OnInit*.

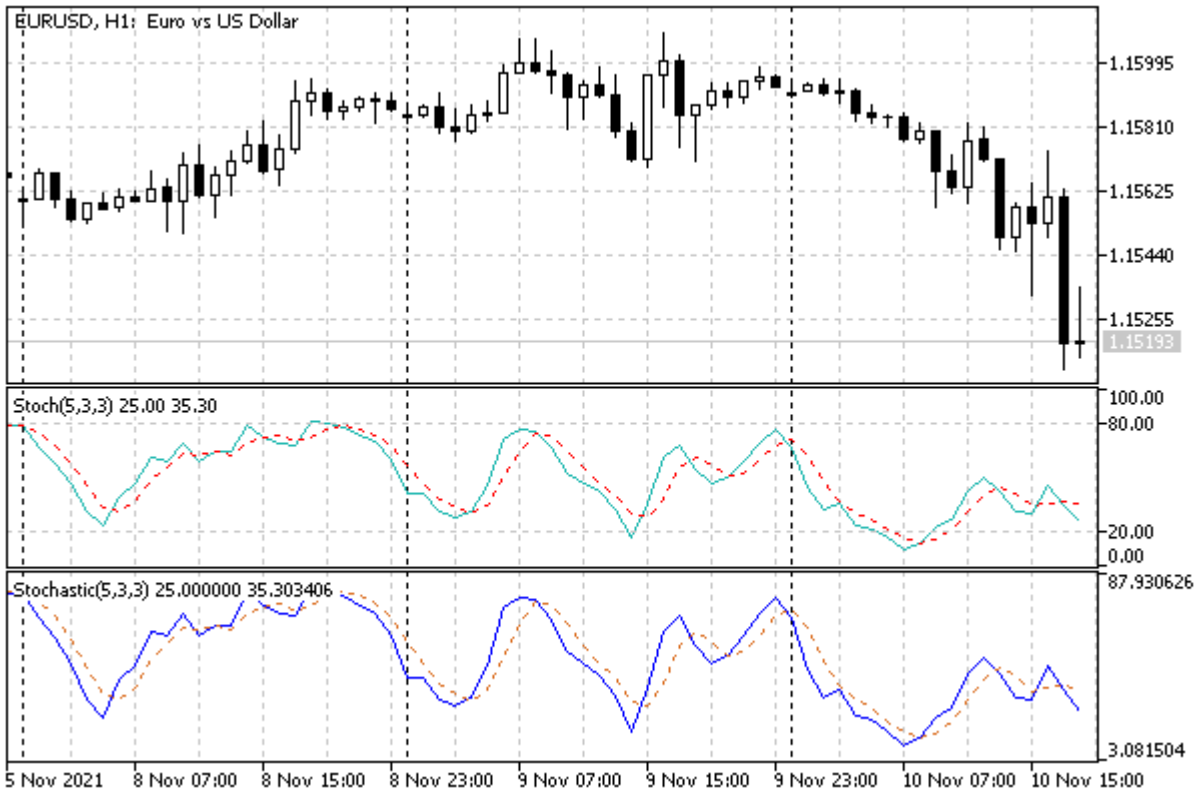
```
int OnInit()
{
   IndicatorSetString(INDICATOR_SHORTNAME,
      StringFormat("Stochastic(%d,%d,%d)", KPeriod, DPeriod, Slowing));
   // binding of arrays as buffers
   SetIndexBuffer(0, MainBuffer);
   SetIndexBuffer(1, SignalBuffer);
   // getting the descriptor Stochastic
   Handle = iStochastic(_Symbol, _Period,
      KPeriod, DPeriod, Slowing, Method, StochasticPrice);
   return Handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}
```

Ahora, en *OnCalculate*, necesitamos leer los datos usando la función *CopyBuffer* tan pronto como el manejador esté listo.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   // waiting for the calculation of the stochastic on all bars
   if(BarsCalculated(Handle) != rates_total)
   {
      return prev_calculated;
   }

   // copy data to our two buffers
   const int n = CopyBuffer(Handle, 0, 0, rates_total - prev_calculated + 1,
      MainBuffer);
   const int m = CopyBuffer(Handle, 1, 0, rates_total - prev_calculated + 1,
      SignalBuffer);

   return n > -1 && m > -1 ? rates_total : 0;
}
```

Tenga en cuenta que estamos llamando a *CopyBuffer* dos veces: para cada búfer por separado (0 y 1 en el segundo parámetro). Un intento de leer un búfer con un índice inexistente, por ejemplo, 2, generaría un error y no recibiríamos ningún dato.

Nuestro indicador no es especialmente útil, ya que no añade nada al estocástico original y no analiza sus lecturas. Por otro lado, podemos asegurarnos de que las líneas del indicador terminal estándar y las creadas en MQL5 coinciden (también se podrían añadir fácilmente niveles y ajustes de precisión, como hacíamos con los indicadores completamente personalizados, pero entonces sería difícil distinguir una copia del original).



Estocástico estándar y personalizado basado en la función iStochastic

Para demostrar el almacenamiento en caché de los indicadores por el terminal, añada a la función *OnInit* un par de líneas.

```
double array[];
Print("This is very first copy of iStochastic with such settings=",
   !(CopyBuffer(Handle, 0, 0, 10, array) > 0));
```

En este caso, hemos utilizado un truco relacionado con las características conocidas: inmediatamente después de que se cree el indicador, tarda algún tiempo en calcularse, y es imposible leer datos del búfer inmediatamente después de recibir el manejador. Esto es válido para el caso del arranque «en frío», cuando el indicador con los parámetros especificados aún no existe en la caché, en la memoria del terminal. Si hay un análogo preparado, entonces podemos acceder instantáneamente al búfer.

Después de compilar un nuevo indicador, debe colocar dos copias del mismo en dos gráficos del mismo símbolo y marco temporal. La primera vez se mostrará un mensaje con la bandera *true* en el registro (ésta es la primera copia), y la segunda vez (y las siguientes, si hay muchos gráficos) será *false*. También puede añadir primero manualmente un indicador «Stochastic Oscillator» estándar al gráfico (con los ajustes predeterminados o los que se aplicarán después en *Use Stochastic*) y después ejecutar *Use Stochastic*: necesitamos obtener también *false*.

Ahora vamos a intentar inventar algo original basado en un indicador estándar. El siguiente indicador *UseM1MA.mq5* está diseñado para calcular precios medios por barra en M5 y marcos temporales superiores (principalmente, intradía). Acumula los precios de las barras M1 que caen dentro del rango de marcas de tiempo de cada barra específica en el marco temporal de trabajo (superior). Esto le permite estimar el precio efectivo de una barra con mucha más precisión que los tipos de precio estándar (*Close*, *Open*, *Median*, *Typical*, *Weighted*, etc.). Además, preveremos la posibilidad de promediar dichos precios a lo largo de un período determinado, pero aquí debe estar preparado para que no funcione una línea especialmente suave.

El indicador se mostrará en la ventana principal y contendrá un único búfer. Los ajustes pueden modificarse con ayuda de 3 parámetros:

```
input uint _BarLimit = 100; // BarLimit
input uint BarPeriod = 1;
input ENUM_APPLIED_PRICE M1Price = PRICE_CLOSE;
```

BarLimit establece el número de barras del historial más cercano para el cálculo. Esto es importante porque los gráficos de marco temporal alto pueden requerir un número muy grande de barras en comparación con el minuto M1 (por ejemplo, se sabe que un día D1 en el trading 24/7 contiene 1440 barras M1). Esto puede dar lugar a que se descarguen datos adicionales en espera de sincronización. Experimente con la configuración de ahorro por defecto (100 barras del marco temporal de trabajo) antes de ajustar este parámetro a 0, lo que significa procesamiento sin límite.

Sin embargo, incluso cuando se establece *BarLimit* en 0, es probable que el indicador no se calcule para todo el historial visible del marco temporal más antiguo: si el terminal tiene un límite en el número de barras del gráfico, entonces también afectará a las solicitudes de barras M1. En otras palabras, la profundidad del análisis está determinada por el tiempo durante el cual el número máximo permitido de barras M1 entra en la historia.

BarPeriod establece el número de barras del marco temporal superior para el que se realiza el promedio. El valor por defecto aquí es 1, lo que le permite ver el precio efectivo de cada barra por separado.

El parámetro *M1Price* especifica el tipo de precio utilizado para los cálculos de las barras M1.

En el contexto global, un array se describe para un búfer, un descriptor y una bandera de autoactualización, que necesitamos para esperar la construcción de una serie temporal del marco temporal M1 «ajeno».

```
double Buffer[];

int Handle;
int BarLimit;
bool PendingRefresh;

const string MyName = "M1MA (" + StringSubstr(EnumToString(M1Price), 6)
   + "," + (string)BarPeriod + "[" + (string)(PeriodSeconds() / 60) + "])";
const uint P = PeriodSeconds() / 60 * BarPeriod;
```

Además, aquí se forman el nombre del indicador y el periodo de promediación *P*. La función *PeriodSeconds*, que devuelve el número de segundos dentro de una barra del marco temporal actual, le permite calcular el número de barras M1 dentro de una barra actual: *PeriodSeconds() / 60* (60 segundos es la duración de la barra M1).

La inicialización habitual se realiza en *OnInit*.

```
int OnInit()
{
   IndicatorSetString(INDICATOR_SHORTNAME, MyName);
   IndicatorSetInteger(INDICATOR_DIGITS, _Digits);

   SetIndexBuffer(0, Buffer);

   Handle = iMA(_Symbol, PERIOD_M1, P, 0, MODE_SMA, M1Price);

   return Handle != INVALID_HANDLE ? INIT_SUCCEEDED : INIT_FAILED;
}
```

Para obtener el precio medio en una barra de marco temporal superior, aplicamos una media móvil simple, llamando a *iMA* con el modo MODE_SMA.

La función *OnCalculate* que se muestra a continuación está simplificada. En la primera ejecución o cambio de historial, borramos el búfer y rellenamos la variable *BarLimit* (esto es necesario porque las variables de entrada no se pueden editar, y queremos interpretar el valor 0 como el número máximo de barras disponibles para el cálculo). Durante las siguientes llamadas, los elementos del búfer se borran sólo en los últimos compases, a partir de *prev_calculated* y no más allá de *BarLimit*.

```
int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   if(prev_calculated == 0)
   {
      ArrayInitialize(Buffer, EMPTY_VALUE);
      if(_BarLimit == 0
      || _BarLimit > (uint)rates_total)
      {
         BarLimit = rates_total;
      }
      else
      {
         BarLimit = (int)_BarLimit;
      }
   }
   else
   {
      for(int i = fmax(prev_calculated - 1, (int)(rates_total - BarLimit));
         i < rates_total; ++i)
      {
         Buffer[i] = EMPTY_VALUE;
      }
   }
```

Antes de leer los datos del indicador *iMA* creado, hay que esperar a que estén listos: para ello comparamos *BarsCalculated* con el número de barras M1.

```
   if(BarsCalculated(Handle) != iBars(_Symbol, PERIOD_M1))
   {
      if(prev_calculated == 0)
      {
         EventSetTimer(1);
         PendingRefresh = true;
      }
      return prev_calculated;
   }
   ...
```

Si los datos no están listos, iniciamos un temporizador para intentar leerlos de nuevo en un segundo.

A continuación, entramos en la parte principal de cálculo del algoritmo y, por lo tanto, debemos detener el temporizador si todavía está en marcha. Esto puede suceder si el siguiente evento de tic llegó más rápido, antes de 1 segundo, e *iMA* M1 ya dio resultados. Lo lógico sería simplemente llamar a la función *EventKillTimer*adecuada. Sin embargo, hay un matiz en su comportamiento: no borra la cola de eventos para un programa MQL de tipo indicador, y si un evento de temporizador ya está colocado en la cola, entonces se llamará una vez al manejador *OnTimer*. Para evitar la actualización innecesaria del gráfico, controlamos el proceso utilizando nuestra propia variable *Pending Refresh*, y aquí le asignamos *false*.

```
   ...
   Pending Refresh =false;// data is ready, the timer will idle
   ...
```

Así es como se organiza todo en el manejador *OnTimer*:

```
void OnTimer()
{
   EventKillTimer();
   if(PendingRefresh)
   {
      ChartSetSymbolPeriod(0, _Symbol, _Period);
   }
}
```

Volvamos a *OnCalculate* y presentemos el flujo de trabajo principal.

```
   for(int i = fmax(prev_calculated - 1, (int)(rates_total - BarLimit));
      i < rates_total; ++i)
   {
      static double result[1];

      // get the last bar M1 corresponding to the i-th bar of the current timeframe
      const datetime dt = time[i] + PeriodSeconds() - 60;
      const int bar = iBarShift(_Symbol, PERIOD_M1, dt);

      if(bar > -1)
      {
         // request MA value on M1
         if(CopyBuffer(Handle, 0, bar, 1, result) == 1)
         {
            Buffer[i] = result[0];
         }
         else
         {
            Print("CopyBuffer failed: ", _LastError);
            return prev_calculated;
         }
      }
   }

   return rates_total;
}
```

El funcionamiento del indicador se ilustra con la siguiente imagen en EURUSD,H1. La línea azul corresponde a la configuración por defecto. Cada valor se obtiene promediando PRICE_CLOSE sobre 60 barras M1. La línea naranja incluye además el suavizado en 5 barras H1, con precios M1 PRICE_TYPICAL.

Dos instancias del indicador UseM1MA en EURUSD,H1

El libro presenta una versión simplificada de *UseM1MASimple.mq5*. Dejamos entre bastidores las particularidades del cálculo de la media de la última barra (incompleta), el tratamiento de las barras vacías (para las que no hay datos en M1) y la configuración correcta de la propiedad PLOT_DRAW_BEGIN, así como el control de la aparición de desfases a corto plazo en el cálculo de la media cuando aparecen nuevas barras. La versión completa está disponible en el archivo *UseM1MA.mq5*.

Como último ejemplo de construcción de indicadores a partir de indicadores estándar, vamos a analizar la mejora del indicador *IndUnityPercent.mq5*, que se presentó en la sección Indicadores multidivisa y de marco temporal múltiple. La primera versión utilizaba los precios de *Close* para los cálculos, obteniéndolos con *CopyBuffer*. En la nueva versión *UseUnityPercentPro.mq5*, vamos a sustituir este método por la lectura de los datos del indicador *iMA*. Esto nos permitirá implementar nuevas funciones:

- Precios medios durante un periodo determinado
- Elija el método de cálculo de medias
- Elija el tipo de precio para el cálculo

Los cambios en el código fuente son mínimos. Añadimos 3 nuevos parámetros y un array global para los manejadores *iMA*:

```
input ENUM_APPLIED_PRICE PriceType = PRICE_CLOSE;
input ENUM_MA_METHOD PriceMethod = MODE_EMA;
input int PricePeriod = 1;
...
int Handles[];
```

En la función de ayuda *InitSymbols*, que se llama desde *OnInit* para analizar una cadena con una lista de símbolos de trabajo, añadimos la asignación de memoria para un nuevo array (su tamaño *SymbolCount* se determina a partir de la lista).

```
string InitSymbols()
{
   SymbolCount = StringSplit(Instruments, ',', Symbols);
   ...
   ArrayResize(Handles, SymbolCount);
   ArrayInitialize(Handles, INVALID_HANDLE);
   ...
   for(int i = 0; i < SymbolCount; i++)
   {
      ...
      Handles[i] = iMA(Symbols[i], PERIOD_CURRENT, PricePeriod, 0,
         PriceMethod, PriceType);
   }
}
```

Al final de la misma función, crearemos los descriptores de los indicadores subordinados necesarios.

En la función *Calculate*, donde se realiza el cálculo principal, sustituimos las llamadas de la forma:

```
CopyClose(Symbols[j], _Period, time0, time1, w);
```

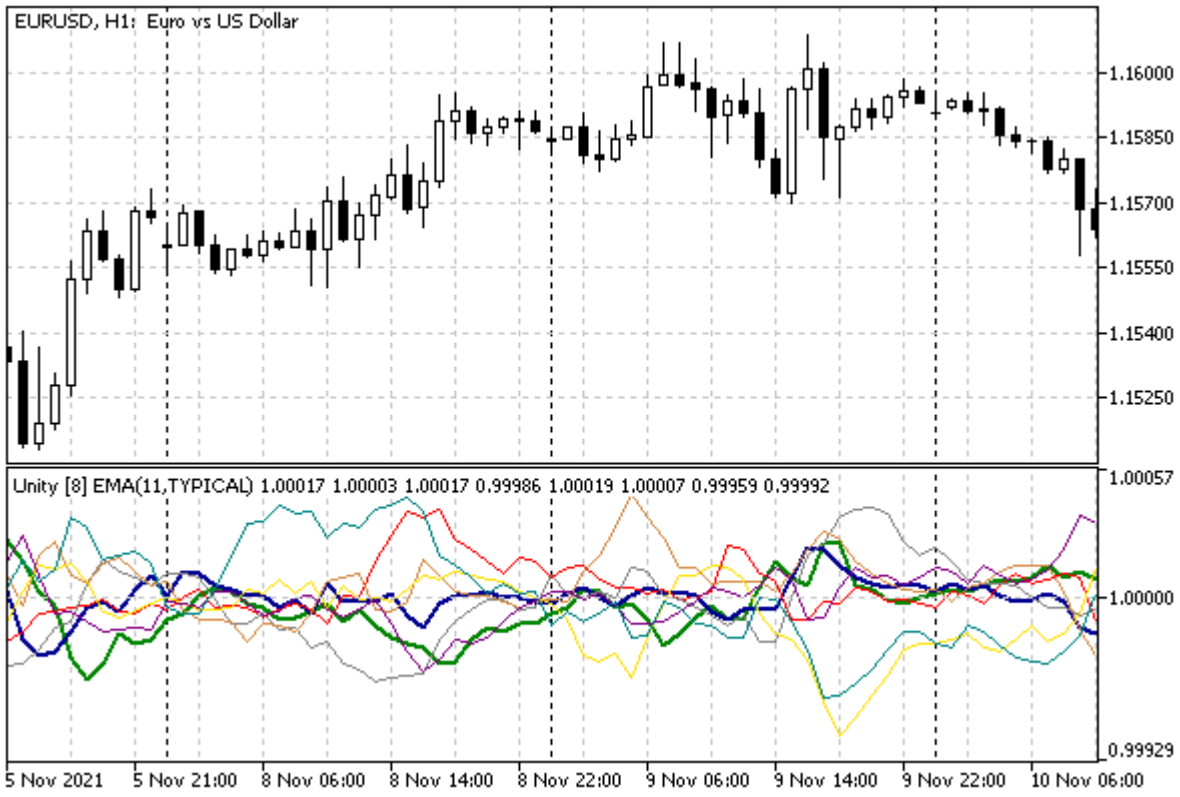
por llamadas:

```
CopyBuffer(Handles[j], 0, time0, time1, w); // j-th handle, 0-th buffer
```

Para mayor claridad, también hemos completado el nombre abreviado del indicador con tres nuevos parámetros:

```
   IndicatorSetString(INDICATOR_SHORTNAME,
      StringFormat("Unity [%d] %s(%d,%s)", workCurrencies.getSize(),
      StringSubstr(EnumToString(PriceMethod), 5), PricePeriod,
      StringSubstr(EnumToString(PriceType), 6)));
```

Esto es lo que ha ocurrido como resultado:



Indicador multisímbolo UseUnityPercentPro con los principales pares Forex

Aquí se muestra una cesta de las 8 principales monedas Forex (configuración por defecto) promediada sobre 11 barras y calculada en base al precio de *typical*. Dos líneas gruesas corresponden al valor relativo de las divisas del gráfico actual: EUR está marcado en azul y USD, en verde.

5.5.8 Forma avanzada de crear indicadores: IndicatorCreate

La creación de un indicador mediante la función *iCustom* o una de las funciones que componen un conjunto de indicadores integrado requiere conocer la lista de parámetros en la fase de codificación. Sin embargo, en la práctica, a menudo es necesario escribir programas lo suficientemente flexibles como para sustituir un indicador por otro.

Por ejemplo, al optimizar un Asesor Experto en el probador, tiene sentido seleccionar no sólo el período de la media móvil, sino también el algoritmo para su cálculo. Por supuesto, si construimos el algoritmo sobre un único indicador *iMA*, puede ofrecer la posibilidad de especificar ENUM_MA_METHOD en la configuración de su método. Sin embargo, a alguien probablemente le gustaría ampliar la elección cambiando entre media móvil doble exponencial, triple exponencial y fractal. A primera vista, esto podría hacerse utilizando *switch* con una llamada de *DEMA*, *iTEMA* y *iFrAMA*, respectivamente. Sin embargo, ¿qué hay acerca de incluir indicadores personalizados en esta lista?

Aunque el nombre del indicador puede sustituirse fácilmente en la llamada a *iCustom*, la lista de parámetros puede diferir significativamente. En el caso general, un Asesor Experto puede necesitar generar señales basadas en una combinación de cualquier indicador que no se conozca de antemano, y no sólo medias móviles.

Para estos casos, MQL5 dispone de un método universal para crear un indicador técnico arbitrario utilizando la función *IndicatorCreate*.

int IndicatorCreate(const string symbol, ENUM_TIMEFRAMES timeframe, ENUM_INDICATOR indicator, int count = 0, const MqlParam &parameters[] = NULL)

La función crea una instancia de indicador para el símbolo y el marco temporal especificados. El tipo de indicador se establece mediante el parámetro *indicator*. Su tipo es la enumeración ENUM_INDICATOR (véase más adelante) que contiene identificadores para todos los indicadores integrados, así como una opción para *iCustom*. El número de parámetros indicadores y sus descripciones se pasan, respectivamente, en el argumento *count* y en el array de estructuras *MqlParam* (véase más abajo).

Cada elemento de este array describe el parámetro de entrada correspondiente del indicador que se está creando, por lo que el contenido y el orden de los elementos deben corresponderse con el prototipo de la función integrada del indicador o, en el caso de un indicador personalizado, con las descripciones de las variables de entrada en su código fuente.

El incumplimiento de esta regla puede dar lugar a un error en la fase de ejecución del programa (véase el ejemplo siguiente) y a la incapacidad de crear un manejador. En el peor de los casos, los parámetros pasados se interpretarán incorrectamente y el indicador no se comportará como se espera; sin embargo, debido a la falta de errores, esto no es fácil de notar. La excepción es pasar un array vacío o no pasarlo en absoluto (porque los argumentos *count* y *parameters* son opcionales): en este caso, el indicador se creará con la configuración predeterminada. Además, en el caso de los indicadores personalizados, puede omitir un número arbitrario de parámetros del final de la lista.

La estructura *MqlParam* ha sido especialmente diseñada para pasar parámetros de entrada cuando se crea un indicador utilizando *IndicatorCreate* o para obtener información sobre los parámetros de un indicador de terceros (realizado en el gráfico) utilizando *IndicatorParameters*.

```
struct MqlParam
{
   ENUM_DATATYPE type;          // input parameter type
   long          integer_value; // field for storing an integer value
   double        double_value;  // field for storing double or float values
   string        string_value;  // field for storing a value of string type
};
```

El valor real del parámetro debe establecerse en uno de los campos *integer_value*, *double_value*, *string_value*, según el valor del primer campo *type*. A su vez, el campo *type* se describe mediante la enumeración ENUM_DATATYPE que contiene identificadores para todos los tipos MQL5 integrados.

Identificador	Tipo de datos
TYPE_BOOL	bool
TYPE_CHAR	char
TYPE_UCHAR	uchar
TYPE_SHORT	short
TYPE_USHORT	ushort
TYPE_COLOR	color
TYPE_INT	int
TYPE_UINT	uint
TYPE_DATETIME	datetime
TYPE_LONG	long
TYPE_ULONG	ulong
TYPE_FLOAT	float
TYPE_DOUBLE	double
TYPE_STRING	string

Si algún parámetro del indicador tiene un tipo de enumeración, debe utilizar el valor TYPE_INT en el campo *type* para describirlo.

La enumeración ENUM_INDICATOR utilizada en el tercer parámetro *IndicatorCreate* para indicar el tipo de indicador contiene las siguientes constantes:

Identificador	Indicador
IND_AC	Oscilador Acelerador
IND_AD	Acumulación/Distribución
IND_ADX	Índice direccional medio
IND_ADXW	ADX de Welles Wilder
IND_ALLIGATOR	Alligator
IND_AMA	Media móvil adaptativa
IND_AO	Oscilador Impresionante
IND_ATR	Rango medio verdadero
IND_BANDS	Bandas de Bollinger
IND_BEARS	Bears Power

Identificador	Indicador
IND_BULLS	Bulls Power
IND_BWMFI	Índice de facilitación del mercado
IND_CCI	Índice del Canal de Materias Primas
IND_CHAIKIN	Oscilador Chaikin
IND_CUSTOM	Indicador personalizado
IND_DEMA	Media móvil exponencial doble
IND_DEMARKER	DeMarker
IND_ENVELOPES	Envelopes
IND_FORCE	Índice de Fuerza
IND_FRACTALS	Fractales
IND_FRAMA	Media móvil adaptativa fractal
IND_GATOR	Oscilador Gator
IND_ICHIMOKU	Ichimoku Kinko Hyo
IND_MA	Media móvil
IND_MACD	MACD
IND_MFI	Índice de flujo de dinero
IND_MOMENTUM	Momentum
IND_OBV	Volumen de balance
IND_OSMA	OsMA
IND_RSI	Índice de Fuerza Relativa
IND_RVI	Índice de Vigor Relativo
IND_SAR	SAR parabólico
IND_STDDEV	Desviación típica
IND_STOCHASTIC	Oscilador estocástico
IND_TEMA	Media móvil exponencial triple
IND_TRIX	Oscilador de medias móviles exponenciales triples
IND_VIDYA	Media Móvil con Periodo de Promediación Dinámico
IND_VOLUMES	Volúmenes
IND_WPR	Rango Porcentual de Williams

Es importante tener en cuenta que si se pasa el valor IND_CUSTOM como tipo de indicador, entonces el primer elemento del array de parámetros debe tener el campo *type* con el valor TYPE_STRING, y el campo *string_value* debe contener el nombre (ruta) del indicador personalizado.

Si tiene éxito, la función *IndicatorCreate* devuelve un manejador del indicador creado, y en caso de fallo devuelve INVALID_HANDLE. El código de error se proporcionará en *_LastError*.

Recuerde que, para probar programas MQL que crean indicadores personalizados cuyos nombres no se conocen en la fase de compilación (lo que también ocurre cuando se utiliza *IndicatorCreate*), debe vincularlos explícitamente mediante la directiva:

```
#property tester_indicator "indicator_name.ex5"
```

Esto permite al probador enviar los indicadores auxiliares necesarios a los agentes de pruebas, pero limita el proceso a sólo los indicadores conocidos de antemano.

Veamos algunos ejemplos. Empecemos con una aplicación sencilla *IndicatorCreate* como alternativa a funciones ya conocidas, y después, para demostrar la flexibilidad del nuevo enfoque, crearemos un indicador universal envolvente para visualizar indicadores arbitrarios integrados o personalizados.

El primer ejemplo de *UseEnvelopesParams1.mq5* crea una copia insertada del indicador *Envelopes*. Para ello, describimos dos búferes, dos trazados, arrays para ellos, y parámetros de entrada que repiten los parámetros de *iEnvelopes*.

```
#property indicator_chart_window
#property indicator_buffers 2
#property indicator_plots   2

// drawing settings
#property indicator_type1   DRAW_LINE
#property indicator_color1  clrBlue
#property indicator_width1  1
#property indicator_label1  "Upper"
#property indicator_style1  STYLE_DOT

#property indicator_type2   DRAW_LINE
#property indicator_color2  clrRed
#property indicator_width2  1
#property indicator_label2  "Lower"
#property indicator_style2  STYLE_DOT

input int WorkPeriod = 14;
input int Shift = 0;
input ENUM_MA_METHOD Method = MODE_EMA;
input ENUM_APPLIED_PRICE Price = PRICE_TYPICAL;
input double Deviation = 0.1; // Deviation, %

double UpBuffer[];
double DownBuffer[];

int Handle; // handle of the subordinate indicator
```

El manejador *OnInit* podría tener este aspecto si utiliza la función *iEnvelopes*:

```
int OnInit()
{
   SetIndexBuffer(0, UpBuffer);
   SetIndexBuffer(1, DownBuffer);

   Handle = iEnvelopes(WorkPeriod, Shift, Method, Price, Deviation);
   return Handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}
```

Los enlaces del búfer seguirán siendo los mismos, pero para crear un manejador, ahora iremos en la otra dirección. Vamos a describir el array *MqlParam*, rellenarlo y llamar a la función *IndicatorCreate*.

```
int OnInit()
{
   ...
   MqlParam params[5] = {};
   params[0].type = TYPE_INT;
   params[0].integer_value = WorkPeriod;
   params[1].type = TYPE_INT;
   params[1].integer_value = Shift;
   params[2].type = TYPE_INT;
   params[2].integer_value = Method;
   params[3].type = TYPE_INT;
   params[3].integer_value = Price;
   params[4].type = TYPE_DOUBLE;
   params[4].double_value = Deviation;
   Handle = IndicatorCreate(_Symbol, _Period, IND_ENVELOPES,
      ArraySize(params), params);
   return Handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}
```

Una vez recibido el manejador, lo utilizamos en *OnCalculate* para rellenar dos de sus búferes.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   if(BarsCalculated(Handle) != rates_total)
   {
      return prev_calculated;
   }

   const int n = CopyBuffer(Handle, 0, 0, rates_total - prev_calculated + 1, UpBuffer
   const int m = CopyBuffer(Handle, 1, 0, rates_total - prev_calculated + 1, DownBuff

   return n > -1 && m > -1 ? rates_total : 0;
}
```

Comprobemos cómo se ve en el gráfico el indicador creado *UseEnvelopesParams1*:

Indicador UseEnvelopesParams1

Arriba se ha mostrado una forma estándar pero no muy elegante de poblar las propiedades. Dado que la llamada a *IndicatorCreate* puede ser necesaria en muchos proyectos, tiene sentido simplificar el procedimiento para el código de llamada. Para este fin desarrollaremos una clase denominada *MqlParamBuilder* (véase el archivo *MqlParamBuilder.mqh*). Su tarea será aceptar valores de parámetros utilizando algunos métodos, determinar su tipo y añadir los elementos apropiados (estructuras correctamente rellenadas) al array.

MQL5 no soporta completamente el concepto de información sobre los tipos de tiempo de ejecución (Run-Time Type Information, RTTI). Con esto, los programas pueden pedir al tiempo de ejecución metadatos descriptivos sobre sus partes constituyentes, incluidas variables, estructuras, clases, funciones, etc. Las pocas características integradas de MQL5 que pueden clasificarse como RTTI son los operadores typename y offsetof. Dado que *typename* devuelve el nombre del tipo como una cadena, vamos a construir nuestro autodetector de tipos en cadenas (véase el archivo *RTTI.mqh*).

```
template<typename T>
ENUM_DATATYPE rtti(T v = (T)NULL)
{
   static string types[] =
   {
      "null",     //               (0)
      "bool",     // 0 TYPE_BOOL=1 (1)
      "char",     // 1 TYPE_CHAR=2 (2)
      "uchar",    // 2 TYPE_UCHAR=3 (3)
      "short",    // 3 TYPE_SHORT=4 (4)
      "ushort",   // 4 TYPE_USHORT=5 (5)
      "color",    // 5 TYPE_COLOR=6 (6)
      "int",      // 6 TYPE_INT=7 (7)
      "uint",     // 7 TYPE_UINT=8 (8)
      "datetime", // 8 TYPE_DATETIME=9 (9)
      "long",     // 9 TYPE_LONG=10 (A)
      "ulong",    // 10 TYPE_ULONG=11 (B)
      "float",    // 11 TYPE_FLOAT=12 (C)
      "double",   // 12 TYPE_DOUBLE=13 (D)
      "string",   // 13 TYPE_STRING=14 (E)
   };
   const string t = typename(T);
   for(int i = 0; i < ArraySize(types); ++i)
   {
      if(types[i] == t)
      {
         return (ENUM_DATATYPE)i;
      }
   }
   return (ENUM_DATATYPE)0;
}
```

La función de plantilla *rtti* utiliza *typename* para recibir una cadena con el nombre del parámetro de tipo de plantilla y la compara con los elementos de un array que contiene todos los tipos integrados de la enumeración ENUM_DATATYPE. El orden de enumeración de los nombres en el array corresponde al valor del elemento de enumeración, por lo que cuando se encuentra una cadena coincidente, basta con convertir el índice al tipo (ENUM_DATATYPE) y devolverlo al código de llamada. Por ejemplo, la llamada a *rtti(1.0)* o *rtti<double>()* dará el valor TYPE_DOUBLE.

Con esta herramienta, podemos volver a trabajar en *MqlParamBuilder*. En la clase se describe el array de estructuras *MqlParam* y la variable *n* que contendrá el índice del último elemento que se va a rellenar.

```
class MqlParamBuilder
{
protected:
   MqlParam array[];
   int n;
   ...
```

Hagamos que el método público para añadir el siguiente valor a la lista de parámetros sea una plantilla. Además, lo implementamos como una sobrecarga del operador '<<' , que devuelve un puntero al propio

objeto «constructor». Esto permitirá escribir múltiples valores en el array en una sola línea; por ejemplo, así: *builder << WorkPeriod << PriceType << SmoothingMode*.

Es en este método que aumentamos el tamaño del array, obtenemos el índice de trabajo *n* para rellenar, e inmediatamente restablecemos esta estructura *n*--ésima.

```
...
public:
   template<typename T>
   MqlParamBuilder *operator<<(T v)
   {
 // expand the array
      n = ArraySize(array);
      ArrayResize(array, n + 1);
      ZeroMemory(array[n]);
      ...
      return &this;
   }
```

Donde haya una elipsis, seguirá la parte principal de trabajo, es decir, rellenar los campos de la estructura. Se podría suponer que determinaremos directamente el tipo del parámetro utilizando un *rtti* de elaboración propia. Pero debe prestar atención a un matiz: si escribimos las instrucciones *array[n].type = rtti(v)*, no funcionará correctamente para las enumeraciones. Cada enumeración es un tipo independiente con su propio nombre, a pesar de que se almacena de la misma forma que los enteros. En el caso de las enumeraciones, la función *rtti* devolverá 0, por lo que deberá sustituirla explícitamente por TYPE_INT.

```
      ...
      // define value type
      array[n].type = rtti(v);
      if(array[n].type == 0) array[n].type = TYPE_INT; // imply enum
      ...
```

Ahora sólo tenemos que poner el valor *v* en uno de los tres campos de la estructura: *integer_value* del tipo *long* (nota: *long* es un entero largo, de ahí el nombre del campo), *double_value* de tipo *double* o *string_value* de tipo *string*. Mientras tanto, el número de tipos integrados es mucho mayor, por lo que se asume que todos los tipos integrales (incluyendo *int*, *short*, *char*, *color*, *datetime* y enumeraciones) deben caer en el campo *integer_value*, los valores *float* deben caer en el campo *double_value*, y sólo para el campo *string_value* tiene una interpretación inequívoca: siempre es *string*.

Para llevar a cabo esta tarea, implementamos varios métodos *assign* sobrecargados: tres con tipos específicos de *float*, *double* y *string*, y una plantilla para todo lo demás.

```
class MqlParamBuilder
{
protected:
   ...
   void assign(const float v)
   {
      array[n].double_value = v;
   }

   void assign(const double v)
   {
      array[n].double_value = v;
   }

   void assign(const string v)
   {
      array[n].string_value = v;
   }

   // here we process int, enum, color, datetime, etc. compatible with long
   template<typename T>
   void assign(const T v)
   {
      array[n].integer_value = v;
   }
   ...
```

Esto completa el proceso de llenado de estructuras, y queda la cuestión de pasar el array generado para el código de llamada. Esta acción se asigna a un método público con una sobrecarga del operador '>>', que tiene un único argumento: una referencia al array receptor *MqlParam*.

```
   // export the inner array to the outside
   void operator>>(MqlParam &params[])
   {
      ArraySwap(array, params);
   }
```

Ahora que todo está listo, podemos trabajar con el código fuente del indicador modificado *UseEnvelopesParams2.mq5*. Los cambios comparados con la primera versión sólo afectan al rellenado del array *MqlParam* en el manejador *OnInit*. En él, describimos el objeto «constructor», le enviamos todos los parámetros mediante '<<' y devolvemos el array terminado mediante '>>'. Todo se hace en una sola línea.

```
int OnInit()
{
   ...
   MqlParam params[];
   MqlParamBuilder builder;
   builder << WorkPeriod << Shift << Method << Price << Deviation >> params;
   ArrayPrint(params);
   /*
       [type] [integer_value] [double_value] [string_value]
   [0]      7              14        0.00000 null            <- "INT" period
   [1]      7               0        0.00000 null            <- "INT" shift
   [2]      7               1        0.00000 null            <- "INT" EMA
   [3]      7               6        0.00000 null            <- "INT" TYPICAL
   [4]     13               0        0.10000 null            <- "DOUBLE" deviation
   */
```

Para el control, enviamos el array al registro (arriba se muestra el resultado para los valores por defecto).

Si el array no está completamente lleno, la llamada a *IndicatorCreate* finalizará con un error. Por ejemplo, si pasa sólo 3 parámetros de los 5 requeridos para *Envelopes*, obtendrá el error 4002 y un manejador no válido.

```
   Handle = PRTF(IndicatorCreate(_Symbol, _Period, IND_ENVELOPES, 3, params));
   // Error example:
   // indicator Envelopes cannot load [4002]
   // IndicatorCreate(_Symbol,_Period,IND_ENVELOPES,3,params)=
      -1 / WRONG_INTERNAL_PARAMETER(4002)
```

Sin embargo, un array más largo que en la especificación del indicador no se considera un error: los valores adicionales simplemente no se tienen en cuenta.

Tenga en cuenta que cuando los tipos de valores difieren de los tipos de parámetros esperados, el sistema realiza una conversión implícita, y esto no genera errores obvios, aunque el indicador generado puede no funcionar como se esperaba. Por ejemplo, si en lugar de *Deviation* enviamos una cadena al indicador, se interpretará como el número 0, como resultado de lo cual el «sobre» se colapsará: ambas líneas se alinearán en la línea central, respecto a la cual la sangría se realiza por el tamaño de *Deviation* (en porcentajes). Del mismo modo, pasar un número real con una parte fraccionaria en un parámetro donde se espera un entero hará que se redondee.

Pero, por supuesto, dejar la versión correcta de la llamada *IndicatorCreate* y obtener un indicador de trabajo, al igual que en la primera versión.

```
   ...
   Handle = PRTF(IndicatorCreate(_Symbol, _Period, IND_ENVELOPES,
      ArraySize(params), params));
   // success:
   // IndicatorCreate(_Symbol,_Period,IND_ENVELOPES,ArraySize(params),params)=10 / ok
   return Handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}
```

Por su aspecto, el nuevo indicador no difiere del anterior.

5.5.9 Creación flexible de indicadores con IndicatorCreate

Después de familiarizarnos con una nueva forma de crear indicadores, pasemos a una tarea más cercana a la realidad. *IndicatorCreate* suele utilizarse en los casos en que el indicador llamado no se conoce de antemano. Tal necesidad, por ejemplo, surge al escribir Asesores Expertos universales capaces de operar con señales arbitrarias configuradas por el usuario. E incluso los nombres de los indicadores pueden ser fijados por el usuario.

Todavía no estamos preparados para desarrollar Asesores Expertos, por lo que estudiaremos esta tecnología utilizando el ejemplo de un indicador envolvente *UseDemoAll.mq5*, capaz de mostrar los datos de cualquier otro indicador.

El proceso debería tener el siguiente aspecto. Cuando ejecutamos *UseDemoAll* en el gráfico, aparece una lista en el cuadro de diálogo de propiedades en la que debemos seleccionar uno de los indicadores integrados o uno personalizado, y en este último caso, además, tendremos que especificar su nombre en el campo de entrada. En otro parámetro de cadena, podemos introducir una lista de parámetros separados por comas. Los tipos de parámetros se determinarán automáticamente en función de su ortografía. Por ejemplo, un número con punto decimal (10.0) se tratará como un doble, un número sin punto (15) como un entero y algo encerrado entre comillas («texto») como una cadena.

Estos son sólo los ajustes básicos de *UseDemoAll*, pero no todos los posibles. Más adelante estudiaremos otras configuraciones.

Tomemos la enumeración ENUM_INDICATOR como base para la solución: ya dispone de elementos para todos los tipos de indicadores, incluidos los personalizados (IND_CUSTOM). A decir verdad, en su forma pura, no encaja por varias razones. En primer lugar, es imposible obtener de él metadatos sobre un indicador concreto, como el número y los tipos de argumentos, el número de búferes y en qué ventana se muestra el indicador (principal o subventana). Esta información es importante para la correcta creación y visualización del indicador. En segundo lugar, si definimos una variable de entrada del tipo ENUM_INDICATOR para que el usuario pueda seleccionar el indicador deseado, en el cuadro de diálogo de propiedades éste se representará mediante una lista desplegable, donde las opciones contienen únicamente el nombre del elemento. En realidad, sería deseable proporcionar pistas al usuario en esta lista (al menos sobre los parámetros). Por ello, describiremos nuestra propia enumeración *IndicatorType*. Recordemos que MQL5 permite que cada elemento especifique un comentario a la derecha, que se muestra en la interfaz.

En cada elemento de la enumeración *IndicatorType*, codificaremos no sólo el identificador (ID) correspondiente de ENUM_INDICATOR, sino también el número de parámetros (P), el número de búferes (B) y el número de la ventana de trabajo (W). Para ello se han desarrollado las siguientes macros:

```
#define MAKE_IND(P,B,W,ID) (int)((W << 24) | ((B & 0xFF) << 16) | ((P & 0xFF) << 8) |
#define IND_PARAMS(X)   ((X >> 8) & 0xFF)
#define IND_BUFFERS(X)  ((X >> 16) & 0xFF)
#define IND_WINDOW(X)   ((uchar)(X >> 24))
#define IND_ID(X)       ((ENUM_INDICATOR)(X & 0xFF))
```

La macro MAKE_IND toma todas las características anteriores como parámetros y las empaqueta en diferentes bytes de un único entero de 4 bytes, formando así un código único para el elemento de la nueva enumeración. Las 4 macros restantes permiten realizar la operación inversa, es decir, calcular todas las características del indicador utilizando el código.

No proporcionaremos aquí toda la enumeración de *IndicatorType*, sino sólo una parte. El código fuente completo se encuentra en el archivo *AutoIndicator.mqh*.

```
enum IndicatorType
{
   iCustom_ = MAKE_IND(0, 0, 0, IND_CUSTOM), // {iCustom}(...)[?]

   iAC_ = MAKE_IND(0, 1, 1, IND_AC), // iAC( )[1]*
   iAD_volume = MAKE_IND(1, 1, 1, IND_AD), // iAD(volume)[1]*
   iADX_period = MAKE_IND(1, 3, 1, IND_ADX), // iADX(period)[3]*
   iADXWilder_period = MAKE_IND(1, 3, 1, IND_ADXW), // iADXWilder(period)[3]*
   ...
   iMomentum_period_price = MAKE_IND(2, 1, 1, IND_MOMENTUM), // iMomentum(period,pric
   iMFI_period_volume = MAKE_IND(2, 1, 1, IND_MFI), // iMFI(period,volume)[1]*
   iMA_period_shift_method_price = MAKE_IND(4, 1, 0, IND_MA), // iMA(period,shift,met
   iMACD_fast_slow_signal_price = MAKE_IND(4, 2, 1, IND_MACD), // iMACD(fast,slow,sig
   ...
   iTEMA_period_shift_price = MAKE_IND(3, 1, 0, IND_TEMA), // iTEMA(period,shift,pric
   iVolumes_volume = MAKE_IND(1, 1, 1, IND_VOLUMES), // iVolumes(volume)[1]*
   iWPR_period = MAKE_IND(1, 1, 1, IND_WPR) // iWPR(period)[1]*
};
```

Los comentarios, que se convertirán en elementos de la lista desplegable visible para el usuario, indican prototipos con parámetros con nombre, el número de búferes entre corchetes y las marcas de estrella de los indicadores que se muestran en su propia ventana. Los propios identificadores también se convierten en informativos, ya que son los que convierte en texto la función *EnumToString* que se utiliza para enviar mensajes al registro.

La lista de parámetros es especialmente importante, ya que el usuario tendrá que introducir los valores apropiados separados por comas en la variable de entrada reservada a tal efecto. También podríamos mostrar los tipos de los parámetros, pero por simplicidad, se ha decidido dejar sólo los nombres con un significado, del que también se puede deducir el tipo. Por ejemplo, *period*, *fast*, *slow* son enteros con un periodo (número de barras), *method* es el método de cálculo de medias ENUM_MA_METHOD, *price* es el tipo de precio ENUM_APPLIED_PRICE, *volume* es el tipo de volumen ENUM_APPLIED_VOLUME.

Para comodidad del usuario (de manera que no tenga que recordar los valores de los elementos de enumeración), el programa admitirá los nombres de todas las enumeraciones. En concreto, el identificador *sma* denota MODO_SMA, *ema* denota MODO_EMA, y así sucesivamente. El precio *close* se convertirá en PRICE_CLOSE, *open* se convertirá en PRICE_OPEN, y otros tipos de precios se comportarán igual, por la última palabra (después del subrayado) en el identificador del elemento de enumeración. Por ejemplo, para la lista de parámetros del indicador iMA (iMA_period_shift_method_price), puede escribir la siguiente línea: *11,0,sma,close*. No es necesario entrecomillar los identificadores. Sin embargo, si es necesario, puede pasar una cadena con el mismo texto, por ejemplo, una lista *1.5,"close"* contiene el número real 1.5 y la cadena «cerrar».

El tipo de indicador, así como cadenas con una lista de parámetros y, opcionalmente, un nombre (si el indicador es personalizado) son los datos principales para el constructor de la clase *AutoIndicator*.

```
class AutoIndicator
{
protected:
   IndicatorTypetype;       // selected indicator type
   string symbols;          // working symbol (optional)
   ENUM_TIMEFRAMES tf;      // working timeframe (optional)
   MqlParamBuilder builder; // "builder" of the parameter array
   int handle;              // indicator handle
   string name;             // custom indicator name
   ...
public:
   AutoIndicator(const IndicatorType t, const string custom, const string parameters,
      const string s = NULL, const ENUM_TIMEFRAMES p = 0):
      type(t), name(custom), symbol(s), tf(p), handle(INVALID_HANDLE)
   {
      PrintFormat("Initializing %s(%s) %s, %s",
         (type == iCustom_ ? name : EnumToString(type)), parameters,
         (symbol == NULL ? _Symbol : symbol), EnumToString(tf == 0 ? _Period : tf));
      // split the string into an array of parameters (formed inside the builder)
      parseParameters(parameters);
      // create and store the handle
      handle = create();
   }

   int getHandle() const
   {
      return handle;
   }
};
```

Aquí y más abajo se omiten algunos fragmentos relacionados con la comprobación de la corrección de los datos de entrada. El libro incluye el código fuente completo.

El proceso de análisis de una cadena con parámetros se confía al método *parseParameters*, que implementa el esquema descrito anteriormente con el reconocimiento de los tipos de valor y su transferencia a un objeto *MqlParamBuilder*, que conocimos en el ejemplo anterior.

```
int parseParameters(const string &list)
{
   string sparams[];
   const int n = StringSplit(list, ',', sparams);

   for(int i = 0; i < n; i++)
   {
      // normalization of the string (remove spaces, convert to lower case)
      StringTrimLeft(sparams[i]);
      StringTrimRight(sparams[i]);
      StringToLower(sparams[i]);

      if(StringGetCharacter(sparams[i], 0) == '"'
      && StringGetCharacter(sparams[i], StringLen(sparams[i]) - 1) == '"')
      {
         // everything inside quotes is taken as a string
         builder << StringSubstr(sparams[i], 1, StringLen(sparams[i]) - 2);
      }
      else
      {
         string part[];
         int p = StringSplit(sparams[i], '.', part);
         if(p == 2) // double/float
         {
            builder << StringToDouble(sparams[i]);
         }
         else if(p == 3) // datetime
         {
            builder << StringToTime(sparams[i]);
         }
         else if(sparams[i] == "true")
         {
            builder << true;
         }
         else if(sparams[i] == "false")
         {
            builder << false;
         }
         else // int
         {
            int x = lookUpLiterals(sparams[i]);
            if(x == -1)
            {
               x = (int)StringToInteger(sparams[i]);
            }
            builder << x;
         }
      }
   }

   return n;
```

```
}
```

La función de ayuda *lookUpLiterals* proporciona la conversión de identificadores a constantes de enumeración estándar.

```
int lookUpLiterals(const string &s)
{
   if(s == "sma") return MODE_SMA;
   else if(s == "ema") return MODE_EMA;
   else if(s == "smma") return MODE_SMMA;
   else if(s == "lwma") return MODE_LWMA;

   else if(s == "close") return PRICE_CLOSE;
   else if(s == "open") return PRICE_OPEN;
   else if(s == "high") return PRICE_HIGH;
   else if(s == "low") return PRICE_LOW;
   else if(s == "median") return PRICE_MEDIAN;
   else if(s == "typical") return PRICE_TYPICAL;
   else if(s == "weighted") return PRICE_WEIGHTED;

   else if(s == "lowhigh") return STO_LOWHIGH;
   else if(s == "closeclose") return STO_CLOSECLOSE;

   else if(s == "tick") return VOLUME_TICK;
   else if(s == "real") return VOLUME_REAL;

   return -1;
}
```

Una vez reconocidos y guardados los parámetros en el array interno del objeto *MqlParamBuilder*, se llama al método *create*. Su propósito es copiar los parámetros al array local, complementarlo con el nombre del indicador personalizado (si lo hay), y llamar a la función *IndicatorCreate*.

```
   int create()
   {
      MqlParam p[];
      // fill 'p' array with parameters collected by 'builder' object
      builder >> p;

      if(type == iCustom_)
      {
         // insert the name of the custom indicator at the very beginning
         ArraySetAsSeries(p, true);
         const int n = ArraySize(p);
         ArrayResize(p, n + 1);
         p[n].type = TYPE_STRING;
         p[n].string_value = name;
         ArraySetAsSeries(p, false);
      }

      return IndicatorCreate(symbol, tf, IND_ID(type), ArraySize(p), p);
   }
```

El método devuelve el manejador recibido.

Resulta especialmente interesante la forma en que se inserta un parámetro de cadena adicional con el nombre del indicador personalizado al principio del array. En primer lugar, el array se asigna al orden de indexación «como en las series temporales» (véase *ArraySetAsSeries*), por lo que el índice del último elemento (físicamente, por ubicación en memoria) pasa a ser igual a 0, y los elementos se cuentan de derecha a izquierda. A continuación, se aumenta el tamaño del array y se escribe el nombre del indicador en el elemento añadido. Debido a la indexación inversa, esta adición no se produce a la derecha de los elementos existentes, sino a la izquierda. Por último, devolvemos el array a su orden de indexación habitual, y en el índice 0 está el nuevo elemento con la cadena que acaba de ser la última.

Opcionalmente, la clase *AutoIndicator* puede formar un nombre abreviado del indicador integrado a partir del nombre de un elemento de enumeración.

```
   ...
   string getName() const
   {
      if(type != iCustom_)
      {
         const string s = EnumToString(type);
         const int p = StringFind(s, "_");
         if(p > 0) return StringSubstr(s, 0, p);
         return s;
      }
      return name;
   }
};
```

Ahora todo está listo para ir directamente al código fuente *UseDemoAll.mq5*. Pero empecemos con una versión ligeramente simplificada *UseDemoAllSimple.mq5*.

En primer lugar, vamos a definir el número de búferes de indicadores. Dado que el número máximo de búferes entre los indicadores integrados es de cinco (para *Ichimoku*), lo tomamos como limitador.

Asignaremos el registro de este número de arrays como búferes a la clase que ya conocemos, *BufferArray* (véase la sección Indicadores multidivisa y de marco temporal múltiple, ejemplo *IndUnityPercent*).

```
#define BUF_NUM 5

#property indicator_chart_window
#property indicator_buffers BUF_NUM
#property indicator_plots   BUF_NUM

#include <MQL5Book/IndBufArray.mqh>

BufferArray buffers(5);
```

Es importante recordar que un indicador puede diseñarse para que aparezca en la ventana principal o en una ventana aparte. MQL5 no permite combinar dos modos. No obstante, no sabemos de antemano qué indicador elegirá el usuario, por lo que tenemos que inventar algún tipo de «solución». Por ahora, vamos a colocar nuestro indicador en la ventana principal, y abordaremos el problema de una ventana separada más adelante.

Desde un punto de vista puramente técnico, no hay ningún obstáculo para copiar los datos de los búferes de indicadores con la propiedad *indicator_separate_window* en sus búferes que se muestran en la ventana principal. Sin embargo, hay que tener en cuenta que el rango de valores de estos indicadores a menudo no coincide con la escala de precios, por lo que es poco probable que pueda verlos en el gráfico (las líneas estarán en algún lugar más allá de la zona visible, en la parte superior o inferior), aunque los valores se seguirán mostrando en *Data window*.

Con la ayuda de las variables de entrada, seleccionaremos el tipo de indicador, el nombre del indicador personalizado y la lista de parámetros. También añadiremos variables para el tipo de representación y el ancho de línea. Dado que los búferes se conectarán para funcionar dinámicamente, dependiendo del número de búferes del indicador fuente, no describiremos los estilos de búfer de forma estática utilizando directivas y lo haremos en *OnInit* mediante llamadas a las funciones *Plot* integradas.

```
input IndicatorType IndicatorSelector = iMA_period_shift_method_price; // Built-in In
input string IndicatorCustom = ""; // Custom Indicator Name
input string IndicatorParameters = "11,0,sma,close"; // Indicator Parameters (comma,s
input ENUM_DRAW_TYPE DrawType = DRAW_LINE; // Drawing Type
input int DrawLineWidth = 1; // Drawing Line Width
```

Vamos a definir una variable global para almacenar el descriptor del indicador.

```
int Handle;
```

En el manejador *OnInit*, utilizamos la clase *AutoIndicator* presentada anteriormente, para analizar un dato de entrada, preparar el array *MqlParam* y obtener un manejador basado en él.

```
#include <MQL5Book/AutoIndicator.mqh>

int OnInit()
{
   AutoIndicator indicator(IndicatorSelector, IndicatorCustom, IndicatorParameters);
   Handle = indicator.getHandle();
   if(Handle == INVALID_HANDLE)
   {
      Alert(StringFormat("Can't create indicator: %s",
         _LastError ? E2S(_LastError) : "The name or number of parameters is incorrec
      return INIT_FAILED;
   }
   ...
```

Para personalizar los trazados, describimos un conjunto de colores y obtenemos el nombre abreviado del indicador del objeto *AutoIndicator*. También calculamos el número de búferes *n* utilizados del indicador integrado utilizando la macro IND_BUFFERS, y para cualquier indicador personalizado (que no se conoce de antemano), a falta de una solución mejor, incluiremos todos los búferes. Además, en el proceso de copia de datos, las llamadas innecesarias a *CopyBuffer* simplemente devolverán un error, y dichos arrays pueden llenarse con valores vacíos.

```
   ...
   static color defColors[BUF_NUM] = {clrBlue, clrGreen, clrRed, clrCyan, clrMagenta}
   const string s = indicator.getName();
   const int n = (IndicatorSelector != iCustom_) ? IND_BUFFERS(IndicatorSelector) : B
   ...
```

En el bucle, estableceremos las propiedades de los gráficos, teniendo en cuenta el limitador n: los búferes por encima de él están ocultos.

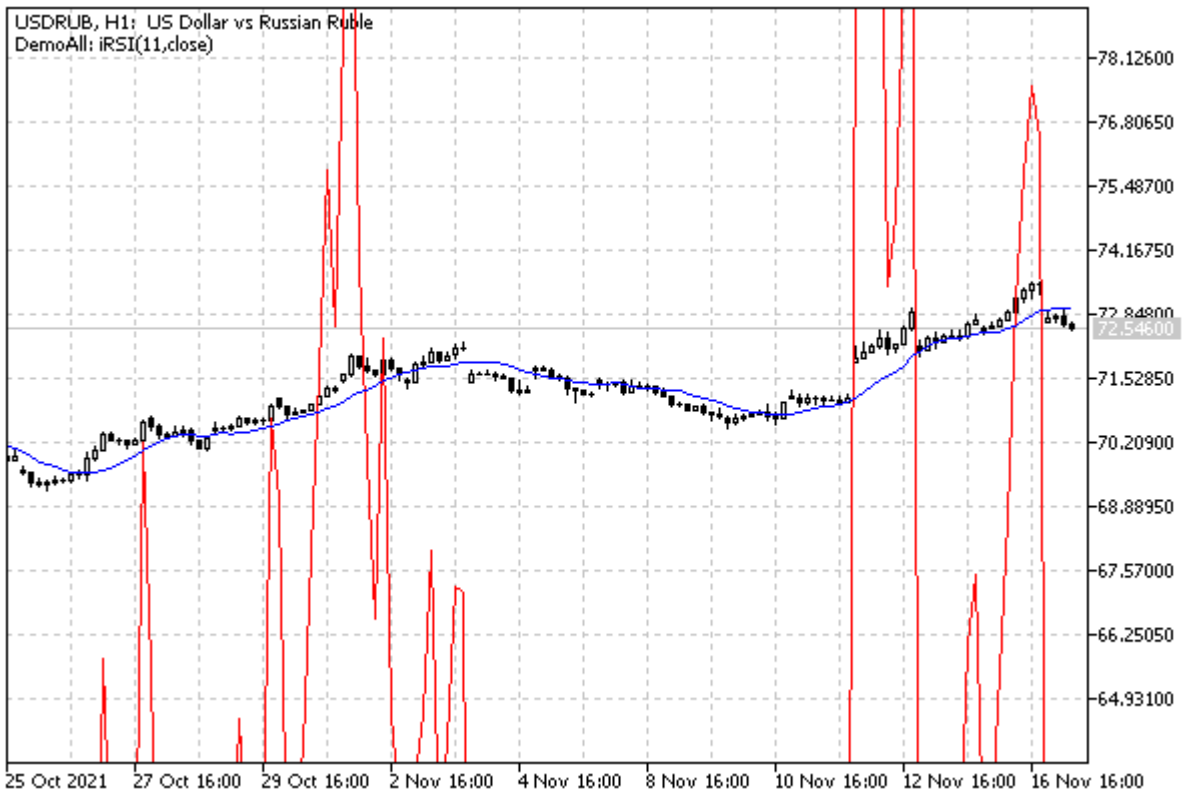
```
   for(int i = 0; i < BUF_NUM; ++i)
   {
      PlotIndexSetString(i, PLOT_LABEL, s + "[" + (string)i + "]");
      PlotIndexSetInteger(i, PLOT_DRAW_TYPE, i < n ? DrawType : DRAW_NONE);
      PlotIndexSetInteger(i, PLOT_LINE_WIDTH, DrawLineWidth);
      PlotIndexSetInteger(i, PLOT_LINE_COLOR, defColors[i]);
      PlotIndexSetInteger(i, PLOT_SHOW_DATA, i < n);
   }

   Comment("DemoAll: ", (IndicatorSelector == iCustom_ ? IndicatorCustom : s),
      "(", IndicatorParameters, ")");

   return INIT_SUCCEEDED;
}
```

En la esquina superior izquierda del gráfico, el comentario mostrará el nombre del indicador con los parámetros.

En el manejador *OnCalculate*, cuando los datos del manejador están listos, los leemos en nuestros arrays.

```
int OnCalculate(ON_CALCULATE_STD_SHORT_PARAM_LIST)
{
   if(BarsCalculated(Handle) != rates_total)
   {
      return prev_calculated;
   }

   const int m = (IndicatorSelector != iCustom_) ? IND_BUFFERS(IndicatorSelector) : B
   for(int k = 0; k < m; ++k)
   {
      // fill our buffers with data form the indicator with the 'Handle' handle
      const int n = buffers[k].copy(Handle,
         k, 0, rates_total - prev_calculated + 1);

      // in case of error clean the buffer
      if(n < 0)
      {
         buffers[k].empty(EMPTY_VALUE, prev_calculated, rates_total - prev_calculated
      }
   }

   return rates_total;
}
```

La implementación anterior está simplificada y coincide con el archivo original *UseDemoAllSimple.mq5*. Nos ocuparemos de su ampliación más adelante, pero por ahora comprobaremos el comportamiento de la versión actual. En la siguiente imagen se muestran 2 instancias del indicador: la línea azul con la configuración por defecto (*iMA_period_shift_method_price*, opciones «*11,0,sma,close*»), y la roja *iRSI_period_price* con los parámetros «*11 close*»:



Dos instancias del indicador UseDemoAllSimple con lecturas iMA e iRSI

El gráfico USDRUB se ha elegido de forma intencionada para la demostración, porque los valores de las cotizaciones aquí coinciden más o menos con el rango del indicador RSI (que debería haberse mostrado en una ventana aparte). En la mayoría de los gráficos de otros símbolos, no notaríamos el RSI. Si sólo le importa el acceso programático a los valores, entonces esto no es gran cosa, pero si tiene requisitos de visualización, se trata de un problema que debe resolverse.

Por lo tanto, debe proporcionar de alguna manera una visualización por separado de los indicadores destinados a la subventana. Básicamente, existe una petición popular de la comunidad de desarrolladores de MQL para permitir la visualización de gráficos tanto en la ventana principal como en una subventana al mismo tiempo. Vamos a presentar una de las soluciones, pero para ello primero tiene que familiarizarse con algunas de las nuevas funciones.

5.5.10 Visión general de las funciones de gestión de indicadores en el gráfico

Como ya nos hemos dado cuenta, los indicadores son el tipo de programas MQL que combinan la parte de cálculo y la visualización. Los cálculos se realizan internamente, de forma imperceptible para el usuario, pero la visualización requiere enlazar con el gráfico. Por eso los indicadores están estrechamente relacionados con los gráficos, y la API de MQL5 contiene incluso un grupo de funciones que gestionan los indicadores en los gráficos. Hablaremos de estas funciones con más detalle en el capítulo sobre gráficos y aquí sólo damos una lista de los mismos.

Función	Propósito
ChartWindowFind	Devuelve el número de la subventana que contiene el indicador actual o un indicador con el nombre dado.
ChartIndicatorAdd	Añade un indicador con el manejador especificado a la ventana del gráfico especificada.
ChartIndicatorDelete	Elimina un indicador con el nombre especificado de la ventana del gráfico especificada.
ChartIndicatorGet	Devuelve el manejador del indicador con el nombre corto especificado en la ventana del gráfico especificada.
ChartIndicatorName	Devuelve el nombre abreviado del indicador por número en la lista de indicadores de la ventana del gráfico especificada.
ChartIndicatorsTotal	Devuelve el número de todos los indicadores adjuntos a la ventana del gráfico especificada.

En la siguiente sección sobre Combinar salida de información en la ventana principal y auxiliar, veremos un ejemplo de *UseDemoAll.mq5*, que utiliza algunas de estas funciones.

5.5.11 Combinar salida a ventanas principal y auxiliar

Volvamos al problema de mostrar gráficos de un indicador en la ventana principal y en una subventana, ya que nos lo encontramos al desarrollar el ejemplo de *UseDemoAllSimple.mq5*. Ahí descubrimos que los indicadores concebidos para una ventana independiente no son adecuados para la visualización en el

gráfico principal, y los indicadores para la ventana principal no tienen ventanas adicionales. Existen varios enfoques alternativos:

- Implementar un indicador padre para una ventana separada y mostrar gráficos allí y usarlo en la ventana principal para mostrar datos del tipo objetos gráficos. Esto es malo, porque los datos de los objetos no se pueden leer de la misma manera que los de una serie temporal, y muchos objetos consumen recursos adicionales.
- Desarrolle su propio panel virtual (clase) para la ventana principal y, con la escala correcta, represente allí las series temporales que deben mostrarse en la subventana.
- Utilice varios indicadores, al menos uno para la ventana principal y otro para la subventana, e intercambie datos entre ellos a través de la memoria compartida (se requiere DLL), recursos o base de datos.
- Duplicar los cálculos (utilizar código fuente común) en los indicadores de la ventana principal y de la subventana.

Presentaremos una de las soluciones que va más allá de un único programa MQL: necesitamos un indicador adicional con la propiedad *indicator_separate_window*. En realidad ya lo tenemos, puesto que creamos su parte calculada solicitando un manejador. Sólo tenemos que mostrarlo de alguna manera en una subventana separada.

En la nueva versión (completa) de *UseDemoAll.mq5*, analizaremos los metadatos del indicador cuya creación se solicita en el elemento de enumeración *IndicatorType* correspondiente. Recordemos que, entre otras cosas, allí se codifica la ventana de trabajo de cada tipo de indicador integrado. Cuando un indicador requiera una ventana separada, crearemos una usando funciones especiales de MQL5, que todavía tenemos que descubrir.

No hay forma de obtener información sobre la ventana de trabajo de los indicadores personalizados. Por lo tanto, vamos a añadir la variable de entrada *IndicatorCustomSubwindow*, en la que el usuario puede especificar que se requiere una subventana.

```
input bool IndicatorCustomSubwindow = false; // Custom Indicator Subwindow
```

En *OnInit*, ocultamos los búferes destinados a la subventana.

```
int OnInit()
{
   ...
   const bool subwindow = (IND_WINDOW(IndicatorSelector) > 0)
      || (IndicatorSelector == iCustom_ && IndicatorCustomSubwindow);
   for(int i = 0; i < BUF_NUM; ++i)
   {
      ...
      PlotIndexSetInteger(i, PLOT_DRAW_TYPE,
         i < n && !subwindow ? DrawType : DRAW_NONE);
   }
   ...
```

Después de esta configuración, tendremos que utilizar un par de funciones que se aplican al trabajo no sólo con indicadores, sino también con gráficos. Los estudiaremos en detalle en el capítulo correspondiente, mientras que en la sección anterior se presenta una visión general introductoria.

Una de las funciones *ChartIndicatorAdd* permite añadir el indicador especificado por el manejador a la ventana, y no sólo a la parte principal, sino también a la subventana. Hablaremos de los identificadores

de gráficos y de la numeración de ventanas en el capítulo sobre gráficos, y por ahora es suficiente saber que la siguiente llamada a la función *ChartIndicatorAdd* añade un indicador con el *handle* al gráfico actual, a una nueva subventana.

```
inthandle = ...// get indicator handle, iCustom or IndicatorCreate

                  // set the current chart (0)
                  // |
                  // |     set the window number to the current number of windows
                  // |                          |
                  // |                          | passing the descriptor
                  // |                          |                       |
                  // v                          v                       v
   ChartIndicatorAdd(  0, (int)ChartGetInteger(0, CHART_WINDOWS_TOTAL), handle);
```

Conociendo esta posibilidad, podemos pensar en llamar a *ChartIndicatorAdd* y pasarle el manejador de un indicador subordinado ya preparado.

La segunda función que necesitamos es *ChartIndicatorName*. Devuelve el nombre corto del indicador por su manejador. Este nombre corresponde a la propiedad INDICATOR_SHORTNAME establecida en el código del indicador y puede diferir del nombre del archivo. El nombre deberá limpiarse a sí mismo, es decir, eliminar el indicador auxiliar y su subventana, después de eliminar o reconfigurar el indicador padre.

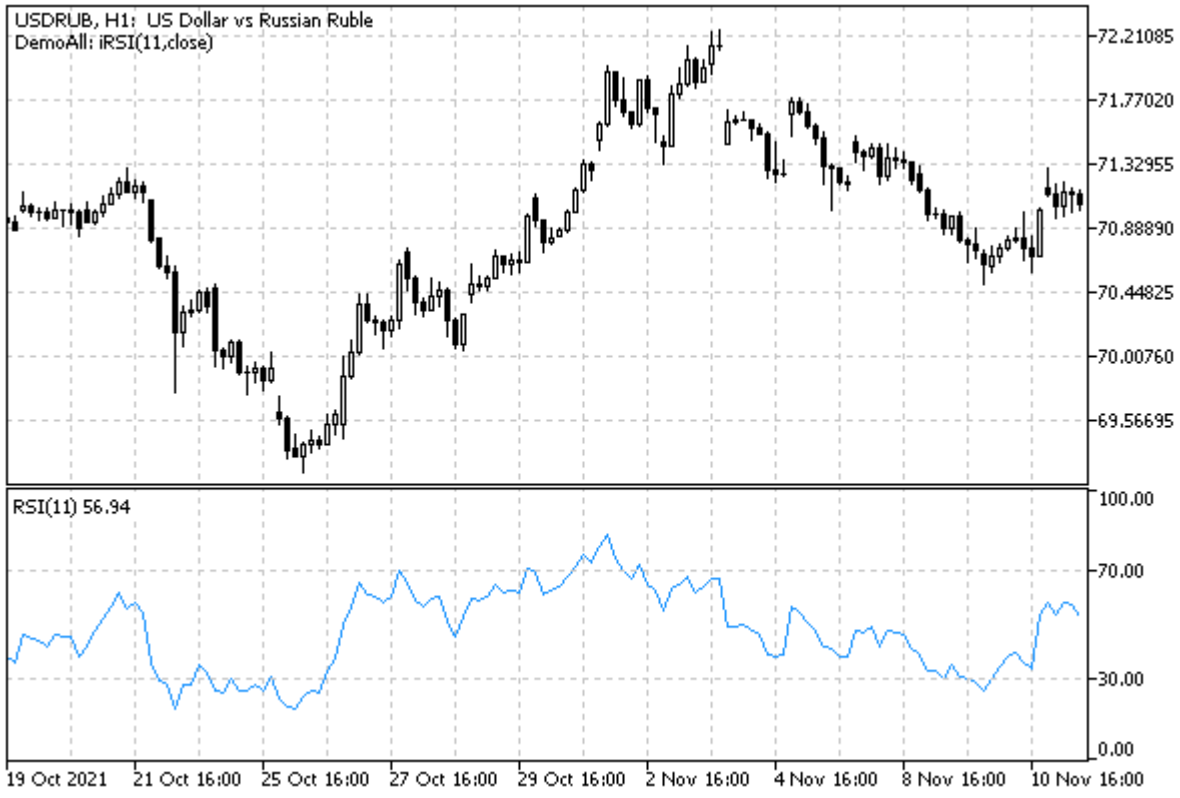
```
string subTitle = "";

int OnInit()
{
   ...
   if(subwindow)
   {
      // show a new indicator in the subwindow
      const int w = (int)ChartGetInteger(0, CHART_WINDOWS_TOTAL);
      ChartIndicatorAdd(0, w, Handle);
      // save the name to remove the indicator in OnDeinit
      subTitle = ChartIndicatorName(0, w, 0);
   }
   ...
}
```

En el manejador *OnDeinit*, utilizamos la función guardada *subTitle* para llamar a otra función que estudiaremos más adelante: *ChartIndicatorDelete*. Esta elimina del gráfico el indicador con el nombre especificado en el último argumento.

```
void OnDeinit(const int)
{
   Print(__FUNCSIG__, (StringLen(subTitle) > 0 ? " deleting " + subTitle : ""));
   if(StringLen(subTitle) > 0)
   {
      ChartIndicatorDelete(0, (int)ChartGetInteger(0, CHART_WINDOWS_TOTAL) - 1,
         subTitle);
   }
}
```

Aquí se supone que sólo nuestro indicador funciona en el gráfico, y sólo en una única instancia. En un caso más general, todas las subventanas deberían analizarse para su correcta eliminación, pero esto requeriría algunas funciones más de las que se presentarán en el capítulo sobre gráficos, por lo que, de momento, nos limitaremos a una versión sencilla.

Si ahora ejecutamos *UseDemoAll* y seleccionamos un indicador marcado con un asterisco (es decir, el que requiere una subventana) de la lista, por ejemplo, RSI, veremos el resultado esperado: RSI en una ventana aparte.



RSI en la subventana creada por el indicador UseDemoAll

5.5.12 Leer datos de gráficos que tienen un desplazamiento

Nuestro nuevo indicador *UseDemoAll* está casi listo; sólo tenemos que considerar una cuestión más.

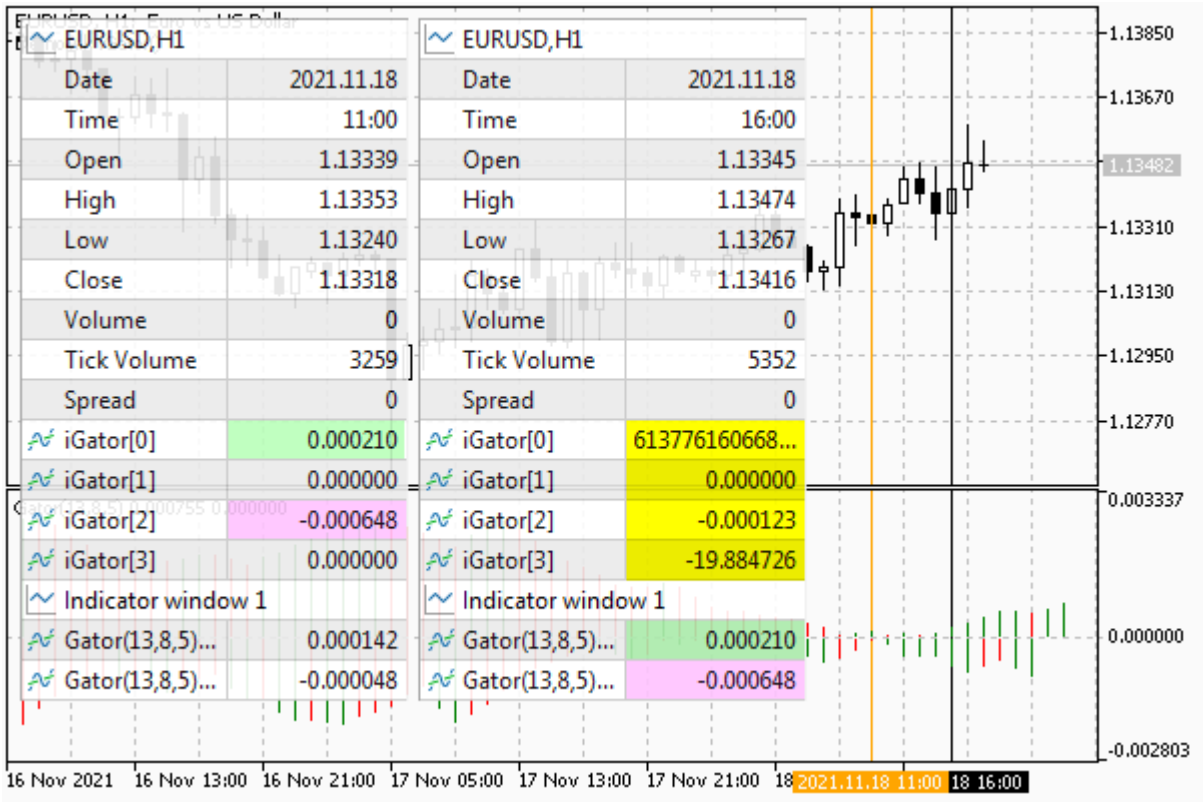
En el indicador subordinado, algunos gráficos pueden tener un desfase establecido por la propiedad PLOT_SHIFT. Por ejemplo, con un desplazamiento positivo, los elementos de la serie temporal se desplazan hacia el futuro y se muestran a la derecha de la barra con índice 0. Sus índices, curiosamente, son negativos. A medida que se desplaza hacia la derecha, los números disminuyen cada

vez más: -1, -2, -3, etc. Esto también afecta a la función *CopyBuffer*. Cuando utilizamos la primera forma de *CopyBuffer*, el parámetro *offset* puesto a 0 se refiere al elemento con el tiempo actual en la serie temporal. Pero si la propia serie temporal se desplaza hacia la derecha, obtendremos datos a partir del elemento numerado N, donde N es el valor de desplazamiento en el indicador de origen. Al mismo tiempo, los elementos situados en nuestro búfer a la derecha del índice N no se llenarán de datos, y la «basura» permanecerá en ellos.

Para demostrar el problema, empecemos con un indicador sin desplazamiento: *Awesome Oscillator* se ajusta perfectamente a este requisito. Recordemos que *UseDemoAll* copia todos los valores en sus arrays, y aunque no son visibles en el gráfico debido a las diferentes escalas de precios y lecturas de los indicadores, podemos comprobarlo en *Data Window*. Dondequiera que movamos el cursor del ratón en el gráfico, los valores de los indicadores en la subventana de *Data Window* y en los búferes de *UseDemoAll* coincidirán. Por ejemplo, en la imagen siguiente, puede ver claramente que en la barra horaria de las 16:00, ambos valores son iguales a 0.001797.

Datos del indicador AO en los búferes UseDemoAll

Ahora, en los ajustes de *UseDemoAll*, seleccionamos el indicador *iGator* (*Gator Oscillator*). Para simplificar, borre el campo con los parámetros *Gator*, de manera que se construya con sus parámetros por defecto. En este caso, el desplazamiento del histograma es de 5 barras (hacia delante), lo que se ve claramente en el gráfico.



Datos del indicador Gator en los búferes UseDemoAll sin corrección para el desplazamiento futuro

La línea vertical negra marca la barra horaria de las 16:00. Sin embargo, los valores del indicador *Gator* en *Data Window* y en nuestros arrays leídos del mismo indicador son diferentes. El color amarillo *UseDemoAll* resalta los búferes que contienen basura.

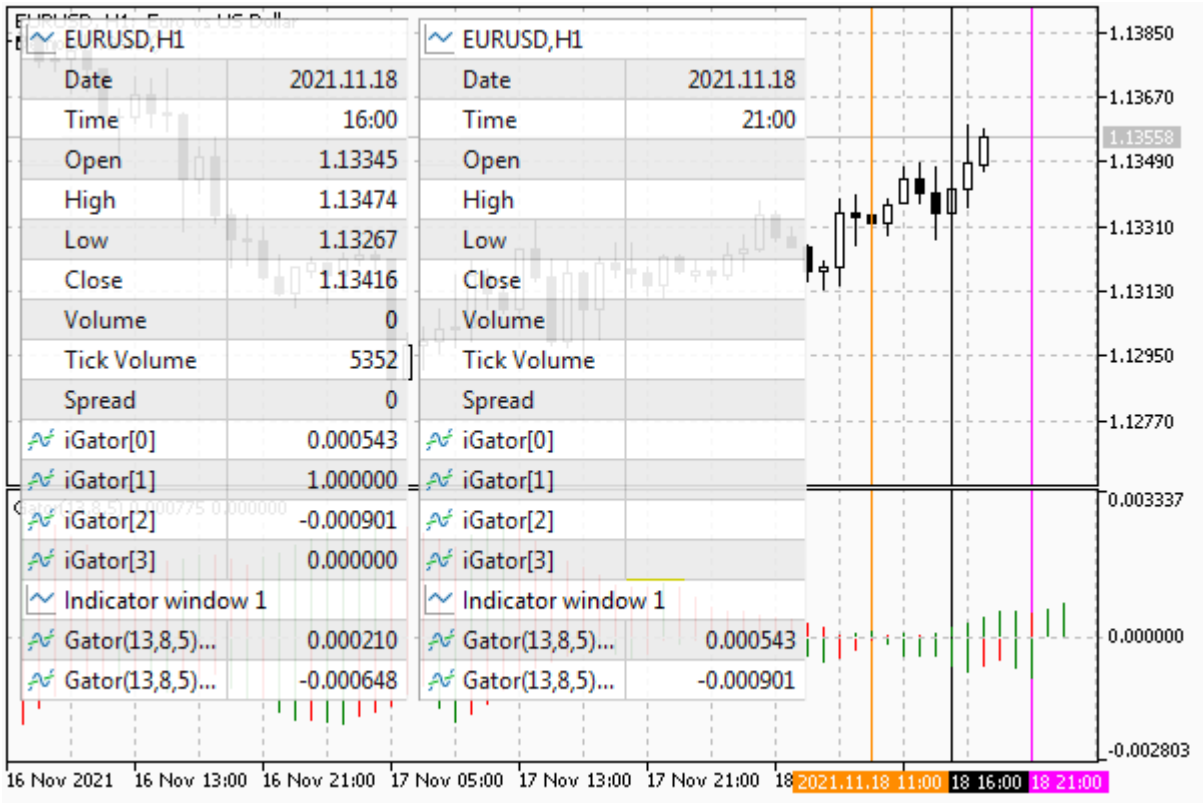
Si examinamos los datos desplazándonos 5 barras hacia el pasado, a las 11:00 (línea vertical naranja), encontraremos allí los valores que *Gator* emite a las 16:00. Los valores correctos por pares de los histogramas superior e inferior se resaltan en verde y rosa, respectivamente.

Para resolver este problema, tenemos que añadir a *UseDemoAll* una variable de entrada para que el usuario especifique un desplazamiento del gráfico, y luego hacer una corrección para ello al llamar a *CopyBuffer*.

```
input int IndicatorShift = 0; // Plot Shift
...
int OnCalculate(ON_CALCULATE_STD_SHORT_PARAM_LIST)
{
   ...
   for(int k = 0; k < m; ++k)
   {
      const int n = buffers[k].copy(Handle, k,
         -IndicatorShift, rates_total - prev_calculated + 1);
      ...
   }
}
```

Por desgracia, es imposible encontrar la propiedad PLOT_SHIFT para un indicador de terceros de MQL5.

Comprobemos cómo la introducción de un desplazamiento de 5 arregla la situación con el indicador Gator (con la configuración por defecto).



Datos del indicador Gator en los búferes UseDemoAll tras ajustar el desplazamiento futuro

Ahora las lecturas de *UseDemoAll* en la barra de las 16:00 corresponden a los datos reales de Gator del futuro virtual 5 barras por delante (línea vertical lila a las 21:00).

Quizá se pregunte por qué en la ventana *Gator* sólo aparecen 2 búferes mientras que en la nuestra hay 4. La cuestión es que el histograma de color de *Gator* utiliza un búfer adicional para la codificación del color, pero sólo hay dos colores, rojo y verde, y los vemos en nuestros arrays como 0 o 1.

5.5.13 Borrar instancias de indicadores: IndicatorRelease

Como se ha mencionado en la parte introductoria de este capítulo, el terminal mantiene un contador de referencia para cada indicador creado y lo deja en funcionamiento mientras al menos un programa MQL o gráfico lo utilice. En un programa MQL, una señal de la necesidad de un indicador es un manejador válido. Normalmente, pedimos un manejador durante la inicialización y lo utilizamos en algoritmos hasta el final del programa.

En el momento en que se descarga el programa, todos los manejadores únicos creados se liberan automáticamente, es decir, sus contadores se decrementan en 1 (y si llegan a cero, esos indicadores también se descargan de la memoria). Por lo tanto, no es necesario liberar explícitamente el manejador.

No obstante, hay situaciones en las que un subindicador resulta innecesario durante el funcionamiento del programa. Entonces, el indicador inútil sigue consumiendo recursos. Por lo tanto, usted debe liberar explícitamente el manejador con *IndicatorRelease*.

bool IndicatorRelease(int handle)

La función borra el manejador del indicador especificado y descarga el propio indicador si nadie más lo utiliza. La descarga se produce con un ligero retraso.

La función devuelve un indicador de éxito (*true*) o de errores (*false*).

Tras la llamada a *IndicatorRelease*, el manejador que se ha pasado deja de ser válido, aunque la propia variable conserva su valor anterior. Un intento de utilizar tal manejador en otras funciones de indicador como *CopyBuffer* fallará con el error 4807 (ERR_INDICATOR_WRONG_HANDLE). Para evitar malentendidos, es conveniente asignar el valor INVALID_HANDLE a la variable correspondiente inmediatamente después de liberar el manejador.

No obstante, si a continuación el programa solicita un manejador para un nuevo indicador, lo más probable es que ese manejador tenga el mismo valor que el liberado anteriormente, pero ahora estará asociado a los datos del nuevo indicador.

Cuando se trabaja en el comprobador de estrategias, no se realiza la función *IndicatorRelease*.

Para demostrar la aplicación de *IndicatorRelease*, vamos a preparar una versión especial de *UseDemoAllLoop.mq5*, que recreará periódicamente un indicador auxiliar en un ciclo de la lista, que incluirá sólo indicadores para la ventana principal (para mayor claridad).

```
IndicatorType MainLoop[] =
{
   iCustom_,
   iAlligator_jawP_jawS_teethP_teethS_lipsP_lipsS_method_price,
   iAMA_period_fast_slow_shift_price,
   iBands_period_shift_deviation_price,
   iDEMA_period_shift_price,
   iEnvelopes_period_shift_method_price_deviation,
   iFractals_,
   iFrAMA_period_shift_price,
   iIchimoku_tenkan_kijun_senkou,
   iMA_period_shift_method_price,
   iSAR_step_maximum,
   iTEMA_period_shift_price,
   iVIDyA_momentum_smooth_shift_price,
};

const int N = ArraySize(MainLoop);
int Cursor = 0; // current position inside the MainLoop array

const string IndicatorCustom = "LifeCycle";
```

El primer elemento del array contiene un indicador personalizado como excepción, *LifeCycle* de la sección Funciones de inicio y parada de programas de diferentes tipos. Aunque este indicador no muestra ninguna línea, es apropiado aquí porque muestra mensajes en el registro cuando se llama a sus manejadores *OnInit/OnDeinit*, lo que le permitirá rastrear su ciclo de vida. Los ciclos de vida de otros indicadores son similares.

En las variables de entrada, dejaremos sólo los ajustes de renderización. La salida por defecto de las etiquetas DRAW_ARROW es óptima para mostrar diferentes tipos de indicadores.

```
input ENUM_DRAW_TYPE DrawType = DRAW_ARROW; // Drawing Type
input int DrawLineWidth = 1; // Drawing Line Width
```

Para recrear los indicadores «sobre la marcha» , vamos a ejecutar el temporizador de 5 segundos en *OnInit*, y toda la inicialización anterior (con algunas modificaciones que se describen a continuación) se trasladará al manejador *OnTimer*.

```
int OnInit()
{
   Comment("Wait 5 seconds to start looping through indicator set");
   EventSetTimer(5);
   return INIT_SUCCEEDED;
}

IndicatorType IndicatorSelector; // currently selected indicator type

void OnTimer()
{
   if(Handle != INVALID_HANDLE && ClearHandles)
   {
      IndicatorRelease(Handle);
      /*
      // descriptor is still 10, but is no longer valid
      // if we uncomment the fragment, we get the following error
      double data[1];
      const int n = CopyBuffer(Handle, 0, 0, 1, data);
      Print("Handle=", Handle, " CopyBuffer=", n, " Error=", _LastError);
      // Handle=10 CopyBuffer=-1 Error=4807 (ERR_INDICATOR_WRONG_HANDLE)
      */
   }
   IndicatorSelector = MainLoop[Cursor];
   Cursor = ++Cursor % N;

   // create a handle with default parameters
   // (because we pass an empty string in the third argument of the constructor)
   AutoIndicator indicator(IndicatorSelector,
      (IndicatorSelector == iCustom_ ? IndicatorCustom : ""), "");
   Handle = indicator.getHandle();
   if(Handle == INVALID_HANDLE)
   {
      Print(StringFormat("Can't create indicator: %s",
         _LastError ? E2S(_LastError) : "The name or number of parameters is incorrec
   }
   else
   {
      Print("Handle=", Handle);
   }

   buffers.empty(); // clear buffers because a new indicator will be displayed
   ChartSetSymbolPeriod(0,NULL,0); // request a full redraw
   ...
   // further setup of diagrams - similar to the previous one
   ...
   Comment("DemoAll: ", (IndicatorSelector == iCustom_ ? IndicatorCustom : s),
      "(default-params)");
}
```

La principal diferencia es que el tipo del indicador creado actualmente *IndicatorSelector* ahora no lo establece el usuario, sino que se selecciona secuencialmente del array *MainLoop* en el índice *Cursor*.

Cada vez que se llama al temporizador, este índice aumenta cíclicamente, es decir, cuando se llega al final del array, saltamos a su principio.

Para todos los indicadores, la línea con los parámetros está vacía. Esto se hace para unificar su inicialización. Como resultado, cada indicador se creará con sus propios valores por defecto.

Al principio del manejador *OnTimer*, llamamos a *IndicatorRelease* para el manejador anterior. Sin embargo, hemos proporcionado una variable de entrada *ClearHandles* para desactivar la rama de operadores *if* dada y ver lo que sucede si no limpia los manejadores.

```
input bool ClearHandles = true;
```

Por defecto, *ClearHandles* es igual a *true*, es decir, los indicadores se borrarán como se espera.

Por último, otro ajuste adicional son las líneas que permiten borrar los búferes y solicitar un redibujado completo del gráfico. Ambas son necesarias, pues hemos sustituido el indicador esclavo que suministra los datos visualizados.

El manejador *OnCalculate* no ha cambiado.

Vamos a ejecutar *UseDemoAllLoop* con la configuración por defecto. En el registro aparecerán las siguientes entradas (sólo se muestra el principio):

```
UseDemoAllLoop (EURUSD,H1) Initializing LifeCycle() EURUSD, PERIOD_H1
UseDemoAllLoop (EURUSD,H1) Handle=10
LifeCycle (EURUSD,H1) Loader::Loader()
LifeCycle (EURUSD,H1) void OnInit() 0 DEINIT_REASON_PROGRAM
UseDemoAllLoop (EURUSD,H1) Initializing iAlligator_jawP_jawS_teethP_teethS_lipsP_lips
UseDemoAllLoop (EURUSD,H1) iAlligator_jawP_jawS_teethP_teethS_lipsP_lipsS_method_pric
UseDemoAllLoop (EURUSD,H1) Handle=10
LifeCycle (EURUSD,H1) void OnDeinit(const int) DEINIT_REASON_REMOVE
LifeCycle (EURUSD,H1) Loader::~Loader()
UseDemoAllLoop (EURUSD,H1) Initializing iAMA_period_fast_slow_shift_price() EURUSD, P
UseDemoAllLoop (EURUSD,H1) iAMA_period_fast_slow_shift_price requires 5 parameters, 0
UseDemoAllLoop (EURUSD,H1) Handle=10
UseDemoAllLoop (EURUSD,H1) Initializing iBands_period_shift_deviation_price() EURUSD,
UseDemoAllLoop (EURUSD,H1) iBands_period_shift_deviation_price requires 4 parameters,
UseDemoAllLoop (EURUSD,H1) Handle=10
...
```

Fíjese en que obtenemos siempre el mismo «número» de manejador (10) porque lo liberamos antes de crear un nuevo manejador.

También es importante que el indicador *LifeCycle* se descargue poco después de que lo hayamos liberado (suponiendo que no se haya añadido al mismo gráfico por sí mismo, porque entonces su cuenta de referencia no se pondría a cero).

En la siguiente imagen se muestra el momento en que nuestro indicador muestra los datos de Alligator.

UseDemoAllLoop en el paso de demostración de Alligator

Si cambia el valor de *ClearHandles* a *false*, veremos una imagen completamente diferente en el registro. A partir de ahora, los números de los manejadores aumentarán constantemente, lo que indica que los indicadores permanecen en el terminal y siguen trabajando, consumiendo recursos en vano. En concreto, no se recibe ningún mensaje de desinicialización del indicador *LifeCycle*.

```
UseDemoAllLoop (EURUSD,H1) Initializing LifeCycle() EURUSD, PERIOD_H1
UseDemoAllLoop (EURUSD,H1) Handle=10
LifeCycle (EURUSD,H1) Loader::Loader()
LifeCycle (EURUSD,H1) void OnInit() 0 DEINIT_REASON_PROGRAM
UseDemoAllLoop (EURUSD,H1) Initializing iAlligator_jawP_jawS_teethP_teethS_lipsP_lips
UseDemoAllLoop (EURUSD,H1) iAlligator_jawP_jawS_teethP_teethS_lipsP_lipsS_method_pric
UseDemoAllLoop (EURUSD,H1) Handle=11
UseDemoAllLoop (EURUSD,H1) Initializing iAMA_period_fast_slow_shift_price() EURUSD, P
UseDemoAllLoop (EURUSD,H1) iAMA_period_fast_slow_shift_price requires 5 parameters, 0
UseDemoAllLoop (EURUSD,H1) Handle=12
UseDemoAllLoop (EURUSD,H1) Initializing iBands_period_shift_deviation_price() EURUSD,
UseDemoAllLoop (EURUSD,H1) iBands_period_shift_deviation_price requires 4 parameters,
UseDemoAllLoop (EURUSD,H1) Handle=13
UseDemoAllLoop (EURUSD,H1) Initializing iDEMA_period_shift_price() EURUSD, PERIOD_H1
UseDemoAllLoop (EURUSD,H1) iDEMA_period_shift_price requires 3 parameters, 0 given
UseDemoAllLoop (EURUSD,H1) Handle=14
UseDemoAllLoop (EURUSD,H1) Initializing iEnvelopes_period_shift_method_price_deviatio
UseDemoAllLoop (EURUSD,H1) iEnvelopes_period_shift_method_price_deviation requires 5
UseDemoAllLoop (EURUSD,H1) Handle=15
...
UseDemoAllLoop (EURUSD,H1) Initializing iVIDyA_momentum_smooth_shift_price() EURUSD,
UseDemoAllLoop (EURUSD,H1) iVIDyA_momentum_smooth_shift_price requires 4 parameters,
UseDemoAllLoop (EURUSD,H1) Handle=22
UseDemoAllLoop (EURUSD,H1) Initializing LifeCycle() EURUSD, PERIOD_H1
UseDemoAllLoop (EURUSD,H1) Handle=10
UseDemoAllLoop (EURUSD,H1) Initializing iAlligator_jawP_jawS_teethP_teethS_lipsP_lips
UseDemoAllLoop (EURUSD,H1) iAlligator_jawP_jawS_teethP_teethS_lipsP_lipsS_method_pric
UseDemoAllLoop (EURUSD,H1) Handle=11
UseDemoAllLoop (EURUSD,H1) Initializing iAMA_period_fast_slow_shift_price() EURUSD, P
UseDemoAllLoop (EURUSD,H1) iAMA_period_fast_slow_shift_price requires 5 parameters, 0
UseDemoAllLoop (EURUSD,H1) Handle=12
UseDemoAllLoop (EURUSD,H1) Initializing iBands_period_shift_deviation_price() EURUSD,
UseDemoAllLoop (EURUSD,H1) iBands_period_shift_deviation_price requires 4 parameters,
UseDemoAllLoop (EURUSD,H1) Handle=13
UseDemoAllLoop (EURUSD,H1) Initializing iDEMA_period_shift_price() EURUSD, PERIOD_H1
UseDemoAllLoop (EURUSD,H1) iDEMA_period_shift_price requires 3 parameters, 0 given
UseDemoAllLoop (EURUSD,H1) Handle=14
UseDemoAllLoop (EURUSD,H1) void OnDeinit(const int)
...
```

Cuando el índice en el bucle sobre el array de tipos de indicadores alcanza el último elemento y dibuja un círculo desde el principio, el terminal empezará a devolver a nuestro código manejadores de indicadores ya existentes (los mismos valores: al manejador 22 le sigue de nuevo el 10).

5.5.14 Obtener la configuración del indicador por su manejador

A veces un programa MQL necesita conocer los parámetros de una instancia de indicador en ejecución. Estos pueden ser indicadores de terceros en el gráfico, o un manejador pasado desde el programa principal a la biblioteca o el archivo de encabezado. Para ello, MQL5 proporciona la función *IndicatorParameters*.

int IndicatorParameters(int handle, ENUM_INDICATOR &type, MqlParam &params[])

Mediante el manejador especificado, la función devuelve el número de parámetros de entrada del indicador, así como sus tipos y valores.

En caso de éxito, la función rellena el array *params* que se le ha pasado, y el tipo de indicador se guarda en el parámetro *type*.

En caso de error, la función devuelve -1.

Como ejemplo del trabajo con esta función, vamos a mejorar el indicador *UseDemoAllLoop.mq5* presentado en la sección sobre Borrar instancias de indicadores. Llamemos a la nueva versión *UseDemoAllParams.mq5*.

Como recordará, creamos secuencialmente algunos indicadores integrados en el bucle de la lista y dejamos vacía la lista de parámetros, lo que provoca que los indicadores utilicen algunos valores predeterminados desconocidos. A este respecto, mostramos un prototipo generalizado en un comentario sobre el gráfico: con un nombre, pero sin valores específicos.

```
// UseDemoAllLoop.mq5
void OnTimer()
{
   ...
   Comment("DemoAll: ", (IndicatorSelector == iCustom_ ? IndicatorCustom : s),
      "(default-params)");
   ...
}
```

Ahora tenemos la oportunidad de averiguar sus parámetros basándonos en el manejador del indicador y mostrárselos al usuario.

```
// UseDemoAllParams.mq5
void OnTimer()
{
   ...
   // read the parameters applied by the indicator by default
   ENUM_INDICATOR itype;
   MqlParam defParams[];
   const int p = IndicatorParameters(Handle, itype, defParams);
   ArrayPrint(defParams);
   Comment("DemoAll: ", (IndicatorSelector == iCustom_ ? IndicatorCustom : s),
      "(" + MqlParamStringer::stringify(defParams) + ")");
   ...
}
```

La conversión del array *MqlParam* en una cadena se implementa en la clase especial *MqlParamStringer* (véase el archivo *MqlParamStringer.mqh*).

```
class MqlParamStringer
{
public:
   static string stringify(const MqlParam &param)
   {
      switch(param.type)
      {
      case TYPE_BOOL:
      case TYPE_CHAR:
      case TYPE_UCHAR:
      case TYPE_SHORT:
      case TYPE_USHORT:
      case TYPE_DATETIME:
      case TYPE_COLOR:
      case TYPE_INT:
      case TYPE_UINT:
      case TYPE_LONG:
      case TYPE_ULONG:
         return IntegerToString(param.integer_value);
      case TYPE_FLOAT:
      case TYPE_DOUBLE:
         return (string)(float)param.double_value;
      case TYPE_STRING:
         return param.string_value;
      }
      return NULL;
   }

   static string stringify(const MqlParam &params[])
   {
      string result = "";
      const int p = ArraySize(params);
      for(int i = 0; i < p; ++i)
      {
         result += stringify(params[i]) + (i < p - 1 ? "," : "");
      }
      return result;
   }
};
```

Después de compilar y ejecutar el nuevo indicador, puede asegurarse de que la lista específica de parámetros del indicador que se está renderizando aparece ahora en la esquina superior izquierda del gráfico.

Para un único indicador personalizado de la lista (*LifeCycle*), el primer parámetro contendrá la ruta y el nombre de archivo del indicador. El segundo parámetro se describe en el código fuente como un número entero, pero el tercer parámetro es interesante porque describe implícitamente la propiedad «Aplicar a», que es inherente a todos los indicadores con una forma abreviada del manejador *OnCalculate*. En este caso, por defecto, el indicador se aplica a PRICE_CLOSE (valor 1).

```
Initializing LifeCycle() EURUSD, PERIOD_H1
Handle=10
[type] [integer_value] [double_value] [string_value]
[0] 14 0 0.00000 "Indicators\MQL5Book\p5\LifeCycle.ex5"
[1] 7 0 0.00000 null
[2] 7 1 0.00000 null
Initializing iAlligator_jawP_jawS_teethP_teethS_lipsP_lipsS_method_price() EURUSD, PE
iAlligator_jawP_jawS_teethP_teethS_lipsP_lipsS_method_price requires 8 parameters, 0
Handle=10
[type] [integer_value] [double_value] [string_value]
[0] 7 13 0.00000 null
[1] 7 8 0.00000 null
[2] 7 8 0.00000 null
[3] 7 5 0.00000 null
[4] 7 5 0.00000 null
[5] 7 3 0.00000 null
[6] 7 2 0.00000 null
[7] 7 5 0.00000 null
Initializing iAMA_period_fast_slow_shift_price() EURUSD, PERIOD_H1
iAMA_period_fast_slow_shift_price requires 5 parameters, 0 given
Handle=10
[type] [integer_value] [double_value] [string_value]
[0] 7 9 0.00000 null
[1] 7 2 0.00000 null
[2] 7 30 0.00000 null
[3] 7 0 0.00000 null
[4] 7 1 0.00000 null
```

Según el registro, los ajustes de los indicadores integrados también corresponden a los predeterminados.

5.5.15 Definir la fuente de datos de un indicador

Entre las variables integradas del programa MQL hay una que sólo puede utilizarse en los indicadores. Se trata de la variable *_AppliedTo* de tipo *int*, que permite leer la propiedad *Apply to* del cuadro de diálogo de configuración del indicador. Además, si el indicador se crea llamando a la función *iCustom*, a la que se pasó el manejador del indicador de terceros, la variable *_AppliedTo* contendrá este manejador.

En la siguiente tabla se describen los posibles valores de la variable *_AppliedTo*.

Valor	Descripción de los datos para el cálculo
0	El indicador utiliza la forma completa de *OnCalculate*, y los datos para el cálculo no se establecen mediante un array de datos
1	Precio de cierre
2	Precio de apertura
3	Precio alto
4	Precio bajo
5	Precio medio = (Máximo+Mínimo)/2
6	Precio típico = (Máximo+Mínimo+Cierre)/3
7	Precio ponderado = (Apertura+Alto+Bajo+Cierre)/4
8	Datos del indicador que se lanzó en el gráfico antes de este indicador
9	Datos del indicador que se lanzó en el gráfico la primera vez
10+	Datos del indicador con el manejador contenido en *_AppliedTo;* este manejador se pasó como último parámetro a la función *iCustom* al crear el indicador.

Para facilitar el análisis de los valores, se adjunta a este libro un archivo de encabezado *AppliedTo.mqh* con la enumeración.

5.6 Trabajar con temporizador

Para muchas tareas aplicadas, es importante poder realizar acciones conforme a un horario, con algún intervalo especificado. En MQL5, esta funcionalidad la proporciona el temporizador, un contador de tiempo del sistema que puede configurarse para enviar notificaciones periódicas a un programa MQL.

Existen varias funciones para establecer o cancelar notificaciones de temporizador en la API de MQL5: *EventSetTimer*, *EventSetMillisecondTimer*, *EventKillTimer*. Las propias notificaciones entran en el programa como eventos de un tipo especial: el manejador *OnTimer* está reservado para ellas en el código fuente. Este grupo de funciones se abordará en este capítulo.

Recuerde que en MQL5 los eventos sólo pueden ser recibidos por programas interactivos que se ejecutan en los gráficos, es decir, indicadores y Asesores Expertos. Scripts y servicios no admiten ningún evento, incluidos los del temporizador.

No obstante, en el capítulo Funciones para trabajar con el tiempo, abordamos ya temas relacionados:

- Obtener las marcas de tiempo del reloj local o del servidor actual (*TimeLocal / TimeCurrent*)
- Pausar la ejecución del programa durante un periodo de tiempo utilizando *Sleep*
- Obtener el estado del contador de tiempo del sistema del ordenador, contado desde el inicio del sistema operativo (*GetTickCount*) o desde el lanzamiento del programa MQL (*GetMicrosecondCount*)

Estas opciones están abiertas a absolutamente todos los tipos de programas MQL.

En los capítulos anteriores ya hemos utilizado muchas veces las funciones de temporizador, aunque su descripción formal sólo se dará ahora. Debido al hecho de que los eventos de temporizador sólo están disponibles en indicadores o Asesores Expertos, sería difícil estudiarlo antes que los programas en sí. Después de haber dominado la creación de indicadores, el tema de los temporizadores será una continuación lógica.

Básicamente, utilizamos temporizadores para esperar a que se construya la serie temporal. Encontrará ejemplos de ello en las secciones Esperar datos, Indicadores multidivisa y de marco temporal múltiple, Soporte para múltiples símbolos y marcos temporales y Utilización de los indicadores integrados.

Además, cronometramos (cada 5 segundos) el tipo de indicador subordinado en la demostración del indicador «animación» en la sección Borrar instancias de indicadores.

5.6.1 Activar y desactivar temporizador: EventSetTimer/EventKillTimer

MQL5 permite activar o desactivar el temporizador estándar para realizar cualquier acción programada. Existen dos funciones para ello: *EventSetTimer* y *EventKillTimer*.

bool EventSetTimer(int seconds)

La función indica al terminal cliente que para este Asesor Experto o indicador es necesario generar eventos del temporizador con la frecuencia especificada, que se establece en segundos (parámetro *seconds*).

La función devuelve un signo de éxito (*true*) o de error (*false*). El código de error puede obtenerse en *_LastError*.

Para poder procesar eventos de temporizador, un Asesor Experto o un indicador debe tener la función *OnTimer* en su código. El primer evento del temporizador no se producirá inmediatamente después de la llamada de *EventSetTimer*, sino después de *seconds* segundos.

Para cada Asesor Experto o indicador que llama a la función *EventSetTimer*, crea su propio temporizador dedicado. El programa recibirá eventos sólo de él. Los temporizadores de los distintos programas funcionan de forma independiente.

Cada programa MQL interactivo colocado en un gráfico tiene una cola de eventos independiente en la que se añaden los eventos que se reciben para él. Si ya hay un evento en la cola *OnTimer* o que está en estado de procesamiento, el nuevo evento *OnTimer* no se pone en cola.

Si el temporizador ya no es necesario, debe desactivarse con la función *EventKillTimer*.

void EventKillTimer(void)

La función detiene el temporizador activado anteriormente por la función *EventSetTimer* (o por *EventSetMillisecondTimer*, de la que hablaremos a continuación). La función también puede llamarse desde el manejador *OnTimer*. Así, en concreto, es posible realizar una acción única retardada.

> La llamada de *EventKillTimer* en los indicadores no despeja la cola, por lo que después de ella se puede obtener el último evento *OnTimer* residual.

Cuando el programa MQL termina, el temporizador es forzosamente destruido si fue creado pero no desactivado por la función *EventKillTimer*.

Cada programa sólo puede establecer un temporizador. Por lo tanto, si desea llamar a diferentes partes del algoritmo en diferentes intervalos, debe habilitar un temporizador con un período que sea el mínimo

común divisor de los períodos requeridos (en el caso límite, con un período mínimo de 1 segundo), y en el manejador *OnTimer* realizar un seguimiento independiente de períodos mayores. Veremos un ejemplo de este enfoque en la próxima sección.

MQL5 también permite crear temporizadores con un periodo inferior a 1 segundo: existe una función para ello, *EventSetMillisecondTimer*.

5.6.2 Evento temporizador: OnTimer

El evento *OnTimer* es uno de los eventos estándar admitidos por los programas MQL5 (véase la sección Visión general de las funciones de gestión de eventos). Para recibir eventos de temporizador en el código del programa, debe describir una función con el siguiente prototipo:

void OnTimer(void)

El evento *OnTimer* es generado periódicamente por el terminal cliente para un Asesor Experto o un indicador que ha activado el temporizador utilizando la función *EventSetTimer* o *EventSetMillisecondTimer* (véase la sección siguiente).

> ¡Atención! En indicadores dependientes creados llamando a *iCustom* o *IndicatorCreate* desde otros programas, el temporizador no funciona, y el evento *OnTimer* no se genera. Esta es una limitación arquitectónica de MetaTrader 5.

Debe entenderse que la presencia de un manejador *OnTimer* y un temporizador habilitado no hace que el programa MQL sea multihilo. No se asigna más de un hilo por programa MQL (un indicador puede incluso compartir un hilo con otros indicadores en el mismo símbolo), por lo que la llamada de *OnTimer* y otros manejadores siempre se produce secuencialmente, de acuerdo con la cola de eventos. Si uno de los manejadores, incluido *OnTimer*, va a iniciar cálculos largos, esto suspenderá la ejecución de todos los demás eventos y secciones del código del programa.

Si necesita organizar el procesamiento de datos en paralelo, debe ejecutar varios programas MQL de forma simultánea (quizás, instancias del mismo programa en diferentes gráficos u objetos gráficos) e intercambiar comandos y datos entre ellos utilizando su propio protocolo; por ejemplo, utilizando eventos personalizados.

Como ejemplo, vamos a crear clases que pueden organizar varios temporizadores lógicos en un programa. Los periodos de todos los temporizadores lógicos se establecerán como un multiplicador del periodo base, es decir, el periodo de un único temporizador de hardware que suministra eventos al manejador estándar *OnTimer*. En este manejador, debemos llamar a cierto método de nuestra nueva clase *MultiTimer* que manejará todos los temporizadores lógicos.

```
void OnTimer()
{
   // call the MultiTimer method to check and call dependent timers when needed
   MultiTimer::onTimer();
}
```

La clase *MultiTimer* y las clases relacionadas de temporizadores individuales se combinarán en un archivo, *MultiTimer.mqh*.

La clase base para los temporizadores de trabajo será *TimerNotification*. En términos estrictos, podría tratarse de una interfaz, pero es conveniente volcar en ella algunos detalles de la implementación general: en concreto, almacenar la lectura del contador *chronometer*, mediante el cual nos aseguraremos de que el temporizador se dispara con un cierto multiplicador del periodo relativo del

temporizador principal, así como un método para comprobar el momento en que el temporizador debe dispararse *isTimeCome*. Por eso *TimerNotification* es una clase abstracta. Carece de la implementación de dos métodos virtuales: *notify*, para las acciones cuando se dispara el temporizador, y *getInterval*, para obtener un multiplicador que determina el periodo de un temporizador concreto en relación con el periodo del temporizador principal.

```
class TimerNotification
{
protected:
   int chronometer; // counter of timer checks (isTimeCome calls)
public:
   TimerNotification(): chronometer(0)
   {
   }

   // timer work event
   // pure virtual method, it is required to be described in the heirs
   virtual void notify() = 0;
   // returns the period of the timer (it can be changed on the go)
   // pure virtual method, it is required to be described in the heirs
   virtual int getInterval() = 0;
   // check if it's time for the timer to fire, and if so, call notify
   virtual bool isTimeCome()
   {
      if(chronometer >= getInterval() - 1)
      {
         chronometer = 0; // reset the counter
         notify();        // notify application code
         return true;
      }

      ++chronometer;
      return false;
   }
};
```

Toda la lógica se proporciona en el método *isTimeCome*. Cada vez que se llama, el contador *chronometer* se incrementa, y si alcanza la última iteración según el método *getInterval*, se llama al método *notify* para notificar el código de la aplicación.

Por ejemplo, si el temporizador principal se inicia con un periodo de 1 segundo (*EventSetTimer(1)*), entonces el objeto hijo *TimerNotification,*, que devolverá 5 de *getInterval*, recibirá llamadas a su método *notify* cada 5 segundos.

Como ya hemos dicho, dichos objetos temporizadores serán gestionados por el objeto administrador *MultiTimer*. Sólo necesitamos un objeto de este tipo. Por lo tanto, su constructor se declara protegido y se crea una única instancia de forma estática dentro de la clase.

```
class MultiTimer
{
protected:
   static MultiTimer _mainTimer;

   MultiTimer()
   {
   }
   ...
```

Dentro de esta clase, organizamos el almacenamiento del array de objetos *TimerNotification* (veremos cómo se rellena dentro de pocos párrafos). Una vez que tenemos el array podemos escribir fácilmente el método *checkTimers* que hace un bucle a través de todos los temporizadores lógicos. Para el acceso externo, este método está duplicado por el método estático público *onTimer*, que ya hemos visto en el manejador global *OnTimer*. Como la única instancia del administrador se crea estáticamente, podemos acceder a ella desde un método estático.

```
   ...
   TimerNotification *subscribers[];

   void checkTimers()
   {
      int n = ArraySize(subscribers);
      for(int i = 0; i < n; ++i)
      {
         if(CheckPointer(subscribers[i]) != POINTER_INVALID)
         {
            subscribers[i].isTimeCome();
         }
      }
   }

public:
   static void onTimer()
   {
      _mainTimer.checkTimers();
   }
   ...
```

El objeto *TimerNotification* se añade al arrayz *subscribers* mediante el método *bind*.

```
void bind(TimerNotification &tn)
{
   int i, n = ArraySize(subscribers);
   for(i = 0; i < n; ++i)
   {
      if(subscribers[i] == &tn) return; // there is already such an object
      if(subscribers[i] == NULL) break; // found an empty slot
   }
   if(i == n)
   {
      ArrayResize(subscribers, n + 1);
   }
   else
   {
      n = i;
   }
   subscribers[n] = &tn;
}
```

El método está protegido contra la adición repetida del objeto y, si es posible, el puntero se coloca en un elemento vacío del array, si lo hay, lo que elimina la necesidad de ampliar el array. Pueden aparecer elementos vacíos en un array si se ha eliminado alguno de los objetos de *TimerNotification* mediante el método *unbind* (se pueden utilizar temporizadores de forma ocasional).

```
void unbind(TimerNotification &tn)
{
   const int n = ArraySize(subscribers);
   for(int i = 0; i < n; ++i)
   {
      if(subscribers[i] == &tn)
      {
         subscribers[i] = NULL;
         return;
      }
   }
}
```

Tenga en cuenta que el administrador no se apropia del objeto temporizador y no intenta llamar a *delete*. Si va a registrar objetos temporizadores asignados dinámicamente en el administrador, puede añadir el siguiente código dentro de *if* antes de la puesta a cero:

```
if(CheckPointer(subscribers[i]) == POINTER_DYNAMIC) delete subscribers[i]
```

Ahora queda entender cómo podemos organizar convenientemente las llamadas a *bind*/*unbind*, para no cargar el código de la aplicación con estas operaciones utilitarias. Si lo hace «manualmente», es fácil que olvide crear o, por el contrario, borrar el temporizador en algún sitio.

Vamos a desarrollar la clase *SingleTimer* derivada de *TimerNotification*, en la que implementamos las llamadas *bind* y *unbind* del constructor y destructor, respectivamente. Además, describimos en él la variable *multiplier* para almacenar el periodo del temporizador.

```
class SingleTimer: public TimerNotification
{
protected:
   int multiplier;
   MultiTimer *owner;

public:
   // creating a timer with the specified base period multiplier, optionally pause
   // automatically register the object in the manager
   SingleTimer(const int m, const bool paused = false): multiplier(m)
   {
      owner = &MultiTimer::_mainTimer;
      if(!paused) owner.bind(this);
   }

   // automatically disconnect the object from the manager
   ~SingleTimer()
   {
      owner.unbind(this);
   }

   // return timer period
   virtual int getInterval() override
   {
      return multiplier;
   }

   // pause this timer
   virtual void stop()
   {
      owner.unbind(this);
   }

   // resume this timer
   virtual void start()
   {
      owner.bind(this);
   }
};
```

El segundo parámetro del constructor (*paused*) permite crear un objeto, pero no iniciar el temporizador inmediatamente. Dicho temporizador retardado puede activarse mediante el método *start*.

> El esquema de suscripción de unos objetos a eventos en otros es uno de los patrones de diseño populares en la POO y se denomina «publisher/subscriber» (publicador/suscriptor).

Es importante señalar que esta clase también es abstracta porque no implementa el método *notify*. Basándonos en *SingleTimer*, vamos a describir las clases de temporizadores con funcionalidad adicional.

Empecemos por la clase *CountableTimer*. Permite especificar cuántas veces debe dispararse, tras lo cual se detendrá automáticamente. Con ella, en concreto, es fácil organizar una única acción diferida. El constructor *CountableTimer* tiene parámetros para establecer el periodo del temporizador, la

bandera de pausa y el número de reintentos. Por defecto, el número de repeticiones no está limitado, por lo que esta clase se convertirá en la base de la mayoría de los temporizadores de aplicación.

```
class CountableTimer: public MultiTimer::SingleTimer
{
protected:
   const uint repeat;
   uint count;

public:
   CountableTimer(const int m, const uint r = UINT_MAX, const bool paused = false):
      SingleTimer(m, paused), repeat(r), count(0) { }

   virtual bool isTimeCome() override
   {
      if(count >= repeat && repeat != UINT_MAX)
      {
         stop();
         return false;
      }
      // delegate the time check to the parent class,
      // increment our counter only if the timer fired (returned true)
      return SingleTimer::isTimeCome() && (bool)++count;
   }
   // reset our counter on stop
   virtual void stop() override
   {
      SingleTimer::stop();
      count = 0;
   }

   uint getCount() const
   {
      return count;
   }

   uint getRepeat() const
   {
      return repeat;
   }
};
```

Para utilizar *CountableTimer* tenemos que describir la clase derivada en nuestro programa de la siguiente manera:

```
// MultipleTimers.mq5
class MyCountableTimer: public CountableTimer
{
public:
   MyCountableTimer(const int s, const uint r = UINT_MAX):
      CountableTimer(s, r) { }

   virtual void notify() override
   {
      Print(__FUNCSIG__, multiplier, " ", count);
   }
};
```

En esta implementación del método *notify*, sólo registramos el período del temporizador y el número de veces que se disparó. Por cierto, este es un fragmento del indicador *MultipleTimers.mq5*, que utilizaremos como ejemplo de trabajo.

Llamemos a la segunda clase derivada de *SingleTimer FunctionalTimer*. Su propósito es proporcionar una implementación sencilla del temporizador para aquellos a los que les gusta el estilo funcional de programación y no les apetece escribir clases derivadas. El constructor de la clase *FunctionalTimer* tomará, además del punto, un puntero a una función de un tipo especial, *TimerHandler*.

```
// MultiTimer.mqh
typedef bool (*TimerHandler)(void);

class FunctionalTimer: public MultiTimer::SingleTimer
{
   TimerHandler func;
public:
   FunctionalTimer(const int m, TimerHandler f):
      SingleTimer(m), func(f) { }

   virtual void notify() override
   {
      if(func != NULL)
      {
         if(!func())
         {
            stop();
         }
      }
   }
};
```

En esta implementación del método *notify*, el objeto llama a la función por el puntero. Con una clase de este tipo, podemos definir una macro que, cuando se coloca delante de un bloque de sentencias entre llaves, lo «convertirá» en el cuerpo de la función del temporizador.

```
// MultiTimer.mqh
#define OnTimerCustom(P) OnTimer##P(); \
FunctionalTimer ft##P(P, OnTimer##P); \
bool OnTimer##P()
```

Entonces, en el código de la aplicación, puede escribir algo así:

```
// MultipleTimers.mq5
bool OnTimerCustom(3)
{
   Print(__FUNCSIG__);
   return true;         // continue the timer
}
```

Esta construcción declara un temporizador con un período de 3 y un conjunto de instrucciones dentro de paréntesis (aquí, sólo la impresión a un registro). Si esta función devuelve *false*, este temporizador se detendrá.

Vamos a analizar más detenidamente el indicador *MultipleTimers.mq5*. Como no proporciona visualización, especificaremos el número de diagramas igual a cero.

```
#property indicator_chart_window
#property indicator_buffers 0
#property indicator_plots   0
```

Para utilizar las clases de temporizadores lógicos, incluimos el archivo de encabezado *MultiTimer.mqh* y añadimos una variable de entrada para el periodo base (global) del temporizador.

```
#include <MQL5Book/MultiTimer.mqh>

input int BaseTimerPeriod = 1;
```

El temporizador base se inicia en *OnInit*.

```
void OnInit()
{
   Print(__FUNCSIG__, " ", BaseTimerPeriod, " Seconds");
   EventSetTimer(BaseTimerPeriod);
}
```

Recordemos que el funcionamiento de todos los temporizadores lógicos está garantizado por la interceptación del evento global *OnTimer*.

```
void OnTimer()
{
   MultiTimer::onTimer();
}
```

Además de la clase de aplicación del temporizador *MyCountableTimer* anterior, vamos a describir otra clase del temporizador suspendido *MySuspendedTimer*.

```
class MySuspendedTimer: public CountableTimer
{
public:
   MySuspendedTimer(const int s, const uint r = UINT_MAX):
      CountableTimer(s, r, true) { }
   virtual void notify() override
   {
      Print(__FUNCSIG__, multiplier, " ", count);
      if(count == repeat - 1) // execute last time
      {
         Print("Forcing all timers to stop");
         EventKillTimer();
      }
   }
};
```

Un poco más abajo veremos cómo empieza. También es importante señalar aquí que después de alcanzar el número especificado de operaciones, este temporizador apagará todos los temporizadores llamando a *EventKillTimer*.

Ahora vamos a mostrar cómo (en el contexto global) se describen los objetos de diferentes temporizadores de estas dos clases.

```
MySuspendedTimer st(1, 5);
MyCountableTimer t1(2);
MyCountableTimer t2(4);
```

El temporizador *st* de la clase *MySuspendedTimer* tiene periodo 1 (*1*BaseTimerPeriod*) y debe detenerse después de 5 operaciones.

Los temporizadores *t1* y *t2* de la clase *MyCountableTimer* tienen periodos 2 (*2 * BaseTimerPeriod*) y 4 (*4 * BaseTimerPeriod*), respectivamente. Con el valor por defecto *BaseTimerPeriod = 1* todos los periodos representan segundos. Estos dos temporizadores se ponen en marcha inmediatamente después del inicio del programa.

También crearemos dos temporizadores de estilo funcional.

```
bool OnTimerCustom(5)
{
   Print(__FUNCSIG__);
   st.start();          // start delayed timer
   return false;        // and stop this timer object
}

bool OnTimerCustom(3)
{
   Print(__FUNCSIG__);
   return true;         // this timer keeps running
}
```

Tenga en cuenta que *OnTimerCustom5* sólo tiene una tarea: 5 periodos después del inicio del programa, necesita iniciar un temporizador retardado *st* y terminar su propia ejecución. Teniendo en cuenta que el temporizador retardado debe desactivar todos los temporizadores después de 5 periodos, obtenemos 10 segundos de actividad del programa con los ajustes por defecto.

El temporizador *OnTimerCustom3* debe activarse tres veces durante este periodo.

Así, tenemos 5 temporizadores con diferentes períodos: 1, 2, 3, 4, 5 segundos.

Analicemos un ejemplo de lo que se envía al registro (las marcas de tiempo se muestran esquemáticamente a la derecha).

```
                                                  // time
  17:08:45.174  void OnInit() 1 Seconds             |
  17:08:47.202  void MyCountableTimer::notify()2 0    |
  17:08:48.216  bool OnTimer3()                        |
  17:08:49.230  void MyCountableTimer::notify()2 1      |
  17:08:49.230  void MyCountableTimer::notify()4 0      |
  17:08:50.244  bool OnTimer5()                          |
  17:08:51.258  void MyCountableTimer::notify()2 2        |
  17:08:51.258  bool OnTimer3()                           |
  17:08:51.258  void MySuspendedTimer::notify()1 0        |
  17:08:52.272  void MySuspendedTimer::notify()1 1         |
  17:08:53.286  void MyCountableTimer::notify()2 3          |
  17:08:53.286  void MyCountableTimer::notify()4 1          |
  17:08:53.286  void MySuspendedTimer::notify()1 2          |
  17:08:54.300  bool OnTimer3()                              |
  17:08:54.300  void MySuspendedTimer::notify()1 3           |
  17:08:55.314  void MyCountableTimer::notify()2 4            |
  17:08:55.314  void MySuspendedTimer::notify()1 4            |
  17:08:55.314  Forcing all timers to stop                    |
```

El primer mensaje del temporizador de dos segundos llega, como era de esperar, unos 2 segundos después del inicio (decimos «unos» porque el temporizador de hardware tiene una limitación en la precisión y, además, otra carga del ordenador afecta a la ejecución). Un segundo después, el temporizador de tres segundos se dispara por primera vez. El segundo golpe del temporizador de dos segundos coincide con la primera salida del temporizador de cuatro segundos. Tras una única ejecución del temporizador de cinco segundos, los mensajes del temporizador de un segundo comienzan a aparecer en el registro regularmente (su contador aumenta de 0 a 4). En su última iteración, detiene todos los temporizadores.

5.6.3 Temporizador de alta precisión: EventSetMillisecondTimer

Si su programa requiere que el temporizador se active con una frecuencia superior a 1 segundo, en lugar de *EventSetTimer* utilice la función *EventSetMillisecondTimer*.

Los temporizadores con unidades diferentes no pueden iniciarse al mismo tiempo: debe utilizarse una función u otra. El tipo de temporizador que se ejecuta en realidad viene determinado por la función a la que se llamó posteriormente. Todas las características inherentes al temporizador estándar siguen siendo válidas para el temporizador de alta precisión.

bool EventSetMillisecondTimer(int milliseconds)

La función indica al terminal cliente que es necesario generar eventos de temporizador para este Asesor Experto o indicador con una frecuencia inferior a un segundo. La periodicidad se fija en milisegundos (parámetro *milliseconds*).

La función devuelve un signo de éxito (*true*) o de error (*false*).

> Cuando trabaje en el probador de estrategias, tenga en cuenta que, cuanto más corto sea el periodo del temporizador, más tiempo requerirá la prueba, ya que aumenta el número de llamadas al manejador de eventos del temporizador.

Durante el funcionamiento normal, los eventos del temporizador no se generan más de una vez cada 10-16 milisegundos, lo que se debe a limitaciones del hardware.

Para demostrar cómo trabajar con el temporizador de milisegundos, ampliemos el ejemplo del indicador *MultipleTimers.mq5*. Dado que la activación del temporizador global se deja al programa de aplicación, podemos cambiar fácilmente el tipo del temporizador, dejando las clases lógicas de temporizador sin cambios. La única diferencia será que sus multiplicadores se aplicarán al periodo base en milisegundos que especificaremos en la función *EventSetMillisecondTimer*.

Para seleccionar el tipo de temporizador, describiremos la enumeración y añadiremos una nueva variable de entrada.

```
enum TIMER_TYPE
{
   Seconds,
   Milliseconds
};

input TIMER_TYPE TimerType = Seconds;
```

Por defecto, utilizamos un segundo temporizador. En *OnInit*, inicie el temporizador del tipo deseado.

```
void OnInit()
{
   Print(__FUNCSIG__, " ", BaseTimerPeriod, " ", EnumToString(TimerType));
   if(TimerType == Seconds)
   {
      EventSetTimer(BaseTimerPeriod);
   }
   else
   {
      EventSetMillisecondTimer(BaseTimerPeriod);
   }
}
```

Veamos qué se mostrará en el registro al elegir un temporizador de milisegundos.

```
                                                // time ms
  17:27:54.483  void OnInit() 1 Milliseconds        |
  17:27:54.514  void MyCountableTimer::notify()2 0    |           +31
  17:27:54.545  bool OnTimer3()                        |          +31
  17:27:54.561  void MyCountableTimer::notify()2 1      |         +16
  17:27:54.561  void MyCountableTimer::notify()4 0      |
  17:27:54.577  bool OnTimer5()                         |        +16
  17:27:54.608  void MyCountableTimer::notify()2 2        |       +31
  17:27:54.608  bool OnTimer3()                           |
  17:27:54.608  void MySuspendedTimer::notify()1 0        |
  17:27:54.623  void MySuspendedTimer::notify()1 1         |      +15
  17:27:54.655  void MyCountableTimer::notify()2 3          |     +32
  17:27:54.655  void MyCountableTimer::notify()4 1          |
  17:27:54.655  void MySuspendedTimer::notify()1 2          |
  17:27:54.670  bool OnTimer3()                              |    +15
  17:27:54.670  void MySuspendedTimer::notify()1 3           |
  17:27:54.686  void MyCountableTimer::notify()2 4            |   +16
  17:27:54.686  void MySuspendedTimer::notify()1 4            |
  17:27:54.686  Forcing all timers to stop                    |
```

La secuencia de generación de eventos es exactamente la misma que vimos para el segundo temporizador, pero todo sucede mucho más rápido, casi instantáneamente.

Debido a que la precisión del temporizador del sistema está limitada a un par de decenas de milisegundos, el intervalo real entre sucesos supera con creces el inalcanzablemente pequeño espacio de tiempo de 1 milisegundo. Además, hay un diferencial del tamaño de un «paso». Así, incluso cuando se utiliza un temporizador de milisegundos, es conveniente no ceñirse a periodos inferiores a unas pocas decenas de milisegundos.

5.7 Trabajar con gráficos

La mayoría de los programas MQL, como scripts, indicadores y Asesores Expertos, se ejecutan en gráficos. Sólo los servicios se ejecutan en segundo plano, sin estar vinculados a un horario. Se proporciona un rico conjunto de funciones para obtener y cambiar las propiedades de los gráficos, analizar su lista y buscar otros programas en ejecución.

Dado que los gráficos son el entorno natural de los indicadores, ya hemos tenido ocasión de familiarizarnos con algunas de estas características en los capítulos anteriores sobre indicadores. En este capítulo estudiaremos todas estas funciones de forma específica.

Cuando trabajemos con gráficos utilizaremos el concepto de ventana. Una ventana es un área dedicada en la que se muestran gráficos de precios y/o gráficos de indicadores. La ventana superior y, por regla general, la más grande contiene gráficos de precios, tiene el número 0 y siempre existe. Todas las ventanas adicionales que se añaden a la parte inferior al colocar los indicadores se numeran de 1 en adelante (numeración de arriba abajo). Cada subventana existe sólo mientras tenga al menos un indicador.

Dado que el usuario puede eliminar todos los indicadores de una subventana arbitraria, incluido el que no es el último (el más bajo), los índices de las subventanas restantes pueden disminuir.

El modelo de eventos de los gráficos relacionado con la recepción y el procesamiento de notificaciones sobre eventos en los gráficos y la generación de eventos personalizados se tratará en un capítulo aparte.

Además de los «gráficos en ventanas» comentados aquí, MetaTrader 5 también permite crear «gráficos en objetos». Nos ocuparemos de los objetos gráficos en el próximo capítulo.

5.7.1 Funciones para obtener las propiedades básicas del gráfico actual

En muchos ejemplos del libro ya hemos tenido que utilizar Variables predefinidas, las cuales contienen las principales propiedades del gráfico y su símbolo de trabajo. Los programas MQL también tienen acceso a funciones que devuelven los valores de algunas de estas variables. No importa lo que se utilice, una variable o una función, por lo que puede utilizar los estilos de código fuente que prefiera.

Cada gráfico se caracteriza por un símbolo de trabajo y un marco temporal, que se pueden encontrar utilizando las funciones *Symbol* y *Period*, respectivamente. Además, MQL5 proporciona un acceso simplificado a las dos propiedades de símbolo más utilizadas: el tamaño del punto de precio (*Point*) y el número asociado de dígitos significativos (*Digits*) después del punto decimal en el precio.

string Symbol()

La función *Symbol* devuelve el nombre del símbolo del gráfico actual, es decir, el valor de la variable del sistema *_Symbol*. Para obtener el símbolo de un gráfico arbitrario, existe la función *ChartSymbol*, que opera a partir del identificador del gráfico. Hablaremos de los métodos para obtener identificadores de gráficos un poco más adelante.

ENUM_TIMEFRAMES Period()

La función *Period* devuelve el valor del marco temporal (ENUM_TIMEFRAMES) del gráfico actual, que corresponde a la variable *_Period*. Para obtener el marco temporal de un gráfico arbitrario, utilice la función *ChartPeriod*, y también necesita un identificador como parámetro.

double Point()

La función *Point* devuelve el tamaño en puntos del instrumento actual en la divisa de cotización, que es el mismo que el valor de la variable *_Point*.

int Digits()

La función devuelve el número de decimales después del punto decimal, que determina la precisión de la medición del precio del símbolo del gráfico actual, lo que equivale a la variable *_Digits*.

Otras propiedades de la herramienta actual le permiten obtener *SymbolInfo*-functions, que en un caso más general proporcionan un análisis de todos los instrumentos.

En el siguiente ejemplo sencillo del script *ChartMainProperties.mq5* se registran las propiedades descritas en esta sección.

```
void OnStart()
{
   PRTF(_Symbol);
   PRTF(Symbol());
   PRTF(_Period);
   PRTF(Period());
   PRTF(_Point);
   PRTF(Point());
   PRTF(_Digits);
   PRTF(Digits());
   PRTF(DoubleToString(_Point, _Digits));
   PRTF(EnumToString(_Period));
}
```

Para el gráfico EURUSD,H1, obtendremos las siguientes entradas de registro:

```
_Symbol=EURUSD / ok
Symbol()=EURUSD / ok
_Period=16385 / ok
Period()=16385 / ok
_Point=1e-05 / ok
Point()=1e-05 / ok
_Digits=5 / ok
Digits()=5 / ok
DoubleToString(_Point,_Digits)=0.00001 / ok
EnumToString(_Period)=PERIOD_H1 / ok
```

5.7.2 Identificación de gráficos

Cada gráfico en MetaTrader 5 opera en una ventana independiente y tiene un identificador único. Para los programadores familiarizados con los principios de funcionamiento de Windows, nos gustaría aclarar que este identificador no es un manejador de ventana del sistema (aunque la API de MQL5 permite obtener este último a través de la propiedad CHART_WINDOW_HANDLE). Como sabemos, además del área principal de trabajo del gráfico con cotizaciones, existen áreas adicionales (subventanas) con indicadores que tienen la propiedad *indicator_separate_window*. Todas las subventanas forman parte del gráfico y pertenecen a la misma ventana de Windows.

long ChartID()

La función devuelve un identificador único para el gráfico actual.

Muchas de las funciones que veremos requieren un ID de gráfico como parámetro, pero puede especificar 0 para el gráfico actual en lugar de llamar a *ChartID*. Tiene sentido utilizar *ChartID* en los casos en que el identificador se envía entre programas MQL; por ejemplo, al intercambiar mensajes (eventos personalizados) en el mismo gráfico o en gráficos diferentes. Si se especifica un ID no válido se producirá el error ERR_CHART_WRONG_ID (4101).

El ID del gráfico suele ser el mismo de una sesión a otra.

Demostraremos la función *ChartID* y el aspecto de los identificadores en el script de ejemplo *ChartList1.mq5* después de estudiar el método para obtener una lista de gráficos.

5.7.3 Obtener la lista de gráficos

Un programa MQL puede obtener una lista de los gráficos abiertos en el terminal (tanto windows como objetos gráficos) mediante las funciones *ChartFirst* y *ChartNext*.

long ChartFirst()
long ChartNext(long chartId)

La función *ChartFirst* devuelve el identificador del primer gráfico del terminal de cliente. MetaTrader 5 mantiene una lista interna de todos los gráficos, el orden de los cuales puede diferir con respecto al que vemos en pantalla; por ejemplo, en las pestañas de las ventanas cuando están maximizadas. En concreto, el orden en la lista puede cambiar como resultado de arrastrar pestañas y desacoplar y acoplar ventanas. Después de cargar el terminal, el orden visible de los marcadores es el mismo que el de la vista de lista interna.

La función *ChartNext* devuelve el ID del gráfico siguiente al gráfico con el *chartId* especificado.

A diferencia de otras funciones para trabajar con gráficos, el valor 0 en el parámetro *ChartId* no significa el gráfico actual, sino el principio de la lista. En otras palabras: la llamada a *ChartNext(0)* equivale a *ChartFirst*.

Si se llega al final de la lista, la función devuelve -1.

El script *ChartList1.mq5* muestra la lista de gráficos en el registro. El trabajo principal lo realiza la función *ChartList* que se llama desde *OnStart*. Al principio de la función obtenemos el identificador del gráfico actual utilizando *ChartID* y a continuación lo marcamos con un asterisco en la lista. Al final, se obtiene el número total de gráficos.

```
void OnStart()
{
   ChartList();
}

void ChartList()
{
   const long me = ChartID();
   long id = ChartFirst();
   // long id = ChartNext(0); - analogue of calling ChartFirst()
   int count = 0, used = 0;
   Print("Chart List\nN, ID, *active");
   // keep iterating over charts until there are none left
   while(id != -1)
   {
      const string header = StringFormat("%d %lld %s",
         count, id, (id == me ? " *" : ""));

      // fields: N, id, label of the current chart
      Print(header);
      count++;
      id = ChartNext(id);
   }
   Print("Total chart number: ", count);
}
```

A continuación se muestra un ejemplo de resultado:

```
Chart List
N, ID, *active
0 132358585987782873
1 132360375330772909  *
2 132544239145024745
3 132544239145024732
4 132544239145024744
Total chart number: 5
```

5.7.4 Obtener el símbolo y el marco temporal de un gráfico arbitrario

Dos propiedades fundamentales de cualquier gráfico son su símbolo de trabajo y su marco temporal. Como vimos anteriormente, estas propiedades para el gráfico actual están disponibles como variables integradas *_Symbol* y *_Period*, así como a través de las funciones *Symbol* y*Period* pertinentes. Las siguientes funciones pueden utilizarse para determinar las mismas propiedades para otros gráficos: *ChartSymbol* y *ChartPeriod*.

string ChartSymbol(long chartId = 0)

La función devuelve el nombre del símbolo del gráfico con el identificador especificado. Si el parámetro es 0, se asume el gráfico actual.

Si el gráfico no existe, se devuelve una cadena vacía («») y *_LastError* establece el código de error ERR_CHART_WRONG_ID (4101).

ENUM_TIMEFRAMES ChartPeriod(long chartId = 0)

La función devuelve el valor del periodo para el gráfico con el identificador especificado.

Si el gráfico no existe, se devuelve 0.

El script *ChartList2.mq5*, similar a *ChartList1.mq5* genera una lista de gráficos que indica el símbolo y el marco temporal.

```
#include <MQL5Book/Periods.mqh>

void OnStart()
{
   ChartList();
}

void ChartList()
{
   const long me = ChartID();
   long id = ChartFirst();
   int count = 0;

   Print("Chart List\nN, ID, Symbol, TF, *active");
   // keep iterating over charts until there are none left
   while(id != -1)
   {
      const string header = StringFormat("%d %lld %s %s %s",
         count, id, ChartSymbol(id), PeriodToString(ChartPeriod(id)),
         (id == me ? " *" : ""));

      // fields: N, id, symbol, timeframe, label of the current chart
      Print(header);
      count++;
      id = ChartNext(id);
   }
   Print("Total chart number: ", count);
}
```

A continuación se muestra un ejemplo del contenido del registro después de ejecutar el script en el gráfico EURUSD, H1 (en la segunda línea).

```
Chart List
N, ID, Symbol, TF, *active
0 132358585987782873 EURUSD M15
1 132360375330772909 EURUSD H1  *
2 132544239145024745 XAUUSD H1
3 132544239145024732 USDRUB D1
4 132544239145024744 EURUSD H1
Total chart number: 5
```

MQL5 permite no sólo identificar, sino también cambiar el símbolo y el marco temporal de cualquier gráfico.

5.7.5 Visión general de funciones para trabajar con el conjunto completo de propiedades de gráfico

Las propiedades de los gráficos son legibles y editables a través de los grupos de funciones *ChartSet-* y *ChartGet-*, cada uno de los cuales contiene propiedades de un tipo determinado: números reales (*double*), números enteros (*long*, *int*, *datetime*, *color*, *bool*, enums) y cadenas.

Todas las funciones reciben el ID del gráfico como primer parámetro. El valor 0 significa el gráfico actual, es decir, equivale a pasar el resultado de la llamada *ChartID()*. Sin embargo, esto no significa que el ID del gráfico actual sea 0.

Las constantes que describen todas las propiedades forman tres enumeraciones ENUM_CHART_PROPERTY_INTEGER, ENUM_CHART_PROPERTY_DOUBLE, ENUM_CHART_PROPERTY_STRING, que se utilizan como parámetros de función para el tipo correspondiente. Puede encontrar una tabla resumen de todas las propiedades en la documentación de MQL5, en la página sobre propiedades de gráficos . En las siguientes secciones de este capítulo iremos abordado poco a poco prácticamente todas las propiedades, agrupándolas según su finalidad. La única excepción son las propiedades de gestión de eventos en el gráfico, que describiremos en la sección correspondiente del capítulo sobre eventos.

A los elementos de las tres enumeraciones se les asignan valores tales que forman una única lista sin intersecciones (repeticiones). Esto permite determinar el tipo de enumeración mediante un valor específico. Por ejemplo, dada una constante, podemos intentar convertirla de forma sistemática en una cadena con el nombre de uno de los enums hasta que lo consigamos.

```
int value = ...;

ResetLastError(); // clear the error code if there was one
EnumToString((ENUM_CHART_PROPERTY_INTEGER)value); // resulting string is not importan
if(_LastError == 0) // analyze if there is a new error
{
   // success is an element of ENUM_CHART_PROPERTY_INTEGER
   return ChartGetInteger(0, (ENUM_CHART_PROPERTY_INTEGER)value);
}

ResetLastError();
EnumToString((ENUM_CHART_PROPERTY_DOUBLE)value);
if(_LastError == 0)
{
   // success is an ENUM_CHART_PROPERTY_DOUBLE element
   return ChartGetDouble(0, (ENUM_CHART_PROPERTY_DOUBLE)value);
}

... // continue a similar check for ENUM_CHART_PROPERTY_STRING
```

Más adelante utilizaremos este enfoque en los scripts de prueba.

Algunas propiedades (por ejemplo, el número de barras visibles) son de sólo lectura y no pueden modificarse. Además, se marcarán con «r/o» (sólo lectura).

Las funciones de lectura de propiedades tienen una forma corta y una forma larga: la forma corta devuelve directamente el valor solicitado, y la forma larga devuelve un atributo booleano de éxito (*true*) o error (*false*), mientras que el valor en sí se coloca en el último parámetro pasado por referencia.

Cuando se utiliza la forma corta es especialmente importante comprobar el código de error en la variable *_LastError*, ya que el valor 0 (NULL) devuelto en caso de problemas puede ser generalmente correcto.

Al acceder a algunas propiedades debe especificar un parámetro adicional *window*, que se utiliza para indicar la ventana/subventana del gráfico. 0 significa la ventana principal. Las subventanas se numeran empezando por 1. Algunas propiedades se aplican al gráfico en su conjunto, por lo que tienen variantes de función sin el parámetro *window*.

A continuación se muestran los prototipos de funciones para leer y escribir propiedades de enteros. Tenga en cuenta que el tipo de valores que contienen es *long*.

```
bool ChartSetInteger(long chartId, ENUM_CHART_PROPERTY_INTEGER property, long value)
bool ChartSetInteger(long chartId, ENUM_CHART_PROPERTY_INTEGER property, int window, long value)
long ChartGetInteger(long chartId, ENUM_CHART_PROPERTY_INTEGER property, int window = 0)
bool ChartGetInteger(long chartId, ENUM_CHART_PROPERTY_INTEGER property, int window, long &value)
```

Las funciones para propiedades reales se describen de forma similar. No hay propiedades reales escribibles para las subventanas, por lo que sólo hay una forma de *ChartSetDouble*, que es sin el parámetro *window*.

```
bool ChartSetDouble(long chartId, ENUM_CHART_PROPERTY_DOUBLE property, double value)
double ChartGetDouble(long chartId, ENUM_CHART_PROPERTY_DOUBLE property, int window = 0)
bool ChartGetDouble(long chartId, ENUM_CHART_PROPERTY_DOUBLE property, int window, double &value)
```

Lo mismo se aplica a las propiedades de las cadenas, pero hay que tener en cuenta un matiz más: la longitud de la cadena no puede superar los 2045 caracteres (se cortarán los caracteres sobrantes).

```
bool ChartSetString(long chartId, ENUM_CHART_PROPERTY_STRING property, string value)
string ChartGetString(long chartId, ENUM_CHART_PROPERTY_STRING property)
bool ChartGetString(long chartId, ENUM_CHART_PROPERTY_STRING property, string &value)
```

Cuando se leen propiedades utilizando la forma abreviada de *ChartGetInteger/ChartGetDouble*, el parámetro *window* es opcional y por defecto se toma la ventana principal (*window=0*).

Las funciones para ajustar las propiedades del gráfico (*ChartSetInteger*, *ChartSetDouble*, *ChartSetString*) son asíncronas y sirven para enviar comandos de cambio al gráfico. Si estas funciones se ejecutan con éxito, el comando se añade a la cola común de eventos del gráfico y se devuelve *true*. Cuando se produce un error, la función devuelve *false*. En este caso, debe comprobar el código de error en la variable *_LastError*.

Las propiedades del gráfico se modifican más tarde, durante el procesamiento de la cola de eventos de este gráfico y, por regla general, con cierto retardo, por lo que no debe esperar una actualización inmediata del gráfico tras aplicar nuevos ajustes. Para forzar la actualización del aspecto y las propiedades del gráfico, utilice la función *ChartRedraw*. Si desea modificar varias propiedades del gráfico a la vez, deberá llamar a las funciones correspondientes en un bloque de código y, a continuación, una vez en *ChartRedraw*.

En general, el terminal actualiza automáticamente el gráfico en respuesta a eventos tales como la llegada de una nueva cotización, cambios en el tamaño de la ventana del gráfico, escalado, desplazamiento, adición de un indicador, etc.

Las funciones para obtener las propiedades de los gráficos (*ChartGetInteger*, *ChartGetDouble*, *ChartGetString*) son síncronas, es decir, el código de llamada espera el resultado de su ejecución.

5.7.6 Propiedades descriptivas de los gráficos

Las funciones de *ChartSetString/ChartGetString* permiten la lectura y el ajuste de las siguientes propiedades de cadena de los gráficos:

Identificador	Descripción
CHART_COMMENT	Texto del comentario del gráfico
CHART_EXPERT_NAME	Nombre del Asesor Experto que se ejecuta en el gráfico (r/o)
CHART_SCRIPT_NAME	Nombre del script que se ejecuta en el gráfico (r/o)

En el capítulo Visualización de mensajes en la ventana de gráficosdescubrimos la función *Comment*, que muestra un mensaje de texto en la esquina superior izquierda del gráfico. La propiedad CHART_COMMENT permite leer el comentario actual del gráfico: *ChartGetString(0, CHART_COMMENT)*. También es posible acceder a los comentarios de otros gráficos pasando sus identificadores a la función. Utilizando *ChartSetString* puede cambiar los comentarios del gráfico actual y de otros gráficos, si conoce su *ID*: *ChartSetString(ID, CHART_COMMENT, "text")*.

Si un Asesor Experto y/o un script se está ejecutando en cualquier gráfico, podemos averiguar sus nombres utilizando estas llamadas: *ChartGetString(ID, CHART_EXPERT_NAME)* y *ChartGetString(ID, CHART_SCRIPT_NAME)*.

El script *ChartList3.mq5*, similar a *ChartList2.mq5*, complementa la lista de gráficos con información sobre Asesores Expertos y scripts. Más adelante añadiremos información sobre los indicadores.

```
void ChartList()
{
   const long me = ChartID();
   long id = ChartFirst();
   int count = 0, used = 0, temp, experts = 0, scripts = 0;

   Print("Chart List\nN, ID, Symbol, TF, *active");
   // keep iterating over charts until there are none left
   while(id != -1)
   {
      temp =0;// sign of MQL programs on this chart
      const string header = StringFormat("%d %lld %s %s %s",
         count, id, ChartSymbol(id), PeriodToString(ChartPeriod(id)),
         (id == me ? " *" : ""));
      // fields: N, id, symbol, timeframe, label of the current chart
      Print(header);
      string expert = ChartGetString(id, CHART_EXPERT_NAME);
      string script = ChartGetString(id, CHART_SCRIPT_NAME);
      if(StringLen(expert) > 0) expert = "[E] " + expert;
      if(StringLen(script) > 0) script = "[S] " + script;
      if(expert != NULL || script != NULL)
      {
         Print(expert, " ", script);
         if(expert != NULL) experts++;
         if(script != NULL) scripts++;
         temp++;
      }
      count++;
      if(temp > 0)
      {
         used++;
      }
      id = ChartNext(id);
   }
   Print("Total chart number: ", count, ", with MQL-programs: ", used);
   Print("Experts: ", experts, ", Scripts: ", scripts);
}
```

Este es un ejemplo de la salida de este script:

```
Chart List
N, ID, Symbol, TF, *active
0 132358585987782873 EURUSD M15
1 132360375330772909 EURUSD H1  *
 [S] ChartList3
2 132544239145024745 XAUUSD H1
3 132544239145024732 USDRUB D1
4 132544239145024744 EURUSD H1
Total chart number: 5, with MQL-programs: 1
Experts: 0, Scripts: 1
```

Aquí puede ver que sólo se está ejecutando un script.

5.7.7 Comprobar el estado de la ventana principal

El par de funciones *ChartSetInteger/ChartGetInteger* permite conocer algunas de las características del estado del gráfico, así como modificar algunas de ellas.

Identificador	Descripción	Tipo de valor
CHART_BRING_TO_TOP	Actividad del gráfico (foco de entrada) por encima de todas las demás	bool
CHART_IS_MAXIMIZED	Gráfico maximizado	bool
CHART_IS_MINIMIZED	Gráfico minimizado	bool
CHART_WINDOW_HANDLE	Manejador Windows de la ventana del gráfico (r/o)	int
CHART_IS_OBJECT	Una bandera de que un gráfico es un objeto Chart (OBJ_CHART); *true* es para un objeto gráfico y *false* es para un gráfico normal (r/o)	bool

Como era de esperar, el manejador de la ventana y el atributo del objeto gráfico son de sólo lectura. Otras propiedades son editables: por ejemplo, llamando a *ChartSetInteger(ID, CHART_BRING_TO_TOP, true)* se activa el gráfico con el ID especificado.

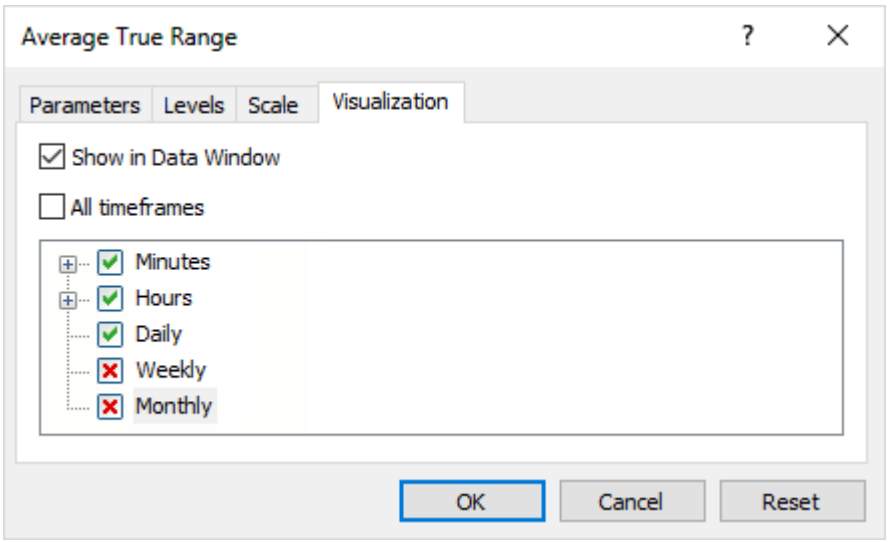
En el script *ChartList4.mq5* de la siguiente sección se ofrece un ejemplo de aplicación de propiedades.

5.7.8 Obtener el número y la visibilidad de las ventanas/subventanas

Mediante la función *ChartGetInteger*, un programa MQL puede averiguar el número de ventanas de un gráfico (incluidas las subventanas), así como su visibilidad.

Identificador	Descripción	Tipo de valor
CHART_WINDOWS_TOTAL	Número total de ventanas del gráfico, incluidas las subventanas del indicador (r/o)	int
CHART_WINDOW_IS_VISIBLE	Visibilidad de la subventana, el parámetro 'window' es el número de subventana (r/o)	bool

Algunas subventanas pueden estar ocultas si los indicadores colocados en ellas están desactivados en el marco temporal actual en el cuadro de diálogo Propiedades, en la pestaña Visualización. Es imposible restablecer todas las banderas: debido a la naturaleza del almacenamiento de plantillas tpl, dicho estado se interpreta como la habilitación de todos los marcos temporales. Por lo tanto, si el usuario desea ocultar la subventana durante algún tiempo, es necesario dejar al menos una bandera activada en el marco temporal menos utilizado.



Configuración de la visibilidad de los indicadores en diferentes marcos temporales

Cabe señalar que no hay herramientas estándar en MQL5 para la determinación programática del estado y la conmutación de banderas específicas. La forma más sencilla de simular dicho control es guardar la plantilla tpl y analizarla, con posible edición y carga posterior (véase la sección Trabajar con plantillas tpl).

En la nueva versión del script *ChartList4.mq5*, mostramos el número de subventanas (una ventana, que es la principal, está siempre presente), una señal de actividad del gráfico, una señal de un objeto gráfico y un manejador de Windows.

```
const int win = (int)ChartGetInteger(id, CHART_WINDOWS_TOTAL);
const string header = StringFormat("%d %lld %s %s %s %s %s %s %lld",
   count, id, ChartSymbol(id), PeriodToString(ChartPeriod(id)),
   (win > 1 ? "#" + (string)(win - 1) : ""), (id == me ? " *" : ""),
   (ChartGetInteger(id, CHART_BRING_TO_TOP, 0) ? "active" : ""),
   (ChartGetInteger(id, CHART_IS_OBJECT) ? "object" : ""),
   ChartGetInteger(id, CHART_WINDOW_HANDLE));
...
for(int i = 0; i < win; i++)
{
   const bool visible = ChartGetInteger(id, CHART_WINDOW_IS_VISIBLE, i);
   if(!visible)
   {
      Print("  ", i, "/Hidden");
   }
}
```

He aquí cuál podría ser el resultado:

```
Chart List
N, ID, Symbol, TF, #subwindows, *active, Windows handle
0 132358585987782873 EURUSD M15 #1    68030
1 132360375330772909 EURUSD H1  * active  68048
 [S] ChartList4
2 132544239145024745 XAUUSD H1     395756
3 132544239145024732 USDRUB D1     395768
4 132544239145024744 EURUSD H1 #2    461286
  2/Hidden
Total chart number: 5, with MQL-programs: 1
Experts: 0, Scripts: 1
```

En el primer gráfico (índice 0) hay una subventana (nº 1). Hay dos subventanas (nº 2) en el último gráfico, y la segunda está actualmente oculta. Más adelante, en la sección Gestionar indicadores en el gráfico, presentaremos la versión completa de *ChartList.mq5*, donde incluimos en el informe información sobre los indicadores situados en las subventanas y en la ventana principal.

> ¡Atención! Un gráfico dentro de un objeto gráfico siempre tiene la propiedad CHART_WINDOW_IS_VISIBLE igual a *true*, incluso si la visualización de objetos está desactivada en el marco temporal actual o en todos los marcos temporales.

5.7.9 Modos de visualización de gráficos

Cuatro propiedades de la enumeración ENUM_CHART_PROPERTY_INTEGER describen los modos de visualización de los gráficos. Todas estas propiedades están disponibles para la lectura a través de *ChartGetInteger*, y para el registro a través de *ChartSetInteger*, que permite cambiar la apariencia del gráfico.

Identificador	Descripción	Tipo de valor
CHART_MODE	Tipo de gráfico (velas, barras o líneas)	ENUM_CHART_MODE
CHART_FOREGROUND	Gráfico de precios en primer plano	bool
CHART_SHIFT	Modo de sangría del gráfico de precios desde el borde derecho	bool
CHART_AUTOSCROLL	Desplazamiento automático al borde derecho del gráfico	bool

Hay una enumeración especial ENUM_CHART_MODE para el modo CHART_MODE en MQL5. Sus elementos se muestran en la siguiente tabla.

Identificador	Descripción	Valor
CHART_BARS	Visualización en forma de barras	0
CHART_CANDLES	Visualización en forma de velas japonesas	1
CHART_LINE	Visualización como una línea trazada a precios de cierre	2

Implementemos el script *ChartMode.mq5*, que monitorizará el estado de los modos e imprimirá mensajes en el registro cuando se detecten cambios. Dado que los algoritmos de procesamiento de propiedades son de carácter general, los colocaremos en un archivo de encabezado independiente *ChartModeMonitor.mqh*, que luego conectaremos a diferentes pruebas.

Vamos a sentar las bases en una clase abstracta *ChartModeMonitorInterface*: proporciona métodos get- y set- sobrecargados para todos los tipos. Las clases derivadas tendrán que comprobar directamente las propiedades en la medida necesaria anulando el método virtual *snapshot*.

```
class ChartModeMonitorInterface
{
public:
   long get(const ENUM_CHART_PROPERTY_INTEGER property, const int window = 0)
   {
      return ChartGetInteger(0, property, window);
   }
   double get(const ENUM_CHART_PROPERTY_DOUBLE property, const int window = 0)
   {
      return ChartGetDouble(0, property, window);
   }
   string get(const ENUM_CHART_PROPERTY_STRING property)
   {
      return ChartGetString(0, property);
   }
   bool set(const ENUM_CHART_PROPERTY_INTEGER property, const long value, const int w
   {
      return ChartSetInteger(0, property, window, value);
   }
   bool set(const ENUM_CHART_PROPERTY_DOUBLE property, const double value)
   {
      return ChartSetDouble(0, property, value);
   }
   bool set(const ENUM_CHART_PROPERTY_STRING property, const string value)
   {
      return ChartSetString(0, property, value);
   }

   virtual void snapshot() = 0;
   virtual void print() { };
   virtual void backup() { }
   virtual void restore() { }
};
```

La clase también tiene métodos reservados: *print*, por ejemplo, para enviar a un registro, *backup* para guardar el estado actual, y *restore* para recuperarlo. Se declaran no abstractos, sino con una implementación vacía, ya que son opcionales.

Tiene sentido definir ciertas clases para propiedades de diferentes tipos como una única plantilla heredada de *ChartModeMonitorInterface* y que acepte tipos paramétricos de valor (T) y enumeración (E). Por ejemplo, para las propiedades de número entero, tendría que configurar *T=long* y *E=ENUM_CHART_PROPERTY_INTEGER*.

El objeto contiene el array *data* para almacenar pares [clave,valor] con todas las propiedades solicitadas. Tiene un tipo genérico *MapArray<K,V>*, que introdujimos anteriormente para el indicador *IndUnityPercent* en el capítulo Indicadores multidivisa y de marco temporal múltiple. Su peculiaridad radica en que, además del acceso habitual a los elementos del array por números, se puede utilizar el direccionamiento por clave.

Para rellenar el array se pasa un array de números enteros al constructor, mientras que primero se comprueba que los enteros cumplen los identificadores de la enumeración E dada mediante el método *detect*. Todas las propiedades correctas se leen inmediatamente a través de la llamada *get*, y los valores resultantes se almacenan en el mapa junto con sus identificadores.

```
#include <MQL5Book/MapArray.mqh>

template<typename T,typename E>
class ChartModeMonitorBase: public ChartModeMonitorInterface
{
protected:
   MapArray<E,T> data; // array-map of pairs [property, value]

   // the method checks if the passed constant is an enumeration element,
   // and if it is, then add it to the map array
   bool detect(const int v)
   {
      ResetLastError();
      EnumToString((E)v); // resulting string is not used
      if(_LastError == 0) // it only matters if there is an error or not
      {
         data.put((E)v, get((E)v));
         return true;
      }
      return false;
   }

public:
   ChartModeMonitorBase(int &flags[])
   {
      for(int i = 0; i < ArraySize(flags); ++i)
      {
         detect(flags[i]);
      }
   }

   virtual void snapshot() override
   {
      MapArray<E,T> temp;
      // collect the current state of all properties
      for(int i = 0; i < data.getSize(); ++i)
      {
         temp.put(data.getKey(i), get(data.getKey(i)));
      }

      // compare with previous state, display differences
      for(int i = 0; i < data.getSize(); ++i)
      {
         if(data[i] != temp[i])
         {
            Print(EnumToString(data.getKey(i)), " ", data[i], " -> ", temp[i]);
         }
      }

      // save for next comparison
      data = temp;
```

```
      }
      ...
   };
```

El método *snapshot* itera a través de todos los elementos del array y solicita el valor de cada propiedad. Como queremos detectar cambios, los nuevos datos se almacenan primero en un array de mapas temporal *temp*. A continuación, los arrays *data* y *temp* se comparan elemento por elemento y, por cada diferencia, se muestra un mensaje con el nombre de la propiedad, su valor antiguo y su nuevo valor. En este ejemplo simplificado sólo se utiliza el diario. No obstante, si es necesario, el programa puede llamar a algunas funciones de la aplicación que adaptan el comportamiento al entorno.

Los métodos *print*, *backup* y *restore* se implementan de la forma más sencilla posible.

```
template<typename T,typename E>
class ChartModeMonitorBase: public ChartModeMonitorInterface
{
protected:
   ...
   MapArray<E,T> store; // backup
public:
   ...
   virtual void print() override
   {
      data.print();
   }
   virtual void backup() override
   {
      store = data;
   }

   virtual void restore() override
   {
      data = store;
      // restore chart properties
      for(int i = 0; i < data.getSize(); ++i)
      {
         set(data.getKey(i), data[i]);
      }
   }
```

Una combinación de métodos *backup/restore* permite guardar el estado del gráfico antes de iniciar experimentos con él y, una vez finalizado el script de prueba, restaurarlo todo como estaba.

Por último, la última clase del archivo *ChartModeMonitor.mqh* es *ChartModeMonitor*. Combina tres instancias de *ChartModeMonitorBase*, creadas para las combinaciones disponibles de tipos de propiedad. Tienen un array de punteros *m* a la interfaz base *ChartModeMonitorInterface*. La propia clase también deriva de ella.

```
#include <MQL5Book/AutoPtr.mqh>

#define CALL_ALL(A,M) for(int i = 0, size = ArraySize(A); i < size; ++i) A[i][].M

class ChartModeMonitor: public ChartModeMonitorInterface
{
   AutoPtr<ChartModeMonitorInterface> m[3];

public:
   ChartModeMonitor(int &flags[])
   {
      m[0] = new ChartModeMonitorBase<long,ENUM_CHART_PROPERTY_INTEGER>(flags);
      m[1] = new ChartModeMonitorBase<double,ENUM_CHART_PROPERTY_DOUBLE>(flags);
      m[2] = new ChartModeMonitorBase<string,ENUM_CHART_PROPERTY_STRING>(flags);
   }

   virtual void snapshot() override
   {
      CALL_ALL(m, snapshot());
   }

   virtual void print() override
   {
      CALL_ALL(m, print());
   }

   virtual void backup() override
   {
      CALL_ALL(m, backup());
   }

   virtual void restore() override
   {
      CALL_ALL(m, restore());
   }
};
```

Para simplificar el código se utiliza aquí la macro CALL_ALL, que llama al método especificado para todos los objetos del array, y lo hace teniendo en cuenta el operador sobrecargado [] de la clase *AutoPtr* (se utiliza para desreferenciar un puntero inteligente y obtener un puntero directo al objeto «protegido»).

El destructor suele encargarse de liberar los objetos, pero en este caso se decidió utilizar el array *AutoPtr* (esta clase se abordó en la sección Plantillas de tipos de objeto). Esto garantiza la eliminación automática de los objetos dinámicos cuando el array *m* se libera normalmente.

En el archivo *ChartModeMonitorFull.mqh* se proporciona una versión más completa del monitor con soporte para números de subventana.

Basándose en la clase *ChartModeMonitor*, puede implementar fácilmente la secuencia de comandos prevista *ChartMode.mq5*. Su tarea consiste en comprobar el estado de un determinado conjunto de propiedades cada medio segundo. Ahora estamos utilizando aquí un bucle infinito y *Sleep*, pero pronto

vamos a aprender a reaccionar a los eventos en los gráficos de una manera diferente: debido a las notificaciones de la terminal.

```
#include <MQL5Book/ChartModeMonitor.mqh>

void OnStart()
{
   int flags[] =
   {
      CHART_MODE, CHART_FOREGROUND, CHART_SHIFT, CHART_AUTOSCROLL
   };
   ChartModeMonitor m(flags);
   Print("Initial state:");
   m.print();
   m.backup();

   while(!IsStopped())
   {
      m.snapshot();
      Sleep(500);
   }
   m.restore();
}
```

Ejecute el script en cualquier gráfico e intente cambiar los modos utilizando los botones de herramientas. De esta forma puede acceder a todos los elementos excepto a CHART_FOREGROUND, que puede cambiarse desde el cuadro de diálogo de propiedades (la pestaña *Common*, bandera *Chart on top*).

Botones de la barra de herramientas para cambiar de modo de gráfico

Por ejemplo, el siguiente registro se creó cambiando la visualización de velas a barras, de barras a líneas y de nuevo a velas, y después activando la sangría y el desplazamiento automático hasta el principio.

```
Initial state:
    [key] [value]
[0]     0       1
[1]     1       0
[2]     2       0
[3]     4       0
CHART_MODE 1 -> 0
CHART_MODE 0 -> 2
CHART_MODE 2 -> 1
CHART_SHIFT 0 -> 1
CHART_AUTOSCROLL 0 -> 1
```

Un ejemplo más práctico del uso de la propiedad CHART_MODE es una versión mejorada del indicador *IndSubChart.mq5* (ya hablamos de su versión simplificada *IndSubChartSimple.mq5* en la sección Indicadores multidivisa y de marco temporal múltiple). El indicador está diseñado para mostrar cotizaciones de un símbolo ajeno en una subventana, y antes teníamos que solicitar al usuario un método de visualización (velas, barras o líneas) a través de un parámetro de entrada. Ahora el parámetro ya no es necesario porque podemos cambiar automáticamente el indicador al modo que se utiliza en la ventana principal.

El modo actual se almacena en la variable global *mode* y se asigna primero durante la inicialización.

```
ENUM_CHART_MODE mode = 0;

int OnInit()
{
   ...
   mode = (ENUM_CHART_MODE)ChartGetInteger(0, CHART_MODE);
   ...
}
```

La detección de un nuevo modo se realiza mejor en un manejador de eventos especialmente diseñado *OnChartEvent*, que estudiaremos en un capítuloaparte. En esta etapa, es importante saber que con cualquier cambio en el gráfico, el programa MQL puede recibir notificaciones del terminal si el código describe una función con este prototipo predefinido (nombre y lista de parámetros). En concreto, su primer parámetro contiene un identificador de evento que describe su significado. Seguimos interesados en el gráfico en sí, por lo que comprobamos si *eventId* es igual a CHARTEVENT_CHART_CHANGE. Esto es necesario porque el manejador también es capaz de rastrear objetos gráficos, teclado, ratón y mensajes arbitrarios del usuario.

```
void OnChartEvent(const int eventId,
                  // parameters not used here
                   const long &, const double &, const string &)
{
   if(eventId == CHARTEVENT_CHART_CHANGE)
   {
      const ENUM_CHART_MODE newmode = (ENUM_CHART_MODE)ChartGetInteger(0, CHART_MODE)
      if(mode != newmode)
      {
         const ENUM_CHART_MODE oldmode = mode;
         mode = newmode;
         // change buffer bindings and rendering type on the go
         InitPlot(0, InitBuffers(mode), Mode2Style(mode));
         // TODO: we will auto-adjust colors later
         // SetPlotColors(0, mode);
         if(oldmode == CHART_LINE || newmode == CHART_LINE)
         {
            // switching to or from CHART_LINE mode requires updating the entire char
            // because the number of buffers changes
            Print("Refresh");
            ChartSetSymbolPeriod(0, _Symbol, _Period);
         }
         else
         {
           // when switching between candles and bars, it is enough
           // just redraw the chart in a new manner,
           // because data doesn't change (previous 4 buffers with values)
            Print("Redraw");
            ChartRedraw();
         }
      }
   }
}
```

Puede probar el nuevo indicador usted mismo ejecutándolo en el gráfico y cambiando los métodos de dibujo.

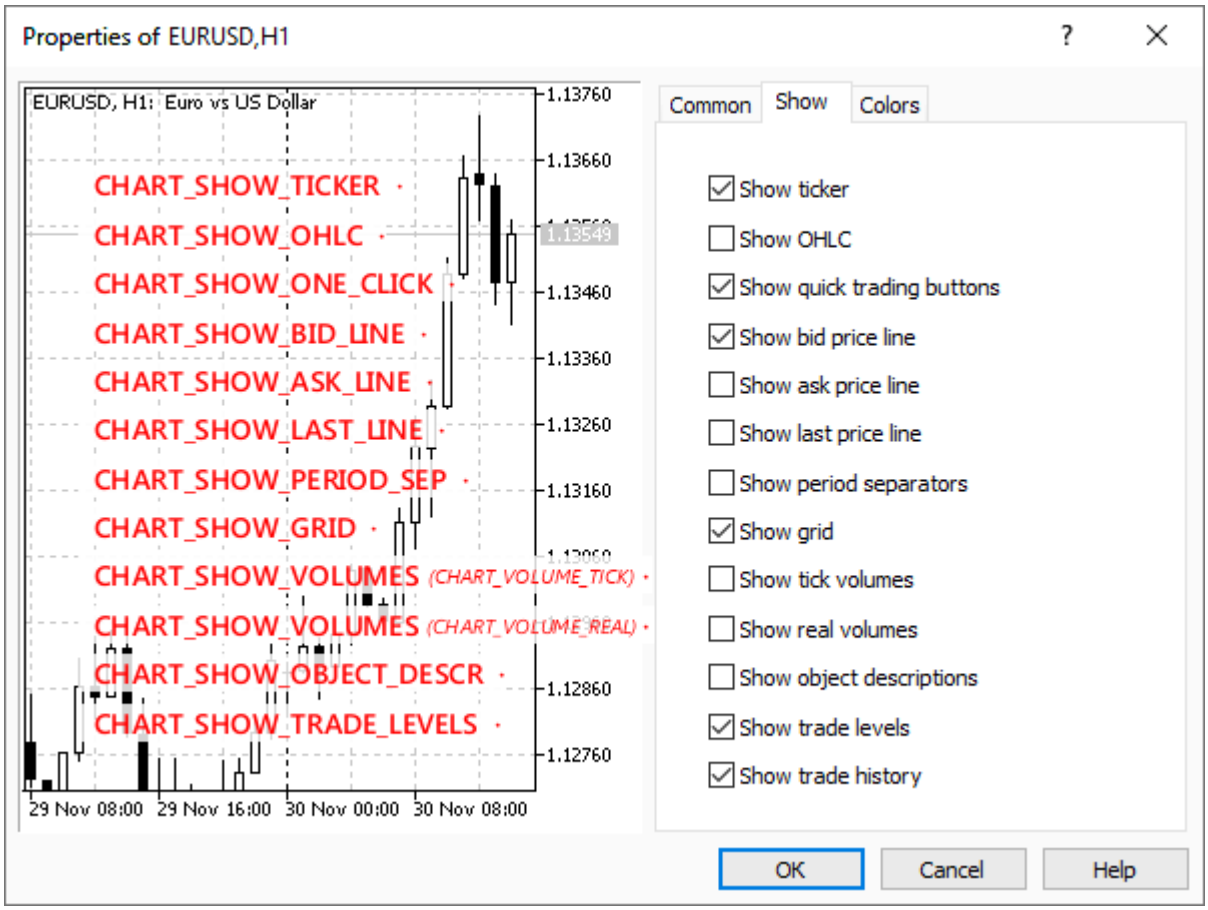
Estas no son todas las mejoras introducidas en *IndSubChart.mq5*. Un poco más adelante, en la sección sobre colores del gráfico, mostraremos el ajuste automático de los gráficos al esquema de colores del gráfico.

5.7.10 Gestionar la visibilidad de los elementos del gráfico

Un amplio conjunto de propiedades de ENUM_CHART_PROPERTY_INTEGER controla la visibilidad de los elementos del gráfico. Casi todos ellos son de tipo booleano: *true* corresponde a mostrar el elemento, y *false* corresponde a ocultarlo. La excepción es CHART_SHOW_VOLUMES, que utiliza la enumeración ENUM_CHART_VOLUME_MODE (véase más abajo).

Identificador	Descripción	Tipo de valor
CHART_SHOW	Visualización general del gráfico de precios. Si se establece en false, se desactiva la representación de cualquier atributo del gráfico de precios y se elimina todo el relleno a lo largo de los bordes del gráfico: escalas de tiempo y precio, barra de navegación rápida, marcadores de eventos del calendario, iconos de operaciones, información sobre herramientas de indicadores y barras, subventanas de indicadores, histogramas de volumen, etc.	bool
CHART_SHOW_TICKER	Mostrar el ticker del símbolo en la esquina superior izquierda. Al desactivar el ticker se desactiva automáticamente OHLC (CHART_SHOW_OHLC)	bool
CHART_SHOW_OHLC	Muestra los valores OHLC en la esquina superior izquierda. Al activar OHLC se activa automáticamente el ticker (CHART_SHOW_TICKER)	bool
CHART_SHOW_BID_LINE	Mostrar el valor de la oferta como una línea horizontal	bool
CHART_SHOW_ASK_LINE	Mostrar el valor Ask como una línea horizontal	bool
CHART_SHOW_LAST_LINE	Mostrar el valor Last como una línea horizontal	bool
CHART_SHOW_PERIOD_SEP	Mostrar separadores verticales entre periodos adyacentes	bool
CHART_SHOW_GRID	Mostrar cuadrícula en el gráfico	bool
CHART_SHOW_VOLUMES	Mostrar volúmenes en un gráfico	ENUM_CHART_VOLUME_MODE
CHART_SHOW_OBJECT_DESCR	Mostrar descripciones de texto de los objetos (las descripciones no se muestran para todos los tipos de objetos)	bool
CHART_SHOW_TRADE_LEVELS	Mostrar niveles de trading en el gráfico (niveles de posiciones abiertas, Stop Loss, Take Profit y órdenes pendientes).	bool
CHART_SHOW_DATE_SCALE	Mostrar la escala de fechas en el gráfico	bool
CHART_SHOW_PRICE_SCALE	Mostrar la escala de precios en el gráfico	bool

Identificador	Descripción	Tipo de valor
CHART_SHOW_ONE_CLICK	Mostrar el panel de trading rápido en el gráfico (opción «trading con un clic»)	bool



Banderas del cuadro de diálogo de configuración para algunas propiedades ENUM_CHART_PROPERTY_INTEGER

Algunas de estas propiedades están disponibles para el usuario desde el menú contextual del gráfico, mientras que otras sólo están disponibles desde el cuadro de diálogo de configuración. También hay ajustes que sólo se pueden cambiar desde MQL5; en concreto, la visualización de las escalas vertical (CHART_SHOW_DATE_SCALE) y horizontal (CHART_SHOW_DATE_SCALE), así como la visibilidad de todo el gráfico (CHART_SHOW). Cabe destacar especialmente el último caso, ya que desactivar la renderización es la solución ideal para crear su propia interfaz de programa utilizando recursos gráficos y objetos gráficos, que siempre se muestran, independientemente del valor de CHART_SHOW.

El libro viene con el script *ChartBlackout.mq5*, que cambia el modo CHART_SHOW de actual a inverso en cada ejecución.

```
void OnStart()
{
   ChartSetInteger(0, CHART_SHOW, !ChartGetInteger(0, CHART_SHOW));
}
```

Así, puede aplicarlo en un gráfico normal para borrar completamente la ventana y, a continuación, volver a aplicarlo para restaurar el aspecto anterior.

La mencionada enumeración ENUM_CHART_VOLUME_MODE contiene los siguientes miembros:

Identificador	Descripción	Valor
CHART_VOLUME_HIDE	Los volúmenes están ocultos.	0
CHART_VOLUME_TICK	Volúmenes de ticks	1
CHART_VOLUME_REAL	Volúmenes de trading (en su caso)	2

De forma similar al script *ChartMode.mq5*, implementamos un monitor de visibilidad para los elementos del gráfico en el script *ChartElements.mq5*. La principal diferencia radica en los distintos conjuntos de banderas controladas.

```
void OnStart()
{
   int flags[] =
   {
      CHART_SHOW,
      CHART_SHOW_TICKER, CHART_SHOW_OHLC,
      CHART_SHOW_BID_LINE, CHART_SHOW_ASK_LINE, CHART_SHOW_LAST_LINE,
      CHART_SHOW_PERIOD_SEP, CHART_SHOW_GRID,
      CHART_SHOW_VOLUMES,
      CHART_SHOW_OBJECT_DESCR,
      CHART_SHOW_TRADE_LEVELS,
      CHART_SHOW_DATE_SCALE, CHART_SHOW_PRICE_SCALE,
      CHART_SHOW_ONE_CLICK
   };
   ...
```

Además, tras crear una copia de seguridad de la configuración, desactivamos intencionadamente las escalas de tiempo y los precios mediante programación (cuando finalice el script, los restaurará a partir de la copia de seguridad).

```
   ...
   m.backup();

   ChartSetInteger(0, CHART_SHOW_DATE_SCALE, false);
   ChartSetInteger(0, CHART_SHOW_PRICE_SCALE, false);
   ...
}
```

A continuación se muestra un fragmento del registro con comentarios sobre las acciones realizadas. Las dos primeras entradas aparecieron exactamente porque las escalas se desactivaron en el código MQL después de crear la copia de seguridad inicial.

```
CHART_SHOW_DATE_SCALE 1 -> 0 // disabled the time scale in the MQL5 code
CHART_SHOW_PRICE_SCALE 1 -> 0 // disabled the price scale in the MQL5 code
CHART_SHOW_ONE_CLICK 0 -> 1 // disabled "One click trading"
CHART_SHOW_GRID 1 -> 0 // disable "Grid"
CHART_SHOW_VOLUMES 0 -> 2 // showed real "Volumes"
CHART_SHOW_VOLUMES 2 -> 1 // showed "Tick volumes"
CHART_SHOW_TRADE_LEVELS 1 -> 0 // disabled "Trade levels"
```

5.7.11 Desplazamientos horizontales

Otro matiz de la visualización de gráficos son las sangrías horizontales de los bordes izquierdo y derecho. Funcionan de forma ligeramente diferente, pero se describen en la misma enumeración ENUM_CHART_PROPERTY_DOUBLE y utilizan el tipo *double*.

Identificador	Descripción
CHART_SHIFT_SIZE	Sangría de la barra cero desde el borde derecho en porcentajes (de 10 a 50). Activo sólo cuando el modo CHART_SHIFT está activado. El desplazamiento se indica en el gráfico mediante un pequeño triángulo gris invertido en el marco superior, a la derecha de la ventana.
CHART_FIXED_POSITION	La ubicación de la posición fija del gráfico desde el borde izquierdo en porcentaje (de 0 a 100). Una posición fija del gráfico se indica mediante un pequeño triángulo gris en el eje temporal horizontal y sólo se muestra si está desactivado el desplazamiento automático hacia la derecha cuando llega un nuevo tick (CHART_AUTOCROLL). Una barra que está en una posición fija permanece en el mismo lugar al acercar y alejar la imagen. Por defecto, el triángulo se encuentra en la esquina inferior izquierda del gráfico.

Representación visual de las propiedades del relleno horizontal

Disponemos del script *ChartShifts.mq5* para comprobar el acceso a estas propiedades, que funciona de forma similar a *ChartMode.mq5* y sólo difiere en el conjunto de propiedades controladas.

```
void OnStart()
{
   int flags[] =
   {
      CHART_SHIFT_SIZE, CHART_FIXED_POSITION
   };
   ChartModeMonitor m(flags);
   ...
}
```

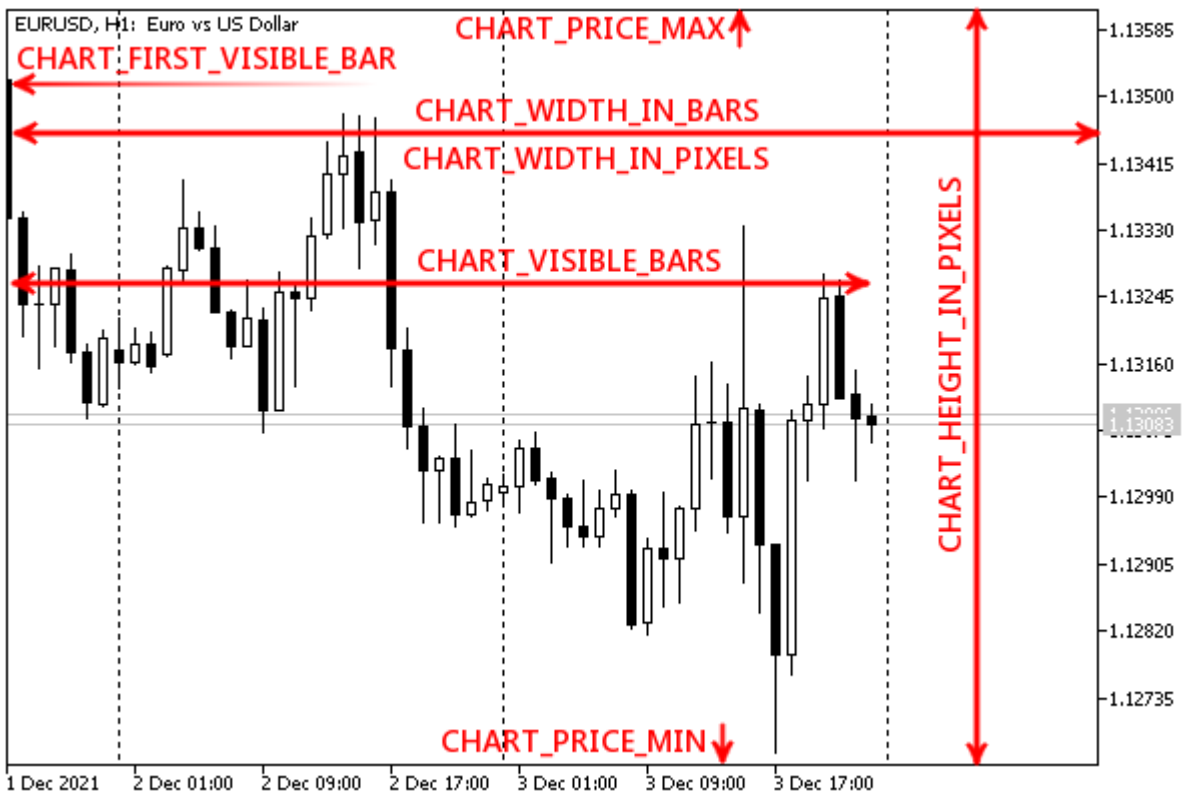
Arrastrando con el ratón una etiqueta de posición fija (abajo a la izquierda) se obtiene esta salida de registro.

```
Initial state:
    [key]  [value]
[0]     3 21.78771
[1]    41 17.87709
CHART_FIXED_POSITION 17.87709497206704 -> 26.53631284916201
CHART_FIXED_POSITION 26.53631284916201 -> 27.93296089385475
CHART_FIXED_POSITION 27.93296089385475 -> 28.77094972067039
CHART_FIXED_POSITION 28.77094972067039 -> 50.0
```

5.7.12 Escala horizontal (por tiempo)

Para determinar la escala y el número de barras a lo largo del eje horizontal, utilice el grupo de propiedades de enteros de ENUM_CHART_PROPERTY_INTEGER. Entre ellos, sólo CHART_SCALE es editable.

Identificador	Descripción
CHART_SCALE	Escala (0 a 5)
CHART_VISIBLE_BARS	Número de barras visibles actualmente en el gráfico (puede ser inferior a CHART_WIDTH_IN_BARS debido a la sangría CHART_SHIFT_SIZE) (sólo lectura)
CHART_FIRST_VISIBLE_BAR	Número de la primera barra visible en el gráfico. La numeración va de derecha a izquierda, como en una serie temporal. (sólo lectura)
CHART_WIDTH_IN_BARS	Anchura del gráfico en barras (capacidad potencial, las barras de los extremos a izquierda y derecha pueden ser parcialmente visibles) (sólo lectura)
CHART_WIDTH_IN_PIXELS	Anchura del gráfico en píxeles (sólo lectura)



Propiedades ENUM_CHART_PROPERTY_INTEGER de un gráfico

Estamos listos para implementar el siguiente script de prueba *ChartScaleTime.mq5*, que permite analizar los cambios en estas propiedades.

```
void OnStart()
{
   int flags[] =
   {
      CHART_SCALE,
      CHART_VISIBLE_BARS,
      CHART_FIRST_VISIBLE_BAR,
      CHART_WIDTH_IN_BARS,
      CHART_WIDTH_IN_PIXELS
   };
   ChartModeMonitor m(flags);
   ...
}
```

A continuación figura una parte del registro con comentarios sobre las medidas adoptadas.

```
Initial state:
    [key] [value]
[0]     5       4
[1]   100      35
[2]   104      34
[3]   105      45
[4]   106     715
                                  // 1) changed the scale to a smaller one:
CHART_SCALE 4 -> 3               // - the value of the "scale" property has changed
CHART_VISIBLE_BARS 35 -> 69        // - increased the number of visible bars
CHART_FIRST_VISIBLE_BAR 34 -> 68 // - the number of the first visible bar has increas
CHART_WIDTH_IN_BARS 45 -> 90 // - increased the potential number of bars
                                  // 2) disabled padding at the right edge
CHART_VISIBLE_BARS 69 -> 89 // - the number of visible bars has increased
CHART_FIRST_VISIBLE_BAR 68 -> 88 // - the number of the first visible bar has increas
                                  // 3) reduced the window size
CHART_VISIBLE_BARS 89 -> 86 // - number of visible bars decreased
CHART_WIDTH_IN_BARS 90 -> 86 // - the potential number of bars has decreased
CHART_WIDTH_IN_PIXELS 715 -> 680 // - decreased width in pixels
                                  // 4) clicked the "End" button to move to the curren
CHART_VISIBLE_BARS 86 -> 85 // - number of visible bars decreased
CHART_FIRST_VISIBLE_BAR 88 -> 84 // - the number of the first visible bar has decreas
```

5.7.13 Escala vertical (por precio y lecturas del indicador)

Las propiedades relacionadas con la escala vertical se establecen y analizan utilizando los elementos de dos enumeraciones: ENUM_CHART_PROPERTY_INTEGER y ENUM_CHART_PROPERTY_DOUBLE. En la siguiente tabla se enumeran las propiedades junto con su tipo de valor.

Algunas propiedades permiten acceder no sólo a la ventana principal sino también a una subventana, para lo cual las funciones *ChartSet* y *ChartGet* deben utilizar el parámetro *window* (0 significa la ventana principal y es el valor por defecto de la forma abreviada de *ChartGet*).

Identificador	Descripción	Tipo de valor
CHART_SCALEFIX	Modo de escala fija	bool
CHART_FIXED_MAX	Máximo fijo de la subventana *window* o máximo inicial de la ventana principal	double
CHART_FIXED_MIN	Mínimo fijo de la subventana *window* o mínimo inicial de la ventana principal	double
CHART_SCALEFIX_11	Modo de escala 1:1	bool
CHART_SCALE_PT_PER_BAR	Modo de indicación de la escala en puntos por barra	bool
CHART_POINTS_PER_BAR	Valor de la escala en puntos por barra	double
CHART_PRICE_MIN	Valores mínimos en la ventana o subventana *window* (sólo lectura)	double
CHART_PRICE_MAX	Valores máximos en la ventana o subventana *window* (sólo lectura)	double
CHART_HEIGHT_IN_PIXELS	Altura fija de la ventana o subventana en píxeles, se requiere el parámetro *window*	int
CHART_WINDOW_YDISTANCE	Distancia en píxeles a lo largo del eje vertical Y entre el marco superior de la subventana *window* y el marco superior de la ventana principal del gráfico. (sólo lectura)	int

Por defecto, los gráficos admiten la escala adaptativa para que las cotizaciones o las líneas indicadoras se ajusten completamente en vertical a un periodo de tiempo visible. Para algunas aplicaciones es conveniente fijar la escala, para lo cual el terminal ofrece varios modos. En ellos, el gráfico puede desplazarse con el ratón o con las teclas (Mayús + flecha) no sólo a izquierda/derecha, sino también arriba/abajo, y en la escala de la derecha aparece una barra deslizante con la que puede desplazar rápidamente el gráfico con el ratón.

El modo fijo se establece activando la bandera CHART_SCALEFIX y especificando el máximo y el mínimo requeridos en los campos CHART_FIXED_MAX y CHART_FIXED_MIN (en la ventana principal, el usuario podrá mover el gráfico hacia arriba o hacia abajo, debido a lo cual los valores CHART_FIXED_MAX y CHART_FIXED_MIN cambiarán sincrónicamente, pero la escala vertical seguirá siendo la misma). El usuario también podrá cambiar la escala vertical pulsando el botón del ratón sobre la escala de precios y, sin soltarlo, moviéndolo hacia arriba o hacia abajo. Las subventanas no permiten la edición interactiva de la escala vertical. A este respecto, más adelante presentaremos un indicador *SubScaler.mq5* (véase la sección de eventos de teclado), que le permitirá al usuario controlar el rango de valores de la subventana mediante el teclado, en lugar de desde el cuadro de diálogo de configuración, utilizando los campos de la pestaña *Scale*.

El modo CHART_SCALEFIX_11 proporciona una igualdad visual aproximada de los lados del cuadrado en la pantalla: X barras en píxeles (horizontalmente) serán iguales a X puntos en píxeles (verticalmente). La igualdad es aproximada, ya que el tamaño de los píxeles, por regla general, no es el mismo vertical y horizontalmente.

Por último, existe un modo para fijar la relación del número de puntos por barra, que se activa mediante la opción CHART_SCALE_PT_PER_BAR, y la propia relación requerida se fija mediante la propiedad CHART_POINTS_PER_BAR. A diferencia del modo CHART_SCALEFIX, el usuario no podrá cambiar interactivamente la escala con el ratón sobre el gráfico. En este modo, la distancia horizontal de una barra se mostrará en la pantalla en la misma proporción con respecto al número especificado de puntos verticales que la relación de aspecto del gráfico (en píxeles). Si los marcos temporales y tamaños de los dos gráficos son iguales, uno se verá comprimido en precio comparado con el otro según la relación de sus valores CHART_POINTS_PER_BAR. Obviamente, cuanto más pequeño sea el marco temporal, menor será el rango de barras y, por lo tanto, con la misma escala, los marcos temporales pequeños parecen más «aplanados».

Establecer programáticamente la propiedad CHART_HEIGHT_IN_PIXELS hace imposible que el usuario edite el tamaño de la ventana/subventana. Esto suele utilizarse para ventanas que alojan paneles de trading con un conjunto predefinido de controles (botones, campos de entrada, etc.). Para eliminar la fijación del tamaño, establezca el valor de la propiedad en -1.

El valor CHART_WINDOW_YDISTANCE es necesario para convertir las coordenadas absolutas del gráfico principal en coordenadas locales de la subventana para trabajar correctamente con objetos gráficos. La cuestión es que cuando se producen eventos de ratón, las coordenadas del cursor se transfieren en relación con la ventana del gráfico principal, mientras que las coordenadas de los objetos gráficos de la subventana del indicador se establecen en relación con la esquina superior izquierda de la subventana.

Vamos a preparar el script *ChartScalePrice.mq5* para analizar los cambios en las escalas y tamaños verticales.

```
void OnStart()
{
   int flags[] =
   {
      CHART_SCALEFIX, CHART_SCALEFIX_11,
      CHART_SCALE_PT_PER_BAR, CHART_POINTS_PER_BAR,
      CHART_FIXED_MAX, CHART_FIXED_MIN,
      CHART_PRICE_MIN, CHART_PRICE_MAX,
      CHART_HEIGHT_IN_PIXELS, CHART_WINDOW_YDISTANCE
   };
   ChartModeMonitor m(flags);
   ...
}
```

Esto reacciona a la manipulación del gráfico de la siguiente manera:

```
Initial state:
    [key] [value]   // ENUM_CHART_PROPERTY_INTEGER
[0]     6       0
[1]     7       0
[2]    10       0
[3]   107     357
[4]   110       0
    [key]  [value]  // ENUM_CHART_PROPERTY_DOUBLE
[0]    11 10.00000
[1]     8  1.13880
[2]     9  1.12330
[3]   108  1.12330
[4]   109  1.13880
// reduced the vertical size of the window
CHART_HEIGHT_IN_PIXELS 357 -> 370
CHART_HEIGHT_IN_PIXELS 370 -> 408
CHART_FIXED_MAX 1.1389 -> 1.1388
CHART_FIXED_MIN 1.1232 -> 1.1233
CHART_PRICE_MIN 1.1232 -> 1.1233
CHART_PRICE_MAX 1.1389 -> 1.1388
// reduced the horizontal scale, which increased the price range
CHART_FIXED_MAX 1.1388 -> 1.139
CHART_FIXED_MIN 1.1233 -> 1.1183
CHART_PRICE_MIN 1.1233 -> 1.1183
CHART_PRICE_MAX 1.1388 -> 1.139
CHART_FIXED_MAX 1.139 -> 1.1406
CHART_FIXED_MIN 1.1183 -> 1.1167
CHART_PRICE_MIN 1.1183 -> 1.1167
CHART_PRICE_MAX 1.139 -> 1.1406
// expand the price range using the mouse (quotes "shrink" vertically)
CHART_FIXED_MAX 1.1406 -> 1.1454
CHART_FIXED_MIN 1.1167 -> 1.1119
CHART_PRICE_MIN 1.1167 -> 1.1119
CHART_PRICE_MAX 1.1406 -> 1.1454
```

5.7.14 Colores

Un programa MQL puede reconocer y cambiar los colores para mostrar todos los elementos del gráfico. Las propiedades correspondientes forman parte de la enumeración ENUM_CHART_PROPERTY_INTEGER.

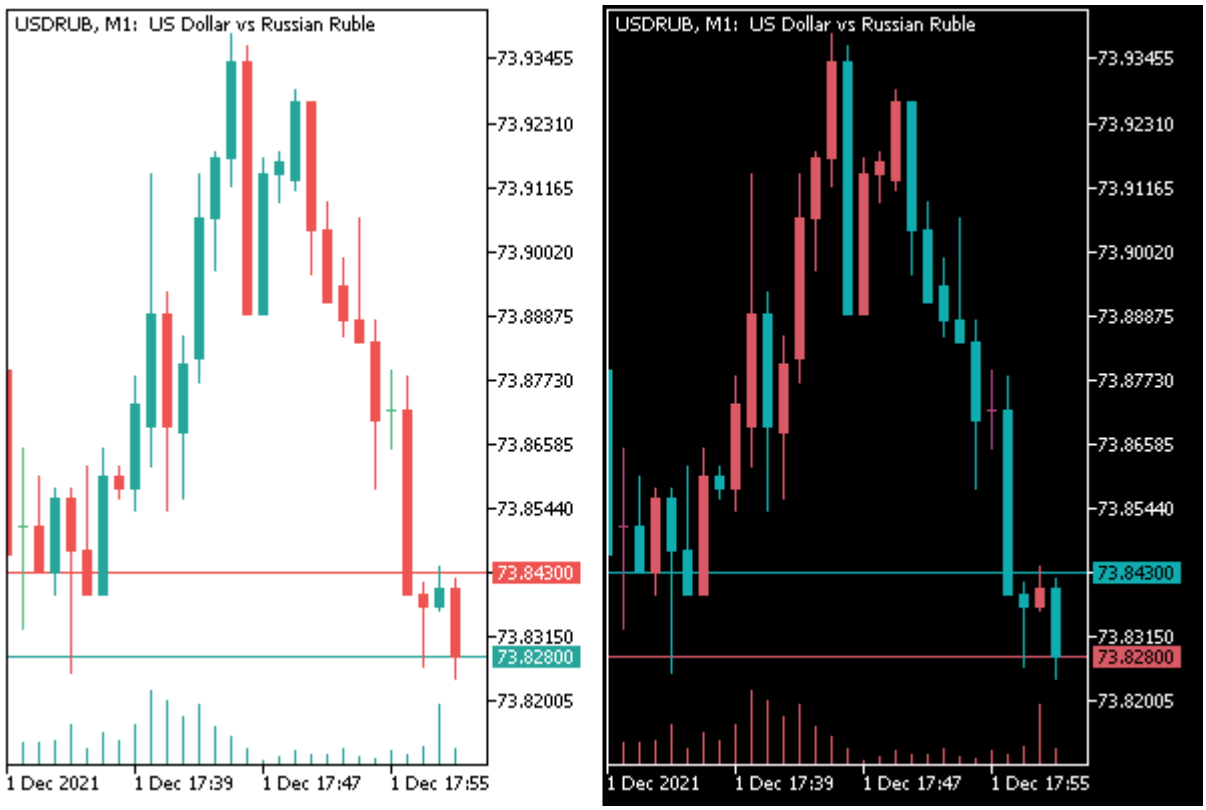
Identificador	Descripción
CHART_COLOR_BACKGROUND	Color de fondo del gráfico
CHART_COLOR_FOREGROUND	Color de los ejes, escalas y líneas OHLC
CHART_COLOR_GRID	Color de la cuadrícula
CHART_COLOR_VOLUME	Color de los volúmenes y niveles de apertura de posiciones

Identificador	Descripción
CHART_COLOR_CHART_UP	El color de la barra ascendente, la sombra y el borde del cuerpo de una vela alcista.
CHART_COLOR_CHART_DOWN	El color de la barra descendente, la sombra y el borde del cuerpo de una vela bajista.
CHART_COLOR_CHART_LINE	El color de la línea del gráfico y de los contornos de las velas japonesas
CHART_COLOR_CANDLE_BULL	Color del cuerpo de la vela alcista
CHART_COLOR_CANDLE_BEAR	Color del cuerpo de la vela bajista
CHART_COLOR_BID	Color de la línea de precio de demanda (Bid)
CHART_COLOR_ASK	Color de la línea de precio de oferta (Ask)
CHART_COLOR_LAST	Color de la línea del último precio negociado (Last)
CHART_COLOR_STOP_LEVEL	Color de los niveles de las órdenes stop (Stop Loss y Take Profit)

Como ejemplo de trabajo con estas propiedades, vamos a crear un script, *ChartColorInverse.mq5*, que cambiará todos los colores del gráfico a inverso, es decir, por la representación en bits del color en el formato RGB XOR ('^',XOR). Así, tras reiniciar el script en el mismo gráfico, se restablecerá su configuración.

```
#define RGB_INVERSE(C) ((color)C ^ 0xFFFFFF)

void OnStart()
{
   ENUM_CHART_PROPERTY_INTEGER colors[] =
   {
      CHART_COLOR_BACKGROUND,
      CHART_COLOR_FOREGROUND,
      CHART_COLOR_GRID,
      CHART_COLOR_VOLUME,
      CHART_COLOR_CHART_UP,
      CHART_COLOR_CHART_DOWN,
      CHART_COLOR_CHART_LINE,
      CHART_COLOR_CANDLE_BULL,
      CHART_COLOR_CANDLE_BEAR,
      CHART_COLOR_BID,
      CHART_COLOR_ASK,
      CHART_COLOR_LAST,
      CHART_COLOR_STOP_LEVEL
   };

   for(int i = 0; i < ArraySize(colors); ++i)
   {
      ChartSetInteger(0, colors[i], RGB_INVERSE(ChartGetInteger(0, colors[i])));
   }
}
```

En la siguiente imagen se combinan las imágenes del gráfico antes y después de aplicar el script:



Invertir los colores del gráfico desde un programa MQL

Ahora vamos a terminar de editar *IndSubChart.mq5*. Tenemos que leer los colores del gráfico principal y aplicarlos a nuestro gráfico indicador. Existe una función para estos fines: *SetPlotColors*, cuya llamada se comentó en el manejador *OnChartEvent* (véase el último ejemplo de la sección Modos de visualización de gráficos).

```
void SetPlotColors(const int index, const ENUM_CHART_MODE m)
{
   if(m == CHART_CANDLES)
   {
      PlotIndexSetInteger(index, PLOT_COLOR_INDEXES, 3);
      PlotIndexSetInteger(index, PLOT_LINE_COLOR, 0, (int)ChartGetInteger(0, CHART_CO
      PlotIndexSetInteger(index, PLOT_LINE_COLOR, 1, (int)ChartGetInteger(0, CHART_CO
      PlotIndexSetInteger(index, PLOT_LINE_COLOR, 2, (int)ChartGetInteger(0, CHART_CO
   }
   else
   {
      PlotIndexSetInteger(index, PLOT_COLOR_INDEXES, 1);
      PlotIndexSetInteger(index, PLOT_LINE_COLOR, (int)ChartGetInteger(0, CHART_COLOR
   }
}
```

En esta nueva función, obtenemos, dependiendo del modo de dibujo del gráfico, el color de los contornos y cuerpos de las velas alcistas y bajistas, o el color de las líneas, y aplicamos los colores a los gráficos. Por supuesto, no olvide llamar a esta función durante la inicialización.

```
int OnInit()
{
   ...
   mode = (ENUM_CHART_MODE)ChartGetInteger(0, CHART_MODE);
   InitPlot(0, InitBuffers(mode), Mode2Style(mode));
   SetPlotColors(0, mode);
   ...
}
```

El indicador está listo. Pruebe a ejecutarlo en la ventana y cambiar los colores en el cuadro de diálogo de propiedades del gráfico. El gráfico debería adaptarse automáticamente a la nueva configuración.

5.7.15 Control del ratón y del teclado

En esta sección nos familiarizaremos con un grupo de propiedades que afectan a la forma en que el gráfico capturará ciertas manipulaciones del ratón y del teclado, que se consideran por defecto acciones de control. En concreto, los usuarios de MetaTrader 5 saben muy bien que el gráfico se puede desplazar con el ratón, y que se puede llamar al menú contextual para ejecutar los comandos más solicitados. MQL5 permite desactivar total o parcialmente este comportamiento del gráfico. Es importante señalar que esto sólo puede hacerse mediante programación: no hay opciones similares en la interfaz de usuario del terminal.

La única excepción es la opción CHART_DRAG_TRADE_LEVELS (véase en la tabla siguiente): la configuración del terminal proporciona la pestaña *Charts* con una lista desplegable que controla el permiso para arrastrar niveles de trading con el ratón.

Todas las propiedades de este grupo son de tipo booleano (*true* para permitido y *false* para deshabilitado) y están en la enumeración ENUM_CHART_PROPERTY_INTEGER.

Identificador	Descripción
CHART_CONTEXT_MENU	Activar/desactivar el acceso al menú contextual pulsando el botón derecho del ratón. El valor *false* desactiva sólo el menú contextual del gráfico, mientras que el menú contextual de los objetos del gráfico sigue estando disponible. El valor por defecto es *true*.
CHART_CROSSHAIR_TOOL	Activar/desactivar el acceso a la herramienta Crosshair pulsando el botón central del ratón. El valor por defecto es *true*.
CHART_MOUSE_SCROLL	Desplazamiento del gráfico con el botón izquierdo o la rueda del ratón. Cuando el desplazamiento está activado, esto se aplica no sólo al desplazamiento horizontal, sino también al vertical, pero este último sólo está disponible cuando se establece una escala fija: una de las propiedades CHART_SCALEFIX, CHART_SCALEFIX_11 o CHART_SCALE_PT_PER_BAR. El valor por defecto es *true*.
CHART_KEYBOARD_CONTROL	Posibilidad de manejar el gráfico desde el teclado (botones *Home*, *End*, *PageUp/PageDown*, +/-, flechas arriba/abajo, etc.). Si se configura en *false*, se puede desactivar el desplazamiento y la escala del gráfico, pero al mismo tiempo es posible recibir eventos de pulsación de estas teclas en *OnChartEvent*. El valor por defecto es *true*.
CHART_QUICK_NAVIGATION	Activación de la barra de navegación rápida en el gráfico, que aparece automáticamente en la esquina izquierda de la línea de tiempo al hacer doble clic con el ratón o pulsar las teclas *Space* o *Input*. Con la barra, puede cambiar rápidamente el símbolo, el marco temporal o la fecha de la primera barra visible. Por defecto, la propiedad está establecida en true y la navegación rápida está activada.
CHART_DRAG_TRADE_LEVELS	Permiso para arrastrar niveles de trading en el gráfico con el ratón. El modo de arrastre está activado por defecto (*true*).

En el script de prueba *ChartInputControl.mq5*, configuraremos el monitor con todas las propiedades anteriores y, además, proporcionaremos variables de entrada para que el usuario configure arbitrariamente los valores. Nuestro script guarda una copia de seguridad de la configuración al iniciarse, por lo que todas las propiedades modificadas se restaurarán cuando finalice el script.

```
#property script_show_inputs

#include <MQL5Book/ChartModeMonitor.mqh>

input bool ContextMenu = true; // CHART_CONTEXT_MENU
input bool CrossHairTool = true; // CHART_CROSSHAIR_TOOL
input bool MouseScroll = true; // CHART_MOUSE_SCROLL
input bool KeyboardControl = true; // CHART_KEYBOARD_CONTROL
input bool QuickNavigation = true; // CHART_QUICK_NAVIGATION
input bool DragTradeLevels = true; // CHART_DRAG_TRADE_LEVELS

void OnStart()
{
   const bool Inputs[] =
   {
      ContextMenu, CrossHairTool, MouseScroll,
      KeyboardControl, QuickNavigation, DragTradeLevels
   };
   const int flags[] =
   {
      CHART_CONTEXT_MENU, CHART_CROSSHAIR_TOOL, CHART_MOUSE_SCROLL,
      CHART_KEYBOARD_CONTROL, CHART_QUICK_NAVIGATION, CHART_DRAG_TRADE_LEVELS
   };
   ChartModeMonitor m(flags);
   Print("Initial state:");
   m.print();
   m.backup();

   for(int i = 0; i < ArraySize(flags); ++i)
   {
      ChartSetInteger(0, (ENUM_CHART_PROPERTY_INTEGER)flags[i], Inputs[i]);
   }

   while(!IsStopped())
   {
      m.snapshot();
      Sleep(500);
   }
   m.restore();
}
```

Por ejemplo, cuando ejecutamos el script, podemos restablecer los permisos para el menú contextual, la herramienta Crosshair, el ratón y los controles de teclado a *false*. El resultado está en el siguiente registro:

```
Initial state:
    [key] [value]
[0]    50       1
[1]    49       1
[2]    42       1
[3]    47       1
[4]    45       1
[5]    43       1
CHART_CONTEXT_MENU 1 -> 0
CHART_CROSSHAIR_TOOL 1 -> 0
CHART_MOUSE_SCROLL 1 -> 0
CHART_KEYBOARD_CONTROL 1 -> 0
```

En este caso, no podrá mover el gráfico ni con el ratón ni con el teclado, ni siquiera llamar al menú contextual. Por lo tanto, para restaurar su rendimiento, tendrá que soltar el mismo script u otro en el gráfico (recuerde que sólo puede haber un script en el gráfico, y cuando se aplica uno nuevo, el anterior se descarga). Basta con soltar una nueva instancia del script, pero no ejecutarlo (pulse *Cancel* en el cuadro de diálogo para introducir variables de entrada).

5.7.16 Desacoplar la ventana del gráfico

Las ventanas de gráficos del terminal pueden desacoplarse de la ventana principal, tras lo cual pueden moverse a cualquier lugar del escritorio, incluidos otros monitores. MQL5 permite averiguar y cambiar esta configuración: las propiedades correspondientes se incluyen en la enumeración ENUM_CHART_PROPERTY_INTEGER.

Identificador	Descripción	Tipo de valor
CHART_IS_DOCKED	La ventana del gráfico está acoplada (true por defecto). Si se establece en false, el gráfico puede arrastrarse fuera del terminal	bool
CHART_FLOAT_LEFT	Coordenada izquierda del gráfico desacoplado respecto a la pantalla virtual	int
CHART_FLOAT_TOP	Coordenada superior del gráfico desacoplado respecto a la pantalla virtual	int
CHART_FLOAT_RIGHT	Coordenada derecha del gráfico desacoplado respecto a la pantalla virtual	int
CHART_FLOAT_BOTTOM	Coordenada inferior del gráfico desacoplado respecto a la pantalla virtual	int

Establezcamos el seguimiento de estas propiedades en el script *ChartDock.mq5*.

```
void OnStart()
{
   const int flags[] =
   {
      CHART_IS_DOCKED,
      CHART_FLOAT_LEFT, CHART_FLOAT_TOP, CHART_FLOAT_RIGHT, CHART_FLOAT_BOTTOM
   };
   ChartModeMonitor m(flags);
   ...
}
```

Si ahora ejecuta el script y, a continuación, desacopla el gráfico mediante el menú contextual (desactive el comando switch *Docked*) y mueve o cambia el tamaño del gráfico, los registros correspondientes se añadirán al diario.

```
Initial state:
    [key] [value]
[0]    51       1
[1]    52       0
[2]    53       0
[3]    54       0
[4]    55       0
                                // undocked
CHART_IS_DOCKED 1 -> 0
CHART_FLOAT_LEFT 0 -> 299
CHART_FLOAT_TOP 0 -> 75
CHART_FLOAT_RIGHT 0 -> 1263
CHART_FLOAT_BOTTOM 0 -> 472
                                // changed the vertical size
CHART_FLOAT_BOTTOM 472 -> 500
CHART_FLOAT_BOTTOM 500 -> 539
                                // changed the horizontal size
CHART_FLOAT_RIGHT 1263 -> 1024
CHART_FLOAT_RIGHT 1024 -> 1023
                                // docked back
CHART_IS_DOCKED 0 -> 1
```

Esta sección completa la descripción de las propiedades gestionadas a través de las funciones *ChartGet* y *ChartSet*, así que vamos a resumir el material utilizando un script común *ChartFullSet.mq5*. Realiza un seguimiento del estado de todas las propiedades de todos los tipos. La inicialización del array de banderas se realiza simplemente rellenando los índices sucesivos en un bucle. El valor máximo se toma con un margen en caso de nuevas propiedades, y los números extra no existentes serán descartados automáticamente por la comprobación integrada en la clase *ChartModeMonitorBase* (recuerde el método *detect*).

Después de activar el script, intente cambiar cualquier configuración mientras observa los mensajes del programa en el registro.

5.7.17 Obtener las coordenadas de caída del programa MQL en un gráfico

Los usuarios suelen arrastrar los programas MQL a un gráfico utilizando el ratón. Además de ser práctico y cómodo, esto le permite establecer algún contexto para el algoritmo. Por ejemplo, un

indicador puede aplicarse en diferentes subventanas, o un script puede colocar una orden pendiente al precio en el que el usuario la colocó en el gráfico. El siguiente grupo de funciones está diseñado para obtener las coordenadas del punto al que se arrastró y soltó el programa.

int ChartWindowOnDropped()

Esta función devuelve el número de la subventana del gráfico en la que el Asesor Experto, script o indicador actual es soltado por el ratón. La ventana principal, como sabemos, tiene el número 0, y las subventanas se numeran empezando por 1. El número de una subventana no depende de si hay subventanas ocultas por encima de ella, ya que sus índices permanecen asignados a ellas. En otras palabras: el número de subventana visible puede diferir de su índice real si hay subventanas ocultas.

double ChartPriceOnDropped()
datetime ChartTimeOnDropped()

Este par de funciones devuelve las coordenadas del punto de caída del programa en unidades de precio y tiempo. Tenga en cuenta que en las subventanas se pueden mostrar datos arbitrarios, y no sólo precios, aunque el nombre de la función *ChartPriceOnDropped* incluya 'Price'.

> ¡Atención! El tiempo del punto objetivo no se redondea por el tamaño del marco temporal del gráfico, por lo que incluso en los gráficos H1 y D1, puede obtener un valor con minutos e incluso segundos.

int ChartXOnDropped()
int ChartYOnDropped()

Estas dos funciones devuelven las coordenadas de pantalla X e Y de un punto en píxeles. El origen de las coordenadas se encuentra en la esquina superior izquierda de la ventana principal del gráfico. Ya hemos hablado de la dirección de los ejes en la sección Especificaciones de la pantalla.

La coordenada Y se cuenta siempre desde la esquina superior izquierda del gráfico principal, aun cuando el punto de caída pertenezca a una subventana. Para traducir este valor a una coordenada *y* relativa a una subventana, utilice la propiedad CHART_WINDOW_YDISTANCE (ver ejemplo).

Vamos a mostrar los valores de todas las funciones mencionadas en el registro del script *ChartDrop.mq5*:

```
void OnStart()
{
   const int w = PRTF(ChartWindowOnDropped());
   PRTF(ChartTimeOnDropped());
   PRTF(ChartPriceOnDropped());
   PRTF(ChartXOnDropped());
   PRTF(ChartYOnDropped());

   // for the subwindow, recalculate the y coordinate to the local one
   if(w > 0)
   {
      const int y = (int)PRTF(ChartGetInteger(0, CHART_WINDOW_YDISTANCE, w));
      PRTF(ChartYOnDropped() - y);
   }
}
```

Por ejemplo, si soltamos este script en la primera subventana donde se está ejecutando el indicador WPR, podemos obtener los siguientes resultados:

```
ChartWindowOnDropped()=1 / ok
ChartTimeOnDropped()=2021.11.30 03:52:30 / ok
ChartPriceOnDropped()=-50.0 / ok
ChartXOnDropped()=217 / ok
ChartYOnDropped()=312 / ok
ChartGetInteger(0,CHART_WINDOW_YDISTANCE,w)=282 / ok
ChartYOnDropped()-y=30 / ok
```

A pesar de que el script se deja caer en EURUSD, gráfico H1, obtuvimos una marca de tiempo con minutos y segundos.

Observe que el valor «precio» es -50 porque el rango de valores WPR es [0,-100].

Además, la coordenada vertical del punto 312 (relativa a toda la ventana del gráfico) se ha convertido en la coordenada local de la subventana: como la distancia vertical desde el inicio del gráfico principal hasta la subventana era de 282, el valor *y* dentro de la subventana resultó ser 30.

5.7.18 Conversión de coordenadas de pantalla a tiempo/precio y viceversa

La presencia de distintos principios para medir el espacio de trabajo del gráfico conduce a la necesidad de recalcular las unidades de medida entre sí. Existen dos funciones para ello.

bool ChartTimePriceToXY(long chartId, int window, datetime time, double price, int &x, int &y)
bool ChartXYToTimePrice(long chartId, int x, int y, int &window, datetime &time, double &price)

La función *ChartTimePriceToXY* convierte las coordenadas del gráfico de la representación tiempo/precio (*time/price*) en coordenadas X e Y en píxeles (*x/y*). La función *ChartXYToTimePrice* realiza la operación inversa: convierte las coordenadas X e Y en valores de tiempo y precio.

Ambas funciones requieren que se especifique el ID del gráfico en el primer parámetro *chartId*. Además, el número de la subventana *window* se pasa en *ChartTimePriceToXY* (debe estar dentro del número de ventanas). Si hay varias subventanas, cada una de ellas tiene su propia serie temporal y una escala a lo largo del eje vertical (denominada condicionalmente «precio» con el parámetro *price*).

El parámetro *window* es la salida en la función *ChartXYToTimePrice*. La función rellena este parámetro junto con *time* y *price*. Esto se debe a que las coordenadas de los píxeles son comunes a toda la pantalla, y el origen *x/y* puede caer en cualquier subventana.

Coordenadas de pantalla, precio y hora

Las funciones devuelven *true* una vez completadas con éxito.

Tenga en cuenta que el área rectangular visible que corresponde a las cotizaciones o a las coordenadas de la pantalla está limitada en ambos sistemas de coordenadas. Por lo tanto, son posibles situaciones en las que, con datos iniciales específicos, la hora, los precios o los píxeles recibidos estarán fuera del área de visibilidad. En concreto, también pueden obtenerse valores negativos. Veremos un ejemplo de recálculo interactivo en el capítulo sobre eventos en gráficos.

En la sección anterior vimos cómo se puede averiguar dónde se lanzó un programa MQL. Aunque físicamente sólo hay un punto de caída final, su representación en coordenadas de pantalla y cotización, por regla general, contiene un error de cálculo. Dos nuevas funciones para convertir píxeles en precio/tiempo y viceversa nos ayudarán a asegurarnos de ello.

El script modificado se llama *ChartXY.mq5*. A grandes rasgos, puede dividirse en 3 etapas. En la primera etapa, obtenemos las coordenadas del punto de caída, como antes.

```
void OnStart()
{
   const int w1 = PRTF(ChartWindowOnDropped());
   const datetime t1 = PRTF(ChartTimeOnDropped());
   const double p1 = PRTF(ChartPriceOnDropped());
   const int x1 = PRTF(ChartXOnDropped());
   const int y1 = PRTF(ChartYOnDropped());
   ...
```

En la segunda etapa, intentamos transformar las coordenadas de pantalla *x1* y *y1* durante (*t2*) y el precio (*p2*), y las comparamos con las obtenidas a partir de las funciones *OnDropped* anteriores.

```
   int w2;
   datetime t2;
   double p2;
   PRTF(ChartXYToTimePrice(0, x1, y1, w2, t2, p2));
   Print(w2, " ", p2, " ", t2);
   PRTF(w1 == w2 && t1 == t2 && p1 == p2);
   ...
```

A continuación, realizamos la transformación inversa: utilizamos las coordenadas de cotización *t1* y *p1* obtenidas para calcular las coordenadas de pantalla *-x2* y *y2*, y también las comparamos con los valores originales *x1* y *y1*.

```
   int x2, y2;
   PRTF(ChartTimePriceToXY(0, w1, t1, p1, x2, y2));
   Print(x2, " ", y2);
   PRTF(x1 == x2 && y1 == y2);
   ...
```

Como veremos más adelante en el registro de ejemplo, todas las comprobaciones anteriores fallarán (habrá ligeras discrepancias en los valores). Así que necesitamos un tercer paso.

Recalculemos las coordenadas de cotización y pantalla con el sufijo 2 en los nombres de las variables y guardémoslas en las variables con el nuevo sufijo 3. A continuación, comparamos entre sí todos los valores de la primera y la tercera etapa.

```
   int w3;
   datetime t3;
   double p3;
   PRTF(ChartXYToTimePrice(0, x2, y2, w3, t3, p3));
   Print(w3, " ", p3, " ", t3);
   PRTF(w1 == w3 && t1 == t3 && p1 == p3);

   int x3, y3;
   PRTF(ChartTimePriceToXY(0, w2, t2, p2, x3, y3));
   Print(x3, " ", y3);
   PRTF(x1 == x3 && y1 == y3);
}
```

Vamos a ejecutar el script en XAUUSD, gráfico H1. Aquí están los datos del punto original.

```
ChartWindowOnDropped()=0 / ok
ChartTimeOnDropped()=2021.11.22 18:00:00 / ok
ChartPriceOnDropped()=1797.7 / ok
ChartXOnDropped()=234 / ok
ChartYOnDropped()=280 / ok
```

La conversión de píxeles en cotizaciones da los siguientes resultados:

```
ChartXYToTimePrice(0,x1,y1,w2,t2,p2)=true / ok
0 1797.16 2021.11.22 18:30:00
w1==w2&&t1==t2&&p1==p2=false / ok
```

Hay diferencias tanto de tiempo como de precio. El recuento retrospectivo tampoco es perfecto en términos de precisión.

```
ChartTimePriceToXY(0,w1,t1,p1,x2,y2)=true / ok
232 278
x1==x2&&y1==y2=false / ok
```

La pérdida de precisión se produce debido a la cuantización de los valores en los ejes según las unidades de medida, en concreto píxeles y puntos.

Para terminar, el último paso demuestra que los errores obtenidos anteriormente no son artefactos de la función, porque el recálculo round-robin conduce al resultado original.

```
ChartXYToTimePrice(0,x2,y2,w3,t3,p3)=true / ok
0 1797.7 2021.11.22 18:00:00
w1==w3&&t1==t3&&p1==p3=true / ok
ChartTimePriceToXY(0,w2,t2,p2,x3,y3)=true / ok
234 280
x1==x3&&y1==y3=true / ok
```

En pseudocódigo, esto puede expresarse mediante las siguientes igualdades:

```
ChartTimePriceToXY(ChartXYToTimePrice(XY)) = XY
ChartXYToTimePrice(ChartTimePriceToXY(TP)) = TP
```

Aplicando la función *ChartTimePriceToXY* a los resultados del trabajo *ChartXYToTimePrice* se obtendrán las coordenadas originales. Lo mismo ocurre con las transformaciones en el otro sentido: aplicando *ChartXYToTimePrice* a los resultados de *ChartTimePriceToXY* se obtendrá una coincidencia.

Por lo tanto, se debe considerar cuidadosamente la implementación de algoritmos que utilicen funciones de recálculo si están sujetos a requisitos de precisión mayores.

Otro ejemplo de uso de *ChartWindowOnDropped* se dará en el script *ChartIndicatorMove.mq5* en la sección Gestionar indicadores en el gráfico.

5.7.19 Desplazamiento de gráficos por el eje temporal

Los usuarios de MetaTrader 5 están familiarizados con el panel de navegación rápida de gráficos, que se abre haciendo doble clic en la esquina izquierda de la línea de tiempo o pulsando las teclas *Space* o *Input*. También existe una posibilidad similar mediante programación utilizando la función *ChartNavigate*.

bool ChartNavigate(long chartId, ENUM_CHART_POSITION position, int shift = 0)

La función desplaza el gráfico *chartId* el número de barras especificado con respecto a la posición predefinida del gráfico especificada por el parámetro *position*. Es de tipo de enumeración ENUM_CHART_POSITION con los siguientes elementos:

Identificador	Descripción
CHART_BEGIN	Inicio del gráfico (precios más antiguos)
CHART_CURRENT_POS	Posición actual
CHART_END	Fin del gráfico (últimos precios)

El parámetro *shift* establece el número de barras en que debe desplazarse el gráfico. Un valor positivo desplaza el gráfico hacia la derecha (hacia el final), y un valor negativo lo desplaza hacia la izquierda (hacia el principio).

La función devuelve *true* en caso de éxito o *false* en caso de error.

Para probar la función, vamos a crear un sencillo script *ChartNavigate.mq5*. Con la ayuda de variables de entrada, el usuario puede elegir un punto de partida y un desplazamiento en barras.

```
#property script_show_inputs

input ENUM_CHART_POSITION Position = CHART_CURRENT_POS;
input int Shift = 0;

void OnStart()
{
   ChartSetInteger(0, CHART_AUTOSCROLL, false);
   const int start = (int)ChartGetInteger(0, CHART_FIRST_VISIBLE_BAR);
   ChartNavigate(0, Position, Shift);
   const int stop = (int)ChartGetInteger(0, CHART_FIRST_VISIBLE_BAR);
   Print("Moved by: ", stop - start, ", from ", start, " to ", stop);
}
```

El registro muestra el número de la primera barra visible antes y después del movimiento.

Un ejemplo más práctico sería el script *ChartSynchro.mq5*, que permite desplazarse de forma sincrónica por todos los gráficos en los que se está ejecutando, en respuesta a que el usuario se desplace manualmente por uno de los gráficos. Así, puede sincronizar ventanas de distintos marcos temporales del mismo instrumento o analizar movimientos de precios paralelos en distintos instrumentos.

```
void OnStart()
{
   datetime bar = 0; // current position (time of the first visible bar)

   conststring namePosition =__FILE__;// global variable name

   ChartSetInteger(0,CHART_AUTOSCroll,false); // disable autoscroll

   while(!IsStopped())
   {
      const bool active = ChartGetInteger(0, CHART_BRING_TO_TOP);
      const int move = (int)ChartGetInteger(0, CHART_FIRST_VISIBLE_BAR);

      // the active chart is the leader, and the rest are slaves
      if(active)
      {
         const datetime first = iTime(_Symbol, _Period, move);
         if(first != bar)
         {
            // if the position has changed, save it in a global variable
            bar = first;
            GlobalVariableSet(namePosition, bar);
            Comment("Chart ", ChartID(), " scrolled to ", bar);
         }
      }
      else
      {
         const datetime b = (datetime)GlobalVariableGet(namePosition);

         if(b != bar)
         {
            // if the value of the global variable has changed, adjust the position
            bar = b;
            const int difference = move - iBarShift(_Symbol, _Period, bar);
            ChartNavigate(0, CHART_CURRENT_POS, difference);
            Comment("Chart ", ChartID(), " forced to ", bar);
         }
      }

      Sleep(250);
   }
   Comment("");
}
```

La alineación se realiza por la fecha y hora de la primera barra visible (CHART_FIRST_VISIBLE_BAR). El script en un bucle comprueba este valor y, si funciona en un gráfico activo, lo escribe en una variable global. Los scripts de otros gráficos leen esta variable y ajustan su posición en consecuencia con *ChartNavigate*. Los parámetros especifican el movimiento relativo del gráfico (CHART_CURRENT_POS), y el número de barras que se van a mover se define como la diferencia entre el número actual de la primera barra visible y el leído de la variable global.

En la siguiente imagen se muestra el resultado de sincronizar los gráficos H1 y M15 para EURUSD.

Ejemplo de script para sincronizar las posiciones de los gráficos

Después de familiarizarnos con el sistema eventos en gráficos convertiremos este script en un indicador y nos desharemos del bucle infinito.

5.7.20 Solicitud para volver a dibujar el gráfico

En la mayoría de los casos, los gráficos responden automáticamente a los cambios en los datos y en la configuración del terminal, actualizando la imagen de la ventana en consecuencia (gráficos de precios, de indicadores, etc.). No obstante, los programas MQL son demasiado versátiles y pueden realizar acciones arbitrarias, para las que no es tan fácil determinar si es necesario redibujar o no. Además, analizar cualquier acción de cada programa MQL en esta cuenta puede consumir muchos recursos y causar una caída en el rendimiento general del terminal. Por lo tanto, la API de MQL5 proporciona la función *ChartRedraw*, con la ayuda de la cual el propio programa MQL puede, si es necesario, solicitar que se vuelva a dibujar el gráfico.

void ChartRedraw(long chartId = 0)

La función provoca un redibujado forzado del gráfico con el identificador especificado (el valor por defecto 0 significa el gráfico actual). Normalmente, se aplica después de que el programa cambie las propiedades del gráfico u objetos colocados en él.

Hemos visto un ejemplo de utilización de *ChartRedraw* en el indicador *IndSubChart.mq5* en la sección Modos de visualización de gráficos. Se ofrecerá otro ejemplo en la sección Abrir y cerrar gráficos.

Esta función afecta exactamente al redibujado del gráfico, sin provocar el recálculo de las series temporales con cotizaciones e indicadores. La última opción para actualizar (de hecho, reconstruir) el gráfico es más «dura» y la realiza la función *ChartSetSymbolPeriod* (véase la sección siguiente).

5.7.21 Cambiar símbolo y marco temporal

A veces, un programa MQL necesita cambiar el símbolo actual o el marco temporal de un gráfico. En concreto, se trata de una funcionalidad familiar para muchos paneles de trading multidivisa y de marco temporal múltiple, o utilidades de análisis del historial de trading. Para ello, la API de MQL5 proporciona la función *ChartSetSymbolPeriod*.

También puede utilizar esta función para iniciar el recálculo de todo el gráfico, incluidos los indicadores situados en él. Puede especificar simplemente el símbolo actual y el marco temporal como parámetros. Esta técnica puede ser útil para indicadores que no pudieron ser calculados completamente en la primera llamada de *OnCalculate*, y esperar la carga de datos de terceros (otros símbolos, ticks o indicadores). Además, el cambio de símbolo/marco temporal provoca la reinicialización de los Asesores Expertos adjuntos al gráfico. El script (si se ejecuta periódicamente en un ciclo) desaparecerá completamente del gráfico durante este procedimiento (se descargará de la antigua combinación símbolo/marco temporal, pero no se cargará automáticamente para la nueva combinación).

bool ChartSetSymbolPeriod(long chartId, string symbol, ENUM_TIMEFRAMES timeframe)

La función cambia el símbolo y el marco temporal del gráfico especificado con el identificador *chartId* a los valores de los parámetros correspondientes: *symbol* y *timeframe*. 0 en el parámetro *chartId* significa el gráfico actual, NULL en el parámetro *symbol* es el carácter actual, y 0 en el parámetro *timeframe* es el marco temporal actual.

Los cambios surten efecto de forma asíncrona, es decir, la función sólo envía un comando al terminal y no espera a su ejecución. El comando se añade a la cola de mensajes del gráfico y se ejecuta sólo después de que se hayan procesado todos los comandos anteriores.

La función devuelve *true* en caso de colocación correcta del comando en la cola de gráficos o *false* en caso de problemas. Encontrará información sobre el error en *_LastError*.

Hemos visto ejemplos de uso de la función para actualizar varios indicadores, en concreto los siguientes:

- *IndDeltaVolume.mq5* (véase Esperar datos y gestionar la visibilidad)
- *IndUnityPercent.mq5* (véase Indicadores multidivisa y de marco temporal múltiple)
- *UseWPRMTF.mq5* (véase Soporte para múltiples símbolos y marcos temporales)
- *UseM1MA.mq5* (véase Utilización de los indicadores integrados)
- *UseDemoAllLoop.mq5* (véase Borrar instancias de indicadores)
- *IndSubChart.mq5* (véase Modos de visualización de gráficos)

5.7.22 Gestionar indicadores en el gráfico

Como ya hemos visto, los gráficos son el entorno de ejecución y visualización de los indicadores. Su estrecha relación encuentra una confirmación adicional en forma de todo un grupo de funciones integradas que proporcionan control sobre los indicadores en los gráficos. En uno de los capítulos anteriores, ya completamos una visión general de estas características. Ahora, tras familiarizarnos con los gráficos, estamos preparados para estudiarlos en detalle.

Todas las funciones están unidas por el hecho de que los dos primeros parámetros están unificados: se trata del identificador del gráfico (*chartId*) y del número de ventana (*window*). Los valores cero de los parámetros denotan el gráfico actual y la ventana principal, respectivamente.

int ChartIndicatorsTotal(long chartId, int window)

La función devuelve el número de todos los indicadores adjuntos a la ventana de gráfico especificada. Puede utilizarse para enumerar todos los indicadores adjuntos a un gráfico determinado. El número de todas las ventanas del gráfico puede obtenerse de la propiedad CHART_WINDOWS_TOTAL mediante la función *ChartGetInteger*.

string ChartIndicatorName(long chartId, int window, int index)

La función devuelve el nombre corto del indicador por el *index* en la lista de indicadores situados en la ventana del gráfico especificado. El nombre corto es el nombre especificado en la propiedad INDICATOR_SHORTNAME por la función *IndicatorSetString* (si no se establece, por defecto es igual al nombre del archivo del indicador).

int ChartIndicatorGet(long chartId, int window, const string shortname)

Devuelve el manejador del indicador con el nombre corto especificado en la ventana de gráfico específica. Podemos decir que la identificación del indicador en la función *ChartIndicatorGet* se hace exactamente por el nombre corto, y por lo tanto se recomienda componerlo de tal manera que contenga los valores de todos los parámetros de entrada. Si esto no es posible por una razón u otra, existe otra forma de identificar una instancia de indicador a través de la lista de sus parámetros, que puede obtenerse mediante un descriptor dado utilizando la función *IndicatorParameters*.

La obtención de un manejador de una función *ChartIndicatorGet* aumenta el contador interno para utilizar este indicador. El sistema de ejecución del terminal mantiene cargados todos los indicadores cuyo contador es mayor que cero. Por lo tanto, un indicador que ya no se necesita debe liberarse explícitamente llamando a *IndicatorRelease*. De lo contrario, el indicador permanecerá inactivo y consumirá recursos.

bool ChartIndicatorAdd(long chartId, int window, int handle)

La función añade un indicador con el descriptor pasado en el último parámetro a la ventana del gráfico especificada. El indicador y el gráfico deben tener la misma combinación de símbolo y marco temporal. En caso contrario, se producirá el error ERR_CHART_INDICATOR_CANNOT_ADD (4114).

Para añadir un indicador a una nueva ventana, el parámetro *window* debe ser mayor en uno que el índice de la última ventana existente, es decir, igual a la propiedad CHART_WINDOWS_TOTAL recibida a través de la llamada a *ChartGetInteger*. Si el valor del parámetro supera el valor de *ChartGetInteger(ID,CHART_WINDOWS_TOTAL)*, no se creará una nueva ventana ni un nuevo indicador.

> Si se añade un indicador a la ventana del gráfico principal, que debería dibujarse en una subventana separada (por ejemplo, un iMACD integrado o un indicador personalizado con la propiedad especificada *#property indicator_separate_window*), entonces dicho indicador puede parecer invisible, aunque estará presente en la lista de indicadores. Esto suele significar que los valores de este indicador no se encuentran dentro del rango mostrado en el gráfico de precios. Los valores de dicho indicador «invisible» pueden observarse en *Data window* y leerse utilizando funciones de otros programas MQL.

Añadir un indicador a un gráfico aumenta el contador interno de su uso debido a su vinculación con el gráfico. Si el programa MQL mantiene su descriptor y ya no es necesario, entonces vale la pena borrarlo llamando a *IndicatorRelease*. En realidad, esto reducirá el contador, pero el indicador permanecerá en el gráfico.

bool ChartIndicatorDelete(long chartId, int window, const string shortname)

La función elimina el indicador con el nombre corto especificado de la ventana con el número *window* en el gráfico con *chartId*. Si hay varios indicadores con el mismo nombre corto en la subventana del gráfico especificado, se eliminará el primero en orden.

Si se calculan otros indicadores utilizando los valores del indicador eliminado en el mismo gráfico, también se eliminarán.

Borrar un indicador de un gráfico no significa que su parte calculada también se borre de la memoria del terminal si el descriptor permanece en el programa MQL. Para liberar el manejador del indicador, utilice la función *IndicatorRelease*.

La función *ChartWindowFind* devuelve el número de la subventana donde se encuentra el indicador. Hay 2 formas diseñadas para buscar el indicador actual en su gráfico o un indicador con un nombre corto dado en un gráfico arbitrario con el identificador *chartId*.

int ChartWindowFind()
int ChartWindowFind(long chartId, string shortname)

La segunda forma puede utilizarse en scripts y Asesores Expertos.

Como primer ejemplo de demostración de estas funciones, vamos a analizar la versión completa del script *ChartList.mq5*. Lo hemos creado y perfeccionado gradualmente en las secciones anteriores, hasta la sección Obtener el número y la visibilidad de las ventanas/subventanas. En comparación con *ChartList4.mq5* que se presentó ahí, añadiremos variables de entrada para poder listar sólo gráficos con programas MQL y suprimir la visualización de ventanas ocultas.

```
input bool IncludeEmptyCharts = true;
input bool IncludeHiddenWindows = true;
```

Con el valor por defecto (*true*) el parámetro *IncludeEmptyCharts* ordena incluir todos los gráficos en la lista, incluidos los vacíos. El parámetro *IncludeHiddenWindows* establece por defecto la visualización de las ventanas ocultas. Estos ajustes corresponden a la lógica de creación de scripts *ChartListN* anterior.

Para calcular el número total de indicadores e indicadores en subventanas, definimos las variables *indicators* y *subs*.

```
void ChartList()
{
   ...
   int indicators = 0, subs = 0;
   ...
```

El bucle de trabajo en las ventanas del gráfico actual ha sufrido cambios importantes.

```
void ChartList()
{
      ...
      for(int i = 0; i < win; i++)
      {
         const bool visible = ChartGetInteger(id, CHART_WINDOW_IS_VISIBLE, i);
         if(!visible && !IncludeHiddenWindows) continue;
         if(!visible)
         {
            Print("  ", i, "/Hidden");
         }
         const int n = ChartIndicatorsTotal(id, i);
         for(int k = 0; k < n; k++)
         {
            if(temp == 0)
            {
               Print(header);
            }
            Print("  ", i, "/", k, " [I] ", ChartIndicatorName(id, i, k));
            indicators++;
            if(i > 0) subs++;
            temp++;
         }
      }
      ...
```

Aquí hemos añadido las llamadas *ChartIndicatorsTotal* y *ChartIndicatorName*. Ahora la lista mencionará programas MQL de todo tipo: [E]: Asesores Expertos; [S]: scripts; [I]: indicadores.

A continuación se muestra un ejemplo de las entradas de registro generadas por el script para la configuración predeterminada.

```
Chart List
N, ID, Symbol, TF, #subwindows, *active, Windows handle
0 132358585987782873 EURUSD M15 #1    133538
  1/0 [I] ATR(11)
1 132360375330772909 EURUSD D1     133514
2 132544239145024745 EURUSD M15   *   395646
 [S] ChartList
3 132544239145024732 USDRUB D1     395688
4 132544239145024744 EURUSD H1 #2  active  2361730
  1/0 [I] %R(14)
  2/Hidden
  2/0 [I] Momentum(15)
5 132544239145024746 EURUSD H1     133584
Total chart number: 6, with MQL-programs: 3
Experts: 0, Scripts: 1, Indicators: 3 (main: 0 / sub: 3)
```

Si fijamos ambos parámetros de entrada en *false*, obtendremos una lista reducida.

```
Chart List
N, ID, Symbol, TF, #subwindows, *active, Windows handle
0 132358585987782873 EURUSD M15 #1    133538
  1/0 [I] ATR(11)
2 132544239145024745 EURUSD M15   * active  395646
 [S] ChartList
4 132544239145024744 EURUSD H1 #2    2361730
  1/0 [I] %R(14)
Total chart number: 6, with MQL-programs: 3
Experts: 0, Scripts: 1, Indicators: 2 (main: 0 / sub: 2)
```

Como segundo ejemplo, vamos a analizar un script interesante *ChartIndicatorMove.mq5*.

Cuando ejecutamos varios indicadores en un gráfico, a menudo podemos necesitar cambiar el orden de los indicadores. MetaTrader 5 no tiene herramientas integradas para ello, lo que le obliga a eliminar algunos indicadores y añadirlos de nuevo, mientras que es importante guardar y restaurar la configuración. El script *ChartIndicatorMove.mq5* ofrece una opción para automatizar este procedimiento. Es importante tener en cuenta que el script sólo transfiere indicadores: si necesita cambiar el orden de las subventanas junto con los objetos gráficos (si están dentro), entonces debe utilizar plantillas tpl.

La base de funcionamiento de *ChartIndicatorMove.mq5* es la siguiente: cuando el script se aplica a un gráfico, determina a qué ventana/subventana se ha añadido, y comienza a listar al usuario los indicadores que allí se encuentran con una petición de confirmación de la transferencia. El usuario puede aceptar o continuar con el listado.

La dirección del movimiento, arriba o abajo, se establece en la variable de entrada *MoveDirection*. La enumeración DIRECTION lo describirá:

```
#property script_show_inputs

enum DIRECTION
{
   Up = -1,
   Down = +1,
};

input DIRECTION MoveDirection = Up;
```

Para transferir el indicador no a la subventana vecina sino a la siguiente, es decir, para intercambiar realmente las subventanas con indicadores en lugares (lo que suele ser necesario), introducimos la variable de entrada *jumpover*.

```
input bool JumpOver = true;
```

El bucle a través de los indicadores de la ventana de destino obtenidos de *ChartWindowOnDropped* comienza en *OnStart*.

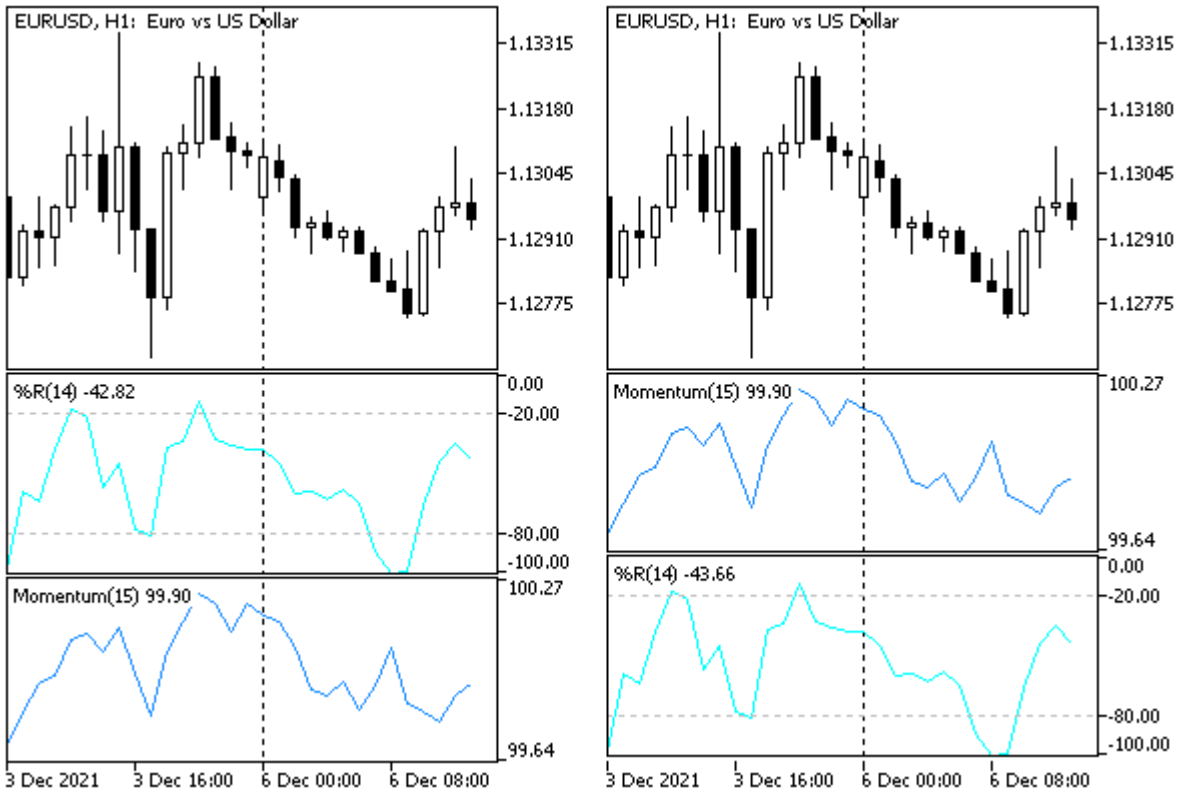
```
void OnStart()
{
   const int w = ChartWindowOnDropped();
   if(w == 0 && MoveDirection == Up)
   {
      Alert("Can't move up from window at index 0");
      return;
   }
   const int n = ChartIndicatorsTotal(0, w);
   for(int i = 0; i < n; ++i)
   {
      ...
   }
}
```

Dentro del bucle, definimos el nombre del siguiente indicador, mostramos un mensaje al usuario y movemos el indicador de una ventana a otra utilizando una secuencia de las siguientes manipulaciones:

- Obtenga el manejador llamando a *ChartIndicatorGet.*
- Añádalo a la ventana situada encima o debajo de la actual a través de *ChartIndicatorAdd*, de acuerdo con la dirección seleccionada, y al desplazarse hacia abajo, se puede crear automáticamente una nueva subventana.
- Elimine el indicador de la ventana anterior con *ChartIndicatorDelete.*
- Libere el descriptor, puesto que ya no lo necesitamos en el programa.

```
...
const string name = ChartIndicatorName(0, w, i);
const string caption = EnumToString(MoveDirection);
const int button = MessageBox("Move '" + name + "' " + caption + "?",
   caption, MB_YESNOCANCEL);
if(button == IDCANCEL) break;
if(button == IDYES)
{
   const int h = ChartIndicatorGet(0, w, name);
   ChartIndicatorAdd(0, w + MoveDirection, h);
   ChartIndicatorDelete(0, w, name);
   IndicatorRelease(h);
   break;
}
...
```

En la siguiente imagen se muestra el resultado de intercambiar subventanas con los indicadores *WPR* y *Momentum*. El script se lanzó soltándolo en la subventana superior con el indicador *WPR*, la dirección de movimiento se eligió hacia abajo (*Down*), el salto (*JumpOver*) se activó por defecto.



Intercambio de indicadores en las subventanas

Tenga en cuenta que, si mueve el indicador de la subventana a la ventana principal, lo más probable es que sus gráficos no sean visibles debido a que los valores van más allá del rango de precios mostrado. Si esto ocurre por error, puede utilizar el script para transferir el indicador de nuevo a la subventana.

5.7.23 Abrir y cerrar gráficos

Un programa MQL no sólo puede analizar la lista de gráficos, sino también modificarla: abrir nuevos o cerrar los existentes. Se asignan dos funciones a estos efectos: *ChartOpen* y *ChartClose*.

long ChartOpen(const string symbol, ENUM_TIMEFRAMES timeframe)

La función abre un nuevo gráfico con el símbolo y el marco temporal especificados y devuelve el ID del nuevo gráfico. Si se produce un error durante la ejecución, el resultado es 0 y el código de error puede leerse en la variable integrada *_LastError*.

Si el parámetro *symbol* es NULL, significa el símbolo del gráfico actual (en el que se está ejecutando el programa MQL). El valor 0 del parámetro *timeframe* corresponde a PERIOD_CURRENT.

El número máximo posible de gráficos abiertos simultáneamente en el terminal no puede superar CHARTS_MAX (100).

Veremos un ejemplo de uso de la función *ChartOpen* en la próxima sección, después de estudiar las funciones para trabajar con plantillas tpl.

> Tenga en cuenta que el terminal le permite crear no sólo ventanas completas con gráficos, sino también objetos gráficos. Se colocan dentro de los gráficos normales del mismo modo que otros objetos gráficos como líneas de tendencia, canales, etiquetas de precios, etc. Los objetos gráficos permiten mostrar dentro de un gráfico estándar varios fragmentos pequeños de series de precios para símbolos y marcos temporales alternativos.

bool ChartClose(long chartId = 0)

La función cierra el gráfico con el ID especificado (el valor por defecto 0 significa el gráfico actual). La función devuelve un indicador de éxito.

Como ejemplo, vamos a implementar el script *ChartCloseIdle.mq5*, que cerrará los gráficos duplicados con combinaciones repetidas de símbolo y marco temporal si no contienen programas MQL y objetos gráficos.

En primer lugar, tenemos que hacer una lista que cuente los gráficos de un determinado par símbolo/marco temporal. Esta tarea se implementa mediante la función *ChartIdleList*, que es muy similar a la que vimos en el script *ChartList.mq5*. La propia lista se forma en el array del mapa *MapArray<string,int> chartCounts*.

```
#include <MQL5Book/Periods.mqh>
#include <MQL5Book/MapArray.mqh>

#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1) - 1] = V)

void OnStart()
{
   MapArray<string,int> chartCounts;
   ulong duplicateChartIDs[];
   // collect duplicate empty charts
   if(ChartIdleList(chartCounts, duplicateChartIDs))
   {
      ...
   }
   else
   {
      Print("No idle charts.");
   }
}
```

Mientras tanto, la función *ChartIdleList* rellena el array *duplicateChartIDs* con identificadores de gráficos libres que coinciden con las condiciones de cierre.

```
int ChartIdleList(MapArray<string,int> &map, ulong &duplicateChartIDs[])
{
   // list charts until their list ends
   for(long id = ChartFirst(); id != -1; id = ChartNext(id))
   {
      // skip objects
      if(ChartGetInteger(id, CHART_IS_OBJECT)) continue;

      // getting the main properties of the chart
      const int win = (int)ChartGetInteger(id, CHART_WINDOWS_TOTAL);
      const string expert = ChartGetString(id, CHART_EXPERT_NAME);
      const string script = ChartGetString(id, CHART_SCRIPT_NAME);
      const int objectCount = ObjectsTotal(id);

      // count the number of indicators
      int indicators = 0;
      for(int i = 0; i < win; ++i)
      {
         indicators += ChartIndicatorsTotal(id, i);
      }

      const string key = ChartSymbol(id) + "/" + PeriodToString(ChartPeriod(id));

      if(map[key] == 0     // the first time we always read a new symbol/TF combinati
                           // otherwise, only empty charts are counted:
         || (indicators == 0           // no indicators
            && StringLen(expert) == 0  // no Expert Advisor
            && StringLen(script) == 0  // no script
            && objectCount == 0))      // no objects
      {
         const int i = map.inc(key);
         if(map[i] > 1)                // duplicate
         {
            PUSH(duplicateChartIDs, id);
         }
      }
   }
   return map.getSize();
}
```

Una vez formada la lista para borrar, en *OnStart* llamamos a la función *ChartClose* en un bucle sobre la lista.

```
void OnStart()
{
   ...
   if(ChartIdleList(chartCounts, duplicateChartIDs))
   {
      for(int i = 0; i < ArraySize(duplicateChartIDs); ++i)
      {
         const ulong id = duplicateChartIDs[i];
         // request to bring the chart to the front
         ChartSetInteger(id, CHART_BRING_TO_TOP, true);
         // update the state of the windows, pumping the queue with the request
         ChartRedraw(id);
         // ask user for confirmation
         const int button = MessageBox(
            "Remove idle chart: "
            + ChartSymbol(id) + "/" + PeriodToString(ChartPeriod(id)) + "?",
            __FILE__, MB_YESNOCANCEL);
         if(button == IDCANCEL) break;
         if(button == IDYES)
         {
            ChartClose(id);
         }
      }
      ...
```

Para cada gráfico, primero se llama a la función *ChartSetInteger(id, CHART_BRING_TO_TOP, true)* para mostrar al usuario qué ventana debe cerrarse. Dado que esta función es asíncrona (sólo pone el comando para activar la ventana en la cola de eventos), es necesario llamar adicionalmente a *ChartRedraw*, que procesa todos los mensajes acumulados. A continuación, se pide al usuario que confirme la acción. El gráfico sólo se cierra al hacer clic en *Yes*. Seleccionando *No* se salta el gráfico actual (lo deja abierto), y el bucle continúa. Pulsando *Cancel*, puede interrumpir el bucle antes de tiempo.

5.7.24 Trabajar con plantillas de gráficos tpl

La API de MQL5 proporciona dos funciones para trabajar con plantillas. Las plantillas son archivos con la extensión *tpl* que guardan el contenido de los gráficos, es decir, todos sus ajustes, junto con los objetos trazados, los indicadores y un EA (si lo hay).

bool ChartSaveTemplate(long chartId, const string filename)

La función guarda la configuración actual del gráfico en una plantilla tpl con el nombre especificado.

El gráfico lo establece *chartId*, 0 significa el gráfico actual.

El nombre del archivo para guardar la plantilla (*filename*) puede especificarse sin la extensión «.tpl», que se añadirá automáticamente. La plantilla predeterminada se guarda en la carpeta *terminal_dir/Profiles/Templates/* y puede utilizarse para aplicación manual en el terminal. No obstante, es posible especificar no sólo un nombre, sino también una ruta relativa al directorio MQL5, en concreto, empezando por «/Archivos/». Así, será posible abrir la plantilla guardada utilizando funciones para trabajar con archivos, analizarlos y, si es necesario, editarlos (véase el ejemplo *ChartTemplate.mq5* más adelante).

Si ya existe un archivo con el mismo nombre en la ruta especificada, se sobrescribirá su contenido.

Más adelante veremos un ejemplo combinado para guardar y aplicar una plantilla.

bool ChartApplyTemplate(long chartId, const string filename)

La función aplica una plantilla del archivo especificado al gráfico *chartId*.

El archivo de plantilla se busca de acuerdo con las siguientes reglas:

- Si *filename* contiene una ruta (que empiece por una barra invertida «\\» o una barra diagonal «/»), el patrón se compara con una ruta relativa *terminal_data_directory/MQL5.*
- Si no hay ruta en el nombre, la plantilla se busca en el mismo lugar en el que se encuentra el ejecutable del archivo EX5 en el que se llama a la función.
- Si la plantilla no se encuentra en los dos primeros lugares, se busca en la carpeta de plantillas estándar *terminal_dir/Profiles/Templates/*.

Tenga en cuenta que *terminal_data_directory* se refiere a la carpeta donde se almacenan los archivos modificados, y su ubicación puede variar en función del tipo de sistema operativo, el nombre de usuario y la configuración de seguridad del ordenador. Normalmente difiere de la carpeta *terminal_dir* aunque en algunos casos (por ejemplo, cuando se trabaja con una cuenta del grupo Administradores), pueden ser la misma. La ubicación de las carpetas *terminal_data_directory* y *terminal_directory* se puede encontrar utilizando la función *TerminalInfoString* (véanse las constantes TERMINAL_DATA_PATH y TERMINAL_PATH, respectivamente).

ChartApplyTemplate crea una orden que se añade a la cola de mensajes del gráfico y sólo se ejecuta después de que se hayan procesado todas las órdenes anteriores.

La carga de una plantilla detiene todos los programas MQL que se ejecutan en el gráfico, incluido el que inició la carga. Si la plantilla contiene indicadores y un Asesor Experto, se lanzarán sus nuevas instancias.

> Por motivos de seguridad, al aplicar una plantilla con un Asesor Experto a un gráfico, los permisos de trading pueden ser limitados. Si el programa MQL que llama a la función *ChartApplyTemplate* no tiene permiso para operar, entonces el Asesor Experto cargado utilizando la plantilla no tendrá permiso para operar, independientemente de la configuración de la plantilla. Si al programa MQL que llama a *ChartApplyTemplate* se le permite operar pero no se permite el trading en la configuración de la plantilla, entonces al Asesor Experto cargado usando la plantilla no se le permitirá operar.

Un ejemplo de script *ChartDuplicate.mq5* le permite crear una copia del gráfico actual.

```
void OnStart()
{
   const string temp = "/Files/ChartTemp";
   if(ChartSaveTemplate(0, temp))
   {
      const long id = ChartOpen(NULL, 0);
      if(!ChartApplyTemplate(id, temp))
      {
         Print("Apply Error: ", _LastError);
      }
   }
   else
   {
      Print("Save Error: ", _LastError);
   }
}
```

En primer lugar, se crea un archivo tpl temporal mediante *ChartSaveTemplate*; a continuación, se abre un nuevo gráfico (llamada a *ChartOpen*) y, por último, la función *ChartApplyTemplate* aplica esta plantilla al nuevo gráfico.

Sin embargo, en muchos casos, el programador se enfrenta a una tarea más difícil: no limitarse a aplicar la plantilla, sino preeditarla.

Usando las plantillas puede cambiar muchas propiedades de los gráficos que no están disponibles usando otras funciones de la API de MQL5; por ejemplo, la visibilidad de los indicadores en el contexto de los marcos temporales, el orden de las subventanas de los indicadores junto con los objetos aplicados a ellas, etc.

El formato del archivo tpl es idéntico al de los archivos chr utilizados por el terminal para almacenar gráficos entre sesiones (en la carpeta *terminal_directory/Profiles/Charts/profile_name*).

Un archivo tpl es un archivo de texto con una sintaxis especial. Las propiedades que contiene pueden ser un par clave=valor escrito en una sola línea o algún tipo de grupo que contenga varias propiedades clave=valor. Estos grupos se denominarán más abajo contenedores porque, además de propiedades individuales, también pueden contener otros contenedores anidados.

El contenedor comienza con una línea parecida a «<etiqueta>», donde *tag* es uno de los tipos de contenedor predefinidos (véase más abajo), y termina con un par de líneas como «</etiqueta>» (los nombres de las etiquetas deben coincidir). En otras palabras: el formato es similar en cierto sentido a XML (sin encabezado), en el que todas las unidades léxicas deben escribirse en líneas separadas y las propiedades de las etiquetas no se indican mediante sus atributos (como en XML dentro de la parte inicial «<tag attribute1=value1...>»), sino en el texto interior de la etiqueta.

Lista de etiquetas admitidas:

- chart - contenedor raíz con las propiedades del gráfico principal y todos los contenedores subordinados;
- expert - contenedor con propiedades generales de un Asesor Experto; por ejemplo, permiso para operar (dentro de un gráfico);
- window - contenedor con propiedades de ventana/subventana y sus contenedores subordinados (dentro del gráfico);
- object - contenedor con propiedades de objeto gráfico (dentro de la ventana);

- indicator - contenedor con las propiedades del indicador (dentro de la ventana);
- graph - contenedor con las propiedades del gráfico del indicador (dentro del indicador);
- level - contenedor con propiedades de nivel de indicador (dentro de indicador);
- period - contenedor con propiedades de visibilidad de un objeto o indicador en un marco temporal específico (dentro de un objeto o indicador);
- inputs - contenedor con configuraciones (variables de entrada) de indicadores personalizados y Asesores Expertos.

La posible lista de propiedades en pares clave=valor es bastante extensa y no tiene documentación oficial. Si es necesario, puede ocuparse usted mismo de estas características de la plataforma.

He aquí fragmentos de un archivo tpl (las sangrías en el formato se hacen para visualizar el anidamiento de contenedores):

```
<chart>
id=0
symbol=EURUSD
description=Euro vs US Dollar
period_type=1
period_size=1
digits=5
...
<window>
  height=117.133747
  objects=0
  <indicator>
    name=Main
    path=
    apply=1
    show_data=1
    ...
    fixed_height=-1
  </indicator>
</window>
<window>
  <indicator>
    name=Momentum
    path=
    apply=6
    show_data=1
    ...
    fixed_height=-1
    period=14
    <graph>
      name=
      draw=1
      style=0
      width=1
      color=16748574
    </graph>
  </indicator>
  ...
</window>
</chart>
```

Disponemos del archivo de encabezado *TplFile.mqh* para trabajar con archivos tpl, con el que podrá analizar y modificar plantillas. Tiene dos clases:

- *Container* - para leer y almacenar elementos de archivos, teniendo en cuenta la jerarquía (anidamiento), así como para escribir en un archivo tras una posible modificación;
- *Selector* - para recorrer secuencialmente los elementos de la jerarquía (objetos Contenedor) en busca de una coincidencia con una determinada consulta que se escribe como una cadena similar a un selector xpath («/ruta/elemento[atributo=valor]»).

Los objetos de la clase *Container* se crean mediante un constructor que toma el descriptor del archivo tpl para leer como primer parámetro y el nombre de la etiqueta como segundo parámetro. Por defecto,

el nombre de la etiqueta es NULL, es decir, el contenedor raíz (todo el archivo). Así, el propio contenedor se llena de contenido en el proceso de lectura del archivo (véase el método *read*).

Se supone que las propiedades del elemento actual, es decir, los pares «clave=valor» situados directamente dentro de este contenedor se añaden al mapa *MapArray<string,string> properties*. Los contenedores anidados se añaden al array *Container *children[]*.

```
#include <MQL5Book/MapArray.mqh>

#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1) - 1] = V)

class Container
{
   MapArray<string,string> properties;
   Container *children[];
   const string tag;
   const int handle;
public:
   Container(const int h, const string t = NULL): handle(h), tag(t) { }
   ~Container()
   {
      for(int i = 0; i < ArraySize(children); ++i)
      {
         if(CheckPointer(children[i]) == POINTER_DYNAMIC) delete children[i];
      }
   }

   bool read(const bool verbose = false)
   {
      while(!FileIsEnding(handle))
      {
         string text = FileReadString(handle);
         const int len = StringLen(text);
         if(len > 0)
         {
            if(text[0] == '<' && text[len - 1] == '>')
            {
               const string subtag = StringSubstr(text, 1, len - 2);
               if(subtag[0] == '/' && StringFind(subtag, tag) == 1)
               {
                  if(verbose)
                  {
                     print();
                  }
                  return true;       // the element is ready
               }

               PUSH(children, new Container(handle, subtag)).read(verbose);
            }
            else
            {
               string pair[];
               if(StringSplit(text, '=', pair) == 2)
               {
                  properties.put(pair[0], pair[1]);
               }
            }
         }
```

```
      }
      return false;
   }
   ...
};
```

En el método *read*, leemos y analizamos el archivo línea por línea. En caso de etiqueta de apertura de la forma «<etiqueta>», creamos un nuevo objeto contenedor y continuamos leyendo en él. En caso de etiqueta de cierre de la forma «</etiqueta>» con el mismo nombre, devolvemos una bandera de éxito (*true*) que significa que el contenedor se ha generado. En las líneas restantes, leemos los pares «clave=valor» y los añadimos al array *properties*.

Hemos preparado *Selector* para buscar elementos en una plantilla. A su constructor se le pasa una cadena con la jerarquía de las etiquetas buscadas. Por ejemplo, la cadena «/gráfico/ventana/indicador» corresponde a un gráfico que tiene una ventana/subventana que, a su vez, contiene cualquier indicador. El resultado de la búsqueda será la primera coincidencia. Esta consulta, por regla general, encontrará el gráfico de cotizaciones, porque está almacenado en la plantilla como un indicador llamado «Principal» y va al principio del archivo, delante de otras subventanas.

Consultas más prácticas que especifican el nombre y el valor de un atributo concreto. En concreto, la cadena modificada «/gráfico/ventana/indicador[nombre=Momentum]» sólo buscará el indicador *Momentum*. Esta búsqueda es diferente a llamar a *ChartWindowFind*, porque aquí el nombre se especifica sin parámetros, mientras que *ChartWindowFind* utiliza un nombre corto del indicador, lo que normalmente incluye valores de parámetros, pero pueden variar.

Para los indicadores integrados, la propiedad *name* contiene el nombre en sí, y para los personalizados, dirá «Indicador personalizado». El enlace al indicador personalizado se indica en la propiedad *path* como una ruta al archivo ejecutable, por ejemplo, "Indicadores\MQL5Book\IndTripleEMA.ex5".

Veamos la estructura interna de la clase *Selector*.

```
class Selector
{
   const string selector;
   string path[];
   int cursor;
public:
   Selector(const string s): selector(s), cursor(0)
   {
      StringSplit(selector, '/', path);
   }
   ...
```

En el constructor, descomponemos la consulta *selector* en componentes independientes y los guardamos en el array *path*. El componente actual de la ruta que se está comparando en el patrón viene dado por la variable *cursor*. Al principio de la búsqueda, estamos en el contenedor raíz (estamos considerando todo el archivo tpl), y el *cursor* es 0. A medida que se encuentren coincidencias, *cursor* debería aumentar (véase el método *accept* más abajo).

El operador [], con cuya ayuda se puede obtener el i-ésimo fragmento de la ruta, está sobrecargado en la clase. También tiene en cuenta que en el fragmento, entre corchetes, se puede especificar el par «[clave=valor]».

```
string operator[](int i) const
{
   if(i < 0 || i >= ArraySize(path)) return NULL;
   const int param = StringFind(path[i], "[");
   if(param > 0)
   {
      return StringSubstr(path[i], 0, param);
   }
   return path[i];
}
...
```

El método *accept* comprueba si el nombre del elemento (*tag*) y sus propiedades (*properties*) coinciden con los datos especificados en la ruta del selector para la posición actual del cursor. El registro *this[cursor]* utiliza la sobrecarga anterior del operador [] .

```
bool accept(const string tag, MapArray<string,string> &properties)
{
   const string name = this[cursor];
   if(!(name == "" && tag == NULL) && (name != tag))
   {
      return false;
   }

   // if the request has a parameter, check it among the properties
   // NB! so far only one attribute is supported, but many "tag[a1=v1][a2=v2]..."
   const int start = StringLen(path[cursor]) > 0 ? StringFind(path[cursor], "[") :
   if(start > 0)
   {
      const int stop = StringFind(path[cursor], "]");
      const string prop = StringSubstr(path[cursor], start + 1, stop - start - 1);

      // NB! only '=' is supported, but it should be '>', '<', etc.
      string kv[];   // key and value
      if(StringSplit(prop, '=', kv) == 2)
      {
         const string value = properties[kv[0]];
         if(kv[1] != value)
         {
            return false;
         }
      }
   }

   cursor++;
   return true;
}
...
```

El método devolverá *false* si el nombre de la etiqueta no coincide con el fragmento actual de la ruta, y también si el fragmento contenía el valor de algún parámetro y no es igual o no está en el array

properties. En otros casos, obtendremos una coincidencia de las condiciones, como resultado de lo cual el cursor se moverá hacia adelante (*cursor*++) y el método devolverá *true*.

El proceso de búsqueda se completará con éxito cuando el cursor alcance el último fragmento de la petición, por lo que necesitamos un método para determinar este momento, que es *isComplete*.

```
bool isComplete() const
{
   return cursor == ArraySize(path);
}

int level() const
{
   return cursor;
}
```

Además, durante el análisis de las plantillas, puede haber situaciones en las que hayamos recorrido la parte de la ruta correspondiente a la jerarquía de contenedores (es decir, hayamos encontrado varias coincidencias), tras lo cual el siguiente fragmento de solicitud no coincida. En este caso, es necesario «volver» a los niveles anteriores de la solicitud, para lo cual se implementa el método *unwind*.

```
bool unwind()
{
   if(cursor > 0)
   {
      cursor--;
      return true;
   }
   return false;
}
};
```

Ahora todo está listo para organizar la búsqueda en la jerarquía de contenedores (que obtenemos después de leer el archivo tpl) utilizando el objeto *Selector*. Todas las acciones necesarias serán realizadas por el método *find* de la clase *Container*. Toma el objeto *Selector* como parámetro de entrada y se llama a sí mismo recursivamente mientras haya coincidencias según el método *Selector::accept*. Alcanzar el final de la solicitud significa éxito, y el método *find* devolverá el contenedor actual al código de llamada.

```
Container *find(Selector *selector)
{
   const string element = StringFormat("%*s", 2 * selector.level(), " ")
      + "<" + tag + "> " + (string)ArraySize(children);
   if(selector.accept(tag, properties))
   {
      Print(element + " accepted");

      if(selector.isComplete())
      {
         return &this;
      }

      for(int i = 0; i < ArraySize(children); ++i)
      {
         Container *c = children[i].find(selector);
         if(c) return c;
      }
      selector.unwind();
   }
   else
   {
      Print(element);
   }

   return NULL;
}
...
```

Observe que, a medida que nos desplazamos por el árbol de objetos, el método *find* registra el nombre de etiqueta del objeto actual y el número de objetos anidados, y lo hace con una sangría proporcional al nivel de anidamiento de los objetos. Si el artículo coincide con la solicitud, a la entrada del registro se le añade la palabra «aceptado».

También es importante señalar que esta implementación devuelve el primer elemento coincidente y no continúa buscando otros candidatos, y en teoría, esto puede ser útil para las plantillas porque a menudo tienen varias etiquetas del mismo tipo dentro del mismo contenedor. Por ejemplo, una ventana puede contener muchos objetos, y un programa MQL puede estar interesado en analizar toda la lista de objetos. Se propone estudiar este aspecto de forma opcional.

Para simplificar la llamada de búsqueda, se ha añadido un método del mismo nombre que toma un parámetro de cadena y crea el objeto *Selector* localmente.

```
Container *find(const string selector)
{
   Selector s(selector);
   return find(&s);
}
```

Dado que vamos a editar la plantilla, deberíamos proporcionar métodos para modificar el contenedor; en concreto, para añadir un par clave=valor y un nuevo contenedor anidado con una etiqueta dada.

```
void assign(const string key, const string value)
{
   properties.put(key, value);
}

Container *add(const string subtag)
{
   return PUSH(children, new Container(handle, subtag));
}

void remove(const string key)
{
   properties.remove(key);
}
```

Tras la edición, deberá volver a escribir el contenido de los contenedores en un archivo (el mismo o distinto). Un método de ayuda *save* guarda el objeto en el formato tpl descrito anteriormente: comienza con la etiqueta de apertura «<etiqueta>», continúa descargando todas las propiedades clave=valor y llama a *save* para los objetos anidados, tras lo cual termina con la etiqueta de cierre «</etiqueta>». El descriptor de archivo se pasa como parámetro para guardar.

```
bool save(const int h)
{
   if(tag != NULL)
   {
      if(FileWriteString(h, "<" + tag + ">\n") <= 0)
         return false;
   }
   for(int i = 0; i < properties.getSize(); ++i)
   {
      if(FileWriteString(h, properties.getKey(i) + "=" + properties[i] + "\n") <=
         return false;
   }
   for(int i = 0; i < ArraySize(children); ++i)
   {
      children[i].save(h);
   }
   if(tag != NULL)
   {
      if(FileWriteString(h, "</" + tag + ">\n") <= 0)
         return false;
   }
   return true;
}
```

El método de alto nivel para escribir una plantilla completa en un archivo se denomina *write*. Su parámetro de entrada (descriptor de archivo) puede ser igual a 0, lo que significa escribir en el mismo archivo del que se ha leído. Sin embargo, el archivo debe abrirse con permiso de escritura.

Es importante tener en cuenta que al sobrescribir un archivo de texto Unicode, MQL5 no escribe la marca UTF inicial (la denominada BOM, Byte Order Mark), y por lo tanto tenemos que hacerlo nosotros. De lo contrario, sin la marca, el terminal no leerá ni aplicará nuestra plantilla.

Si el código de llamada pasa en el parámetro *h* otro archivo abierto exclusivamente para escribir en formato Unicode, MQL5 escribirá la BOM automáticamente.

```
bool write(int h = 0)
{
   bool rewriting = false;
   if(h == 0)
   {
      h = handle;
      rewriting = true;
   }
   if(!FileGetInteger(h, FILE_IS_WRITABLE))
   {
      Print("File is not writable");
      return false;
   }

   if(rewriting)
   {
      // NB! We write the BOM manually because MQL5 does not do this when overwrit
      ushort u[1] = {0xFEFF};
      FileSeek(h, SEEK_SET, 0);
      FileWriteString(h, ShortArrayToString(u));
   }

   bool result = save(h);

   if(rewriting)
   {
      // NB! MQL5 does not allow to reduce file size,
      // so we fill in the extra ending with spaces
      while(FileTell(h) < FileSize(h) && !IsStopped())
      {
         FileWriteString(h, " ");
      }
   }
   return result;
}
```

Para demostrar las capacidades de las nuevas clases, considere el problema de ocultar la ventana de un indicador específico. Como sabe, el usuario puede conseguirlo restableciendo las banderas de visibilidad de los marcos temporales en el cuadro de diálogo de propiedades del indicador (pestaña *Display*). De forma programática, esto no puede hacerse directamente. Aquí es donde entra en juego la capacidad de editar la plantilla.

En la plantilla, la visibilidad del indicador para los marcos temporales se especifica en el contenedor <indicador>, dentro del cual se escribe un contenedor separado para cada marco temporal <período> visible. Por ejemplo, la visibilidad en el marco temporal M15 tiene este aspecto:

```
<period>
period_type=0
period_size=15
</period>
```

Dentro del contenedor <periodo> se utilizan las propiedades *period_type* y *period_size*. *period_type* es una unidad de medida, una de las siguientes:

- 0 para minutos
- 1 para horas
- 2 para semanas
- 3 para meses

period_size es el número de unidades de medida en el marco temporal. Debe tenerse en cuenta que el marco temporal diario es de 24 horas.

Cuando no hay ningún contenedor anidado <período> en el contenedor <indicador>, el indicador se muestra en todos los marcos temporales.

El libro viene con el script *ChartTemplate.mq5*, que añade el indicador Momentum al gráfico (si no está ya presente) y lo hace visible en un único marco temporal mensual.

```
void OnStart()
{
   // if Momentum(14) is not on the chart yet, add it
   const int w = ChartWindowFind(0, "Momentum(14)");
   if(w == -1)
   {
      const int momentum = iMomentum(NULL, 0, 14, PRICE_TYPICAL);
      ChartIndicatorAdd(0, (int)ChartGetInteger(0, CHART_WINDOWS_TOTAL), momentum);
      // not necessarily here because the script will exit soon,
      // however explicitly declares that the handle will no longer be needed in the
      IndicatorRelease(momentum);
   }
   ...
```

A continuación, guardamos la plantilla de gráfico actual en un archivo, que luego abrimos para escribir y leer. Sería posible asignar un archivo separado para la escritura.

```
   const string filename = _Symbol + "-" + PeriodToString(_Period) + "-momentum-rw";
   if(PRTF(ChartSaveTemplate(0, "/Files/" + filename)))
   {
      int handle = PRTF(FileOpen(filename + ".tpl",
         FILE_READ | FILE_WRITE | FILE_TXT | FILE_SHARE_READ | FILE_SHARE_WRITE));
      // alternative - another file open for writing only
      // int writer = PRTF(FileOpen(filename + "w.tpl",
      //    FILE_WRITE | FILE_TXT | FILE_SHARE_READ | FILE_SHARE_WRITE));
```

Una vez recibido un descriptor de archivo, creamos un contenedor raíz *main* y leemos todo el archivo en él (los contenedores anidados y todas sus propiedades se leerán automáticamente).

```
      Container main(handle);
      main.read();
```

A continuación, definimos un selector para buscar el indicador *Momentum*. En teoría, un enfoque más riguroso también requeriría comprobar el periodo especificado (14), pero nuestras clases no admiten la consulta de varias propiedades al mismo tiempo (esta posibilidad se deja para su estudio por separado).

Usando el selector, buscamos e imprimimos el objeto encontrado (sólo como referencia) y añadimos su contenedor anidado <periodo> con la configuración para mostrar el marco temporal mensual.

```
      Container *found = main.find("/chart/window/indicator[name=Momentum]");
      if(found)
      {
         found.print();
         Container *period = found.add("period");
         period.assign("period_type", "3");
         period.assign("period_size", "1");
      }
```

Por último, escribimos la plantilla modificada en el mismo archivo, la cerramos y la aplicamos en el gráfico.

```
      main.write(); // or main.write(writer);
      FileClose(handle);

      PRTF(ChartApplyTemplate(0, "/Files/" + filename));
   }
}
```

Al ejecutar el script en un gráfico limpio, veremos dichas entradas en el registro:

```
ChartSaveTemplate(0,/Files/+filename)=true / ok
FileOpen(filename+.tpl,FILE_READ|FILE_WRITE|FILE_TXT| »
» FILE_SHARE_READ|FILE_SHARE_WRITE|FILE_UNICODE)=1 / ok
 <> 1 accepted
  <chart> 2 accepted
    <window> 1 accepted
      <indicator> 0
    <window> 1 accepted
      <indicator> 1 accepted
Tag: indicator
                    [key]    [value]
[ 0] "name"               "Momentum"
[ 1] "path"               ""
[ 2] "apply"              "6"
[ 3] "show_data"          "1"
[ 4] "scale_inherit"      "0"
[ 5] "scale_line"         "0"
[ 6] "scale_line_percent" "50"
[ 7] "scale_line_value"   "0.000000"
[ 8] "scale_fix_min"      "0"
[ 9] "scale_fix_min_val"  "0.000000"
[10] "scale_fix_max"      "0"
[11] "scale_fix_max_val"  "0.000000"
[12] "expertmode"         "0"
[13] "fixed_height"       "-1"
[14] "period"             "14"
ChartApplyTemplate(0,/Files/+filename)=true / ok
```

Aquí se puede ver que antes de encontrar el indicador deseado (marcado como «aceptado»), el algoritmo encontró el indicador en la ventana anterior principal, pero no encajaba, porque su nombre no es igual al «Momentum» deseado.

Ahora, si abre la lista de indicadores en el gráfico, habrá *momentum*, y en su cuadro de diálogo de propiedades, en la pestaña *Display*, el único marco temporal habilitado es *Month*.

El libro va acompañado de una versión ampliada del archivo *TplFileFull.mqh*, que admite distintas operaciones de comparación en las condiciones de selección de etiquetas y su selección múltiple en arrays. Un ejemplo de su uso puede encontrarse en el script *ChartUnfix.mq5*, que desajusta los tamaños de todas las subventanas del gráfico.

5.7.25 Guardar la imagen de un gráfico

En los programas MQL, a menudo se hace necesario documentar el estado actual del propio programa y del entorno de trading. Por regla general, para ello se utiliza la salida de diversos indicadores analíticos o financieros al diario, pero algunas cosas se representan más claramente con la imagen del gráfico; por ejemplo, en el momento de la transacción. La API de MQL5 incluye una función que permite guardar la imagen de un gráfico en un archivo.

bool ChartScreenShot(long chartId, string filename, int width, int height, ENUM_ALIGN_MODE alignment = ALIGN_RIGHT)

La función toma una instantánea del gráfico especificado en formato GIF, PNG o BMP dependiendo de la extensión en la línea con el nombre del archivo *filename* (máximo 63 caracteres). La captura de pantalla se coloca en el directorio *MQL5/Files*.

Los parámetros *width* y *height* establecen la anchura y la altura de la imagen en píxeles.

El parámetro *alignment* influye en qué parte del gráfico se incluirá en el archivo. El valor ALIGN_RIGHT (por defecto) significa que la instantánea se toma para los precios más recientes (esto puede concebirse como que el terminal hace una transición silenciosamente al pulsar *End* antes de la instantánea). El valor ALIGN_LEFT garantiza que las barras aparezcan en la imagen, empezando por la primera barra visible a la izquierda en ese momento. Por lo tanto, si necesita hacer una captura de pantalla de un gráfico desde una posición determinada, primero debe posicionar el gráfico manualmente o con ayuda de la función *ChartNavigate*.

La función *ChartScreenShot* devuelve *true* en caso de éxito.

Vamos a probar la función en el script *ChartPanorama.mq5*. Su tarea es guardar una copia del gráfico desde la barra visible a la izquierda hasta la hora actual. Desplazando primero el inicio del gráfico hasta la profundidad deseada del historial se puede obtener un panorama bastante extenso. En este caso, no es necesario pensar qué ancho de imagen elegir. No obstante, tenga en cuenta que un historial demasiado largo requerirá una imagen enorme, que podría superar las capacidades del formato gráfico o del software.

La altura de la imagen se determinará automáticamente igual a la altura actual del gráfico.

```
void OnStart()
{
   // the exact width of the price scale is not known, we take it empirically
   const int scale = 60;

   // calculating the total height, including gaps between windows
   const int w = (int)ChartGetInteger(0, CHART_WINDOWS_TOTAL);
   int height = 0;
   int gutter = 0;
   for(int i = 0; i < w; ++i)
   {
      if(i == 1)
      {
         gutter = (int)ChartGetInteger(0, CHART_WINDOW_YDISTANCE, i) - height;
      }
      height += (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS, i);
   }

   Print("Gutter=", gutter, ", total=", gutter * (w - 1));
   height += gutter * (w - 1);
   Print("Height=", height);

   // calculate the total width based on the number of pixels in one bar,
   // and also including chart offset from the right edge and scale width
   const int shift = (int)(ChartGetInteger(0, CHART_SHIFT) ?
      ChartGetDouble(0, CHART_SHIFT_SIZE) * ChartGetInteger(0, CHART_WIDTH_IN_PIXELS)
   Print("Shift=", shift);
   const int pixelPerBar = (int)MathRound(1.0 * ChartGetInteger(0, CHART_WIDTH_IN_PIX
      / ChartGetInteger(0, CHART_WIDTH_IN_BARS));
   const int width = (int)ChartGetInteger(0, CHART_FIRST_VISIBLE_BAR) * pixelPerBar +
   Print("Width=", width);

   // write a file with a picture in PNG format
   const string filename = _Symbol + "-" + PeriodToString() + "-panorama.png";
   if(ChartScreenShot(0, filename, width, height, ALIGN_LEFT))
   {
      Print("File saved: ", filename);
   }
}
```

También podríamos utilizar el modo ALIGN_RIGHT, pero entonces tendríamos que forzar la desactivación del desplazamiento desde el borde derecho, ya que se recalcula para la imagen, dependiendo de su tamaño, y el resultado será completamente diferente a como se ve en la pantalla (la sangría a la derecha se hará demasiado grande, ya que se especifica como un porcentaje de la anchura).

A continuación se muestra un ejemplo del registro después de ejecutar el script en el gráfico XAUUSD,H1.

```
Gutter=2, total=2
Height=440
Shift=74
Width=2086
File saved: XAUUSD-H1-panorama.png
```

Teniendo en cuenta la navegación a un historial no muy lejano, se obtuvo la siguiente captura de pantalla (representada como una copia reducida 4 veces):

Panorama del gráfico

5.8 Objetos gráficos

Los usuarios de MetaTrader 5 conocen bien el concepto de objetos gráficos: líneas de tendencia, etiquetas de precios, canales, niveles de Fibonacci, formas geométricas y muchos otros elementos visuales que se utilizan para el marcado analítico de gráficos. El lenguaje MQL5 permite crear, editar y eliminar objetos gráficos mediante programación. Esto puede ser útil, por ejemplo, cuando se desean mostrar ciertos datos simultáneamente en una subventana y en la ventana principal de un indicador. Dado que la plataforma sólo admite la salida de búferes de indicadores en una ventana, podemos generar objetos en la otra ventana. Con el marcado creado a partir de objetos gráficos es fácil organizar operaciones semiautomatizadas utilizando Asesores Expertos. Además, los objetos se utilizan a menudo para construir interfaces gráficas personalizadas para programas MQL, tales como botones, campos de entrada y banderas. Estos programas se pueden controlar sin necesidad de abrir el cuadro de diálogo de propiedades, y los paneles creados en MQL pueden ofrecer una flexibilidad mucho mayor que las variables de entrada estándar.

Cada objeto existe en el contexto de un gráfico concreto. Por eso, las funciones que analizaremos en este capítulo comparten una característica común: el primer parámetro especifica el ID del gráfico. Además, cada objeto gráfico se caracteriza por un nombre que es único dentro de un gráfico, incluidas todas las subventanas. Cambiar el nombre de un objeto gráfico implica borrar el objeto con el nombre antiguo y crear el mismo objeto con un nombre nuevo. No se pueden crear dos objetos con el mismo nombre.

Las funciones que definen las propiedades de los objetos gráficos, así como las operaciones de creación (*ObjectCreate*) y desplazamiento (*ObjectMove*) de objetos en el gráfico, sirven esencialmente para enviar órdenes asíncronas al gráfico. Si estas funciones se ejecutan correctamente, el comando entra en la cola de eventos compartidos del gráfico. La modificación visual de las propiedades de los objetos gráficos se produce durante el procesamiento de la cola de eventos para ese gráfico en particular. Por lo tanto, la representación externa del gráfico puede reflejar el cambio de estado de los objetos con cierto retraso tras las llamadas a las funciones.

En general, la actualización de los objetos gráficos del gráfico la realiza automáticamente el terminal en respuesta a eventos relacionados con el gráfico, tales como la recepción de una nueva cotización, el cambio de tamaño de la ventana, etc. Para forzar la actualización de los objetos gráficos, puede utilizar

la función de solicitud de redibujado de gráficos (*ChartRedraw*). Esto es especialmente importante tras la creación o modificación masiva de objetos.

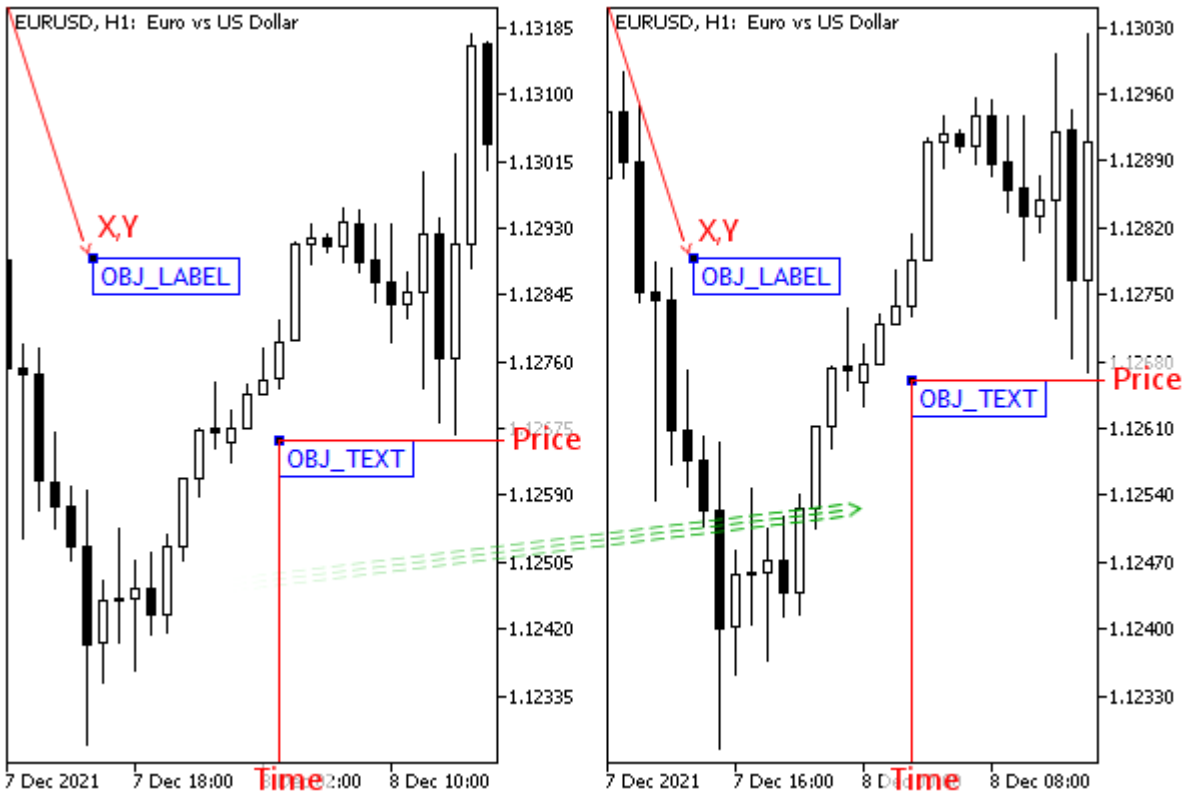
Los objetos sirven como fuente de eventos programáticos, como la creación, la eliminación, la modificación de sus propiedades y los clics del ratón. Todos los aspectos relacionados con la aparición y gestión de incidentes se tratan en un capítuloaparte, junto con los eventos en el contexto de ventana general.

Comenzaremos por los fundamentos teóricos y pasaremos gradualmente a los aspectos prácticos.

5.8.1 Tipos de objetos y características de la especificación de sus coordenadas

Como sabemos por el capítulo sobre gráficos, en la ventana hay dos sistemas de coordenadas: coordenadas de pantalla (píxel) y coordenadas de cotización (hora y precio). A este respecto, el conjunto total de tipos de objetos admitidos se divide en dos grandes grupos: los objetos vinculados a la pantalla y los vinculados al gráfico de precios. Los primeros siempre permanecen en su sitio respecto a una de las esquinas de la ventana (qué esquina es la de referencia lo determina el usuario o programador en las propiedades del objeto). Los últimos se desplazan junto con el área de trabajo de la ventana.

En la siguiente imagen se muestran dos objetos con etiquetas de texto para comparar: uno unido a la pantalla (OBJ_LABEL), y el otro al gráfico de precios (OBJ_TEXT). Sus tipos, dados entre paréntesis, así como las propiedades por las que se fijan las coordenadas, los estudiaremos en los apartados correspondientes de este capítulo. Es importante tener en cuenta que al desplazarse por el gráfico de precios, el texto OBJ_TEXT se mueve sincrónicamente con él, mientras que la inscripción OBJ_LABEL permanece en el mismo lugar.



Dos sistemas de coordenadas diferentes para los objetos

Además, los objetos difieren en el número de puntos de anclaje. Por ejemplo, una etiqueta de precio única («flecha») requiere un punto de tiempo/precio, y una línea de tendencia requiere dos de esos puntos. Existen tipos de objetos con más puntos de anclaje, como los Canales Equidistantes, los Triángulos o las Ondas de Elliott.

Cuando se selecciona un objeto (por ejemplo, en el cuadro de diálogo *Object List*, haciendo doble clic o un solo clic en el gráfico, dependiendo de la pestaña *Charts* / opción *Select objects with a single mouse click*), sus puntos de anclaje se indican mediante pequeños cuadrados de color contrastado. Son los puntos de anclaje los que se utilizan para arrastrar el objeto y cambiar su tamaño y orientación.

Todos los tipos de objeto admitidos se describen en la enumeración ENUM_OBJECT. Puede leerlo íntegramente en la documentación de MQL5. Consideraremos sus elementos gradualmente, por partes.

5.8.2 Objetos vinculados a tiempo y precio

En la siguiente tabla se proporcionan objetos con sus coordenadas de tiempo y precio, sus identificadores en la enumeración ENUM_OBJECT y el número de puntos de anclaje.

Identificador	Nombre	Puntos de anclaje
Líneas rectas simples		
OBJ_VLINE	Vertical (sólo coordenada de tiempo)	1
OBJ_HLINE	Horizontal (sólo coordenada de precio)	1
OBJ_TREND	Tendencia	2
OBJ_ARROWED_LINE	Con una flecha al final	2
Líneas verticales que se repiten periódicamente		
OBJ_CYCLES	Cíclico	2
Canales		
OBJ_CHANNEL	Equidistante	3
OBJ_STDDEVCHANNEL	Desviación típica	2
OBJ_REGRESSION	Regresión lineal	2
OBJ_PITCHFORK	Horquilla de Andrews	3
Herramientas de Fibonacci		
OBJ_FIBO	Niveles	2
OBJ_FIBOTIMES	Husos horarios	2
OBJ_FIBOFAN	Abanico	2
OBJ_FIBOARC	Arcos	2
OBJ_FIBOCHANNEL	Canal	3

Identificador	Nombre	Puntos de anclaje
OBJ_EXPANSION	Expansión	3
Herramientas de Gann		
OBJ_GANNLINE	Línea	2
OBJ_GANNFAN	Abanico	2
OBJ_GANNGRID	Red	2
Ondas de Elliot		
OBJ_ELLIOTWAVE5	Impulso	5
OBJ_ELLIOTWAVE3	Correctiva	3
Formas		
OBJ_RECTANGLE	Rectángulo	2
OBJ_TRIANGLE	Triángulo	3
OBJ_ELLIPSE	Elipse	3
Marcas y etiquetas individuales		
OBJ_ARROW_THUMB_UP	Bien	1*
OBJ_ARROW_THUMB_DOWN	Mal	1*
OBJ_ARROW_UP	Flecha arriba	1*
OBJ_ARROW_DOWN	Flecha abajo	1*
OBJ_ARROW_STOP	Marca de «Stop»	1*
OBJ_ARROW_CHECK	Marca de verificación	1*
OBJ_ARROW_LEFT_PRICE	Etiqueta de precio izquierda	1
OBJ_ARROW_RIGHT_PRICE	Etiqueta de precio derecha	1
OBJ_ARROW_BUY	Señal de compra (flecha azul hacia arriba)	1
OBJ_ARROW_SELL	Señal de venta (flecha roja hacia abajo)	1
OBJ_ARROW	Carácter Wingdings arbitrario	1*
Texto y gráficos		
OBJ_TEXT	Texto	1*
OBJ_BITMAP	Imagen	1*
Eventos		

Identificador	Nombre	Puntos de anclaje
OBJ_EVENT	Marca de tiempo en la parte inferior de la ventana principal (sólo coordenadas de tiempo)	1

Un asterisco marca aquellos objetos para los que se permite seleccionar un punto de anclaje en el objeto (por ejemplo, en una de las esquinas del objeto o en el centro de uno de los lados). Los métodos de selección pueden variar según el tipo de objeto, y los detalles se explicarán en la sección sobre Definición del punto de anclaje del objeto. Los puntos de anclaje son necesarios porque los objetos tienen un tamaño determinado, y sin ellos habría ambigüedad respecto a la posición.

5.8.3 Objetos vinculados a coordenadas de pantalla

En la siguiente tabla se enumeran los nombres y los identificadores ENUM_OBJECT de los objetos posicionados en función de las coordenadas de la pantalla. Casi todos ellos, excepto el objeto gráfico, están diseñados para crear una interfaz de usuario para los programas. En concreto, hay controles básicos como un botón y un campo de entrada, así como etiquetas y paneles para la agrupación visual de objetos. A partir de ellos, puede crear controles más complejos (por ejemplo, listas desplegables o casillas de verificación). Junto con el terminal, se suministra una biblioteca de clases con controles ya preparados en forma de un conjunto de archivos de encabezado (véase el directorio *MQL5/Include/Controls*).

Identificador	Nombre	Configuración punto de anclaje
OBJ_LABEL	Etiqueta de texto	Sí
OBJ_RECTANGLE_LABEL	Panel rectangular	
OBJ_BITMAP_LABEL	Panel con una imagen	Sí
OBJ_BUTTON	Botón	
OBJ_EDIT	Campo de entrada	
OBJ_CHART	Objeto gráfico	

Todos estos objetos requieren determinar la esquina de anclaje en la ventana del gráfico. De manera predeterminada, sus coordenadas son relativas a la esquina superior izquierda de la ventana.

Los tipos de esta lista también utilizan un punto de anclaje en el objeto, y sólo uno, que es editable en algunos objetos y está codificado en otros. Por ejemplo, un panel rectangular, un botón, un campo de entrada y un objeto gráfico siempre se anclan en su esquina superior izquierda. Y para una etiqueta o un panel con una imagen, hay muchas opciones disponibles. La elección se realiza a partir de la enumeración ENUM_ANCHOR_POINT descrita en la sección sobre Definición del punto de anclaje del objeto.

La etiqueta de texto (OBJ_LABEL) proporciona salida de texto sin posibilidad de editarlo. Para la edición, utilice el campo de entrada (OBJ_EDIT).

5.8.4 Crear objetos

Para crear un objeto, se requiere un conjunto mínimo de atributos comunes a todos los tipos. Las propiedades adicionales específicas de cada tipo pueden establecerse o modificarse posteriormente en un objeto ya existente. Los atributos necesarios incluyen el identificador del gráfico en el que debe crearse el objeto, el nombre del objeto, el número de la ventana/subventana y dos coordenadas para el primer punto de anclaje: tiempo y precio.

Aunque haya un grupo de objetos posicionados en coordenadas de pantalla, para crearlos sigue siendo necesario pasar dos valores, normalmente cero porque no se utilizan.

En general, un prototipo de la función *ObjectCreate* tiene el siguiente aspecto:

```
bool ObjectCreate(long chartId, const string name, ENUM_OBJECT type, int window,
 datetime time1, double price1, datetime time2 = 0, double price2 = 0, ...)
```

Un valor de 0 para *chartId* implica el gráfico actual. El parámetro *name* debe ser único en todo el gráfico, incluidas las subventanas, y no debe superar los 63 caracteres.

En las secciones anteriores hemos dado tipos de objeto para el parámetro *type*: son los elementos de la enumeración ENUM_OBJECT.

Como sabemos, la numeración de ventanas/subventanas para el parámetro *window* empieza en 0, que significa la ventana principal del gráfico. Si se especifica un índice mayor para una subventana, debe existir, ya que, de lo contrario, la función terminará con un error y devolverá *false*.

Sólo para recordárselo, la bandera de éxito devuelta (*true*) sólo indica que el comando para crear el objeto se ha colocado con éxito en la cola. El resultado de su ejecución no se conoce de inmediato. Es la otra cara de la llamada asíncrona, que se emplea para mejorar el rendimiento.

Para comprobar el resultado de la ejecución, puede utilizar la función *ObjectFind* o cualquiera de las funciones ObjectGet, que consultan las propiedades de un objeto. Pero debe tener en cuenta que tales funciones esperan la ejecución de toda la cola de comandos del gráfico y sólo entonces devuelven el resultado real (el estado del objeto). Este proceso puede llevar algún tiempo, durante el cual se suspenderá el código del programa MQL. En otras palabras: las funciones de comprobación del estado de los objetos son síncronas, a diferencia de las funciones de creación y modificación de objetos.

Los puntos de anclaje adicionales, a partir del segundo, son opcionales. El número permitido de puntos de anclaje, hasta 30, se proporciona para uso futuro, y no se utilizan más de 5 en los tipos de objeto actuales.

Es importante tener en cuenta que la llamada a la función *ObjectCreate* con el nombre de un objeto ya existente simplemente cambia el punto o puntos de anclaje (si las coordenadas se han modificado desde la llamada anterior). Esto es conveniente para escribir código unificado sin bifurcarse en condiciones basadas en la presencia o ausencia de un objeto. En otras palabras: una llamada incondicional a *ObjectCreate* garantiza la existencia del objeto, si no nos importa si existía antes o no. Sin embargo, hay un matiz: si, al llamar a *ObjectCreate*, el tipo de objeto o el índice de la subventana es diferente de un objeto ya existente, los datos relevantes siguen siendo los mismos y no se produce ningún error.

Al llamar a *ObjectCreate* se pueden dejar todos los puntos de anclaje con valores por defecto (null), siempre que después de esta instrucción se llame a funciones ObjectSet con las propiedades OBJPROP_TIME y OBJPROP_PRICE adecuadas.

El orden en que se especifican los puntos de anclaje puede ser importante para algunos tipos de objetos. Para canales como OBJ_REGRESSION (Canal de regresión lineal) y OBJ_STDDEVCHANNEL (Canal de desviación estándar), es obligatorio que se cumplan las condiciones *time1<time2*. De lo contrario, el canal no se construirá normalmente, aunque el objeto se creará sin errores.

Como ejemplo de la función, tomemos el script *ObjectSimpleShowcase.mq5*, que crea varios objetos de diferentes tipos en las últimas barras del gráfico, necesitando un único punto de anclaje.

Todos los ejemplos de trabajo con objetos utilizarán el archivo de encabezado *ObjectPrefix.mqh*, que contiene una definición de cadena con un prefijo común para los nombres de objetos. Así, nos resultará más cómodo, en caso necesario, limpiar los gráficos de objetos «propios».

```
const string ObjNamePrefix = "ObjShow-";
```

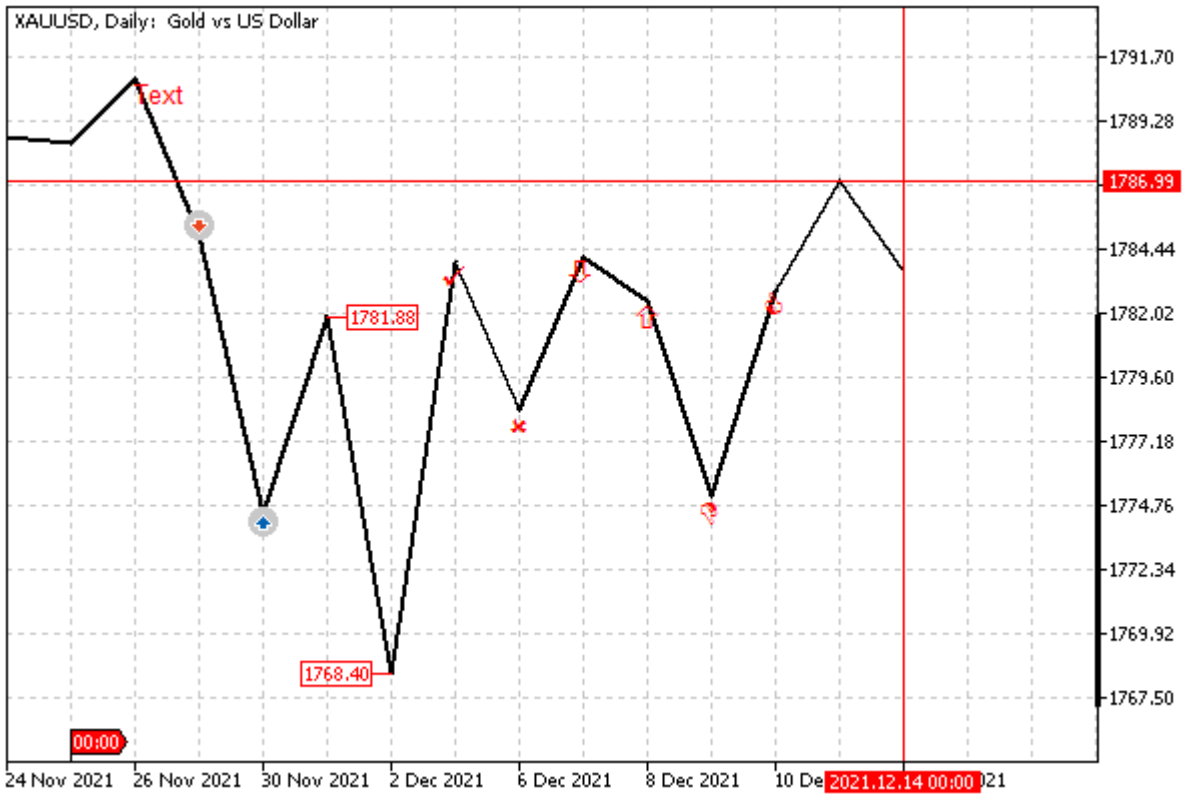
En la función *OnStart* se define un array que contiene tipos de objetos.

```
void OnStart()
{
   ENUM_OBJECT types[] =
   {
      // straight lines
      OBJ_VLINE, OBJ_HLINE,
      // labels (arrows and other signs)
      OBJ_ARROW_THUMB_UP, OBJ_ARROW_THUMB_DOWN,
      OBJ_ARROW_UP, OBJ_ARROW_DOWN,
      OBJ_ARROW_STOP, OBJ_ARROW_CHECK,
      OBJ_ARROW_LEFT_PRICE, OBJ_ARROW_RIGHT_PRICE,
      OBJ_ARROW_BUY, OBJ_ARROW_SELL,
      // OBJ_ARROW, // see the ObjectWingdings.mq5 example

      // text
      OBJ_TEXT,
      // event flag (like in a calendar) at the bottom of the window
      OBJ_EVENT,
   };
```

A continuación, en el bucle a través de sus elementos, creamos objetos en la ventana principal, pasando la hora y el precio de cierre de la barra i-ésima.

```
   const int n = ArraySize(types);
   for(int i = 0; i < n; ++i)
   {
      ObjectCreate(0, ObjNamePrefix + (string)iTime(_Symbol, _Period, i), types[i],
         0, iTime(_Symbol, _Period, i), iClose(_Symbol, _Period, i));
   }

   PrintFormat("%d objects of various types created", n);
}
```

Este es el posible resultado de ejecutar el script:



Objetos de tipos simples en los puntos de cierre de las últimas barras

El trazado de líneas mediante el precio *Close* y la visualización de la cuadrícula están activados en este ejemplo. Más adelante aprenderemos a ajustar el tamaño, el color y otros atributos de los objetos. En concreto, los puntos de anclaje de la mayoría de los iconos están situados por defecto en el centro de la parte superior, por lo que quedan visualmente desplazados bajo la línea. No obstante, el icono de venta está por encima de la línea porque el punto de anclaje siempre está en el centro de la parte inferior.

> Tenga en cuenta que los objetos creados mediante programación no se muestran por defecto en la lista de objetos del cuadro de diálogo del mismo nombre. Para verlos ahí, haga clic en el botón *All*.

5.8.5 Borrar objetos

La API de MQL5 ofrece dos funciones para eliminar objetos. Para la eliminación masiva de objetos que cumplan las condiciones de un prefijo en el nombre, tipo o número de subventana, utilice *ObjectsDeleteAll*. Si necesita seleccionar los objetos que desea eliminar por algún otro criterio (por ejemplo, por una coordenada de fecha y hora obsoleta), o si se trata de un único objeto, utilice la función *ObjectDelete*.

La función *ObjectsDeleteAll* tiene dos formas: con un parámetro para el prefijo del nombre y sin él.

```
int ObjectsDeleteAll(long chartId, int window = -1, int type = -1)
int ObjectsDeleteAll(long chartId, const string prefix, int window = -1, int type = -1)
```

La función borra todos los objetos del gráfico con la dirección *chartId* especificada, teniendo en cuenta la subventana, el tipo y la subcadena inicial del nombre.

El valor 0 del parámetro *chartId* representa el gráfico actual, como de costumbre.

Los valores por defecto (-1) en los parámetros *window* y *type* definen todas las subventanas y todos los tipos de objetos, respectivamente.

Si el prefijo está vacío, se eliminarán los objetos con cualquier nombre.

La función se ejecuta de forma sincrónica, es decir, bloquea hasta su finalización el programa MQL que está llamando y devuelve el número de objetos eliminados. Dado que la función espera a que se ejecuten todos los comandos que estaban en la cola del gráfico antes de llamarla, la acción puede tardar algún tiempo.

bool ObjectDelete(long chartId, const string name)

La función borra un objeto con el nombre especificado en el gráfico con *chartId*.

A diferencia de *ObjectsDeleteAll*, *ObjectDelete* se ejecuta de forma asíncrona, es decir, envía una orden a los gráficos para eliminar el objeto e inmediatamente devuelve el control al programa MQL. El resultado de *true* indica que el comando se ha colocado correctamente en la cola. Para comprobar el resultado de la ejecución, puede utilizar la función *ObjectFind* o cualquiera de las funciones *ObjectGet****, que consultan las propiedades de un objeto.

Como ejemplo, vea el script *ObjectCleanup1.mq5*. Su tarea consiste en eliminar los objetos con el prefijo «nuestro», generados por el script *ObjectSimpleShowcase.mq5* de la sección anterior.

En el caso más sencillo, podríamos escribir lo siguiente:

```
#include "ObjectPrefix.mqh"

void OnStart()
{
   const int n = ObjectsDeleteAll(0, ObjNamePrefix);
   PrintFormat("%d objects deleted", n);
}
```

Sin embargo, para añadir variedad, también podemos proporcionar la opción de eliminar objetos utilizando la función *ObjectDelete* a través de múltiples llamadas. Por supuesto, este enfoque no tiene sentido cuando *ObjectsDeleteAll* cumple todos los requisitos. No obstante, no siempre es así: cuando los objetos deben seleccionarse según condiciones especiales, es decir, no sólo por prefijo y tipo, *ObjectsDeleteAll* ya no servirá de ayuda.

Más adelante, cuando nos familiaricemos con las funciones de lectura de las propiedades de los objetos, completaremos el ejemplo. Mientras tanto, introduciremos sólo una variable de entrada para cambiar al modo de borrado «avanzado» (*UseCustomDeleteAll*).

```
#property script_show_inputs
input bool UseCustomDeleteAll = false;
```

En la función *OnStart*, dependiendo del modo seleccionado, llamaremos al estándar *ObjectsDeleteAll*, o a nuestra propia implementación *CustomDeleteAllObjects*.

```
void OnStart()
{
   const int n = UseCustomDeleteAll ?
      CustomDeleteAllObjects(0, ObjNamePrefix) :
      ObjectsDeleteAll(0, ObjNamePrefix);

   PrintFormat("%d objects deleted", n);
}
```

Primero vamos a esbozar esta función y luego a perfeccionarla.

```
int CustomDeleteAllObjects(const long chart, const string prefix,
   const int window = -1, const int type = -1)
{
   int count = 0;
   const int n = ObjectsTotal(chart, window, type);

   // NB: cycle through objects in reverse order of the internal chart list
   // to keep numbering as we move away from the tail
   for(int i = n - 1; i >= 0; --i)
   {
      const string name = ObjectName(chart, i, window, type);
      if(StringLen(prefix) == 0 || StringFind(name, prefix) == 0)
      // additional checks that ObjectsDeleteAll does not provide,
      // for example, by coordinates, color or anchor point
      ...
      {
         // send a command to delete a specific object
         count += ObjectDelete(chart, name);
      }
   }
   return count;
}
```

Aquí vemos varias novedades que se describirán en la sección siguiente (*ObjectsTotal*, *ObjectName*). La cuestión debería quedar clara en general: la primera función devuelve el índice de los objetos en el gráfico, y la segunda devuelve el nombre del objeto bajo el índice especificado.

También cabe señalar que el bucle a través de los objetos va en orden descendente de índice. Si lo hiciéramos de la forma habitual, la eliminación de objetos al principio de la lista supondría una violación de la numeración. En términos estrictos, ni siquiera el bucle actual garantiza el borrado completo, suponiendo que otro programa MQL comience a añadir objetos en paralelo a nuestro borrado. De hecho, se puede añadir un nuevo objeto «extraño» al principio de la lista (se forma por orden alfabético de los nombres de los objetos) y aumentar los índices restantes, empujando «nuestro» próximo objeto a eliminar más allá del índice actual *i*. Cuantos más objetos nuevos se añadan al principio, más probable será que no se borren los propios.

Por lo tanto, para mejorar la fiabilidad, sería posible comprobar después del bucle que el número de objetos restantes es igual a la diferencia entre el número inicial y el número de objetos retirados. Aunque esto no da una garantía del 100 %, ya que otros programas podrían borrar objetos en paralelo. Dejaremos estos detalles para su estudio de forma independiente.

En la implementación actual, nuestro script debería borrar todos los objetos con el prefijo «nuestro», independientemente de cambiar el modo *UseCustomDeleteAll*. El registro debería mostrar algo como lo siguiente:

```
ObjectSimpleShowcase (XAUUSD,H1) 14 objects of various types created
ObjectCleanup1 (XAUUSD,H1) 14 objects deleted
```

Vamos a descubrir las funciones *ObjectsTotal* y *ObjectName*, que acabamos de utilizar, y luego volveremos a la versión *ObjectCleanup2.mq5* del script.

5.8.6 Encontrar objetos

Existen tres funciones para buscar objetos en el gráfico. Los dos primeros, *ObjectsTotal* y *ObjectName*, permiten ordenar los objetos por su nombre y, a continuación, si es necesario, utilizar el nombre de cada objeto para analizar sus otras propiedades (describiremos cómo se hace en la siguiente sección). La tercera función, *ObjectFind*, permite comprobar la existencia de un objeto con un nombre conocido. Lo mismo podría hacerse simplemente solicitando alguna propiedad a través de la función *ObjectGet*: si no hay ningún objeto con el nombre pasado, obtendremos un error en *_LastError*, pero esto es menos conveniente que llamar a *ObjectFind*. Además, la función devuelve inmediatamente el número de la ventana en la que se encuentra el objeto.

int ObjectsTotal(long chartId, int window = -1, int type = -1)

La función devuelve el número de objetos del gráfico con el identificador *chartId* (0 significa gráfico actual). En el cálculo sólo se tienen en cuenta los objetos de la subventana con el número *window* especificado (0 representa la ventana principal, -1 representa la ventana principal y todas las subventanas). Observe que sólo se tienen en cuenta los objetos del tipo específico especificado en el parámetro *type* (-1 indica todos los tipos por defecto). El valor de *type* puede ser un elemento de la enumeración ENUM_OBJECT.

La función se ejecuta de forma sincrónica, es decir, bloquea la ejecución del programa MQL que llama hasta que se recibe el resultado.

string ObjectName(long chartId, int index, int window = -1, int type = -1)

La función devuelve el nombre del objeto bajo el número *index* en el gráfico con el identificador *chartId*. Al compilar la lista interna, dentro de la cual se busca el objeto, se tienen en cuenta el número de subventana (*window*) y el tipo de objeto (*type*) especificados. La lista se ordena por nombres de objetos en orden lexicográfico, es decir, en concreto, alfabéticamente, distinguiendo mayúsculas de minúsculas.

Al igual que *ObjectsTotal*, durante su ejecución, *ObjectName* espera a que se recupere toda la cola de comandos de gráficos y, a continuación, devuelve el nombre del objeto de la lista actualizada de objetos.

En caso de error, se obtendrá una cadena vacía y el código de error OBJECT_NOT_FOUND (4202) se almacenará en *_LastError*.

Para probar la funcionalidad de estas dos funciones, vamos a crear un script llamado *ObjectFinder.mq5* que registra todos los objetos en todos los gráficos. Utiliza las funciones de iteración del gráfico (*ChartFirst* y *ChartNext*), así como funciones para obtener propiedades del gráfico (*ChartSymbol*, *ChartPeriod* y *ChartGetInteger*).

```
#include <MQL5Book/Periods.mqh>

void OnStart()
{
   int count = 0;
   long id = ChartFirst();
   // loop through charts
   while(id != -1)
   {
      PrintFormat("%s %s (%lld)", ChartSymbol(id), PeriodToString(ChartPeriod(id)), i
      const int win = (int)ChartGetInteger(id, CHART_WINDOWS_TOTAL);
      // loop through windows
      for(int k = 0; k < win; ++k)
      {
         PrintFormat("  Window %d", k);
         const int n = ObjectsTotal(id, k);
         // loop through objects
         for(int i = 0; i < n; ++i)
         {
            const string name = ObjectName(id, i, k);
            const ENUM_OBJECT type = (ENUM_OBJECT)ObjectGetInteger(id, name, OBJPROP_
            PrintFormat("    %s %s", EnumToString(type), name);
            ++count;
         }
      }
      id = ChartNext(id);
   }

   PrintFormat("%d objects found", count);
}
```

Para cada gráfico, determinamos el número de subventanas (*ChartGetInteger(id, CHART_WINDOWS_TOTAL)*), llamamos a *ObjectsTotal* para cada subventana, y llamamos a *ObjectName* en el bucle interno. A continuación, buscamos por nombre el tipo de objeto y los mostramos juntos en el registro.

A continuación se muestra una versión del posible resultado del script (con abreviaturas).

```
EURUSD H1 (132358585987782873)
  Window 0
    OBJ_FIBO H1 Fibo 58513
    OBJ_TEXT H1 Text 40688
    OBJ_TREND H1 Trendline 3291
    OBJ_VLINE H1 Vertical Line 28732
    OBJ_VLINE H1 Vertical Line 33752
    OBJ_VLINE H1 Vertical Line 35549
  Window 1
  Window 2
EURUSD D1 (132360375330772909)
  Window 0
EURUSD M15 (132544239145024745)
  Window 0
    OBJ_VLINE H1 Vertical Line 27032
...
XAUUSD D1 (132544239145024746)
  Window 0
    OBJ_EVENT ObjShow-2021.11.25 00:00:00
    OBJ_TEXT ObjShow-2021.11.26 00:00:00
    OBJ_ARROW_SELL ObjShow-2021.11.29 00:00:00
    OBJ_ARROW_BUY ObjShow-2021.11.30 00:00:00
    OBJ_ARROW_RIGHT_PRICE ObjShow-2021.12.01 00:00:00
    OBJ_ARROW_LEFT_PRICE ObjShow-2021.12.02 00:00:00
    OBJ_ARROW_CHECK ObjShow-2021.12.03 00:00:00
    OBJ_ARROW_STOP ObjShow-2021.12.06 00:00:00
    OBJ_ARROW_DOWN ObjShow-2021.12.07 00:00:00
    OBJ_ARROW_UP ObjShow-2021.12.08 00:00:00
    OBJ_ARROW_THUMB_DOWN ObjShow-2021.12.09 00:00:00
    OBJ_ARROW_THUMB_UP ObjShow-2021.12.10 00:00:00
    OBJ_HLINE ObjShow-2021.12.13 00:00:00
    OBJ_VLINE ObjShow-2021.12.14 00:00:00
...
35 objects found
```

Aquí, en particular, puede ver que en el gráfico XAUUSD, D1 hay objetos generados por el script *ObjectSimpleShowcase.mq5*. No hay objetos en algunos gráficos y en algunas subventanas.

int ObjectFind(long chartId, const string name)

La función busca un objeto por su nombre en el gráfico especificado por el identificador y, si tiene éxito, devuelve el número de la ventana donde se ha encontrado.

Si no se encuentra el objeto, la función devuelve un número negativo. Al igual que las funciones anteriores de esta sección, la función *ObjectFind* utiliza una llamada sincrónica.

Veremos un ejemplo de uso de esta función en el script *ObjectCopy.mq5* en la siguiente sección.

5.8.7 Visión general de las funciones de acceso a las propiedades de los objetos

Los objetos tienen varios tipos de propiedades que pueden leerse y establecerse utilizando las funciones *ObjectGet* y *ObjectSet*. Como sabemos, este principio ya se ha aplicado al gráfico (véase la sección Visión general de las funciones para trabajar con el conjunto completo de propiedades de gráfico).

Todas estas funciones toman como primeros tres parámetros un identificador de gráfico, un nombre de objeto y un identificador de propiedad, que debe pertenecer a una de las enumeraciones ENUM_OBJECT_PROPERTY_INTEGER, ENUM_OBJECT_PROPERTY_DOUBLE o ENUM_OBJECT_PROPERTY_STRING. Estudiaremos propiedades específicas gradualmente en las siguientes secciones. Sus tablas dinámicas completas se pueden encontrar en la documentación de MQL5, en la página con Propiedades de los objetos.

Cabe señalar que los identificadores de propiedad de las tres enumeraciones no se intersecan, lo que permite combinar su tratamiento conjunto en un único código unificado. Lo utilizaremos en los ejemplos.

Algunas propiedades son de sólo lectura y se marcarán con «r/o» (read-only).

Como en el caso de la API de trazado, las funciones de lectura de propiedades tienen una forma corta y una forma larga: la forma corta devuelve directamente el valor solicitado, y la forma larga devuelve un booleano de éxito (*true*) o errores (*false*), y el valor en sí se coloca en el último parámetro pasado por referencia. La ausencia de error al llamar al formulario corto debe comprobarse utilizando la variable incorporada *_LastError*.

Al acceder a algunas propiedades, debe especificar un parámetro adicional (*modifier*), que se utiliza para indicar el número de valor o el nivel si la propiedad es multivalorada. Por ejemplo, si un objeto tiene varios puntos de anclaje, el modificador permite seleccionar uno en concreto.

A continuación se muestran los prototipos de funciones para leer y escribir propiedades de enteros. Tenga en cuenta que el tipo de valores en ellos es *long*, lo que le permite almacenar propiedades no sólo de los tipos *int* o *long*, sino también *bool*, *color*, *datetime* y varias enumeraciones (véase más adelante).

```
bool ObjectSetInteger(long chartId, const string name, ENUM_OBJECT_PROPERTY_INTEGER
property, long value)
bool ObjectSetInteger(long chartId, const string name, ENUM_OBJECT_PROPERTY_INTEGER
property, int modifier, long value)
long ObjectGetInteger(long chartId, const string name, ENUM_OBJECT_PROPERTY_INTEGER
property, int modifier = 0)
bool ObjectGetInteger(long chartId, const string name, ENUM_OBJECT_PROPERTY_INTEGER
property, int modifier, long &value)
```

Las funciones para propiedades reales se describen de forma similar.

bool ObjectSetDouble(long chartId, const string name, ENUM_OBJECT_PROPERTY_DOUBLE property, double value)
bool ObjectSetDouble(long chartId, const string name, ENUM_OBJECT_PROPERTY_DOUBLE property, int modifier, double value)
double ObjectGetDouble(long chartId, const string name, ENUM_OBJECT_PROPERTY_DOUBLE property, int modifier = 0)
bool ObjectGetDouble(long chartId, const string name, ENUM_OBJECT_PROPERTY_DOUBLE property, int modifier, double &value)

Por último, existen cuatro funciones idénticas para las cadenas.

bool ObjectSetString(long chartId, const string name, ENUM_OBJECT_PROPERTY_STRING property, const string value)
bool ObjectSetString(long chartId, const string name, ENUM_OBJECT_PROPERTY_STRING property, int modifier, const string value)
string ObjectGetString(long chartId, const string name, ENUM_OBJECT_PROPERTY_STRING property, int modifier = 0)
bool ObjectGetString(long chartId, const string name, ENUM_OBJECT_PROPERTY_STRING property, int modifier, string &value)

Para mejorar el rendimiento, todas las funciones para establecer las propiedades de los objetos (*ObjectSetInteger*, *ObjectSetDouble* y *ObjectSetString*) son asíncronas y esencialmente envían comandos al gráfico para modificar el objeto. Una vez ejecutadas con éxito estas funciones, los comandos se colocan en la cola de eventos compartidos del gráfico, indicada por el resultado devuelto de *true*. Cuando se produce un error, las funciones devolverán *false*, y el código de error debe comprobarse en la variable *_LastError*.

Las propiedades de los objetos se modifican con cierto retraso, durante el procesamiento de la cola de eventos del gráfico. Para forzar la actualización del aspecto y las propiedades de los objetos del gráfico, especialmente después de cambiar muchos objetos a la vez, utilice la función *ChartRedraw*.

Las funciones para obtener las propiedades de los gráficos (*ObjectGetInteger*, *ObjectGetDouble* y *ObjectGetString*) son síncronas, es decir, el código de llamada espera el resultado de su ejecución. En este caso, se ejecutan todos los comandos de la cola de gráficos para obtener el valor real de las propiedades.

Volvamos al ejemplo del script para borrar objetos; más concretamente, a su nueva versión, *ObjectCleanup2.mq5*. Recordemos que en la función *CustomDeleteAllObjects* queríamos implementar la posibilidad de seleccionar objetos en función de sus propiedades. Digamos que estas propiedades deben ser el color y el punto de anclaje. Para obtenerlos, utilice la función *ObjectGetInteger* y un par de elementos de enumeración ENUM_OBJECT_PROPERTY_INTEGER: OBJPROP_COLOR y OBJPROP_ANCHOR. Los analizaremos en detalle más adelante.

Dada esta información, el código se completaría con las siguientes comprobaciones (aquí, por simplicidad, el color y el punto de anclaje vienen dados por las constantes *clrRed* y ANCHOR_TOP. De hecho, les proporcionaremos variables de entrada).

```
int CustomDeleteAllObjects(const long chart, const string prefix,
   const int window = -1, const int type = -1)
{
   int count = 0;

   for(int i = ObjectsTotal(chart, window, type) - 1; i >= 0; --i)
   {
      const string name = ObjectName(chart, i, window, type);
      // condition on the name and additional properties, such as color and anchor pc
      if((StringLen(prefix) == 0 || StringFind(name, prefix) == 0)
         && ObjectGetInteger(0, name, OBJPROP_COLOR) == clrRed
         && ObjectGetInteger(0, name, OBJPROP_ANCHOR) == ANCHOR_TOP)
      {
         count += ObjectDelete(chart, name);
      }
   }
   return count;
}
```

Preste atención a las líneas con *ObjectGetInteger*.

Su entrada es larga y contiene cierta tautología porque las propiedades específicas están vinculadas a funciones *ObjectGet* de tipos conocidos. Además, a medida que aumenta el número de condiciones, puede parecer redundante repetir el ID del gráfico y el nombre del objeto.

Para simplificar el registro, pasemos a la tecnología que probamos en el archivo *ChartModeMonitor.mqh* en la sección sobre Modos de visualización de gráficos. Su significado es describir una clase intermediaria con sobrecargas de métodos para leer y escribir propiedades de todos los tipos. Pongamos nombre al nuevo archivo de encabezado *ObjectMonitor.mqh*.

La clase *ObjectProxy* reproduce fielmente la estructura de la clase *ChartModeMonitorInterface* para gráficos. La principal diferencia es la presencia de métodos virtuales para establecer y obtener el ID del gráfico y el nombre del objeto.

```
class ObjectProxy
{
public:
   long get(const ENUM_OBJECT_PROPERTY_INTEGER property, const int modifier = 0)
   {
      return ObjectGetInteger(chart(), name(), property, modifier);
   }
   double get(const ENUM_OBJECT_PROPERTY_DOUBLE property, const int modifier = 0)
   {
      return ObjectGetDouble(chart(), name(), property, modifier);
   }
   string get(const ENUM_OBJECT_PROPERTY_STRING property, const int modifier = 0)
   {
      return ObjectGetString(chart(), name(), property, modifier);
   }
   bool set(const ENUM_OBJECT_PROPERTY_INTEGER property, const long value,
      const int modifier = 0)
   {
      return ObjectSetInteger(chart(), name(), property, modifier, value);
   }
   bool set(const ENUM_OBJECT_PROPERTY_DOUBLE property, const double value,
      const int modifier = 0)
   {
      return ObjectSetDouble(chart(), name(), property, modifier, value);
   }
   bool set(const ENUM_OBJECT_PROPERTY_STRING property, const string value,
      const int modifier = 0)
   {
      return ObjectSetString(chart(), name(), property, modifier, value);
   }

   virtual string name() = 0;
   virtual void name(const string) { }
   virtual long chart() { return 0; }
   virtual void chart(const long) { }
};
```

Vamos a implementar estos métodos en la clase descendiente (más adelante complementaremos la jerarquía de clases con el monitor de propiedades del objeto, similar al monitor de propiedades del gráfico).

```
class ObjectSelector: public ObjectProxy
{
protected:
   long host; // chart ID
   string id; // chart ID
public:
   ObjectSelector(const string _id, const long _chart = 0): id(_id), host(_chart) { }

   virtual string name()
   {
      return id;
   }
   virtual void name(const string _id)
   {
      id = _id;
   }
   virtual void chart(const long _chart) override
   {
      host = _chart;
   }
};
```

Hemos separado la interfaz abstracta *ObjectProxy* y su implementación mínima en *ObjectSelector* porque más adelante podemos necesitar implementar un array de proxies para múltiples objetos del mismo tipo, por ejemplo. Entonces basta con almacenar un array de nombres o su prefijo común en la nueva clase «multiselector» y asegurarse de que uno de ellos es devuelto desde el método *name* llamando al operador sobrecargado []:*multiSelector[i].get(OBJPROP_XYZ)*.

Ahora volvamos al script *ObjectCleanup2.mq5* y describamos dos variables de entrada para especificar un color y un punto de anclaje como condiciones adicionales para seleccionar los objetos que se van a eliminar.

```
// ObjectCleanup2.mq5
...
input color CustomColor = clrRed;
input ENUM_ARROW_ANCHOR CustomAnchor = ANCHOR_TOP;
```

Pasemos estos valores a la función *CustomDeleteAllObjects*, y las nuevas comprobaciones de condiciones en el bucle sobre objetos podrán formularse de forma más compacta gracias a la clase mediadora.

```
#include <MQL5Book/ObjectMonitor.mqh>

void OnStart()
{
   const int n = UseCustomDeleteAll ?
      CustomDeleteAllObjects(0, ObjNamePrefix, CustomColor, CustomAnchor) :
      ObjectsDeleteAll(0, ObjNamePrefix);
   PrintFormat("%d objects deleted", n);
}

int CustomDeleteAllObjects(const long chart, const string prefix,
   color clr, ENUM_ARROW_ANCHOR anchor,
   const int window = -1, const int type = -1)
{
   int count = 0;
   for(int i = ObjectsTotal(chart, window, type) - 1; i >= 0; --i)
   {
      const string name = ObjectName(chart, i, window, type);

      ObjectSelector s(name);
      ResetLastError();
      if((StringLen(prefix) == 0 || StringFind(s.get(OBJPROP_NAME), prefix) == 0)
      && s.get(OBJPROP_COLOR) == CustomColor
      && s.get(OBJPROP_ANCHOR) == CustomAnchor
      && _LastError != 4203) // OBJECT_WRONG_PROPERTY
      {
         count += ObjectDelete(chart, name);
      }
   }
   return count;
}
```

Es importante señalar que especificamos el nombre del objeto (y el identificador implícito del gráfico 0 actual) sólo una vez al crear el objeto *ObjectSelector*. Además, todas las propiedades se solicitan mediante el método *get* con un único parámetro que describe la propiedad deseada, y el compilador elegirá automáticamente la función *ObjectGet* adecuada.

La comprobación adicional del código de error 4203 (OBJECT_WRONG_PROPERTY) permite filtrar los objetos que no tienen la propiedad solicitada, como OBJPROP_ANCHOR. De este modo, en concreto, es posible hacer una selección en la que caerán todos los tipos de flechas (sin necesidad de solicitar por separado distintos tipos de OBJ_ARROW_XYZ), pero las líneas y los «eventos» quedarán excluidos del procesamiento.

Esto es fácil de comprobar ejecutando primero el script *ObjectSimpleShowcase.mq5* en el gráfico (creará 14 objetos de diferentes tipos) y luego *ObjectCleanup2.mq5*. Si activa el modo *UseCustomDeleteAll*, habrá 5 objetos no borrados en el gráfico: OBJ_VLINE, OBJ_HLINE, OBJ_ARROW_BUY, OBJ_ARROW_SELL, and OBJ_EVENT. Los dos primeros y el último no tienen la propiedad OBJPROP_ANCHOR, y las flechas de compra y venta no pasan por color (se asume que el color de todos los demás objetos creados es rojo por defecto).

No obstante, *ObjectSelector* se proporciona no sólo por el bien de la sencilla aplicación anterior: es la base para crear un monitor de propiedades para un solo objeto, similar a lo que se implementó para los gráficos. Así, el archivo de encabezado *ObjectMonitor.mqh* contiene algo más interesante.

```
class ObjectMonitorInterface: public ObjectSelector
{
public:
   ObjectMonitorInterface(const string _id, const long _chart = 0):
      ObjectSelector(_id, _chart) { }
   virtual int snapshot() = 0;
   virtual void print() { };
   virtual int backup() { return 0; }
   virtual void restore() { }
   virtual void applyChanges(ObjectMonitorInterface *reference) { }
};
```

Este conjunto de métodos debería recordarle *ChartModeMonitorInterface* de *ChartModeMonitor.mqh*. La única innovación es el método *applyChanges*, que copia las propiedades de un objeto a otro.

Basado en *ObjectMonitorInterface*, aquí está la descripción de la implementación básica de un monitor de propiedades para un par de tipos de plantilla: un tipo de valor de propiedad (uno de *long*, *double* o *string*) y el tipo de enumeración (uno de ENUM_OBJECT_PROPERTY_ish).

```
template<typename T,typename E>
class ObjectMonitorBase: public ObjectMonitorInterface
{
protected:
   MapArray<E,T> data;  // array of pairs [property, value], current state
   MapArray<E,T> store; // backup (filled on demand)
   MapArray<E,T> change;// committed changes between two states
   ...
```

El constructor *ObjectMonitorBase* tiene dos parámetros: el nombre del objeto y un array de banderas con identificadores de las propiedades que deben observarse en el objeto especificado. Una parte importante de este código es casi idéntica a *ChartModeMonitor*. En concreto, como antes, se pasa un array de banderas al método de ayuda *detect*, cuyo propósito principal es identificar aquellas constantes de enteros que son elementos de la enumeración *E*, y eliminar el resto. El único añadido que hay que aclarar es la obtención de una propiedad con el número de niveles de un objeto a través de *ObjectGetInteger(0, id, OBJPROP_LEVELS)*. Esto es necesario para admitir la iteración de propiedades con múltiples valores debido a la presencia de niveles (por ejemplo, Fibonacci). Para objetos sin niveles, obtendremos la cantidad 0, y tal propiedad será la escalar habitual.

```
public:
   ObjectMonitorBase(const string _id, const int &flags[]): ObjectMonitorInterface(_i
   {
      const int levels = (int)ObjectGetInteger(0, id, OBJPROP_LEVELS);
      for(int i = 0; i < ArraySize(flags); ++i)
      {
         detect(flags[i], levels);
      }
   }
   ...
```

Por supuesto, el método *detect* es algo diferente de lo que vimos en *ChartModeMonitor*. Recordemos que, para empezar, contiene un fragmento con una comprobación de si la constante *v* pertenece a la enumeración *E*, mediante una llamada a la función *EnumToString*: si no existe tal elemento en la enumeración, se generará un código de error. Si el elemento existe, añadimos el valor de la propiedad correspondiente al array *data*.

```
// ChartModeMonitor.mqh
bool detect(const int v)
{
   ResetLastError();
   conststrings = EnumToString((E)v); // resulting string is not important
   if(_LastError == 0)                // analyze the error code
   {
      data.put((E)v, get((E)v));
      return true;
   }
   return false;
}
```

En el monitor de objetos, nos vemos obligados a complicar este esquema, ya que algunas propiedades son multivaloradas debido al parámetro *modifier* de las funciones *ObjectGet* y *ObjectSet*.

Así que introducimos un array estático *modifiables* con una lista de aquellas propiedades que los modificadores admiten (cada propiedad será discutida en detalle más adelante). La conclusión es que, para este tipo de propiedades multivaloradas, es necesario leerlas y almacenarlas en el array *data* no una vez, sino varias.

```
// ObjectMonitor.mqh
   bool detect(const int v, const int levels)
   {
      // the following properties support multiple values
      static const int modifiables[] =
      {
         OBJPROP_TIME,        // anchor point by time
         OBJPROP_PRICE,       // anchor point by price
         OBJPROP_LEVELVALUE,  // level value
         OBJPROP_LEVELTEXT,   // inscription on the level line
         // NB: the following properties do not generate errors when exceeded
         // actual number of levels or files
         OBJPROP_LEVELCOLOR,  // level line color
         OBJPROP_LEVELSTYLE,  // level line style
         OBJPROP_LEVELWIDTH,  // width of the level line
         OBJPROP_BMPFILE,     // image files
      };
      ...
```

Aquí también utilizamos el truco con *EnumToString* para comprobar la existencia de una propiedad con el identificador *v*. Si tiene éxito, comprobamos si está en la lista de *modifiables* y establecemos la correspondiente bandera *modifiable* en *true* o *false*.

```
      bool result = false;
      ResetLastError();
  conststrings =EnumToString((E)v); // resulting string is not important
  if(_LastError ==0)// analyze the error code
      {
         bool modifiable = false;
         for(int i = 0; i < ArraySize(modifiables); ++i)
         {
            if(v == modifiables[i])
            {
               modifiable = true;
               break;
            }
         }
         ...
```

Por defecto, cualquier propiedad se considera inequívoca y, por lo tanto, el número necesario de lecturas a través de la función *ObjectGet* o de entradas a través de la función *ObjectSet* es igual a 1 (la variable *k* a continuación).

```
      int k = 1;
      // for properties with modifiers, set the correct amount
      if(modifiable)
      {
         if(levels > 0) k = levels;
         else if(v == OBJPROP_TIME || v == OBJPROP_PRICE) k = MOD_MAX;
         else if(v == OBJPROP_BMPFILE) k = 2;
      }
```

Si un objeto admite niveles, limitamos el número potencial de lecturas/escrituras con el parámetro *levels* (como recordamos, se obtiene en el código de llamada a partir de la propiedad OBJPROP_LEVELS).

Para la propiedad OBJPROP_BMPFILE, como pronto aprenderemos, sólo se permiten dos estados: activado (botón pulsado, bandera activada) o desactivado (botón liberado, bandera desactivada), por lo que *k = 2*.

Por último, las coordenadas de objeto (OBJPROP_TIME y OBJPROP_PRICE) son convenientes porque generan un error cuando se intenta leer/escribir un punto de anclaje inexistente. Por lo tanto, asignamos a *k* algún valor obviamente grande de MOD_MAX, y entonces podemos interrumpir el ciclo de lectura de puntos en un valor distinto de cero *_LastError*.

```
      // read property value - one or many
      for(int i = 0; i < k; ++i)
      {
         ResetLastError();
         T temp = get((E)v, i);
         // if there is no i-th modifier, we will get an error and break the loop
         if(_LastError != 0) break;
         data.put((E)MOD_COMBINE(v, i), temp);
         result = true;
      }
   }
   return result;
}
```

Dado que una propiedad puede tener varios valores, que se leen en un bucle hasta *k*, ya no podemos escribir simplemente *data.put((E)v, get((E)v))*. Necesitamos combinar de alguna manera el identificador de la propiedad *v* y su número de modificación *i*. Afortunadamente, el número de propiedades también está limitado en una constante de entero (tipo *int*): no se ocupan más de dos bytes inferiores. Así que podemos utilizar operadores a nivel de bits para poner *i* al byte superior. Para ello se ha desarrollado la macro MOD_COMBINE.

```
#define MOD_COMBINE(V,I) (V | (I << 24))
```

Por supuesto, se proporcionan macros inversas para recuperar el ID de la propiedad y el número de revisión.

```
#define MOD_GET_NAME(V)  (V & 0xFFFFFF)
#define MOD_GET_INDEX(V) (V >> 24)
```

Por ejemplo, aquí podemos ver cómo se utilizan en el método *snapshot*.

```
virtual int snapshot() override
{
   MapArray<E,T> temp;
   change.reset();

   // collect all required properties in temp
   for(int i = 0; i < data.getSize(); ++i)
   {
      const E e = (E)MOD_GET_NAME(data.getKey(i));
      const int m = MOD_GET_INDEX(data.getKey(i));
      temp.put((E)data.getKey(i), get(e, m));
   }

   int changes = 0;
   // compare previous and new state
   for(int i = 0; i < data.getSize(); ++i)
   {
      if(data[i] != temp[i])
      {
         // save the differences in the change array
         if(changes == 0) Print(id);
         const E e = (E)MOD_GET_NAME(data.getKey(i));
         const int m = MOD_GET_INDEX(data.getKey(i));
         Print(EnumToString(e), (m > 0 ? (string)m : ""), " ", data[i], " -> ", te
         change.put(data.getKey(i), temp[i]);
         changes++;
      }
   }

   // save the new state as current
   data = temp;
   return changes;
}
```

Este método repite toda la lógica del método del mismo nombre en *ChartModeMonitor.mqh*; sin embargo, para leer propiedades en todas partes, primero debe extraer el nombre de la propiedad de la clave almacenada usando MOD_GET_NAME y el número usando MOD_GET_INDEX.

Una complicación similar tiene que hacerse en el método *restore*.

```
virtual void restore() override
{
   data = store;
   for(int i = 0; i < data.getSize(); ++i)
   {
      const E e = (E)MOD_GET_NAME(data.getKey(i));
      const int m = MOD_GET_INDEX(data.getKey(i));
      set(e, data[i], m);
   }
}
```

La innovación más interesante de *ObjectMonitorBase* es cómo funciona con los cambios.

```
   MapArray<E,T> * const getChanges()
   {
      return &change;
   }

   virtual void applyChanges(ObjectMonitorInterface *intf) override
   {
      ObjectMonitorBase *reference = dynamic_cast<ObjectMonitorBase<T,E> *>(intf);
      if(reference)
      {
         MapArray<E,T> *event = reference.getChanges();
         if(event.getSize() > 0)
         {
            Print("Modifing ", id, " by ", event.getSize(), " changes");
            for(int i = 0; i < event.getSize(); ++i)
            {
               data.put(event.getKey(i), event[i]);
               const E e = (E)MOD_GET_NAME(event.getKey(i));
               const int m = MOD_GET_INDEX(event.getKey(i));
               Print(EnumToString(e), " ", m, " ", event[i]);
               set(e, event[i], m);
            }
         }
      }
   }
```

Si pasamos al método *applyChanges* de estados del monitor de otro objeto, podemos adoptar todos los últimos cambios del mismo.

Para admitir propiedades de los tres tipos básicos (*long*,*double*,*string*) necesitamos implementar la clase *ObjectMonitor* (análoga a *ChartModeMonitor* de *ChartModeMonitor.mqh*).

```
class ObjectMonitor: public ObjectMonitorInterface
{
protected:
   AutoPtr<ObjectMonitorInterface> m[3];

   ObjectMonitorInterface *getBase(const int i)
   {
      return m[i][];
   }

public:
   ObjectMonitor(const string objid, const int &flags[]): ObjectMonitorInterface(obji
   {
      m[0] = new ObjectMonitorBase<long,ENUM_OBJECT_PROPERTY_INTEGER>(objid, flags);
      m[1] = new ObjectMonitorBase<double,ENUM_OBJECT_PROPERTY_DOUBLE>(objid, flags);
      m[2] = new ObjectMonitorBase<string,ENUM_OBJECT_PROPERTY_STRING>(objid, flags);
   }
   ...
```

Aquí también se conserva la estructura de código anterior, y sólo se han añadido métodos para admitir cambios y nombres (los gráficos, como recordamos, no tienen nombres).

```
...
virtual string name() override
{
   return m[0][].name();
}

virtual void name(const string objid) override
{
   m[0][].name(objid);
   m[1][].name(objid);
   m[2][].name(objid);
}

virtual void applyChanges(ObjectMonitorInterface *intf) override
{
   ObjectMonitor *monitor = dynamic_cast<ObjectMonitor *>(intf);
   if(monitor)
   {
      m[0][].applyChanges(monitor.getBase(0));
      m[1][].applyChanges(monitor.getBase(1));
      m[2][].applyChanges(monitor.getBase(2));
   }
}
```

Basándose en el monitor de objetos creado, es fácil implementar varios trucos que no se admiten en el terminal. En concreto, se trata de la creación de copias de objetos y la edición en grupo de objetos.

Script ObjectCopy

El script *ObjectCopy.mq5* demuestra cómo copiar objetos seleccionados. Al principio de su función *OnStart*, llenamos el array *flags* con enteros consecutivos que son candidatos a elementos de enumeraciones ENUM_OBJECT_PROPERTY_ de diferentes tipos. La numeración de los elementos de enumeración tiene una marcada agrupación por finalidad, y hay grandes espacios entre los grupos (aparentemente, un margen para futuros elementos), por lo que el array formado es bastante grande: 2048 elementos.

```
#include <MQL5Book/ObjectMonitor.mqh>

#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1) - 1] = V)

void OnStart()
{
   int flags[2048];
   // filling the array with consecutive integers, which will be
   // checked against the elements of enumerations of object properties,
   // invalid values will be discarded in the monitor's detect method
   for(int i = 0; i < ArraySize(flags); ++i)
   {
      flags[i] = i;
   }
   ...
```

A continuación, recopilamos en un array los nombres de los objetos que están seleccionados actualmente en el gráfico. Para ello, utilizamos la propiedad OBJPROP_SELECTED.

```
   string selected[];
   const int n = ObjectsTotal(0);
   for(int i = 0; i < n; ++i)
   {
      const string name = ObjectName(0, i);
      if(ObjectGetInteger(0, name, OBJPROP_SELECTED))
      {
         PUSH(selected, name);
      }
   }
   ...
```

Por último, en el bucle principal sobre los elementos seleccionados, leemos las propiedades de cada objeto, formamos el nombre de su copia y creamos un objeto bajo él con el mismo conjunto de atributos.

```
      for(int i = 0; i < ArraySize(selected); ++i)
      {
         const string name = selected[i];

        // make a backup of the properties of the current object using the monitor
         ObjectMonitor object(name, flags);
         object.print();
         object.backup();
         // form a correct, appropriate name for the copy
         const string copy = GetFreeName(name);

         if(StringLen(copy) > 0)
         {
            Print("Copy name: ", copy);
            // create an object of the same type OBJPROP_TYPE
            ObjectCreate(0, copy,
               (ENUM_OBJECT)ObjectGetInteger(0, name, OBJPROP_TYPE),
               ObjectFind(0, name), 0, 0);
            // change the name of the object in the monitor to a new one
            object.name(copy);
            // restore all properties from the backup to a new object
            object.restore();
         }
         else
         {
            Print("Can't create copy name for: ", name);
         }
      }
   }
```

Es importante señalar aquí que la propiedad OBJPROP_TYPE es una de las pocas propiedades de sólo lectura, y por lo tanto es vital crear un objeto del tipo requerido para empezar.

La función de ayuda *GetFreeName* intenta añadir la cadena «/Copy #x» al nombre del objeto, donde x es el número de copia. Así, ejecutando el script varias veces, puede crear la 2ª, 3ª y sucesivas copias.

```
string GetFreeName(const string name)
{
   const string suffix = "/Copy №";
   // check if there is a copy in the suffix name
   const int pos = StringFind(name, suffix);
   string prefix;
   int n;

   if(pos <= 0)
   {
      // if suffix is not found, assume copy number 1
      const string candidate = name + suffix + "1";
      // checking if the copy name is free, and if so, return it
      if(ObjectFind(0, candidate) < 0)
      {
         return candidate;
      }
      // otherwise, prepare for a loop with iteration of copy numbers
      prefix = name;
      n = 0;
   }
   else
   {
      // if the suffix is found, select the name without it
      prefix = StringSubstr(name, 0, pos);
      // and find the copy number in the string
      n = (int)StringToInteger(StringSubstr(name, pos + StringLen(suffix)));
   }

   Print("Found: ", prefix, " ", n);
   // loop trying to find a free copy number above n, but no more than 1000
   for(int i = n + 1; i < 1000; ++i)
   {
      const string candidate = prefix + suffix + (string)i;
      // check for the existence of an object with a name ending "Copy #i"
      if(ObjectFind(0, candidate) < 0)
      {
         return candidate; // return vacant copy name
      }
   }
   return NULL; // too many copies
}
```

El terminal recuerda la última configuración de un determinado tipo de objeto, y si se crean uno tras otro, esto equivale a copiar. No obstante, los ajustes suelen cambiar en el proceso de trabajo con diferentes gráficos, y si después de un tiempo hay necesidad de duplicar algún objeto «antiguo», entonces los ajustes para él, por regla general, tienen que hacerse por completo. Esto es especialmente costoso para los tipos de objetos con un gran número de propiedades, por ejemplo, las herramientas de Fibonacci. En tales casos, este script le resultará muy útil.

Algunas de las imágenes de este capítulo, que contienen objetos del mismo tipo, se han creado utilizando este script.

Indicador ObjectGroupEdit

El segundo ejemplo de utilización de *ObjectMonitor* es el indicador *ObjectGroupEdit.mq5*, que permite editar a la vez las propiedades de un grupo de objetos seleccionados.

Imaginemos que hemos seleccionado varios objetos en el gráfico (no necesariamente del mismo tipo), para los que es necesario cambiar uniformemente una u otra propiedad. A continuación, abrimos el cuadro de diálogo de propiedades de cualquiera de estos objetos, lo configuramos y, pulsando *OK*, estos cambios se aplican a todos los objetos seleccionados. Así es como funciona nuestro siguiente programa MQL.

Necesitábamos un indicador como tipo de programa porque implica eventos gráficos. Para este aspecto de la programación MQL5 dispondrá de un capítulo entero dedicado en exclusiva al respecto, pero ahora descubriremos algunos de los aspectos básicos.

Como el indicador no tiene gráficos, las directivas *#property* contienen ceros y la función *OnCalculate* está prácticamente vacía.

```
#property indicator_chart_window
#property indicator_buffers 0
#property indicator_plots   0

int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
   return rates_total;
}
```

Para generar automáticamente un conjunto completo de todas las propiedades de un objeto, utilizaremos de nuevo un array de 2048 elementos con valores enteros consecutivos. También proporcionaremos un array para los nombres de los elementos seleccionados y un array de objetos monitor de la clase *ObjectMonitor*.

```
int consts[2048];
string selected[];
ObjectMonitor *objects[];
```

En el manejador *OnInit*, inicializamos el array de números y ponemos en marcha el temporizador.

```
void OnInit()
{
   for(int i = 0; i < ArraySize(consts); ++i)
   {
      consts[i] = i;
   }

   EventSetTimer(1);
}
```

En el manejador del temporizador, guardamos los nombres de los objetos seleccionados en un array. Si la lista de selección ha cambiado, es necesario reconfigurar los objetos monitor, para lo cual se llama a la función auxiliar *TrackSelectedObjects*.

```
void OnTimer()
{
   string updates[];
   const int n = ObjectsTotal(0);
   for(int i = 0; i < n; ++i)
   {
      const string name = ObjectName(0, i);
      if(ObjectGetInteger(0, name, OBJPROP_SELECTED))
      {
         PUSH(updates, name);
      }
   }

   if(ArraySize(selected) != ArraySize(updates))
   {
      ArraySwap(selected, updates);
      Comment("Selected objects: ", ArraySize(selected));
      TrackSelectedObjects();
   }
}
```

La función *TrackSelectedObjects* en sí es bastante sencilla: borrar los monitores antiguos y crear otros nuevos. Si lo desea, puede hacerlo más inteligente manteniendo la parte no modificada de la selección.

```
void TrackSelectedObjects()
{
   for(int j = 0; j < ArraySize(objects); ++j)
   {
      delete objects[j];
   }

   ArrayResize(objects, 0);

   for(int i = 0; i < ArraySize(selected); ++i)
   {
      const string name = selected[i];
      PUSH(objects, new ObjectMonitor(name, consts));
   }
}
```

Recordemos que, cuando se crea un objeto monitor, se toma inmediatamente un «molde» de todas las propiedades del objeto gráfico correspondiente.

Ahora llegamos por fin a la parte en la que entran en juego los eventos. Como ya se mencionó en la visión general de las funciones de eventos, el manejador es responsable de los eventos *OnChartEvent* en el gráfico. En este ejemplo nos interesa un evento CHARTEVENT_OBJECT_CHANGE específico: ocurre cuando el usuario cambia cualquier atributo en el cuadro de diálogo de propiedades del objeto. El nombre del objeto modificado se pasa en el parámetro *sparam*.

Si este nombre coincide con uno de los objetos monitorizados, pedimos al monitor que haga una nueva instantánea de sus propiedades, es decir, llamamos a *objects[i].snapshot()*.

```
void OnChartEvent(const int id,
   const long &lparam, const double &dparam, const string &sparam)
{
   if(id == CHARTEVENT_OBJECT_CHANGE)
   {
      Print("Object changed: ", sparam);
      for(int i = 0; i < ArraySize(selected); ++i)
      {
         if(sparam == selected[i])
         {
            const int changes = objects[i].snapshot();
            if(changes > 0)
            {
               for(int j = 0; j < ArraySize(objects); ++j)
               {
                  if(j != i)
                  {
                     objects[j].applyChanges(objects[i]);
                  }
               }
            }
            ChartRedraw();
            break;
         }
      }
   }
}
```

Si los cambios se confirman (y es poco probable que no sea así), su número en la variable *changes* será mayor que 0. A continuación se inicia un bucle sobre todos los objetos seleccionados, y los cambios detectados se aplican a cada uno de ellos, excepto al original.

Dado que potencialmente podemos cambiar muchos objetos, llamamos a la petición de redibujado del gráfico con *ChartRedraw*.

En el manejador *OnDeinit*, eliminamos todos los monitores.

```
void OnDeinit(const int)
{
   for(int j = 0; j < ArraySize(objects); ++j)
   {
      delete objects[j];
   }
   Comment("");
}
```

Eso es todo: la nueva herramienta está lista.

Este indicador permite personalizar el aspecto general de varios grupos de objetos de etiqueta en la sección sobre Definición del punto de anclaje del objeto.

Por cierto: de acuerdo con un principio similar con la ayuda de *ObjectMonitor*, usted puede hacer otra herramienta popular que no está disponible en el terminal: para deshacer ediciones a las propiedades del objeto, dado que el método *restore* ya está listo.

5.8.8 Propiedades principales de los objetos

Todos los objetos tienen algunos atributos universales. Los principales se enumeran en el cuadro siguiente. Más adelante veremos otras propiedades generales de uso especial (véanse las secciones Gestión del estado de los objetos, Orden Z y Visibilidad de los objetos en el contexto de marcos temporales).

Identificador	Descripción	Tipo
OBJPROP_NAME	Nombre del objeto	string
OBJPROP_TYPE	Tipo de objeto (r/o)	ENUM_OBJECT
OBJPROP_CREATETIME	Hora de creación del objeto (r/o)	datetime
OBJPROP_TEXT	Descripción del objeto (texto contenido en el objeto)	string
OBJPROP_TOOLTIP	Texto de información sobre herramientas al pasar el ratón	string

La propiedad OBJPROP_NAME es un identificador de objeto. Editarla equivale a borrar el objeto antiguo y crear uno nuevo.

Para algunos tipos de objetos capaces de mostrar texto (como etiquetas o botones), la propiedad OBJPROP_TEXT siempre se muestra directamente en el gráfico, dentro del objeto. Para otros objetos (por ejemplo, líneas), esta propiedad contiene una descripción que se muestra en el gráfico junto al objeto y sólo si la opción «Mostrar descripciones de objetos» está activada en la configuración del gráfico. En cualquier caso, OBJPROP_TEXT se muestra en la información sobre herramientas.

La propiedad OBJPROP_CREATETIME sólo existe hasta el final de la sesión actual y no se escribe en los archivos chr.

Puede cambiar el nombre de un objeto mediante programación o manualmente (en el cuadro de diálogo de propiedades del objeto), mientras que su hora de creación seguirá siendo la misma. De cara al futuro, observamos que el cambio de nombre mediante programación no provoca ningún evento sobre los objetos del gráfico. Como vamos a descubrir en el capítulo siguiente, el cambio de nombre manual desencadena tres eventos:

- borrado de un objeto con el nombre antiguo (CHARTEVENT_OBJECT_DELETE),
- creación de un objeto con un nuevo nombre (CHARTEVENT_OBJECT_CREATE) y
- modificación de un nuevo objeto (CHARTEVENT_OBJECT_CHANGE).

Si la propiedad OBJPROP_TOOLTIP no está definida, se muestra información sobre herramientas para el objeto, generado automáticamente por el terminal. Para desactivar dicha información sobre herramientas, establezca su valor en «\n» (salto de línea).

Adaptemos el script *ObjectFinder.mq5* de la sección Encontrar objetos para registrar todas las propiedades anteriores de los objetos del gráfico actual. Vamos a ponerle al nuevo script el nombre *ObjectListing.mq5*.

Al principio de *OnStart*, crearemos o modificaremos una línea recta vertical situada en la última barra (en el momento en que se lanza el script). Si existe la opción de mostrar descripciones de objetos en la configuración del gráfico, entonces veremos el texto «Última barra en este momento» a lo largo de la línea vertical derecha.

```
void OnStart()
{
   const string vline = ObjNamePrefix + "current";
   ObjectCreate(0, vline, OBJ_VLINE, 0, iTime(NULL, 0, 0), 0);
   ObjectSetString(0, vline, OBJPROP_TEXT, "Latest Bar At The Moment");
   ...
```

A continuación, en un bucle a través de las subventanas, consultaremos todos los objetos hasta *ObjectsTotal* y sus principales propiedades.

```
   int count = 0;
   const long id = ChartID();
   const int win = (int)ChartGetInteger(id, CHART_WINDOWS_TOTAL);
   // loop through subwindows
   for(int k = 0; k < win; ++k)
   {
      PrintFormat("  Window %d", k);
      const int n = ObjectsTotal(id, k);
      //loop through objects
      for(int i = 0; i < n; ++i)
      {
         const string name = ObjectName(id, i, k);
         const ENUM_OBJECT type =
            (ENUM_OBJECT)ObjectGetInteger(id, name, OBJPROP_TYPE);
         const datetime created =
            (datetime)ObjectGetInteger(id, name, OBJPROP_CREATETIME);
         const string description = ObjectGetString(id, name, OBJPROP_TEXT);
         const string hint = ObjectGetString(id, name, OBJPROP_TOOLTIP);
         PrintFormat("    %s %s %s %s %s", EnumToString(type), name,
            TimeToString(created), description, hint);
         ++count;
      }
   }

   PrintFormat("%d objects found", count);
}
```

Obtenemos las siguientes entradas en el registro.

```
  Window 0
    OBJ_VLINE ObjShow-current 2021.12.21 20:20 Latest Bar At The Moment
    OBJ_VLINE abc 2021.12.21 19:25
    OBJ_VLINE xyz 1970.01.01 00:00
3 objects found
```

Un valor cero de OBJPROP_CREATETIME (1970.01.01 00:00) significa que el objeto no se creó durante la sesión actual, sino antes.

5.8.9 Coordenadas de tiempo y precio

Para los objetos de los tipos que existen en el sistema de coordenadas de las cotizaciones, la API de MQL5 admite un par de propiedades para especificar las vinculaciones de tiempo y precio. En caso de que un objeto tenga varios puntos de anclaje, las propiedades requieren la especificación de un parámetro modificador que contenga el índice del punto de anclaje al llamar a las funciones *ObjectSet* y *ObjectGet*.

Identificador	Descripción	Tipo de valor
OBJPROP_TIME	Coordenada de tiempo	datetime
OBJPROP_PRICE	Coordenada de precio	double

Estas propiedades están disponibles para absolutamente todos los objetos, pero no tiene sentido establecerlas o leerlas para objetos con coordenadas de pantalla.

Para demostrar cómo trabajar con coordenadas, analicemos el indicador sin búfer *ObjectHighLowChannel.mq5*, que dibuja dos líneas de tendencia para un determinado segmento de barras. Sus puntos de comienzo y fin en el eje temporal coinciden con la primera y la última barra del segmento, y a lo largo del eje de precios, los valores se calculan de forma diferente para cada una de las líneas: para la línea superior se utilizan los precios *High* más alto y más bajo, y para la línea inferior, los precios *Low* más alto y más bajo. A medida que el gráfico se actualice, nuestro improvisado canal debería moverse con los precios.

El rango de barras se establece mediante dos variables de entrada: el número de la barra inicial *BarOffset* y el número de barras *BarCount*. Por defecto, las líneas se trazan a los precios más recientes, ya que *bar offset = 0*.

```
input int BarOffset = 0;
input int BarCount = 10;

const string Prefix = "HighLowChannel-";
```

Los objetos tienen el prefijo de nombre común «HighLowChannel-».

En el manejador *OnCalculate* supervisamos la aparición de nuevas barras sobre la hora *iTime* de la barra 0-ésima. En cuanto se forma la barra, se analizan los precios en el segmento especificado, se toman los valores máximo y mínimo de los precios de cada uno de los dos tipos (MODE_HIGH, MODE_LOW) y se llama a la función auxiliar *DrawFigure* para ellos, y aquí es donde tiene lugar el trabajo con los objetos: la creación y modificación de coordenadas.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
   static datetime now = 0;
   if(now != iTime(NULL, 0, 0))
   {
      const int hh = iHighest(NULL, 0, MODE_HIGH, BarCount, BarOffset);
      const int lh = iLowest(NULL, 0, MODE_HIGH, BarCount, BarOffset);
      const int ll = iLowest(NULL, 0, MODE_LOW, BarCount, BarOffset);
      const int hl = iHighest(NULL, 0, MODE_LOW, BarCount, BarOffset);

      datetime t[2] = {iTime(NULL, 0, BarOffset + BarCount), iTime(NULL, 0, BarOffset
      double ph[2] = {iHigh(NULL, 0, fmax(hh, lh)), iHigh(NULL, 0, fmin(hh, lh))};
      double pl[2] = {iLow(NULL, 0, fmax(ll, hl)), iLow(NULL, 0, fmin(ll, hl))};

      DrawFigure(Prefix + "Highs", t, ph, clrBlue);
      DrawFigure(Prefix + "Lows", t, pl, clrRed);

      now = iTime(NULL, 0, 0);
   }
   return rates_total;
}
```

Y aquí está la función *DrawFigure* propiamente dicha:

```
bool DrawFigure(const string name, const datetime &t[], const double &p[],
   const color clr)
{
   if(ArraySize(t) != ArraySize(p)) return false;

   ObjectCreate(0, name, OBJ_TREND, 0, 0, 0);

   for(int i = 0; i < ArraySize(t); ++i)
   {
      ObjectSetInteger(0, name, OBJPROP_TIME, i, t[i]);
      ObjectSetDouble(0, name, OBJPROP_PRICE, i, p[i]);
   }

   ObjectSetInteger(0, name, OBJPROP_COLOR, clr);
   return true;
}
```

Después de la llamada a *ObjectCreate* que garantiza la existencia de un objeto, se llama a las funciones ObjectSet apropiadas para OBJPROP_TIME y OBJPROP_PRICE en todos los puntos de anclaje (dos en este caso).

En la imagen siguiente se muestra el resultado del indicador:

Canal en dos líneas de tendencia a precios máximos y mínimos

Puede ejecutar el indicador en el probador visual para ver cómo cambian las coordenadas de la línea sobre la marcha.

5.8.10 Anclar la esquina de la ventana y las coordenadas de la pantalla

Para los objetos que utilizan un sistema de coordenadas en forma de puntos (píxeles) en el gráfico, debe seleccionar una de las cuatro esquinas de la ventana, con respecto a la cual se contarán los valores a lo largo del eje horizontal X y del eje vertical Y hasta el punto de anclaje en el objeto. Estos aspectos se controlan mediante las propiedades de la tabla siguiente:

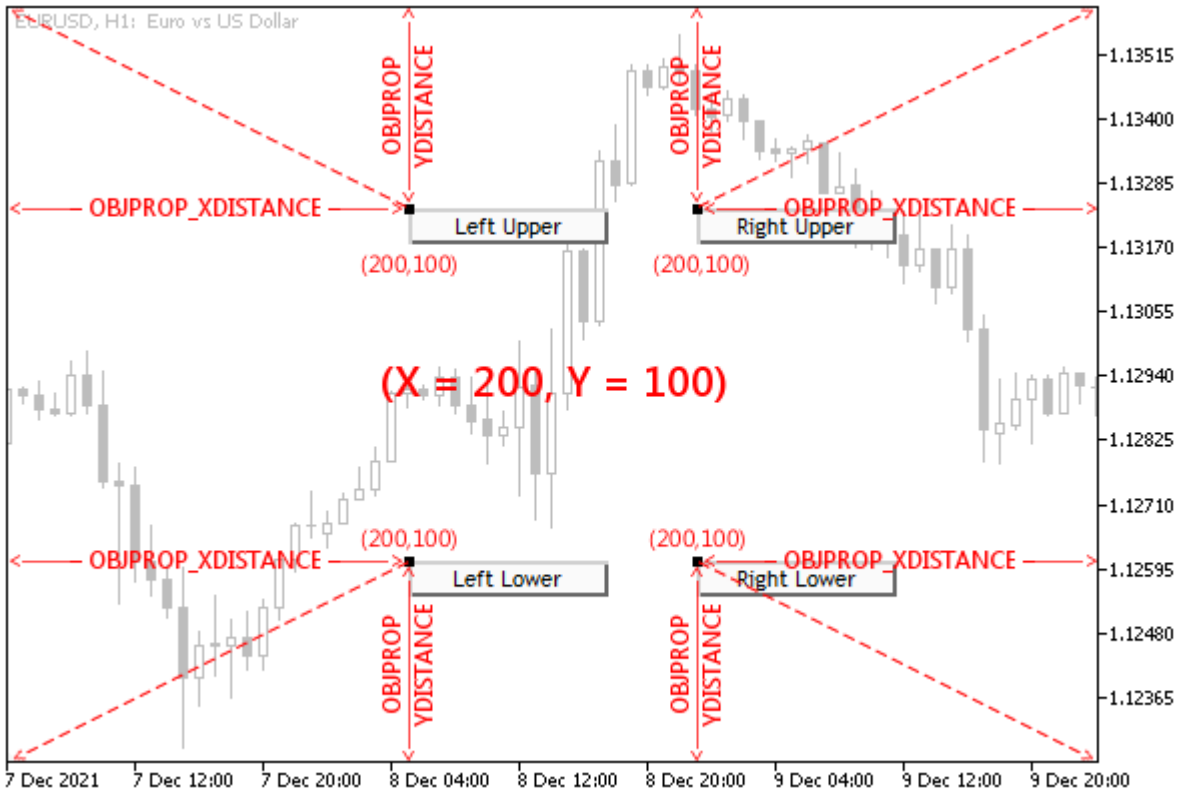
Identificador	Descripción	Tipo
OBJPROP_CORNER	Esquina del gráfico para anclar el objeto gráfico	ENUM_BASE_CORNER
OBJPROP_XDISTANCE	Distancia en píxeles a lo largo del eje X desde la esquina de anclaje	int
OBJPROP_YDISTANCE	Distancia en píxeles a lo largo del eje Y desde la esquina de anclaje	int

Las opciones válidas para OBJPROP_CORNER se resumen en la enumeración ENUM_BASE_CORNER.

Identificador	Ubicación del centro de coordenadas
CORNER_LEFT_UPPER	Esquina superior izquierda de la ventana
CORNER_LEFT_LOWER	Esquina inferior izquierda de la ventana
CORNER_RIGHT_LOWER	Esquina inferior derecha de la ventana
CORNER_RIGHT_UPPER	Esquina superior derecha de la ventana

El valor predeterminado es la esquina superior izquierda.

En la siguiente figura se muestran cuatro objetos Botón con el mismo tamaño y distancia desde la esquina de anclaje en la ventana. Cada uno de estos objetos sólo difiere en el ángulo de unión. Recuerde que los botones tienen un punto de anclaje que siempre está situado en la esquina superior izquierda del botón.

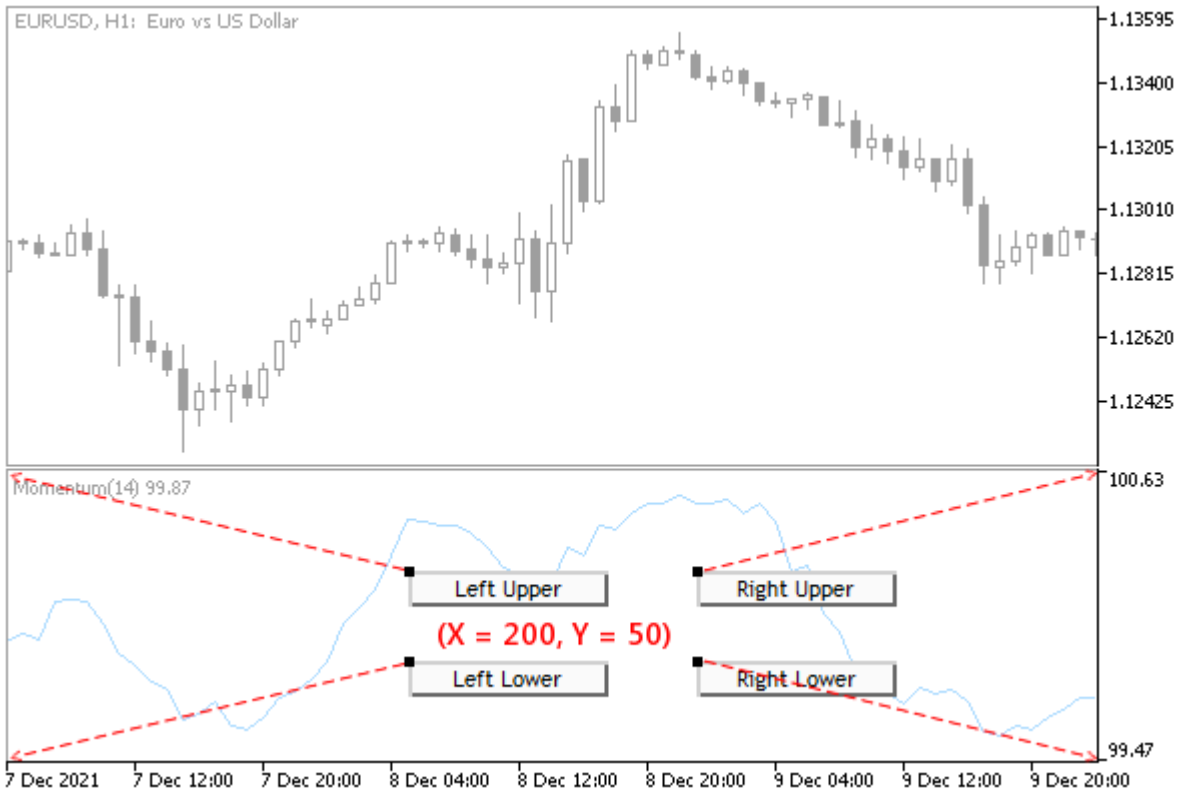


Disposición de objetos vinculados a distintas esquinas de la ventana principal

Los cuatro objetos están actualmente seleccionados en el gráfico, por lo que sus puntos de anclaje aparecen resaltados en un color contrastado.

Cuando hablamos de esquinas de ventana nos referimos a la ventana o subventana concreta en la que se encuentra el objeto y no a todo el gráfico. En otras palabras: en los objetos de las subventanas, la coordenada Y se mide desde el borde superior o inferior de esta subventana.

En la siguiente ilustración se muestran objetos similares en una subventana, ajustados a las esquinas de la subventana.



Ubicación de objetos con vinculación a diferentes esquinas de la subventana

Mediante el script *ObjectCornerLabel.mq5* el usuario puede probar el movimiento de una inscripción de texto, para lo cual el ángulo de anclaje en la ventana se especifica en el parámetro de entrada *Corner*.

```
#property script_show_inputs

input ENUM_BASE_CORNER Corner = CORNER_LEFT_UPPER;
```

Las coordenadas cambian periódicamente y aparecen en el texto de la propia inscripción. Así, la inscripción se desplaza en la ventana y, cuando llega al borde, rebota en él. El objeto se crea en la ventana o subventana en la que se soltó el script mediante el ratón.

```
void OnStart()
{
   const int t = ChartWindowOnDropped();
   const string legend = EnumToString(Corner);

   const string name = "ObjCornerLabel-" + legend;
   int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS, t);
   int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
   int x = w / 2;
   int y = h / 2;
   ...
```

Para un posicionamiento correcto, hallamos las dimensiones de la ventana (y a continuación comprobamos si han cambiado) y buscamos el punto medio para la colocación inicial del objeto: las variables con las coordenadas *x* y *y*.

A continuación, creamos y configuramos una inscripción, sin coordenadas todavía. Es importante tener en cuenta que habilitamos la posibilidad de seleccionar un objeto (OBJPROP_SELECTABLE) y

seleccionarlo (OBJPROP_SELECTED), ya que esto nos permite ver el punto de anclaje en el propio objeto, con el que se mide la distancia desde la esquina de la ventana (centro de coordenadas). Estas dos propiedades se describen con más detalle en la sección sobre Gestión del estado de los objetos.

```
ObjectCreate(0, name, OBJ_LABEL, t, 0, 0);
ObjectSetInteger(0, name, OBJPROP_SELECTABLE, true);
ObjectSetInteger(0, name, OBJPROP_SELECTED, true);
ObjectSetInteger(0, name, OBJPROP_CORNER, Corner);
...
```

En las variables *px* y *py,* registraremos los incrementos de coordenadas para la emulación del movimiento. La propia modificación de coordenadas se realizará en un bucle infinito hasta que sea interrumpida por el usuario. El contador de iteraciones permitirá periódicamente, cada 50 iteraciones, cambiar la dirección del movimiento de forma aleatoria.

```
int px = 0, py = 0;
int pass = 0;

for( ;!IsStopped(); ++pass)
{
   if(pass % 50 == 0)
   {
      h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS, t);
      w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
      px = rand() * (w / 20) / 32768 - (w / 40);
      py = rand() * (h / 20) / 32768 - (h / 40);
   }

   // bounce off window borders so object doesn't hide
   if(x + px > w || x + px < 0) px = -px;
   if(y + py > h || y + py < 0) py = -py;
   // recalculate label positions
   x += px;
   y += py;

   // update the coordinates of the object and add them to the text
   ObjectSetString(0, name, OBJPROP_TEXT, legend
      + "[" + (string)x + "," + (string)y + "]");
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);

   ChartRedraw();
   Sleep(100);
}

ObjectDelete(0, name);
}
```

Intente ejecutar el script varias veces, especificando diferentes esquinas de anclaje.

En la siguiente sección aumentaremos este script controlando también el punto de anclaje en el objeto.

5.8.11 Definir el punto de anclaje en un objeto

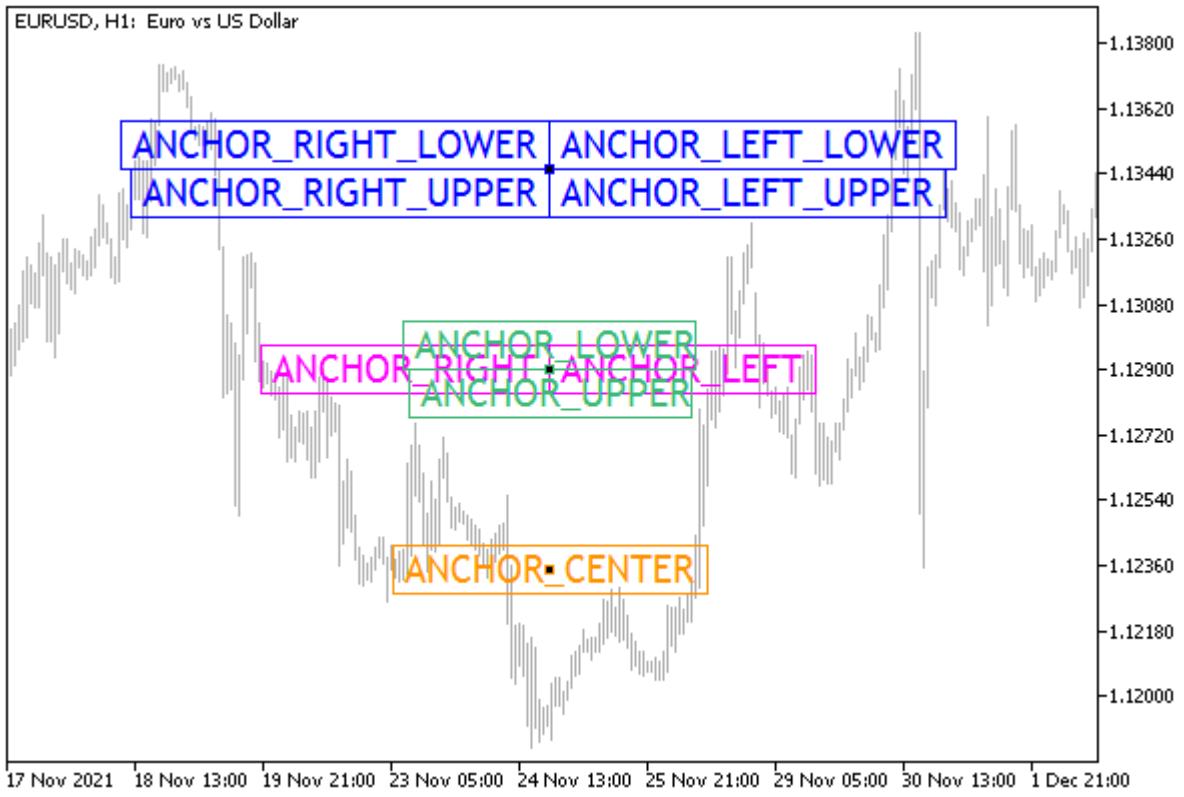
Algunos tipos de objetos permiten seleccionar un punto de anclaje. Los tipos que entran en esta categoría incluyen la etiqueta de texto (OBJ_TEXT) y la imagen de mapa de bits (OBJ_BITMAP) vinculadas a cotizaciones, así como la leyenda (OBJ_LABEL) y el panel con imagen (OBJ_BITMAP_LABEL), posicionados en coordenadas de pantalla.

Para leer y establecer el punto de anclaje, utilice las funciones *ObjectGetInteger* y *ObjectSetInteger* con la propiedad OBJPROP_ANCHOR.

Todas las opciones de selección de puntos se recogen en la enumeración ENUM_ANCHOR_POINT.

Identificador	Localización del punto de anclaje
ANCHOR_LEFT_UPPER	En la esquina superior izquierda
ANCHOR_LEFT	Centro izquierda
ANCHOR_LEFT_LOWER	En la esquina inferior izquierda
ANCHOR_LOWER	Centro inferior
ANCHOR_RIGHT_LOWER	En la esquina inferior derecha
ANCHOR_RIGHT	Centro derecha
ANCHOR_RIGHT_UPPER	En la esquina superior derecha
ANCHOR_UPPER	Centro superior
ANCHOR_CENTER	Estrictamente en el centro del objeto

Los puntos se muestran claramente en la imagen siguiente, donde se aplican varios objetos de etiqueta al gráfico.



Objetos de texto OBJ_LABEL con diferentes puntos de anclaje

El grupo superior de cuatro etiquetas tiene el mismo par de coordenadas (X,Y); sin embargo, debido al anclaje en diferentes esquinas del objeto, están situadas en diferentes lados del punto. Una situación similar se da en el segundo grupo de cuatro etiquetas de texto; sin embargo, allí el anclaje se realiza en los puntos medios de diferentes lados de los objetos. Por último, la leyenda se muestra por separado en la parte inferior, anclada en su centro, de modo que el punto quede dentro del objeto.

El botón (OBJ_BUTTON), el panel rectangular (OBJ_RECTANGLE_LABEL), el campo de entrada (OBJ_EDIT) y el objeto gráfico (OBJ_CHART) tienen un punto de anclaje fijo en la esquina superior izquierda (ANCHOR_LEFT_UPPER).

Algunos objetos gráficos del grupo de marcas de precio simples (OBJ_ARROW, OBJ_ARROW_THUMB_UP, OBJ_ARROW_THUMB_DOWN, OBJ_ARROW_UP, OBJ_ARROW_DOWN, OBJ_ARROW_STOP, OBJ_ARROW_CHECK) tienen dos formas de anclar sus coordenadas, especificadas por identificadores de otra enumeración ENUM_ARROW_ANCH OR.

Identificador	Localización del punto de anclaje
ANCHOR_TOP	Centro superior
ANCHOR_BOTTOM	Centro inferior

El resto de los objetos de este grupo tienen puntos de anclaje predefinidos: las flechas de compra (OBJ_ARROW_BUY) y venta (OBJ_ARROW_SELL) están en el centro de los lados superior e inferior, respectivamente, y las etiquetas de precio (OBJ_ARROW_PRICE_RIGHT, OBJ_ARROW_LEFT_PRICE) están a izquierda y derecha.

De forma similar al script *ObjectCornerLabel.mq5* de la sección anterior, vamos a crear el script *ObjectAnchorLabel.mq5*. En la nueva versión, además de mover la inscripción, cambiaremos aleatoriamente el punto de anclaje de la misma.

La esquina de la ventana para el anclaje será seleccionada, como antes, por el usuario al lanzar el script.

```
input ENUM_BASE_CORNER Corner = CORNER_LEFT_UPPER;
```

Mostraremos el nombre del ángulo en el gráfico como comentario.

```
void OnStart()
{
   Comment(EnumToString(Corner));
   ...
```

En un bucle infinito se genera uno de los 9 posibles valores del punto de anclaje en los momentos seleccionados.

```
   ENUM_ANCHOR_POINT anchor = 0;
   for( ;!IsStopped(); ++pass)
   {
      if(pass % 50 == 0)
      {
        ...
         anchor = (ENUM_ANCHOR_POINT)(rand() * 9 / 32768);
         ObjectSetInteger(0, name, OBJPROP_ANCHOR, anchor);
      }
      ...
```

El nombre del punto de anclaje se convierte en el contenido del texto de la etiqueta, junto con las coordenadas actuales.

```
      ObjectSetString(0, name, OBJPROP_TEXT, EnumToString(anchor)
         + "[" + (string)x + "," + (string)y + "]");
```

El resto de los fragmentos de código se ha mantenido prácticamente sin cambios.

Tras compilar y ejecutar el script, observe cómo la inscripción cambia su posición respecto a las coordenadas actuales (x, y) en función del punto de anclaje seleccionado.

Por ahora, controlamos e impedimos que el propio punto de anclaje salga de la ventana. No obstante, el objeto tiene algunas dimensiones, por lo que puede resultar que la mayor parte de la inscripción esté cortada. En el futuro, tras estudiar las propiedades pertinentes, nos ocuparemos de este problema (véase el ejemplo *ObjectSizeLabel.mq5* en la sección sobre Determinar ancho y alto del objeto).

5.8.12 Gestión del estado de los objetos

Entre las propiedades generales de los objetos, hay varias que controlan el estado de los mismos. Todas estas propiedades son de tipo booleano, lo que significa que pueden activarse (*true*) o desactivarse (*false*) y, por tanto, requieren el uso de las funciones *ObjectGetInteger* y *ObjectSetInteger*.

Identificador	Descripción
OBJPROP_HIDDEN	Desactivar la visualización del nombre de un objeto gráfico en la lista de objetos del cuadro de diálogo correspondiente (al que se accede desde el menú contextual del gráfico o pulsando Ctrl+B).
OBJPROP_SELECTED	Selección de objetos
OBJPROP_SELECTABLE	Disponibilidad de un objeto para su selección

Un valor de *true* para OBJPROP_HIDDEN permite ocultar un objeto innecesario de la lista del usuario. Por defecto, *true* está configurado para los objetos que muestran eventos de calendario, el historial de operaciones, así como para los objetos creados a partir de programas MQL. Para ver estos objetos gráficos y acceder a sus propiedades, pulse el botón *All* en el cuadro de diálogo *Object List*.

Un objeto oculto en la lista permanece visible en el gráfico. Para ocultar un objeto del gráfico sin borrarlo, puede utilizar la opción Visibilidad de los objetos en el contexto de marcos temporales.

El usuario no puede seleccionar y cambiar las propiedades de los objetos para los que OBJPROP_SELECTABLE es igual a *false*. Los objetos creados mediante programación no pueden seleccionarse por defecto. Como vimos en los scripts *ObjectCornerLabel.mq5* y *ObjectAnchorLabel.mq5* en las secciones anteriores, era necesario establecer explícitamente OBJPROP_SELECTABLE en *true* para desbloquear la capacidad de incluir también OBJPROP_SELECTED. Así resaltamos los puntos de anclaje en el objeto.

Normalmente, los programas MQL permiten la selección de sus objetos sólo si estos objetos sirven como controles. Por ejemplo, una línea de tendencia con un nombre predefinido, que el usuario mueve a voluntad, puede significar una condición para enviar una orden de operación cuando el precio la cruce.

5.8.13 Prioridad de los objetos (orden Z)

Los objetos en el gráfico proporcionan no sólo la presentación de la información, sino también la interacción con el usuario y los programas MQL a través de eventos, que se abordarán en detalle en el capítulo siguiente. Una de las fuentes de eventos es el puntero del ratón. El gráfico es capaz, en concreto, de seguir el movimiento del ratón y la pulsación de sus botones.

Si un objeto se encuentra bajo el ratón se puede realizar una gestión de eventos específica para él. Sin embargo, los objetos pueden solaparse entre sí (cuando sus coordenadas se solapan, teniendo en cuenta los tamaños). En este caso entra en juego la propiedad de tipo entero OBJPROP_ZORDER. Establece la prioridad del objeto gráfico para recibir eventos de ratón. Cuando los objetos se solapan, sólo uno de ellos, cuya prioridad es superior a la del resto, recibirá el evento.

De manera predeterminada, cuando se crea un objeto, su orden Z es cero, pero puede aumentarlo si es necesario.

> Es importante tener en cuenta que el orden Z sólo afecta al manejo de los eventos del ratón, no al dibujo de los objetos. Los objetos se dibujan siempre en el orden en que se añadieron al gráfico. Esto puede ser fuente de malentendidos. Por ejemplo, puede que no se muestre un información sobre herramientas para un objeto que está visualmente encima de otro porque el objeto superpuesto tiene una prioridad Z más alta (ver ejemplo).

En el script *ObjectZorder.mq5* crearemos 12 objetos de tipo OBJ_RECTANGLE_LABEL, colocándolos en un círculo, como en la esfera de un reloj. El orden de adición de los objetos corresponde a las horas: de 1 a 12. Para mayor claridad, todos los rectángulos tendrán un color aleatorio (para la propiedad OBJPROP_BGCOLOR, véase la sección siguiente), así como la prioridad aleatoria. Al pasar el ratón por encima de los objetos, el usuario podrá determinar a qué objeto pertenece mediante un «tooltip», o información sobre herramientas.

Para facilitar la configuración de las propiedades de los objetos, definimos la clase especial *ObjectBuilder*, derivada de *Object Selector*.

```
#include "ObjectPrefix.mqh"
#include <MQL5Book/ObjectMonitor.mqh>

class ObjectBuilder: public ObjectSelector
{
protected:
   const ENUM_OBJECT type;
   const int window;
public:
   ObjectBuilder(const string _id, const ENUM_OBJECT _type,
      const long _chart = 0, const int _win = 0):
      ObjectSelector(_id, _chart), type(_type), window(_win)
   {
      ObjectCreate(host, id, type, window, 0, 0);
   }

   // changing the name and chart is prohibited
   virtual void name(const string _id) override = delete;
   virtual void chart(const long _chart) override = delete;
};
```

Los campos con identificadores del objeto (*id*) y del gráfico (*host*) ya están en la clase *ObjectSelector*. En la derivada, añadimos un tipo de objeto (*ENUM_OBJECT type*) y un número de ventana (*int window*). El constructor llama a *ObjectCreate*.

La configuración y lectura de propiedades se hereda completamente como un grupo de métodos *get* y *set* de *ObjectSelector*.

Como en los scripts de prueba anteriores, determinamos la ventana donde se suelta el script, las dimensiones de la ventana y las coordenadas del centro.

```
void OnStart()
{
   const int t = ChartWindowOnDropped();
   int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS, t);
   int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
   int x = w / 2;
   int y = h / 2;
   ...
```

Dado que el tipo de objeto OBJ_RECTANGLE_LABEL admite dimensiones de píxel explícitas, calculamos la anchura de *dx* y la altura de *dy* de cada rectángulo como un cuarto de ventana. Las utilizamos para

establecer las propiedades OBJPROP_XSIZE y OBJPROP_YSIZE que se abordan en la sección sobre Determinar ancho y alto del objeto.

```
const int dx = w / 4;
const int dy = h / 4;
...
```

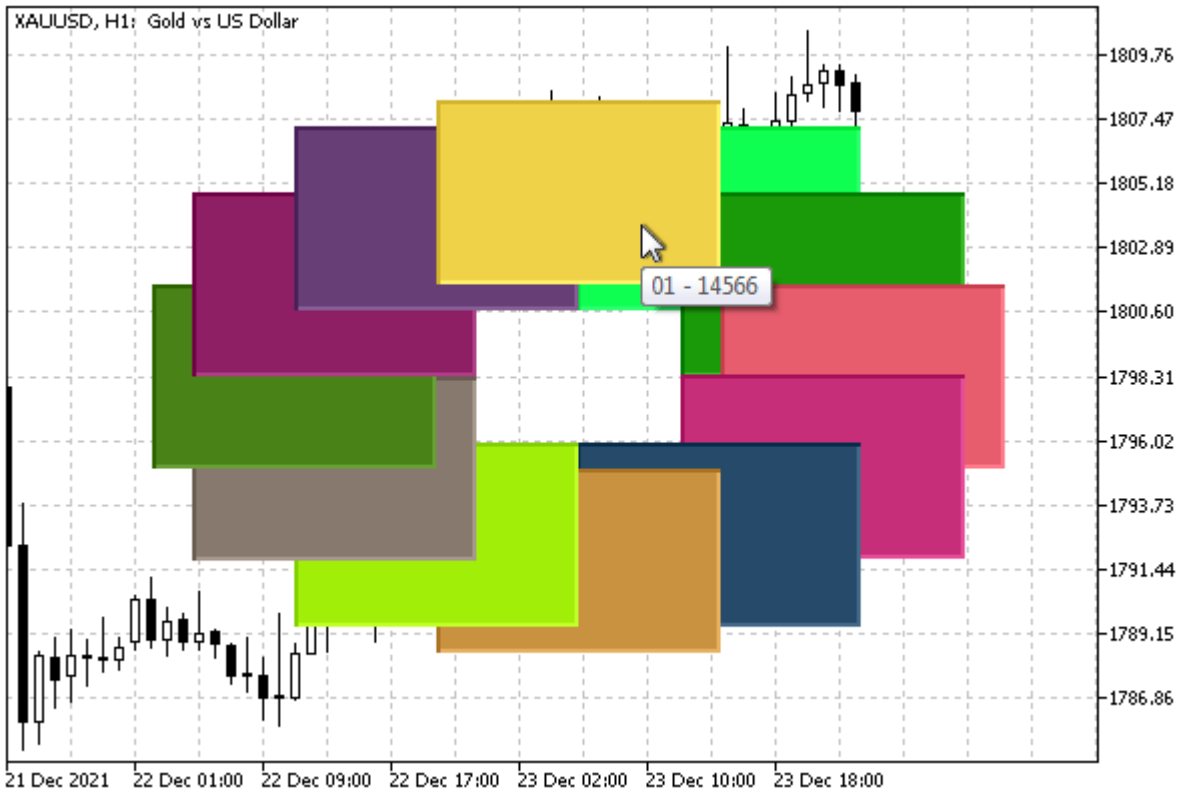
A continuación, en el bucle, creamos 12 objetos. Las variables *px* y *py* contienen el desplazamiento de la siguiente «marca» en el «dial» con respecto al centro (x, y). La prioridad de *z* se elige aleatoriamente. El nombre del objeto y su información sobre herramientas (OBJPROP_TOOLTIP) incluyen una cadena como «XX - YYY»; XX es el número de la «hora» (la posición en el dial es de 1 a 12) e YYY es la prioridad.

```
for(int i = 0; i < 12; ++i)
{
   const int px = (int)(MathSin((i + 1) * 30 * M_PI / 180) * dx) - dx / 2;
   const int py = -(int)(MathCos((i + 1) * 30 * M_PI / 180) * dy) - dy / 2;

   const int z = rand();
   const string text = StringFormat("%02d - %d", i + 1, z);

   ObjectBuilder *builder =
      new ObjectBuilder(ObjNamePrefix + text, OBJ_RECTANGLE_LABEL);
   builder.set(OBJPROP_XDISTANCE, x + px).set(OBJPROP_YDISTANCE, y + py)
   .set(OBJPROP_XSIZE, dx).set(OBJPROP_YSIZE, dy)
   .set(OBJPROP_TOOLTIP, text)
   .set(OBJPROP_ZORDER, z)
   .set(OBJPROP_BGCOLOR, (rand() << 8) | rand());
   delete builder;
}
```

Después de llamar al constructor *ObjectBuilder*, para el nuevo objeto *builder* se encadenan las llamadas al método sobrecargado *set* para diferentes propiedades (el método *set* devuelve un puntero al objeto en sí).

Dado que el objeto MQL ya no es necesario tras la creación y configuración del objeto gráfico, borramos inmediatamente *builder*.

Como resultado de la ejecución del script, aparecerán aproximadamente los siguientes objetos en el gráfico:



Información sobre herramientas de superposición de objetos y prioridad de orden Z

Los colores y las prioridades serán diferentes cada vez que lo ejecute, pero la superposición visual de los rectángulos será siempre la misma, en el orden de creación desde el 1 en la parte inferior hasta el 12 en la parte superior (aquí nos referimos a la superposición de objetos, no al hecho de que el 12 esté situado en la parte superior de la esfera del reloj).

En la imagen, el cursor del ratón se sitúa en un lugar en el que existen dos objetos, es decir, 01 (verde lima fluorescente) y 12 (arena). En este caso, la información sobre herramientas del objeto 01 está visible, aunque visualmente el objeto 12 se muestra encima del objeto 01. Esto se debe a que 01 se generó aleatoriamente con una prioridad mayor que 12.

Sólo se muestra una información sobre herramientas en cada momento, por lo que puede comprobar la relación de prioridad moviendo el cursor del ratón a otras zonas en las que no haya superposición de objetos y la información de la información sobre herramientas pertenezca al único objeto situado bajo el cursor.

Cuando hablemos del manejo de eventos del ratón en el próximo capítulo, podremos mejorar este ejemplo y comprobar el efecto del orden Z en los clics del ratón sobre los objetos.

Para eliminar los objetos creados, puede utilizar el script *ObjectCleanup1.mq5*.

5.8.14 Ajustes de visualización de objetos: color, estilo y marco

La apariencia de los objetos puede modificarse mediante diversas propiedades, que exploraremos en esta sección, empezando por el color, el estilo, el ancho de línea y los bordes. Otros aspectos del formato, como el tipo de letra, la inclinación y la alineación del texto, se abordarán en las secciones siguientes.

Todas las propiedades de la siguiente tabla tienen tipos compatibles con enteros y, por tanto, son gestionadas por las funciones *ObjectGetInteger* y *ObjectSetInteger*.

Identificador	Descripción	Tipo de propiedad
OBJPROP_COLOR	El color de la línea y del elemento principal del objeto (por ejemplo, fuente o relleno)	color
OBJPROP_STYLE	Estilo de línea	ENUM_LINE_STYLE
OBJPROP_WIDTH	Grosor de la línea en píxeles	int
OBJPROP_FILL	Rellenar un objeto con color (para OBJ_RECTANGLE, OBJ_TRIANGLE, OBJ_ELLIPSE, OBJ_CHANNEL, OBJ_STDDEVCHANNEL, OBJ_REGRESSION)	bool
OBJPROP_BACK	Objeto en segundo plano	bool
OBJPROP_BGCOLOR	Color de fondo para OBJ_EDIT, OBJ_BUTTON, OBJ_RECTANGLE_LABEL	color
OBJPROP_BORDER_TYPE	Tipo de marco para panel rectangular OBJ_RECTANGLE_LABEL	ENUM_BORDER_TYPE
OBJPROP_BORDER_COLOR	Color del marco para el campo de entrada OBJ_EDIT y el botón OBJ_BUTTON	color

A diferencia de la mayoría de los objetos con líneas (verticales y horizontales separadas, de tendencia, cíclicas, canales, etc.), donde la propiedad OBJPROP_COLOR define el color de la línea, para las imágenes OBJ_BITMAP_LABEL y OBJ_BITMAP define el color del marco, y OBJPROP_STYLE define el tipo de dibujo del marco.

Ya hemos conocido la enumeración ENUM_LINE_STYLE, utilizada para OBJPROP_STYLE, en el capítulo sobre indicadores, en la sección sobre Configuración de trazado.

Es necesario distinguir el relleno realizado por el color de primer plano OBJPROP_COLOR del color de fondo OBJPROP_BGCOLOR. Ambos son compatibles con diferentes grupos de tipos de objetos, que se enumeran en la tabla.

La propiedad OBJPROP_BACK requiere una explicación aparte. El hecho es que los objetos y los indicadores se muestran por defecto en la parte superior del gráfico de precios. El usuario puede cambiar este comportamiento para todo el gráfico yendo al cuadro de diálogo *Setting* del gráfico, y más allá del marcador *Shared*, la opción *Chart on top*. Este indicador también tiene un equivalente en el software, la propiedad CHART_FOREGROUND (véase Modos de visualización de gráficos). Sin embargo, a veces es conveniente no eliminar todos los objetos, sino sólo los seleccionados, en el fondo. A

continuación, para ellos, puede establecer OBJPROP_BACK en *true*. En este caso, el objeto se solapará incluso con los separadores de cuadrícula y de punto, si están activados en el gráfico.

Cuando el modo de relleno OBJPROP_FILL está activado, el color de las barras que caen dentro de la forma depende de la propiedad OBJPROP_BACK. Por defecto, con OBJPROP_BACK igual a *false*, las barras que se superponen al objeto se dibujan en color invertido con respecto a OBJPROP_COLOR (el color invertido se obtiene cambiando todos los bits del valor de color por los opuestos, por ejemplo, se obtiene 0x00FF7F para 0xFF0080). Con OBJPROP_BACK igual a *true*, las barras se dibujan de la forma habitual, ya que el objeto se muestra en segundo plano, «debajo» del gráfico (véase un ejemplo más abajo).

La enumeración ENUM_BORDER_TYPE contiene los siguientes elementos:

Identificador	**Apariencia**
BORDER_FLAT	Plano
BORDER_RAISED	Convexo
BORDER_SUNKEN	Cóncavo

Cuando el borde es plano (BORDER_FLAT), se dibuja como una línea con color, estilo y anchura según las propiedades OBJPROP_COLOR, OBJPROP_STYLE, OBJPROP_WIDTH. Las versiones convexa y cóncava imitan chaflanes de volumen alrededor del perímetro en tonos de OBJPROP_BGCOLOR.

Cuando no se establece el color del borde OBJPROP_BORDER_COLOR (por defecto, que corresponde a *clrNone*), el campo de entrada queda enmarcado por una línea del color principal OBJPROP_COLOR, y alrededor del botón se dibuja un marco tridimensional con chaflanes en los tonos de OBJPROP_BGCOLOR.

Para probar las nuevas propiedades, vea el script *ObjectStyle.mq5*. En él, crearemos 5 rectángulos del tipo OBJ_RECTANGLE, es decir, con referencia al tiempo y a los precios. Estarán espaciados uniformemente por todo el ancho de la ventana, resaltando el rango entre el precio máximo *High* y el precio mínimo *Low* en cada uno de los cinco periodos de tiempo. Para todos los objetos ajustaremos y cambiaremos periódicamente el color, el estilo y el grosor de las líneas, así como la opción de relleno y visualización detrás del gráfico.

Vamos a utilizar de nuevo la clase auxiliar *ObjectBuilder*, derivada de *Object Selector*. A diferencia de lo que vimos en la sección anterior, añadimos a *ObjectBuilder* un destructor en el que llamaremos a *ObjectDelete*.

```
#include <MQL5Book/ObjectMonitor.mqh>
#include <MQL5Book/AutoPtr.mqh>

class ObjectBuilder: public ObjectSelector
{
...
public:
   ~ObjectBuilder()
   {
      ObjectDelete(host, id);
   }
   ...
};
```

Esto permitirá asignar a esta clase no sólo la configuración de objetos, sino también su eliminación automática al finalizar el script.

En la función *OnStart* averiguamos el número de barras visibles y el índice de la primera barra, y también calculamos la anchura de un rectángulo en barras.

```
#define OBJECT_NUMBER 5

void OnStart()
{
   const string name = "ObjStyle-";
   const int bars = (int)ChartGetInteger(0, CHART_VISIBLE_BARS);
   const int first = (int)ChartGetInteger(0, CHART_FIRST_VISIBLE_BAR);
   const int rectsize = bars / OBJECT_NUMBER;
   ...
```

Reservemos un array de punteros inteligentes para los objetos con el fin de garantizar la llamada de los destructores *ObjectBuilder*.

```
   AutoPtr<ObjectBuilder> objects[OBJECT_NUMBER];
```

Defina una paleta de colores y cree 5 objetos de rectángulo.

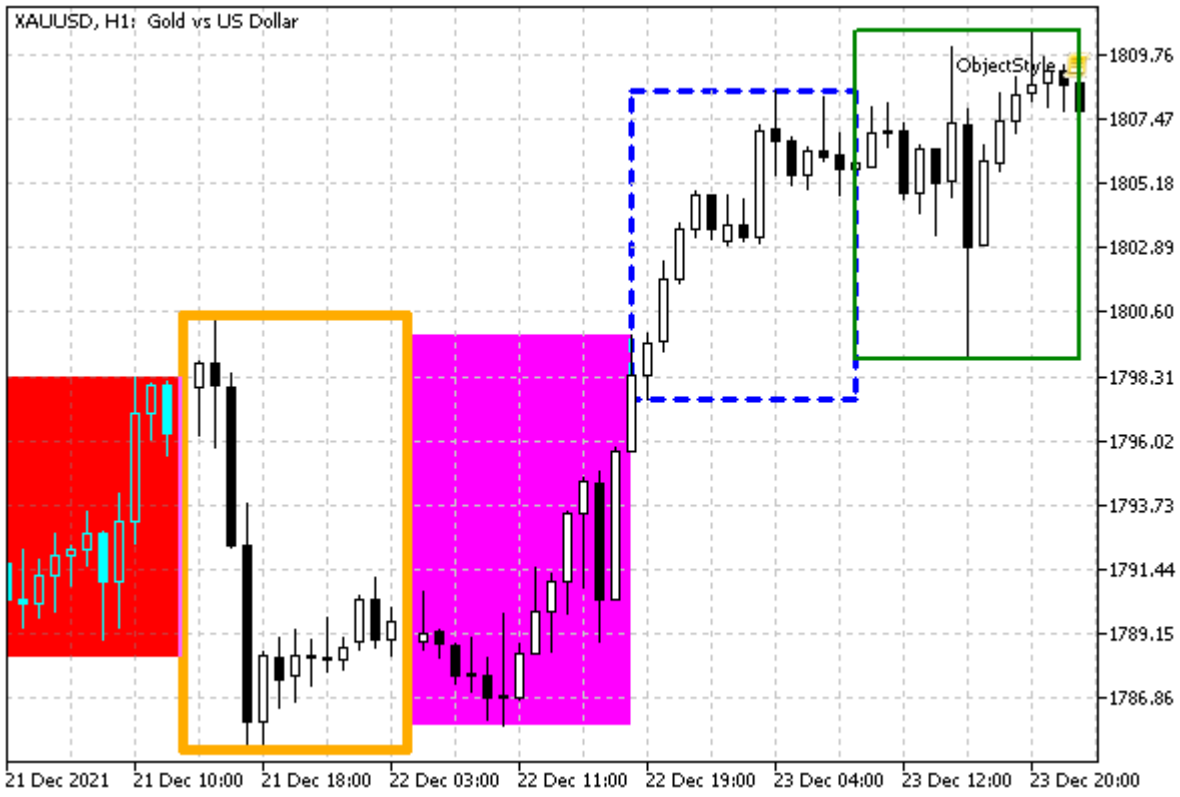
```
color colors[OBJECT_NUMBER] = {clrRed, clrGreen, clrBlue, clrMagenta, clrOrange};

for(int i = 0; i < OBJECT_NUMBER; ++i)
{
   // find the indexes of the bars that determine the range of prices in the i-th
   const int h = iHighest(NULL, 0, MODE_HIGH, rectsize, i * rectsize);
   const int l = iLowest(NULL, 0, MODE_LOW, rectsize, i * rectsize);
   // create and set up an object in the i-th subrange
   ObjectBuilder *object = new ObjectBuilder(name + (string)(i + 1), OBJ_RECTANGLE
   object.set(OBJPROP_TIME, iTime(NULL, 0, i * rectsize), 0);
   object.set(OBJPROP_TIME, iTime(NULL, 0, (i + 1) * rectsize), 1);
   object.set(OBJPROP_PRICE, iHigh(NULL, 0, h), 0);
   object.set(OBJPROP_PRICE, iLow(NULL, 0, l), 1);
   object.set(OBJPROP_COLOR, colors[i]);
   object.set(OBJPROP_WIDTH, i + 1);
   object.set(OBJPROP_STYLE, (ENUM_LINE_STYLE)i);
   // save to array
   objects[i] = object;
}
...
```

Aquí, para cada objeto, se calculan las coordenadas de dos puntos de anclaje; se establecen el color, el estilo y el ancho de línea iniciales.

A continuación, en un bucle infinito, cambiamos las propiedades de los objetos. Cuando *ScrollLock* está activado, la animación puede pausarse.

```
const int key = TerminalInfoInteger(TERMINAL_KEYSTATE_SCRLOCK);
int pass = 0;
int offset = 0;

for( ;!IsStopped(); ++pass)
{
   Sleep(200);
   if(TerminalInfoInteger(TERMINAL_KEYSTATE_SCRLOCK) != key) continue;
   // change color/style/width/fill/background from time to time
   if(pass % 5 == 0)
   {
      ++offset;
      for(int i = 0; i < OBJECT_NUMBER; ++i)
      {
         objects[i][].set(OBJPROP_COLOR, colors[(i + offset) % OBJECT_NUMBER]);
         objects[i][].set(OBJPROP_WIDTH, (i + offset) % OBJECT_NUMBER + 1);
         objects[i][].set(OBJPROP_FILL, rand() > 32768 / 2);
         objects[i][].set(OBJPROP_BACK, rand() > 32768 / 2);
      }
   }
   ChartRedraw();
}
```

Este es el aspecto que tiene en un gráfico:



Rectángulos OBJ_RECTANGLE con diferentes configuraciones de visualización

El rectángulo rojo situado más a la izquierda tiene activado el modo de relleno y está en primer plano. Así, las barras de su interior se muestran en azul brillante contrastado (*clrAqua*, también conocido comúnmente como *cyan*, que es *clrRed* invertido). El rectángulo púrpura también tiene un relleno, pero con una opción de fondo, por lo que las barras en él se muestran de una manera estándar.

Tenga en cuenta que el rectángulo naranja cubre completamente las barras al principio y al final de su subrango debido a la gran anchura de las líneas y a la visualización en la parte superior del gráfico.

Cuando el relleno está activado, no se tiene en cuenta la anchura de la línea. Cuando la anchura del borde es superior a 1, no se aplican algunos estilos de línea discontinua.

ObjectShapesDraw

Para el segundo ejemplo de esta sección, recuerde el hipotético programa de dibujo de formas que esbozamos en la Parte 3 cuando descubrimos POO. Nuestro progreso se detuvo en el hecho de que en el método de dibujo virtual (y se llamaba dibujar) sólo podíamos imprimir un mensaje en el registro de que estábamos dibujando una forma específica. Ahora, tras familiarizarnos con los objetos gráficos, tenemos la oportunidad de poner en práctica el dibujo.

Tomemos el script *Shapes5stats.mq5* como punto de partida. La versión actualizada se llamará *ObjectShapesDraw.mq5*.

Recordemos que, además de la clase base *Shape*, hemos descrito varias clases de formas: *Rectangle*, *Ellipse*, *Triangle*, *Square*, *Circle*. Todas ellas superponen con éxito objetos gráficos de los tipos OBJ_RECTANGLE, OBJ_ELLIPSE, OBJ_TRIANGLE. Pero hay algunos matices.

Todos los objetos especificados están vinculados a coordenadas de tiempo y precio, mientras que nuestro programa de dibujo asume ejes X e Y unificados con posicionamiento puntual. A este respecto,

necesitaremos configurar un gráfico para dibujar de forma especial y utilizar la función *ChartXYToTimePrice* para recalcular los puntos de la pantalla en tiempo y precio.

Además, los objetos OBJ_ELLIPSE y OBJ_TRIANGLE permiten rotaciones arbitrarias (en concreto, los radios pequeño y grande de una elipse pueden rotarse), mientras que OBJ_RECTANGLE siempre tiene sus lados orientados horizontal y verticalmente. Para simplificar el ejemplo, nos limitamos a la posición estándar de todas las formas.

En teoría, la nueva aplicación debería considerarse una demostración de objetos gráficos, y no un programa de dibujo. Un enfoque más correcto para el dibujo completo, desprovisto de las restricciones que imponen los objetos gráficos (ya que están destinados a otros fines en general, como el marcado de gráficos), es utilizar recursos gráficos. Por ello, volveremos a replantearnos el programa de dibujo en el capítulo dedicado a los recursos.

En la nueva clase *Shape* vamos a deshacernos de la estructura anidada *Pair* con coordenadas de objetos: esta estructura servía para demostrar varios principios de la programación orientada a objetos (POO), pero ahora es más fácil devolver la descripción original de los campos *int x, y* directamente a la clase *Shape*. También añadiremos un campo con el nombre del objeto.

```
class Shape
{
   ...
protected:
   int x, y;
   color backgroundColor;
   const string type;
   string name;

   Shape(int px, int py, color back, string t) :
      x(px), y(py),
      backgroundColor(back),
      type(t)
   {
   }

public:
   ~Shape()
   {
      ObjectDelete(0, name);
   }
   ...
```

El campo *name* será necesario para establecer las propiedades de un objeto gráfico, así como para eliminarlo del gráfico, lo que es lógico hacer en el destructor.

Dado que los distintos tipos de formas requieren un número diferente de puntos o tamaños característicos, añadiremos el método *setup*, además del método virtual *draw*, en la interfaz *Shape*:

```
virtual void setup(const int &parameters[]) = 0;
```

Recordemos que en el script hemos implementado una clase anidada *Shape::Registrator*, que se encargaba de contar el número de formas por tipo. Ha llegado el momento de confiarle algo más de responsabilidad para que funcione como una fábrica de formas. Las clases o métodos «fábrica» son buenos porque permiten crear objetos de diferentes clases de forma unificada.

Para ello, añadimos a *Registrator* un método para crear una forma con los parámetros que incluyen las coordenadas obligatorias del primer punto, un color, y un array de parámetros adicionales (cada forma será capaz de interpretarlo de acuerdo con sus propias reglas, y en el futuro, leer o escribir en un archivo).

```
virtual Shape *create(const int px, const int py, const color back,
         const int &parameters[]) = 0;
```

El método es virtual abstracto porque ciertos tipos de formas sólo pueden ser creados por clases registradoras derivadas descritas en clases descendientes de *Shape*. Para simplificar la escritura de clases de generación de registro derivadas, introducimos una clase de plantilla *MyRegistrator* con una implementación del método *create* adecuada para todos los casos.

```
template<typename T>
class MyRegistrator : public Shape::Registrator
{
public:
   MyRegistrator() : Registrator(typename(T))
   {
   }

   virtual Shape *create(const int px, const int py, const color back,
      const int &parameters[]) override
   {
      T *temp = new T(px, py, back);
      temp.setup(parameters);
      return temp;
   }
};
```

Aquí llamamos al constructor de alguna forma T previamente desconocida, la ajustamos llamando a *setup* y devolvemos una instancia al código de llamada.

Así es como se utiliza en la clase *Rectangle*, que tiene dos parámetros adicionales para la anchura y la altura.

```
class Rectangle : public Shape
{
   static MyRegistrator<Rectangle> r;

protected:
   int dx, dy; // dimensions (width, height)

   Rectangle(int px, int py, color back, string t) :
      Shape(px, py, back, t), dx(1), dy(1)
   {
   }

public:
   Rectangle(int px, int py, color back) :
      Shape(px, py, back, typename(this)), dx(1), dy(1)
   {
      name = typename(this) + (string)r.increment();
   }

   virtual void setup(const int &parameters[]) override
   {
      if(ArraySize(parameters) < 2)
      {
         Print("Insufficient parameters for Rectangle");
         return;
      }
      dx = parameters[0];
      dy = parameters[1];
   }
   ...
};

static MyRegistrator<Rectangle> Rectangle::r;
```

Al crear una forma, su nombre contendrá no sólo el nombre de la clase (*typename*), sino también el número ordinal de la instancia, calculado en la llamada a *r.increment()*.

Otras clases de formas se describen de forma similar.

Ahora es el momento de examinar el método *draw* para *Rectangle*. En él, traducimos un par de puntos (x,y) y (x + dx, y + dy) a coordenadas tiempo/precio utilizando *ChartXYToTimePrice* y creamos un objeto OBJ_RECTANGLE.

```
void draw() override
{
   // Print("Drawing rectangle");
   int subw;
   datetime t;
   double p;
   ChartXYToTimePrice(0, x, y, subw, t, p);
   ObjectCreate(0, name, OBJ_RECTANGLE, 0, t, p);
   ChartXYToTimePrice(0, x + dx, y + dy, subw, t, p);
   ObjectSetInteger(0, name, OBJPROP_TIME, 1, t);
   ObjectSetDouble(0, name, OBJPROP_PRICE, 1, p);

   ObjectSetInteger(0, name, OBJPROP_COLOR, backgroundColor);
   ObjectSetInteger(0, name, OBJPROP_FILL, true);
}
```

Por supuesto, no olvide establecer el color en OBJPROP_COLOR y el relleno en OBJPROP_FILL.

En el caso de la clase *Square*, no es necesario modificar nada en sí: basta con igualar *dx* y *dy*.

Para la clase *Ellipse*, dos opciones adicionales, *dx* y *dy*, determinan los radios pequeño y grande trazados en relación con el centro (x,y). En consecuencia, en el método *draw* calculamos 3 puntos de anclaje y creamos un objeto OBJ_ELLIPSE.

```
class Ellipse : public Shape
{
   static MyRegistrator<Ellipse> r;
protected:
   int dx, dy; // large and small radii
   ...
public:
   void draw() override
   {
      // Print("Drawing ellipse");
      int subw;
      datetime t;
      double p;

      // (x, y) center
      // p0: x + dx, y
      // p1: x - dx, y
      // p2: x, y + dy

      ChartXYToTimePrice(0, x + dx, y, subw, t, p);
      ObjectCreate(0, name, OBJ_ELLIPSE, 0, t, p);
      ChartXYToTimePrice(0, x - dx, y, subw, t, p);
      ObjectSetInteger(0, name, OBJPROP_TIME, 1, t);
      ObjectSetDouble(0, name, OBJPROP_PRICE, 1, p);
      ChartXYToTimePrice(0, x, y + dy, subw, t, p);
      ObjectSetInteger(0, name, OBJPROP_TIME, 2, t);
      ObjectSetDouble(0, name, OBJPROP_PRICE, 2, p);

      ObjectSetInteger(0, name, OBJPROP_COLOR, backgroundColor);
      ObjectSetInteger(0, name, OBJPROP_FILL, true);
   }
};

static MyRegistrator<Ellipse> Ellipse::r;
```

Circle es un caso especial de elipse con radios iguales.

Por último, en esta fase sólo se admiten los triángulos equiláteros: el tamaño del lado está contenido en un campo adicional *dx*. Le invitamos a conocer su método*draw* en el código fuente de forma independiente.

El nuevo script generará, como antes, un número determinado de formas aleatorias. Se crean mediante la función *addRandomShape*.

```
Shape *addRandomShape()
{
   const int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
   const int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS);

   const int n = random(Shape::Registrator::getTypeCount());

   int cx = 1 + w / 4 + random(w / 2), cy = 1 + h / 4 + random(h / 2);
   int clr = ((random(256) << 16) | (random(256) << 8) | random(256));
   int custom[] = {1 + random(w / 4), 1 + random(h / 4)};
   return Shape::Registrator::get(n).create(cx, cy, clr, custom);
}
```

Aquí es donde vemos el uso del método de fábrica *create*, llamado sobre un objeto registrador seleccionado aleatoriamente con el número *n*. Si más adelante decidimos añadir otras clases de formas, no tendremos que cambiar nada en la lógica de generación.

Todas las formas se colocan en la parte central de la ventana y tienen dimensiones no superiores a un cuarto de la ventana.

Queda por considerar directamente las llamadas a la función *addRandomShape*, y la configuración especial del horario que ya hemos mencionado.

Para obtener una representación «cuadrada» de los puntos en la pantalla, ajuste el modo CHART_SCALEFIX_11. Además, elegiremos la escala más densa (comprimida) a lo largo del eje temporal CHART_SCALE (0), porque en ella una barra ocupa 1 píxel horizontal (máxima precisión). Por último, desactive la visualización del propio gráfico estableciendo CHART_SHOW en *false*.

```
void OnStart()
{
   const int scale = (int)ChartGetInteger(0, CHART_SCALE);
   ChartSetInteger(0, CHART_SCALEFIX_11, true);
   ChartSetInteger(0, CHART_SCALE, 0);
   ChartSetInteger(0, CHART_SHOW, false);
   ChartRedraw();
   ...
```

Para almacenar las formas, reservemos un array de punteros inteligentes y llenémoslo de formas aleatorias.

```
#define FIGURES 21
...
void OnStart()
{
   ...
   AutoPtr<Shape> shapes[FIGURES];

   for(int i = 0; i < FIGURES; ++i)
   {
      Shape *shape = shapes[i] = addRandomShape();
      shape.draw();
   }

   ChartRedraw();
   ...
```

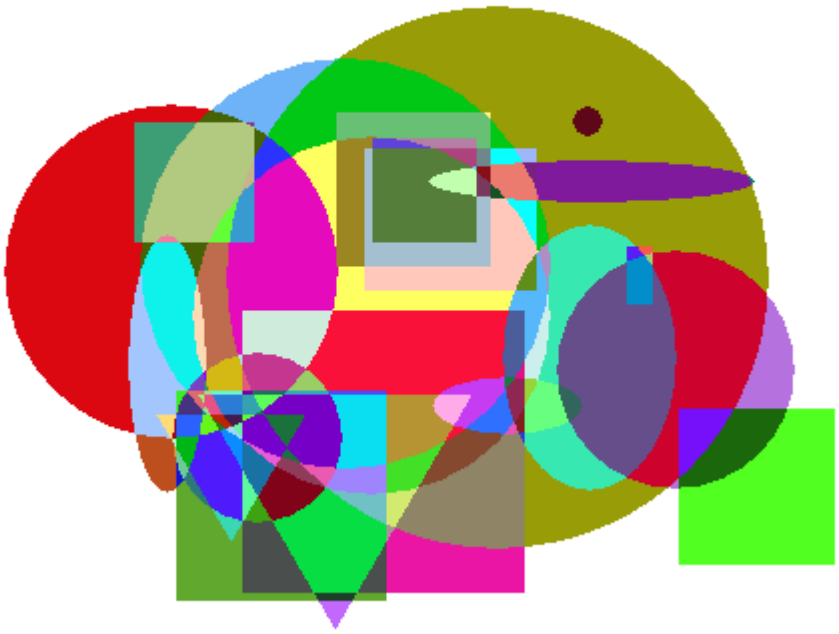
A continuación, ejecutamos un bucle infinito hasta que el usuario detiene el script, en el que movemos ligeramente las formas utilizando el método *move*.

```
   while(!IsStopped())
   {
      Sleep(250);
      for(int i = 0; i < FIGURES; ++i)
      {
         shapes[i][].move(random(20) - 10, random(20) - 10);
         shapes[i][].draw();
      }
      ChartRedraw();
   }
   ...
```

Al final, restauramos la configuración del gráfico.

```
   // it's not enough to disable CHART_SCALEFIX_11, you need CHART_SCALEFIX
   ChartSetInteger(0, CHART_SCALEFIX, false);
   ChartSetInteger(0, CHART_SCALE, scale);
   ChartSetInteger(0, CHART_SHOW, true);
}
```

En la siguiente captura de pantalla se muestra el aspecto que podría tener un gráfico con las formas dibujadas.



Objetos de forma del gráfico

La particularidad de dibujar objetos es la «multiplicación» de los colores en los lugares donde se superponen.

Debido a que el eje Y sube y baja, todos los triángulos están al revés, pero eso no es crítico, porque vamos a rehacer el programa de pintura basado en recursos de todos modos.

5.8.15 Ajustes de fuente

Todos los tipos de objetos permiten establecer determinados textos para ellos (OBJPROP_TEXT). Muchos de ellos muestran el texto especificado directamente en el gráfico, para el resto se convierte en una parte informativa de la información sobre herramientas.

Cuando se muestra texto dentro de un objeto (para los tipos OBJ_TEXT, OBJ_LABEL, OBJ_BUTTON y OBJ_EDIT), puede elegir un nombre y tamaño de fuente. Para los objetos de otros tipos no se aplican los ajustes de fuente: sus descripciones se muestran siempre en la fuente estándar del gráfico.

Identificador	Descripción	Tipo
OBJPROP_FONTSIZE	Tamaño de fuente en píxeles	int
OBJPROP_FONT	Fuente	string

No se puede ajustar el tamaño de la fuente en puntos de impresión aquí.

El script de prueba *ObjectFont.mq5* crea objetos con texto y cambia el nombre y el tamaño de la fuente. Utilicemos la clase *ObjectBuilder* del script anterior.

Al principio de *OnStart*, el script calcula el centro de la ventana tanto en coordenadas de pantalla como en los ejes tiempo/precio. Esto es necesario porque los objetos de distintos tipos que participan en la prueba utilizan sistemas de coordenadas diferentes.

```
void OnStart()
{
   const string name = "ObjFont-";

   const int bars = (int)ChartGetInteger(0, CHART_WIDTH_IN_BARS);
   const int first = (int)ChartGetInteger(0, CHART_FIRST_VISIBLE_BAR);

   const datetime centerTime = iTime(NULL, 0, first - bars / 2);
   const double centerPrice =
      (ChartGetDouble(0, CHART_PRICE_MIN)
      + ChartGetDouble(0, CHART_PRICE_MAX)) / 2;

   const int centerX = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS) / 2;
   const int centerY = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS) / 2;
   ...
```

La lista de tipos de objetos comprobados se especifica en el array *types*. Para algunos de ellos, en particular OBJ_HLINE y OBJ_VLINE, la configuración de la fuente no tendrá ningún efecto, aunque el texto de las descripciones aparecerá en la pantalla (para asegurarnos de ello, activamos el modo CHART_SHOW_OBJECT_DESCR).

```
   ChartSetInteger(0, CHART_SHOW_OBJECT_DESCR, true);

   ENUM_OBJECT types[] =
   {
      OBJ_HLINE,
      OBJ_VLINE,
      OBJ_TEXT,
      OBJ_LABEL,
      OBJ_BUTTON,
      OBJ_EDIT,
   };
   int t = 0; // cursor
   ...
```

La variable *t* se utilizará para cambiar secuencialmente de un tipo a otro.

El array *fonts* contiene las fuentes estándar más populares de Windows.

```
string fonts[] =
{
   "Comic Sans MS",
   "Consolas",
   "Courier New",
   "Lucida Console",
   "Microsoft Sans Serif",
   "Segoe UI",
   "Tahoma",
   "Times New Roman",
   "Trebuchet MS",
   "Verdana"
};

int f = 0; // cursor
...
```

Iteraremos en ellas utilizando la variable *f*.

Dentro del bucle de demostración, ordenamos a *ObjectBuilder* que cree un objeto del tipo actual *types[t]* en el centro de la ventana (por unificación, las coordenadas se especifican en ambos sistemas de coordenadas, a fin de no hacer diferencias en el código dependiendo del tipo: las coordenadas no admitidas por el objeto simplemente no tendrán efecto).

```
while(!IsStopped())
{

   const string str = EnumToString(types[t]);
   ObjectBuilder *object = new ObjectBuilder(name + str, types[t]);
   object.set(OBJPROP_TIME, centerTime);
   object.set(OBJPROP_PRICE, centerPrice);
   object.set(OBJPROP_XDISTANCE, centerX);
   object.set(OBJPROP_YDISTANCE, centerY);
   object.set(OBJPROP_XSIZE, centerX / 3 * 2);
   object.set(OBJPROP_YSIZE, centerY / 3 * 2);
   ...
```

A continuación, configuramos el texto y la fuente (el tamaño se elige al azar).

```
   const int size = rand() * 15 / 32767 + 8;
   Comment(str + " " + fonts[f] + " " + (string)size);
   object.set(OBJPROP_TEXT, fonts[f] + " " + (string)size);
   object.set(OBJPROP_FONT, fonts[f]);
   object.set(OBJPROP_FONTSIZE, size);
   ...
```

Para la siguiente pasada, movemos los cursores en los arrays de tipos de objeto y nombres de fuente.

```
   t = ++t % ArraySize(types);
   f = ++f % ArraySize(fonts);
   ...
```

Por último, actualizamos el gráfico, esperamos 1 segundo y borramos el objeto para crear otro.

```
         ChartRedraw();
         Sleep(1000);
         delete object;
      }
   }
```

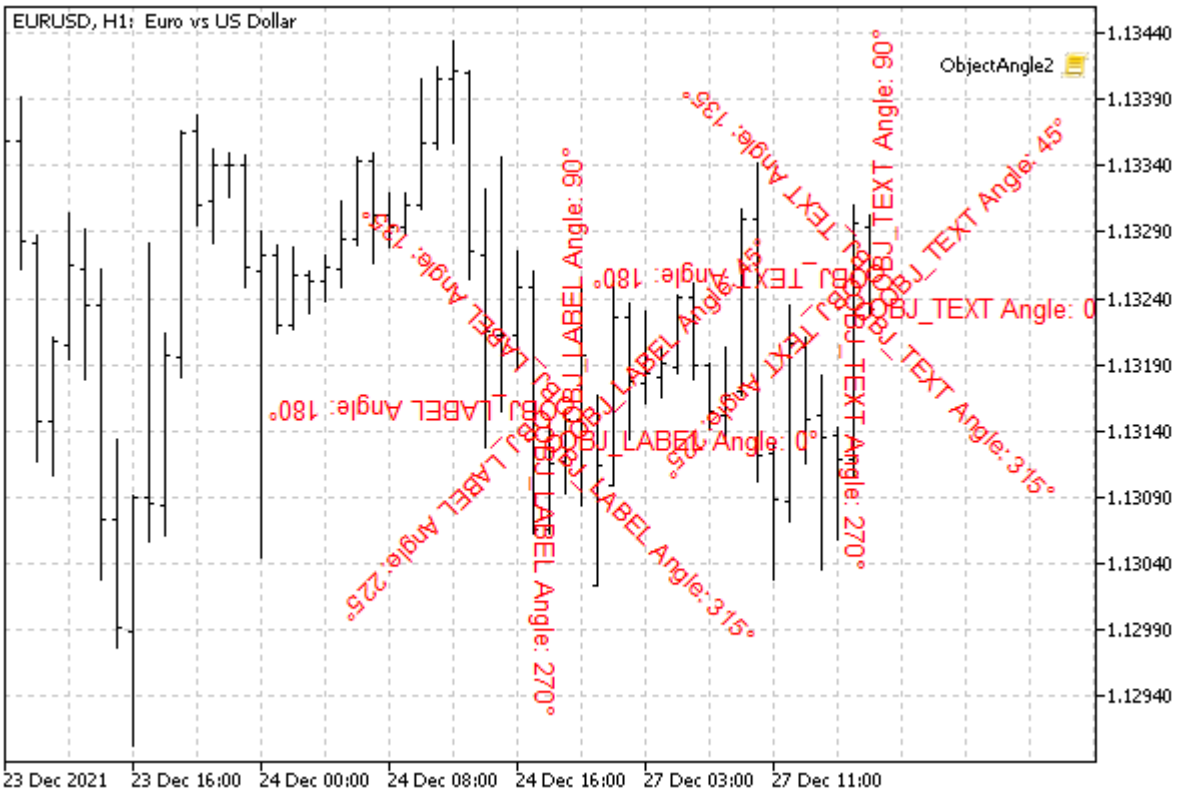
En la siguiente imagen se muestra el momento en que se ejecuta el script.

Botón con configuración de fuente personalizada

5.8.16 Rotar un texto en un ángulo arbitrario

Los objetos de tipo texto (etiqueta OBJ_TEXT (en coordenadas de cotización) y panel OBJ_LABEL (en coordenadas de pantalla)) permiten girar la etiqueta de texto en un ángulo arbitrario. Para ello, existe la propiedad OBJPROP_ANGLE del tipo *double*. Contiene el ángulo en grados relativo a la posición normal del objeto. Los valores positivos giran el objeto en el sentido contrario a las agujas del reloj, y los negativos en el sentido de las agujas del reloj.

No obstante, hay que tener en cuenta que los ángulos cuya diferencia es múltiplo de 360 grados son idénticos, es decir, por ejemplo, +315 y -45 son iguales. La rotación se realiza alrededor del punto de anclaje del objeto (por defecto, arriba a la izquierda).



Rotar objetos OBJ_LABEL y OBJ_TEXT en ángulos múltiplos de 45 grados

Puede comprobar el efecto de la propiedad OBJPROP_ANGLE en un objeto utilizando el script *ObjectAngle.mq5*. Crea una etiqueta de texto OBJ_LABEL en el centro de la ventana, tras lo cual comienza a girar periódicamente 45 grados hasta que el usuario detiene el proceso.

```
void OnStart()
{
   const string name = "ObjAngle";
   ObjectCreate(0, name, OBJ_LABEL, 0, 0, 0);
   const int centerX = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS) / 2;
   const int centerY = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS) / 2;
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, centerX);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, centerY);
   ObjectSetInteger(0, name, OBJPROP_ANCHOR, ANCHOR_CENTER);

   int angle = 0;
   while(!IsStopped())
   {
      ObjectSetString(0, name, OBJPROP_TEXT, StringFormat("Angle: %d°", angle));
      ObjectSetDouble(0, name, OBJPROP_ANGLE, angle);
      angle += 45;

      ChartRedraw();
      Sleep(1000);
   }
   ObjectDelete(0, name);
}
```

El texto muestra el valor actual del ángulo.

5.8.17 Determinar ancho y alto del objeto

Algunos tipos de objetos permiten establecer sus dimensiones en píxeles. Entre ellos se incluyen OBJ_BUTTON, OBJ_CHART, OBJ_BITMAP, OBJ_BITMAP_LABEL, OBJ_EDIT y OBJ_RECTANGLE_LABEL. Además, los objetos OBJ_LABEL admiten la lectura (pero no la configuración) de tamaños porque las etiquetas se expanden o contraen automáticamente para ajustarse al texto que contienen. Si se intenta acceder a propiedades de otros tipos de objetos, se producirá un error OBJECT_WRONG_PROPERTY (4203).

Identificador	Descripción
OBJPROP_XSIZE	Anchura del objeto a lo largo del eje X en píxeles
OBJPROP_YSIZE	Altura del objeto a lo largo del eje Y en píxeles

Ambos tamaños son enteros y, por lo tanto, se manejan con las funciones *ObjectGetInteger/ObjectSetInteger*.

Para los objetos OBJ_BITMAP y OBJ_BITMAP_LABEL se realiza un tratamiento especial de las dimensiones.

Sin asignar una imagen, estos objetos permiten establecer un tamaño arbitrario. Al mismo tiempo, se dibujan transparentes (sólo el marco es visible si no se «oculta» también configurando el color *clrNone*), pero reciben todos los eventos, en concreto, sobre los movimientos del ratón (con una descripción de texto, si la hay, en una información sobre herramientas) y los clics de sus botones sobre el objeto.

Cuando se asigna una imagen, ésta tiene por defecto la altura y la anchura del objeto. Sin embargo, un programa MQL puede establecer tamaños más pequeños y seleccionar un fragmento de una imagen para mostrar; encontrará más información al respecto en la sección sobre sincronización de tramas o frames. Si intenta establecer la altura o la anchura mayores que el tamaño de la imagen, ésta deja de mostrarse y las dimensiones del objeto no cambian.

Como ejemplo, vamos a desarrollar una versión mejorada del script *ObjectAnchorLabel.mq5* de la sección titulada Definir el punto de anclaje en el objeto. En esa sección, movíamos la etiqueta de texto alrededor de la ventana y la invertíamos cuando alcanzaba alguno de los bordes de la misma, pero lo hacíamos teniendo en cuenta sólo el punto de anclaje. Debido a esto, dependiendo de la ubicación del punto de anclaje en el objeto, podría darse la situación de que la etiqueta se desplazara casi por completo más allá de la ventana. Por ejemplo, si el punto de anclaje estuviera en el lado derecho del objeto, el desplazamiento hacia la izquierda provocaría que casi todo el texto sobrepasara el borde izquierdo de la ventana antes de que el punto de anclaje tocara el borde.

En el nuevo script *ObjectSizeLabel.mq5*, tendremos en cuenta el tamaño del objeto y cambiaremos la dirección del movimiento en cuanto toque el borde de la ventana con cualquiera de sus lados.

Para la correcta implementación de este modo, debe tenerse en cuenta que cada esquina de ventana utilizada como centro de referencia de coordenadas al punto de anclaje sobre el objeto determina la dirección característica de los ejes X e Y. Por ejemplo, si el usuario selecciona la esquina superior izquierda en la variable de entrada ENUM_BASE_CORNER, entonces X aumenta de izquierda a derecha e Y aumenta de arriba abajo. Si se considera que el centro es la esquina inferior derecha, entonces X aumenta de derecha a izquierda de la misma, e Y aumenta de abajo a arriba.

Una combinación mutua diferente de la esquina de anclaje en la ventana y el punto de anclaje en el objeto requiere diferentes ajustes de las distancias entre los bordes del objeto y los bordes de la ventana. En concreto, cuando se selecciona una de las esquinas de la derecha y uno de los puntos de anclaje del lado derecho del objeto, entonces no es necesaria la corrección en el borde derecho de la ventana, y en el lado opuesto, el izquierdo, hay que tener en cuenta la anchura del objeto (de manera que sus dimensiones no se salgan de la ventana hacia la izquierda).

Esta regla sobre la corrección del tamaño de un objeto puede generalizarse:

- En el borde de la ventana adyacente a la esquina de anclaje, la corrección es necesaria cuando el punto de anclaje se encuentra en el lado más alejado del objeto con respecto a esta esquina;
- En el borde de la ventana opuesto a la esquina de anclaje, la corrección es necesaria cuando el punto de anclaje se encuentra en el lado cercano del objeto con respecto a esta esquina.

En otras palabras: si el nombre de la esquina (en el elemento ENUM_BASE_CORNER) y el punto de anclaje (en el elemento ENUM_ANCHOR_POINT) contienen una palabra común (por ejemplo, DERECHA), la corrección es necesaria en el lado más alejado de la ventana (es decir, lejos de la esquina seleccionada). Si se encuentran direcciones opuestas en la combinación de los lados ENUM_BASE_CORNER y ENUM_ANCHOR_POINT (por ejemplo, IZQUIERDA y DERECHA), la corrección es necesaria en el lado más cercano de la ventana. Estas reglas funcionan igual para los ejes horizontal y vertical.

Además, hay que tener en cuenta que el punto de anclaje puede estar en el centro de cualquier lado del objeto. A continuación, en la dirección perpendicular se requiere una sangría desde los bordes de la ventana, igual a la mitad del tamaño del objeto.

Un caso especial es el punto de anclaje en el centro del objeto. Para ello, debe tener siempre un margen de distancia en cualquier dirección, igual a la mitad del tamaño del objeto.

La lógica descrita se implementa en una función especial denominada *GetMargins*, la cual toma como entradas la esquina y el punto de anclaje seleccionados, así como las dimensiones del objeto (*dx* y *dy*). La función devuelve una estructura con 4 campos que contienen los tamaños de las sangrías adicionales que deben apartarse del punto de anclaje en la dirección de los bordes cercano y lejano de la ventana para que el objeto no se pierda de vista. Las sangrías reservan la distancia en función de las dimensiones y la posición relativa del propio objeto.

```
struct Margins
{
   int nearX; // X increment between the object point and the window border adjacent
   int nearY; // Y increment between the object point and the window border adjacent
   int farX;  // X increment between the object's point and the opposite corner of th
   int farY;  // Y increment between the object's point and the opposite corner of th
};

Margins GetMargins(const ENUM_BASE_CORNER corner, const ENUM_ANCHOR_POINT anchor,
   int dx, int dy)
{
   Margins margins = {}; // zero corrections by default
   ...
   return margins;
}
```

Para unificar el algoritmo se introducen las siguientes definiciones macro de direcciones (lados):

```
#define LEFT 0x1
#define LOWER 0x2
#define RIGHT 0x4
#define UPPER 0x8
#define CENTER 0x16
```

Con su ayuda se definen máscaras de bits (combinaciones) que describen los elementos de las enumeraciones ENUM_BASE_CORNER y ENUM_ANCHOR_POINT.

```
const int corner_flags[] = // flags for ENUM_BASE_CORNER elements
{
   LEFT | UPPER,
   LEFT | LOWER,
   RIGHT | LOWER,
   RIGHT | UPPER
};

const int anchor_flags[] = // flags for ENUM_ANCHOR_POINT elements
{
   LEFT | UPPER,
   LEFT,
   LEFT | LOWER,
   LOWER,
   RIGHT | LOWER,
   RIGHT,
   RIGHT | UPPER,
   UPPER,
   CENTER
};
```

Cada uno de los arrays, *corner_flags* y *anchor_flags*, contiene exactamente tantos elementos como existen en la enumeración correspondiente.

A continuación viene el código de la función principal. En primer lugar, tratemos la opción más sencilla: el punto de anclaje central.

```
if(anchor == ANCHOR_CENTER)
{
   margins.nearX = margins.farX = dx / 2;
   margins.nearY = margins.farY = dy / 2;
}
else
{
   ...
}
```

Para analizar el resto de situaciones, utilizaremos las máscaras de bits de los arrays anteriores direccionándolos directamente por los valores recibidos *corner* y *anchor*.

```
   const int mask = corner_flags[corner] & anchor_flags[anchor];
   ...
```

Si la esquina y el punto de anclaje están en el mismo lado horizontal, funcionará la siguiente condición y se ajustará la anchura del objeto en el borde más alejado de la ventana.

```
if((mask & (LEFT | RIGHT)) != 0)
{
   margins.farX = dx;
}
...
```

Si no están en el mismo lado, pueden estar en lados opuestos, o puede darse el caso de que el punto de anclaje esté en medio del lado horizontal (arriba o abajo). La comprobación de un punto de anclaje en el centro se realiza mediante la expresión *(anchor_flags[anchor] & (LEFT | RIGHT)) == 0*; entonces, la corrección es igual a la mitad de la anchura del objeto.

```
else
{
   if((anchor_flags[anchor] & (LEFT | RIGHT)) == 0)
   {
      margins.nearX = dx / 2;
      margins.farX = dx / 2;
   }
   else
   {
      margins.nearX = dx;
   }
}
...
```

En caso contrario, con la orientación opuesta de la esquina y el punto de anclaje, realizamos un ajuste de la anchura del objeto en el borde cercano de la ventana.

Se realizan comprobaciones similares para el eje Y.

```
if((mask & (UPPER | LOWER)) != 0)
{
   margins.farY = dy;
}
else
{
   if((anchor_flags[anchor] & (UPPER | LOWER)) == 0)
   {
      margins.farY = dy / 2;
      margins.nearY = dy / 2;
   }
   else
   {
      margins.nearY = dy;
   }
}
```

Ahora la función *GetMargins* está lista, y podemos proceder al código principal del script en la función *OnStart*. Como antes, determinamos el tamaño de la ventana, calculamos las coordenadas iniciales en el centro, creamos un objeto OBJ_LABEL y lo seleccionamos.

```
void OnStart()
{
   const int t = ChartWindowOnDropped();
   Comment(EnumToString(Corner));

   const string name = "ObjSizeLabel";
   int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS, t) - 1;
   int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS) - 1;
   int x = w / 2;
   int y = h / 2;

   ObjectCreate(0, name, OBJ_LABEL, t, 0, 0);
   ObjectSetInteger(0, name, OBJPROP_SELECTABLE, true);
   ObjectSetInteger(0, name, OBJPROP_SELECTED, true);
   ObjectSetInteger(0, name, OBJPROP_CORNER, Corner);
   ...
```

Para la animación, un bucle infinito proporciona las variables *pass* (contador de iteraciones) y *anchor* (el punto de anclaje, que se elegirá periódicamente de forma aleatoria).

```
   int pass = 0;
   ENUM_ANCHOR_POINT anchor = 0;
   ...
```

Pero hay algunos cambios en comparación con *ObjectAnchorLabel.mq5*.

No generaremos movimientos aleatorios del objeto; en lugar de ello, vamos a establecer una velocidad constante de 5 píxeles en diagonal.

```
   int px = 5, py = 5;
```

Para almacenar el tamaño de la etiqueta de texto, reservaremos dos nuevas variables.

```
   int dx = 0, dy = 0;
```

El resultado del recuento de sangrías adicionales se almacenará en una variable *m* de tipo *Margins*.

```
   Margins m = {};
```

A esto le sigue directamente el bucle de mover y modificar el objeto. En él, cada 75 iteraciones (una iteración de 100 ms, véase más adelante), seleccionamos aleatoriamente un nuevo punto de anclaje, formamos un nuevo texto (el contenido del objeto) a partir de él y esperamos a que los cambios se apliquen al objeto (llamando a *ChartRedraw*). Esto último es necesario porque el tamaño de la inscripción se ajusta automáticamente al contenido, y el nuevo tamaño es importante para que podamos calcular correctamente las sangrías en la llamada a *GetMargins*.

Obtenemos las dimensiones utilizando las llamadas *ObjectGetInteger* con las propiedades OBJPROP_XSIZE y OBJPROP_YSIZE.

```
for( ;!IsStopped(); ++pass)
{
   if(pass % 75 == 0)
   {
      // ENUM_ANCHOR_POINT consists of 9 elements: randomly choose one
      const int r = rand() * 8 / 32768 + 1;
      anchor = (ENUM_ANCHOR_POINT)((anchor + r) % 9);
      ObjectSetInteger(0, name, OBJPROP_ANCHOR, anchor);
      ObjectSetString(0, name, OBJPROP_TEXT, " " + EnumToString(anchor)
         + StringFormat("[%3d,%3d] ", x, y));
      ChartRedraw();
      Sleep(1);

      dx = (int)ObjectGetInteger(0, name, OBJPROP_XSIZE);
      dy = (int)ObjectGetInteger(0, name, OBJPROP_YSIZE);

      m = GetMargins(Corner, anchor, dx, dy);
   }
   ...
```

Una vez que conocemos el punto de anclaje y todas las distancias, movemos el objeto. Si este «choca» contra la pared, cambiamos el sentido del movimiento al opuesto (*px* a *-px* o *py* a *-py*, según el lado).

```
// bounce off window borders, object fully visible
if(x + px >= w - m.farX)
{
   x = w - m.farX + px - 1;
   px = -px;
}
else if(x + px < m.nearX)
{
   x = m.nearX + px;
   px = -px;
}

if(y + py >= h - m.farY)
{
   y = h - m.farY + py - 1;
   py = -py;
}
else if(y + py < m.nearY)
{
   y = m.nearY + py;
   py = -py;
}

// calculate the new label position
x += px;
y += py;
...
```

Queda por actualizar el estado del propio objeto: mostrar las coordenadas actuales en la etiqueta de texto y asignarlas a las propiedades OBJPROP_XDISTANCE y OBJPROP_YDISTANCE.

```
ObjectSetString(0, name, OBJPROP_TEXT, " " + EnumToString(anchor)
   + StringFormat("[%3d,%3d] ", x, y));
ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);
...
```

Después de cambiar el objeto, llamamos a *ChartRedraw* y esperamos 100 ms para garantizar una animación razonablemente suave.

```
ChartRedraw();
Sleep(100);
...
```

Al final del bucle, volvemos a comprobar el tamaño de la ventana, ya que el usuario puede cambiarlo mientras se ejecuta el script, y también repetimos la petición de tamaño.

```
h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS, t) - 1;
w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS) - 1;

dx = (int)ObjectGetInteger(0, name, OBJPROP_XSIZE);
dy = (int)ObjectGetInteger(0, name, OBJPROP_YSIZE);
m = GetMargins(Corner, anchor, dx, dy);
```

```
   }
```

Hemos omitido algunas otras innovaciones del script *ObjectSizeLabel.mq5* para que la explicación sea concisa. Quienes lo deseen pueden consultar el código. En concreto, se han utilizado colores distintivos para la inscripción: cada color específico corresponde a su propio punto de anclaje, lo que hace que los puntos de cambio sean más perceptibles. También puede hacer clic en *Delete* mientras se ejecuta el script: esto eliminará el objeto seleccionado del gráfico y el script finalizará automáticamente.

5.8.18 Visibilidad de los objetos en el contexto de marcos temporales

Los usuarios de MetaTrader 5 conocen la pestaña Mostrar del cuadro de diálogo de propiedades del objeto, donde puede utilizar los interruptores (switches) para seleccionar en qué marcos temporales se mostrará el objeto y en cuáles estará oculto. En concreto, puede ocultar temporalmente el objeto por completo en todos los marcos temporales.

MQL5 tiene una propiedad de programa similar, OBJPROP_TIMEFRAMES, que controla la visibilidad del objeto en un marco temporal. El valor de esta propiedad puede ser cualquier combinación de banderas de enteros especiales: cada bandera (constante) contiene un bit correspondiente a un marco temporal (véase la tabla). Para establecer/obtener la propiedad OBJPROP_TIMEFRAMES, utilice las funciones *ObjectSetInteger/ObjectGetInteger*.

Constante	Valor	Visibilidad en marcos temporales
OBJ_NO_PERIODS	0	El objeto está oculto en todos los marcos temporales
OBJ_PERIOD_M1	0x00000001	M1
OBJ_PERIOD_M2	0x00000002	M2
OBJ_PERIOD_M3	0x00000004	M3
OBJ_PERIOD_M4	0x00000008	M4
OBJ_PERIOD_M5	0x00000010	M5
OBJ_PERIOD_M6	0x00000020	M6
OBJ_PERIOD_M10	0x00000040	M10
OBJ_PERIOD_M12	0x00000080	M12
OBJ_PERIOD_M15	0x00000100	M15
OBJ_PERIOD_M20	0x00000200	M20
OBJ_PERIOD_M30	0x00000400	M30
OBJ_PERIOD_H1	0x00000800	H1
OBJ_PERIOD_H2	0x00001000	H2
OBJ_PERIOD_H3	0x00002000	H3
OBJ_PERIOD_H4	0x00004000	H4

Constante	Valor	Visibilidad en marcos temporales
OBJ_PERIOD_H6	0x00008000	H6
OBJ_PERIOD_H8	0x00010000	H8
OBJ_PERIOD_H12	0x00020000	H12
OBJ_PERIOD_D1	0x00040000	D1
OBJ_PERIOD_W1	0x00080000	W1
OBJ_PERIOD_MN1	0x00100000	MN1
OBJ_ALL_PERIODS	0x001fffff	Todos los marcos temporales

Las banderas pueden combinarse utilizando el operador OR («|»), por ejemplo, la superposición de las banderas OBJ_PERIOD_M15 | OBJ_PERIOD_H4 significa que el objeto será visible en los marcos temporales de 15 minutos y 4 horas.

Obsérvese que cada bandera puede obtenerse desplazando en 1 a la izquierda el número de bits igual al número de la constante en la tabla. Esto facilita la generación dinámica de banderas cuando el algoritmo opera en múltiples marcos temporales en lugar de en uno concreto.

Utilizaremos esta función en el script de prueba *ObjectTimeframes.mq5*. Su tarea consiste en crear un montón de grandes etiquetas de texto en el gráfico con los nombres de los marcos temporales, y cada título debe mostrarse sólo en el marco temporal correspondiente. Por ejemplo, una etiqueta grande «D1» sólo será visible en el gráfico diario, y al cambiar a H4, veremos «H4».

Para obtener el nombre abreviado del marco temporal, sin el prefijo «PERIOD_», se implementa una sencilla función auxiliar.

```
string GetPeriodName(const int tf)
{
   const static int PERIOD_ = StringLen("PERIOD_");
   return StringSubstr(EnumToString((ENUM_TIMEFRAMES)tf), PERIOD_);
}
```

Para obtener la lista de todos los marcos temporales a partir de la enumeración ENUM_TIMEFRAMES, utilizaremos la función EnumToArray que se presentó en la sección sobre conversión de Enumeraciones.

```
#include "ObjectPrefix.mqh"
#include <MQL5Book/EnumToArray.mqh>

void OnStart()
{
   ENUM_TIMEFRAMES tf = 0;
   int values[];
   const int n = EnumToArray(tf, values, 0, USHORT_MAX);
   ...
```

Todas las etiquetas se mostrarán en el centro de la ventana en el momento en que se inicie el script. Si se cambia el tamaño de la ventana una vez finalizado el script, los subtítulos creados dejarán de estar centrados. Esto es consecuencia del hecho de que MQL5 sólo admite el anclaje en las esquinas de la

ventana, pero no en el centro. Si desea mantener automáticamente la posición de los objetos debe implementar un algoritmo similar en el indicador y responder a eventos de cambio de tamaño de ventana. Como alternativa, podríamos mostrar las etiquetas en una esquina; por ejemplo, la inferior derecha.

```
const int centerX = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS) / 2;
const int centerY = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS) / 2;
...
```

En el ciclo a través de los marcos temporales, creamos un objeto OBJ_LABEL para cada uno de ellos, y lo colocamos en el centro de la ventana anclada en el centro del objeto.

```
for(int i = 1; i < n; ++i)
{
   // create and setup text label for each timeframe
   const string name = ObjNamePrefix + (string)i;
   ObjectCreate(0, name, OBJ_LABEL, 0, 0, 0);
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, centerX);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, centerY);
   ObjectSetInteger(0, name, OBJPROP_ANCHOR, ANCHOR_CENTER);
   ...
```

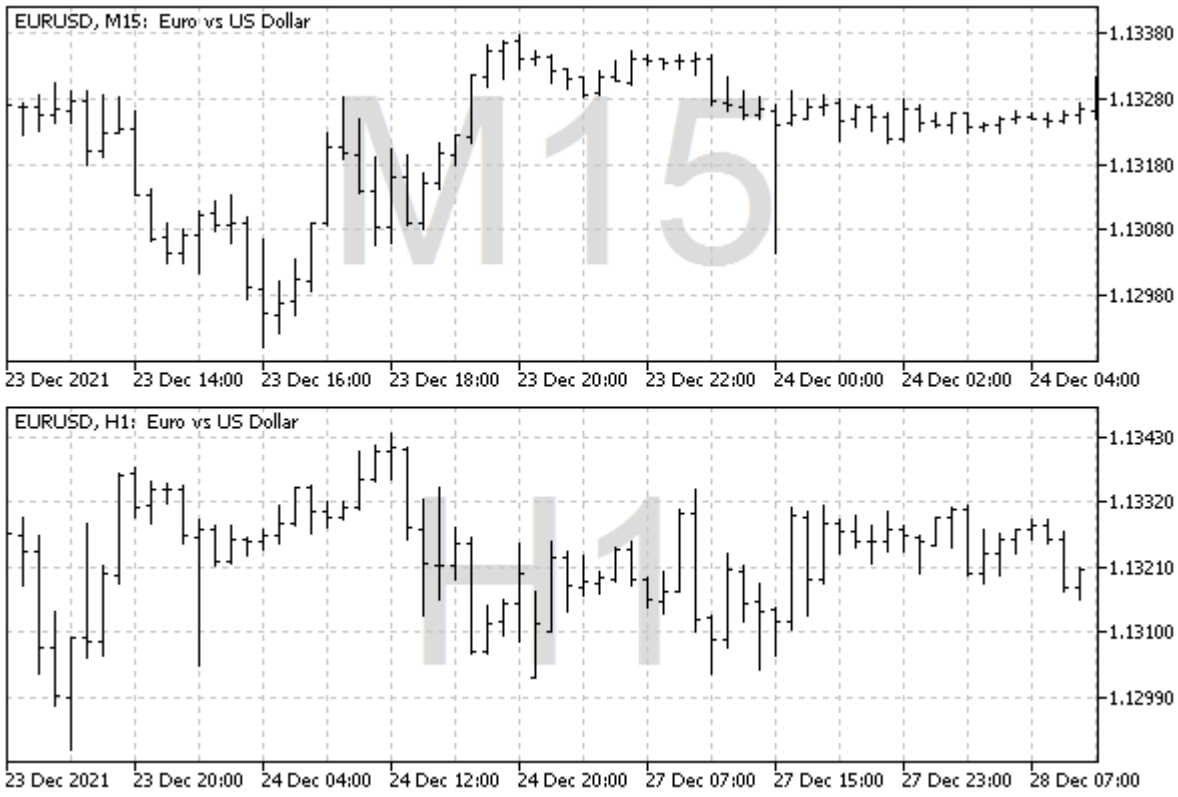
A continuación, establecemos el texto (nombre del marco temporal), el tamaño de fuente grande, el color gris y la propiedad de visualización en segundo plano.

```
   ObjectSetString(0, name, OBJPROP_TEXT, GetPeriodName(values[i]));
   ObjectSetInteger(0, name, OBJPROP_FONTSIZE, fmin(centerY, centerX));
   ObjectSetInteger(0, name, OBJPROP_COLOR, clrLightGray);
   ObjectSetInteger(0, name, OBJPROP_BACK, true);
   ...
```

Por último, generamos la bandera de visibilidad correcta para el marco temporal *i*ésimo y la escribimos en la propiedad OBJPROP_TIMEFRAMES.

```
   const int flag = 1 << (i - 1);
   ObjectSetInteger(0, name, OBJPROP_TIMEFRAMES, flag);
}
```

Vea lo que ha ocurrido en el gráfico al cambiar de marco temporal.



Etiquetas con nombres de marcos temporales

Si abre el cuadro de diálogo *Object List* y activa *All* objetos en la lista, es fácil asegurarse de que haya etiquetas generadas para todos los marcos temporales y comprobar sus banderas de visibilidad.

Para eliminar objetos, puede ejecutar el script *ObjectCleanup1.mq5*.

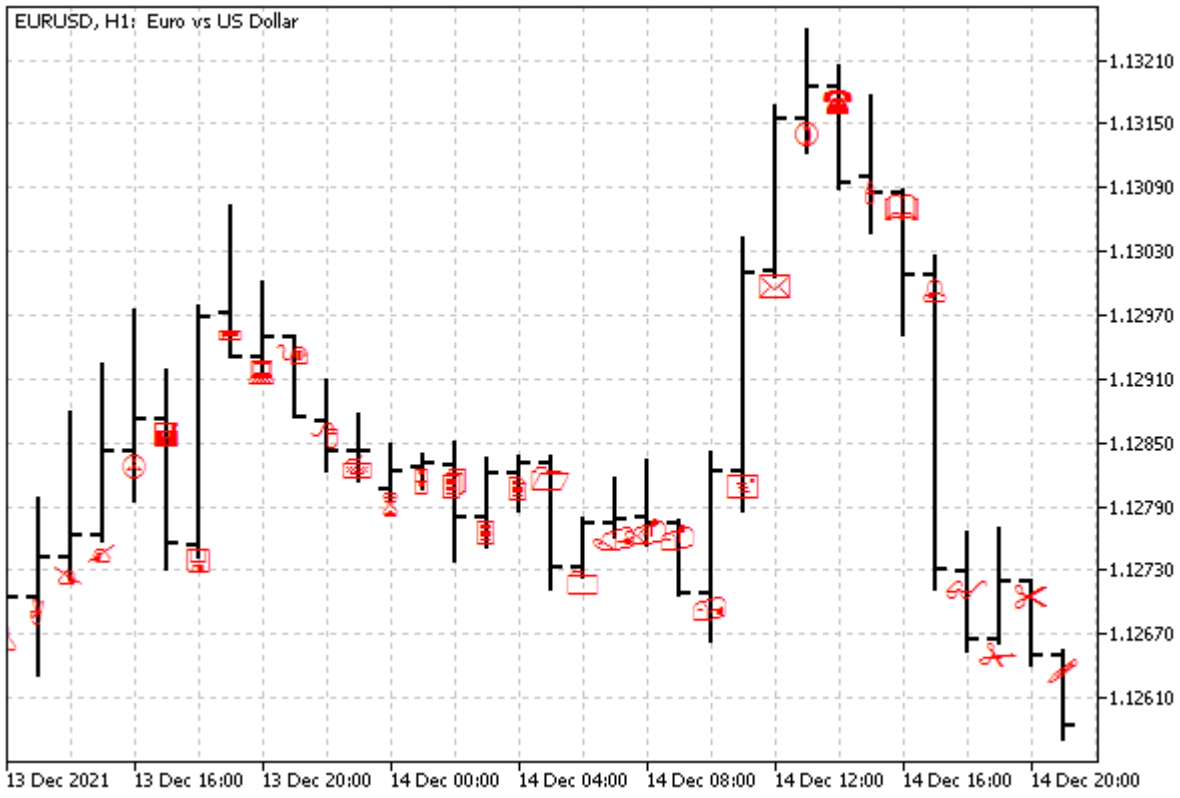
5.8.19 Asignar un código de carácter a una etiqueta

Como se mencionó en la reseña de Objetos vinculados al tiempo y al precio, la etiqueta OBJ_ARROW le permite mostrar un símbolo de fuente Wingdings arbitrario en el gráfico (la lista completa de símbolos disponibles se proporciona en la Documentación MQL5). El código de caracteres del propio objeto viene determinado por la propiedad de enteros OBJPROP_ARROWCODE.

El script permite demostrar todos los caracteres de *ObjectWingdings.mq5 font*. En él, creamos etiquetas con distintos caracteres en un bucle, colocándolas una a una en la barra.

```
#include "ObjectPrefix.mqh"

void OnStart()
{
   for(int i = 33; i < 256; ++i) // character codes
   {
      const int b = i - 33; // bar number
      const string name = ObjNamePrefix + "Wingdings-"
         + (string)iTime(_Symbol, _Period, b);
      ObjectCreate(0, name, OBJ_ARROW,
         0, iTime(_Symbol, _Period, b), iOpen(_Symbol, _Period, b));
      ObjectSetInteger(0, name, OBJPROP_ARROWCODE, i);
   }

   PrintFormat("%d objects with arrows created", 256 - 33);
}
```

En la siguiente captura de pantalla se muestra cómo se ve en el gráfico.



Caracteres Wingdings en las etiquetas OBJ_ARROW

5.8.20 Propiedades de los rayos para objetos con líneas rectas

Entre los objetos gráficos existen varios tipos en los que las líneas entre puntos de anclaje pueden mostrarse como segmentos (es decir, estrictamente entre un par de puntos) o como líneas rectas infinitas que continúan en una u otra dirección a lo largo de toda el área de visibilidad de la ventana. Tales objetos son:

- Línea de tendencia

- Línea de tendencia por ángulo
- Todos los tipos de canales (equidistantes, desviaciones estándar, regresión, horquilla de Andrews)
- Línea de Gann
- Líneas de Fibonacci
- Canal de Fibonacci
- Expansión de Fibonacci

Para ellos, puede activar por separado la continuación de línea hacia la izquierda o hacia la derecha mediante las propiedades OBJPROP_RAY_LEFT y OBJPROP_RAY_RIGHT, respectivamente. Además, para una línea vertical, puede especificar si debe dibujarse en todas las subventanas del gráfico o sólo en la actual (donde se encuentra el punto de anclaje): la propiedad OBJPROP_RAY se encarga de ello. Todas las propiedades son booleanas, lo que significa que pueden estar activadas (*true*) o desactivadas (*false*).

Identificador	Descripción
OBJPROP_RAY_LEFT	El rayo continúa hacia la izquierda.
OBJPROP_RAY_RIGHT	El rayo continúa hacia la derecha.
OBJPROP_RAY	La línea vertical se extiende a todas las ventanas del gráfico.

Puede comprobar el funcionamiento de los rayos utilizando el script *ObjectRays.mq5*. Crea 3 canales de desviación estándar con diferentes configuraciones de rayo.

La función de ayuda *SetupChannel* crea y configura un objeto específico. Mediante sus parámetros se establece la longitud del canal en barras y la anchura del canal (desviación), así como las opciones de visualización de los rayos a izquierda y derecha, y el color.

```
#include "ObjectPrefix.mqh"

void SetupChannel(const int length, const double deviation = 1.0,
   const bool right = false, const bool left = false,
   const color clr = clrRed)
{
   const string name = ObjNamePrefix + "Channel"
      + (right ? "R" : "") + (left ? "L" : "");
   // NB: Anchor point 0 must have an earlier time than anchor point 1,
   // otherwise the channel will degenerate
   ObjectCreate(0, name, OBJ_STDDEVCHANNEL, 0, iTime(NULL, 0, length), 0);
   ObjectSetInteger(0, name, OBJPROP_TIME, 1, iTime(NULL, 0, 0));
   // deviation
   ObjectSetDouble(0, name, OBJPROP_DEVIATION, deviation);
   // color and description
   ObjectSetInteger(0, name, OBJPROP_COLOR, clr);
   ObjectSetString(0, name, OBJPROP_TEXT, StringFormat("%2.1", deviation)
      + ((!right && !left) ? " NO RAYS" : "")
      + (right ? " RIGHT RAY" : "") + (left ? " LEFT RAY" : ""));
   // properties of rays
   ObjectSetInteger(0, name, OBJPROP_RAY_RIGHT, right);
   ObjectSetInteger(0, name, OBJPROP_RAY_LEFT, left);
   // lighting up objects by highlighting
   // (besides, it's easier for the user to remove them)
   ObjectSetInteger(0, name, OBJPROP_SELECTABLE, true);
   ObjectSetInteger(0, name, OBJPROP_SELECTED, true);
}
```

En la función *OnStart*, llamamos a *SetupChannel* para 3 canales diferentes.

```
void OnStart()
{
   SetupChannel(24, 1.0, true);
   SetupChannel(48, 2.0, false, true, clrBlue);
   SetupChannel(36, 3.0, false, false, clrGreen);
}
```

Como resultado, obtenemos un gráfico de la siguiente forma.

Canales con diferentes configuraciones de las propiedades OBJPROP_RAY_LEFT y OBJPROP_RAY_RIGHT

Cuando los rayos están activados, es posible solicitar al objeto que extrapole valores de tiempo y precio utilizando las funciones que describiremos en la sección Obtener la hora o el precio en determinados puntos de las líneas.

5.8.21 Gestionar el estado pulsado de los objetos

Para objetos como botones (OBJ_BUTTON) y paneles con una imagen (OBJ_BITMAP_LABEL), el terminal admite una propiedad especial que cambia visualmente el objeto del estado normal (liberado) al estado pulsado y viceversa. La constante OBJPROP_STATE está reservada para ello. La propiedad es de tipo booleano: cuando el valor es *true*, se considera que el objeto está pulsado, y cuando es *false*, se considera que está liberado (por defecto).

Para OBJ_BUTTON, el efecto de un marco tridimensional es dibujado por el propio terminal, mientras que para OBJ_BITMAP_LABEL el programador debe especificar dos imágenes (como archivos o recursos) que proporcionará una representación externa adecuada. Dado que esta propiedad es técnicamente sólo un conmutador, es fácil utilizarla para otros fines, y no sólo para efectos de «pulsar» y «soltar». Por ejemplo, con la ayuda de imágenes adecuadas, puede implementar una bandera (opción).

El uso de imágenes en objetos se abordará en la siguiente sección.

El estado del objeto suele cambiar en los programas MQL interactivos que responden a las acciones del usuario, en particular a los clics del ratón. Analizaremos esta posibilidad en el capítulo sobre eventos.

Ahora vamos a probar la propiedad en botones simples, en modo estático. El script *ObjectButtons.mq5* crea dos botones en el gráfico: uno en estado pulsado y otro en estado liberado.

La configuración de un solo botón se proporciona a la función *SetupButton* con parámetros que especifican el nombre y el texto del botón, así como sus coordenadas, tamaño y estado.

```
#include "ObjectPrefix.mqh"

void SetupButton(const string button,
   const int x, const int y,
   const int dx, const int dy,
   const bool state = false)
{
   const string name = ObjNamePrefix + button;
   ObjectCreate(0, name, OBJ_BUTTON, 0, 0, 0);
   // position and size
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);
   ObjectSetInteger(0, name, OBJPROP_XSIZE, dx);
   ObjectSetInteger(0, name, OBJPROP_YSIZE, dy);
   // label on the button
   ObjectSetString(0, name, OBJPROP_TEXT, button);

   // pressed (true) / released (false)
   ObjectSetInteger(0, name, OBJPROP_STATE, state);
}
```

A continuación, en *OnStart* llamamos a esta función dos veces.

```
void OnStart()
{
   SetupButton("Pressed", 100, 100, 100, 20, true);
   SetupButton("Normal", 100, 150, 100, 20);
}
```

Los botones resultantes podrían tener este aspecto:

Botones OBJ_BUTTON pulsados y soltados

Curiosamente, puede hacer clic en cualquiera de los botones con el ratón, y el botón cambiará de estado. Sin embargo, aún no hemos hablado de cómo interceptar una notificación al respecto.

Es importante tener en cuenta que este cambio de estado automático sólo se realiza si la opción *Disable selection* está marcada en las propiedades del objeto, pero esta condición es la predeterminada para todos los objetos creados mediante programación. Recuerde que, si es necesario, esta selección puede activarse: para ello, debe establecer explícitamente la propiedad OBJPROP_SELECTABLE en *true*. La hemos utilizado en algunos ejemplos anteriores.

Para eliminar botones que se han vuelto innecesarios, utilice el script *ObjectCleanup1.mq5*.

5.8.22 Ajustar imágenes en objetos bitmap

Los objetos de tipo OBJ_BITMAP_LABEL (un panel con una imagen situada en coordenadas de pantalla) permiten mostrar imágenes de mapa de bits. Por imágenes de mapa de bits se entiende el formato gráfico BMP: aunque en principio existen muchos otros formatos de ráster (por ejemplo, PNG o GIF), actualmente no se admiten en MQL5, igual que los vectoriales.

La propiedad de cadena OBJPROP_BMPFILE permite especificar una imagen para un objeto. Debe contener el nombre del archivo BMP o recurso.

Dado que este objeto admite la posibilidad de conmutación de estados en dos posiciones (véase OBJPROP_STATE), debe utilizarse un parámetro modificador para ello: una imagen para el estado «activado»/«pulsado» se establece en el índice 0, y el estado «desactivado»/«liberado» se establece en el índice 1. Si sólo especifica una imagen (sin modificador, que equivale a 0), se utilizará para ambos estados. El estado por defecto de un objeto es «desactivado»/«liberado».

El tamaño del objeto se hace igual al tamaño de la imagen, pero puede cambiarse especificando valores más pequeños en las propiedades OBJPROP_XSIZE y OBJPROP_YSIZE: en este caso, sólo se muestra una parte de la imagen (para más detalles, véase la siguiente sección sobre sincronización de tramas o frames).

La longitud de la cadena OBJPROP_BMPFILE no debe superar los 63 caracteres. Puede contener no sólo el nombre del archivo, sino también la ruta hacia él. Si la cadena comienza con un carácter separador de ruta (barra diagonal '/' o doble barra invertida '\\'), entonces el archivo se busca en relación a *terminal_data_directory/MQL5/*. De lo contrario, el archivo se busca en relación con la carpeta donde se encuentra el programa MQL.

Por ejemplo, la cadena «\\Images\euro.bmp» (o «/Images/euro.bmp») hace referencia a un archivo del directorio *MQL5/Images/euro.bmp*. El paquete estándar de entrega de terminales incluye la carpeta *Images* en el directorio MQL5, y hay un par de archivos de prueba *euro.bmp* y *dollar.bmp*, por lo que la ruta está en funcionamiento. Si especifica la cadena «Images\euro.bmp» o («Images/euro.bmp»), esto implicará, por ejemplo, para un script lanzado desde *MQL5/Scripts/MQL5Book/*, que la carpeta *Images* con el archivo *euro.bmp* debe estar ubicada directamente allí, es decir, toda la ruta será *MQL5/Scripts/MQL5Book/Images/euro.bmp*. No existe tal archivo en nuestro libro, y esto provocaría un error al cargar la imagen. No obstante, esta disposición de los archivos gráficos junto al programa tiene sus ventajas: es más fácil controlar el montaje y no hay confusión con imágenes mezcladas de distintos programas.

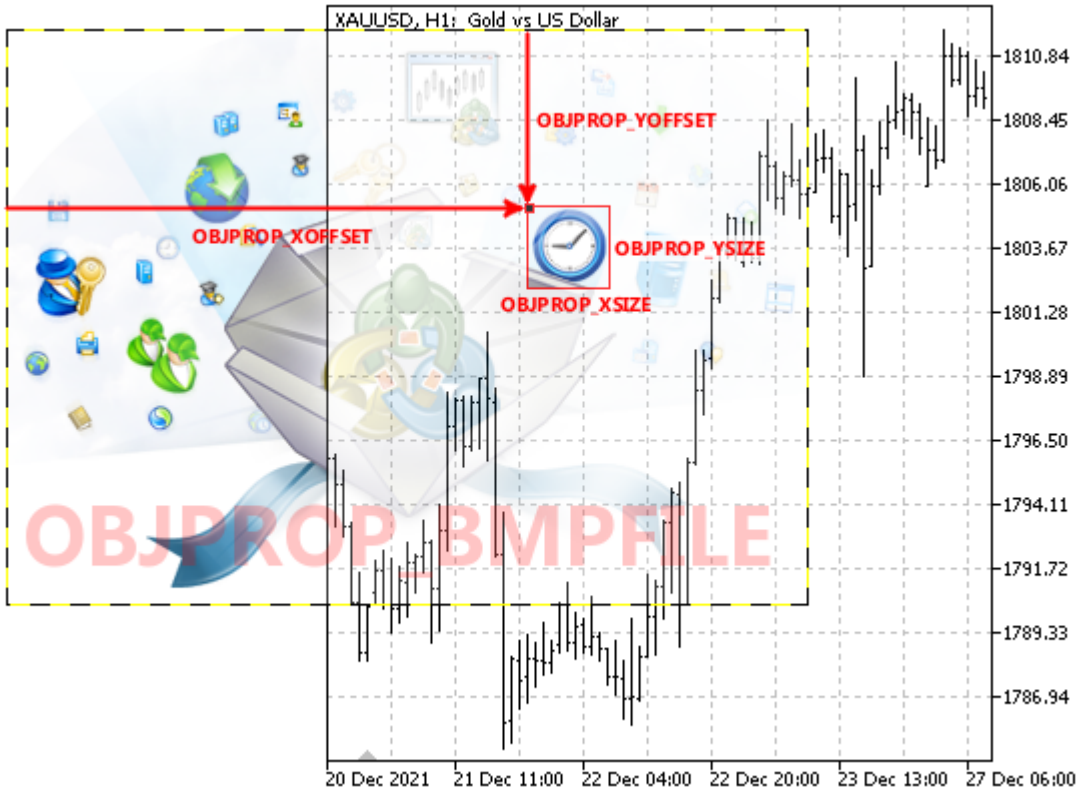
El script *ObjectBitmap.mq5* crea un panel con una imagen en el gráfico y le asigna dos imágenes: «\\Images\\dollar.bmp» e «\\Images\\euro.bmp».

```
#include "ObjectPrefix.mqh"

void SetupBitmap(const string button, const int x, const int y,
   const string imageOn, const string imageOff = NULL)
{
   // creating a panel
   const string name = ObjNamePrefix + "Bitmap";
   ObjectCreate(0, name, OBJ_BITMAP_LABEL, 0, 0, 0);
   // set position
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);
   // include images
   ObjectSetString(0, name, OBJPROP_BMPFILE, 0, imageOn);
   if(imageOff != NULL) ObjectSetString(0, name, OBJPROP_BMPFILE, 1, imageOff);
}

void OnStart()
{
   SetupBitmap("image", 100, 100,
      "\\Images\\dollar.bmp", "\\Images\\euro.bmp");
}
```

Al igual que con el resultado del script de la sección anterior, aquí también puede hacer clic en el objeto de imagen y ver que cambia de la imagen del dólar a la del euro y viceversa.

5.8.23 Recortar (representar parte) de una imagen

Para objetos gráficos con imágenes (OBJ_BITMAP_LABEL y OBJ_BITMAP), MQL5 permite habilitar la visualización de una parte de la imagen especificada por la propiedad OBJPROP_BMPFILE. Para ello, debe establecer el tamaño del objeto (OBJPROP_XSIZE y OBJPROP_YSIZE) para que sea más pequeño que el de la imagen y establecer las coordenadas de la esquina superior izquierda del fragmento rectangular visible utilizando las propiedades enteras OBJPROP_XOFFSET y OBJPROP_YOFFSET. Estas dos propiedades establecen, respectivamente, la sangría a lo largo de X e Y en píxeles desde los bordes izquierdo y superior de la imagen original.



Representación de parte de una imagen para un objeto

Normalmente, para los iconos de las barras de herramientas (conjuntos de botones, menús, etc.) se utiliza una técnica similar que utiliza parte de una imagen grande: un único archivo con todos los iconos proporciona un consumo de recursos más eficiente que muchos archivos pequeños con iconos individuales.

El script de prueba *ObjectBitmapOffset.mq5* crea varios paneles con imágenes (OBJ_BITMAP_LABEL), y para todos ellos se especifica el mismo archivo gráfico en la propiedad OBJPROP_BMPFILE. No obstante, debido a las propiedades OBJPROP_XOFFSET y OBJPROP_YOFFSET, todos los objetos muestran diferentes partes de la imagen.

```
void SetupBitmap(const int i, const int x, const int y, const int size,
   const string imageOn, const string imageOff = NULL)
{
   // create an object
   const string name = ObjNamePrefix + "Tool-" + (string)i;
   ObjectCreate(0, name, OBJ_BITMAP_LABEL, 0, 0, 0);
   ObjectSetInteger(0, name, OBJPROP_CORNER, CORNER_RIGHT_UPPER);
   ObjectSetInteger(0, name, OBJPROP_ANCHOR, ANCHOR_RIGHT_UPPER);
   // position and size
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);
   ObjectSetInteger(0, name, OBJPROP_XSIZE, size);
   ObjectSetInteger(0, name, OBJPROP_YSIZE, size);
   // offset in the original image, according to which the i-th fragment is read
   ObjectSetInteger(0, name, OBJPROP_XOFFSET, i * size);
   ObjectSetInteger(0, name, OBJPROP_YOFFSET, 0);
   // generic image (file)
   ObjectSetString(0, name, OBJPROP_BMPFILE, imageOn);
}

void OnStart()
{
   const int icon = 46; // size of one icon
   for(int i = 0; i < 7; ++i) // loop through the icons in the file
   {
      SetupBitmap(i, 10, 10 + i * icon, icon,
         "\\Files\\MQL5Book\\icons-322-46.bmp");
   }
}
```

La imagen original contiene varios iconos pequeños de 46 x 46 píxeles cada uno. El script los «recorta» uno a uno y los coloca verticalmente en el borde derecho de la ventana.

A continuación se muestra un archivo genérico (*/Files/MQL5Book/icons-322-46.bmp*) y lo que ha sucedido en el gráfico:

Archivo BMP con iconos

Objetos de botón con iconos en el gráfico

5.8.24 Propiedades de los campos de entrada: alineación de texto y sólo lectura

Para los objetos de tipo OBJ_EDIT (campo de entrada), un programa MQL puede establecer dos propiedades específicas definidas mediante las funciones *ObjectSetInteger/ObjectGetInteger*.

Identificador	Descripción	Tipo de valor
OBJPROP_ALIGN	Alineación horizontal del texto	ENUM_ALIGN_MODE
OBJPROP_READONLY	Posibilidad de editar texto	bool

La enumeración ENUM_ALIGN_MODE contiene los siguientes miembros:

Identificador	Descripción
ALIGN_LEFT	Alineación a la izquierda
ALIGN_CENTER	Alineación central
ALIGN_RIGHT	Alineación a la derecha

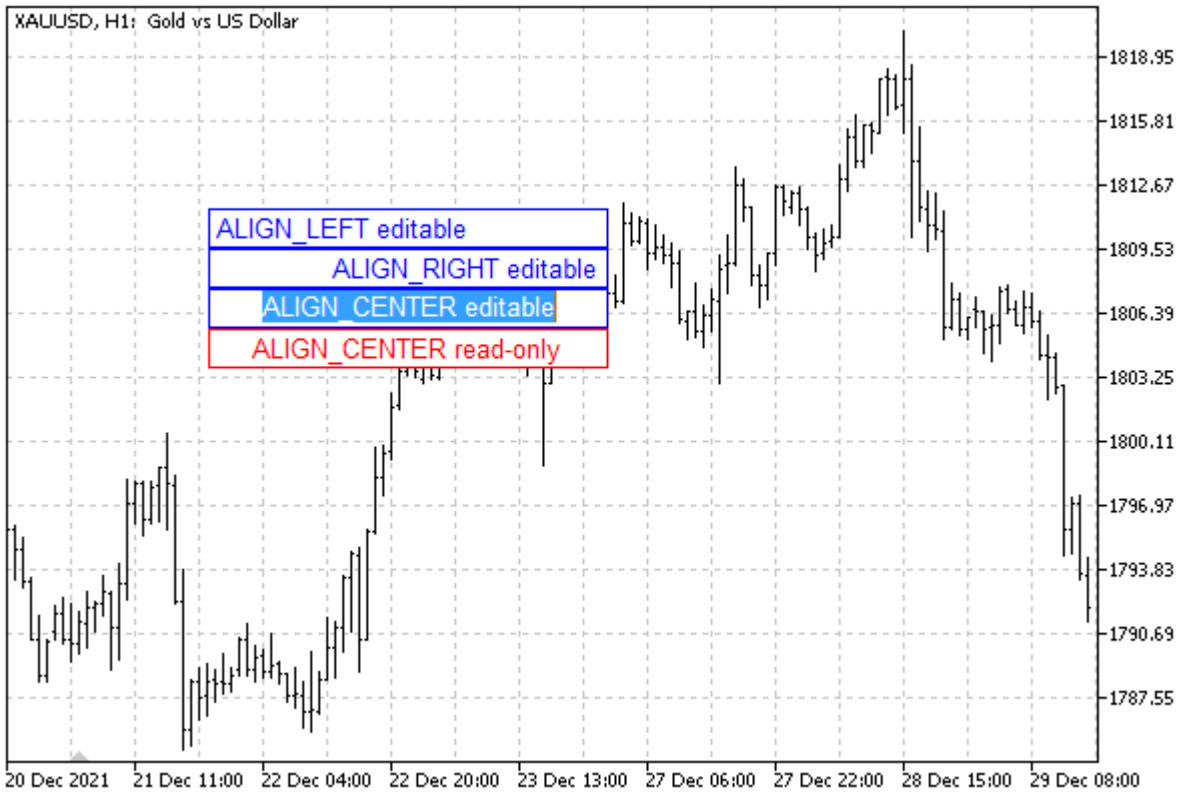
Tenga en cuenta que, a diferencia de los objetos OBJ_TEXT y OBJ_LABEL, el campo de entrada no se redimensiona automáticamente para ajustarse al texto introducido, por lo que para cadenas largas, puede que necesite establecer explícitamente la propiedad OBJPROP_XSIZE.

En el modo de edición, el desplazamiento horizontal del texto funciona dentro del campo de entrada.

El script *ObjectEdit.mq5* crea cuatro objetos OBJ_EDIT: tres de ellos son editables con diferentes métodos de alineación de texto y el cuarto está en modo de sólo lectura.

```
#include "ObjectPrefix.mqh"

void SetupEdit(const int x, const int y, const int dx, const int dy,
   const ENUM_ALIGN_MODE alignment = ALIGN_LEFT, const bool readonly = false)
{
   // create an object with a description of the properties
   const string props = EnumToString(alignment)
      + (readonly ? " read-only" : " editable");
   const string name = ObjNamePrefix + "Edit" + props;
   ObjectCreate(0, name, OBJ_EDIT, 0, 0, 0);
   // position and size
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);
   ObjectSetInteger(0, name, OBJPROP_XSIZE, dx);
   ObjectSetInteger(0, name, OBJPROP_YSIZE, dy);
   // specific properties of input fields
   ObjectSetInteger(0, name, OBJPROP_ALIGN, alignment);
   ObjectSetInteger(0, name, OBJPROP_READONLY, readonly);
   // colors (different depending on editability)
   ObjectSetInteger(0, name, OBJPROP_BGCOLOR, clrWhite);
   ObjectSetInteger(0, name, OBJPROP_COLOR, readonly ? clrRed : clrBlue);
   // content
   ObjectSetString(0, name, OBJPROP_TEXT, props);
   // tooltip for editable
   ObjectSetString(0, name, OBJPROP_TOOLTIP,
      (readonly ? "\n" : "Click me to edit"));
}

void OnStart()
{
   SetupEdit(100, 100, 200, 20);
   SetupEdit(100, 120, 200, 20, ALIGN_RIGHT);
   SetupEdit(100, 140, 200, 20, ALIGN_CENTER);
   SetupEdit(100, 160, 200, 20, ALIGN_CENTER, true);
}
```

El resultado del script se muestra en la siguiente imagen:



Campos de entrada en diferentes modos

Puede hacer clic en cualquier campo editable y cambiar su contenido.

5.8.25 Anchura del canal de desviación estándar

El canal de desviación estándar OBJ_STDDEVCHANNEL tiene una propiedad especial que define el ancho del canal como un multiplicador para la desviación estándar (media cuadrática). La propiedad se denomina OBJPROP_DEVIATION y puede tomar valores reales positivos (*double*). Por defecto, es igual a 1.0.

Ya hemos visto un ejemplo de su uso en el script *ObjectRays.mq5* en la sección sobre Propiedades de los rayos para objetos con líneas rectas.

5.8.26 Establecer niveles en objetos de nivel

Algunos objetos gráficos se construyen utilizando varios niveles (líneas repetitivas). Entre ellos figuran:

- Horquilla de Andrews OBJ_PITCHFORK
- Herramientas de Fibonacci:
  - Niveles OBJ_FIBO
  - Zonas horarias OBJ_FIBOTIMES
  - Abanico OBJ_FIBOFAN
  - Arcos OBJ_FIBOARC
  - Canal OBJ_FIBOCHANNEL
  - Extensión OBJ_EXPANSION

MQL5 permite establecer propiedades de nivel para dichos objetos. Las propiedades incluyen su número, colores, valores y etiquetas.

Identificador	Descripción	Tipo
OBJPROP_LEVELS	Número de niveles	int
OBJPROP_LEVELCOLOR	Color de la línea de nivel	color
OBJPROP_LEVELSTYLE	Estilo de línea de nivel	ENUM_LINE_STYLE
OBJPROP_LEVELWIDTH	Anchura de la línea de nivel	int
OBJPROP_LEVELTEXT	Descripción del nivel	string
OBJPROP_LEVELVALUE	Valor del nivel	double

Al llamar a las funciones *ObjectGet* y *ObjectSet* para todas las propiedades excepto OBJPROP_LEVELS, es necesario proporcionar un parámetro modificador adicional con el número de un nivel específico.

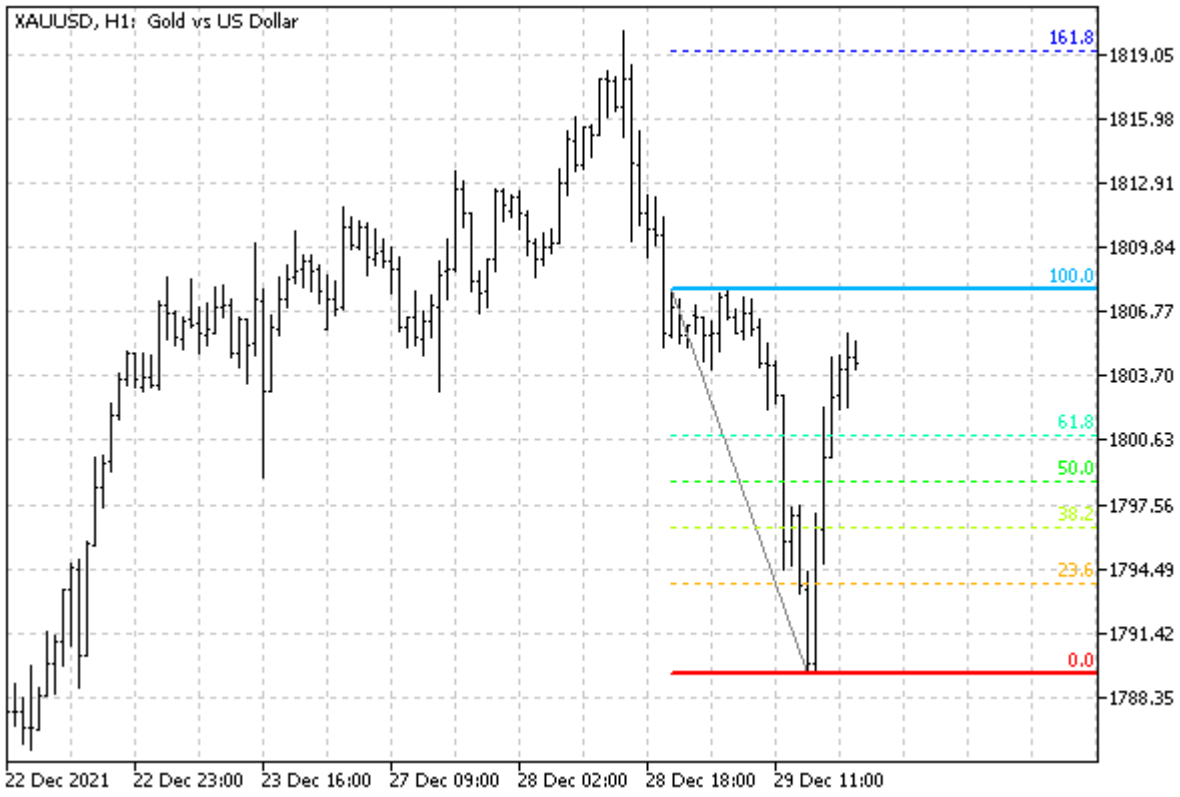
Tomemos como ejemplo el indicador *ObjectHighLowFibo.mq5*. Para un rango de barras determinado, que se define como el número de la última barra (*baroffset*) y el número de barras (*BarCount*) a la izquierda de la misma, el indicador encuentra los precios *High* y *Low*, y, a continuación, crea el objeto OBJ_FIBO para estos puntos. A medida que se formen nuevas barras, los niveles de Fibonacci se desplazarán hacia la derecha, hacia precios más actuales.

```
#property indicator_chart_window
#property indicator_buffers 0
#property indicator_plots   0

#include <MQL5Book/ColorMix.mqh>

input int BarOffset = 0;
input int BarCount = 24;

const string Prefix = "HighLowFibo-";

int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
   static datetime now = 0;
   if(now != iTime(NULL, 0, 0))
   {
      const int hh = iHighest(NULL, 0, MODE_HIGH, BarCount, BarOffset);
      const int ll = iLowest(NULL, 0, MODE_LOW, BarCount, BarOffset);

      datetime t[2] = {iTime(NULL, 0, hh), iTime(NULL, 0, ll)};
      double p[2] = {iHigh(NULL, 0, hh), iLow(NULL, 0, ll)};

      DrawFibo(Prefix + "Fibo", t, p, clrGray);

      now = iTime(NULL, 0, 0);
   }
   return rates_total;
}
```

La configuración directa del objeto se realiza en la función auxiliar *DrawFibo*. En ella, en concreto, los niveles se pintan con los colores del arco iris, y su estilo y grosor se determinan en función de si los valores correspondientes están «redondeados» (sin parte fraccionaria).

```
bool DrawFibo(const string name, const datetime &t[], const double &p[],
   const color clr)
{
   if(ArraySize(t) != ArraySize(p)) return false;

   ObjectCreate(0, name, OBJ_FIBO, 0, 0, 0);
   // anchor points
   for(int i = 0; i < ArraySize(t); ++i)
   {
      ObjectSetInteger(0, name, OBJPROP_TIME, i, t[i]);
      ObjectSetDouble(0, name, OBJPROP_PRICE, i, p[i]);
   }
   // general settings
   ObjectSetInteger(0, name, OBJPROP_COLOR, clr);
   ObjectSetInteger(0, name, OBJPROP_RAY_RIGHT, true);
   // level settings
   const int n = (int)ObjectGetInteger(0, name, OBJPROP_LEVELS);
   for(int i = 0; i < n; ++i)
   {
      const color gradient = ColorMix::RotateColors(ColorMix::HSVtoRGB(0),
         ColorMix::HSVtoRGB(359), n, i);
      ObjectSetInteger(0, name, OBJPROP_LEVELCOLOR, i, gradient);
      const double level = ObjectGetDouble(0, name, OBJPROP_LEVELVALUE, i);
      if(level - (int)level > DBL_EPSILON * level)
      {
         ObjectSetInteger(0, name, OBJPROP_LEVELSTYLE, i, STYLE_DOT);
         ObjectSetInteger(0, name, OBJPROP_LEVELWIDTH, i, 1);
      }
      else
      {
         ObjectSetInteger(0, name, OBJPROP_LEVELSTYLE, i, STYLE_SOLID);
         ObjectSetInteger(0, name, OBJPROP_LEVELWIDTH, i, 2);
      }
   }

   return true;
}
```

He aquí una variante del aspecto que puede tener un objeto en un gráfico:



Objeto de Fibonacci con ajustes de niveles

5.8.27 Propiedades adicionales de los objetos de Gann, Fibonacci y Elliot

Los objetos de Gann, Fibonacci y Elliot tienen propiedades específicas y únicas. Dependiendo del tipo de propiedad, utilice las funciones *ObjectGetInteger/ObjectSetInteger* o *ObjectGetDouble/ObjectSetDouble*.

Identificador	Descripción	Tipo
OBJPROP_DIRECTION	Tendencia del objeto de Gann (Fan OBJ_GANNFAN y Grid OBJ_GANNGRID)	ENUM_GANN_DIRECTION
OBJPROP_DEGREE	Niveles de grado de la onda de Elliot	ENUM_ELLIOT_WAVE_DEGREE
OBJPROP_DRAWLINES	Visualización de las líneas para los niveles de la onda de Elliot	bool
OBJPROP_SCALE	Escala en puntos por barra (propiedad de los objetos de Gann y del objeto Arcos de Fibonacci)	double
OBJPROP_ELLIPSE	Visualización de la elipse completa para el objeto Arcos de Fibonacci (OBJ_FIBOARC)	bool

La enumeración ENUM_GANN_DIRECTION tiene los siguientes miembros:

Constante	Dirección de la tendencia
GANN_UP_TREND	Líneas ascendentes
GANN_DOWN_TREND	Líneas descendentes

ENUM_ELLIOT_WAVE_DEGREE se utiliza para establecer el tamaño (método de etiquetado) de las ondas de Elliot.

Constante	Descripción
ELLIOTT_GRAND_SUPERCYCLE	Gran Superciclo
ELLIOTT_SUPERCYCLE	Superciclo
ELLIOTT_CYCLE	Ciclo
ELLIOTT_PRIMARY	Ciclo primario
ELLIOTT_INTERMEDIATE	Enlace intermedio
ELLIOTT_MINOR	Ciclo menor
ELLIOTT_MINUTE	Minuto
ELLIOTT_MINUETTE	Minuette
ELLIOTT_SUBMINUETTE	Subminuette

5.8.28 Objeto gráfico

El objeto de gráfico OBJ_CHART permite crear miniaturas de otros gráficos dentro del gráfico para otros instrumentos y marcos temporales. Los objetos de gráfico se incluyen en la lista de gráficos, que hemos obtenido mediante programación utilizando las funciones *ChartFirst* y *ChartNext*. Como se menciona en la sección sobre Comprobar el estado de la ventana principal, la propiedad especial del gráfico CHART_IS_OBJECT permite averiguar por identificador si se trata de una ventana completa o de un objeto gráfico. En este último caso, la llamada a *ChartGetInteger(id, CHART_IS_OBJECT)* devolverá *true*.

El objeto gráfico tiene un conjunto de propiedades específicas.

Identificador	Descripción	Tipo
OBJPROP_CHART_ID	ID del gráfico (r/o)	long
OBJPROP_PERIOD	Período del gráfico	ENUM_TIMEFRAMES
OBJPROP_DATE_SCALE	Mostrar la escala de tiempo	bool
OBJPROP_PRICE_SCALE	Mostrar la escala de precios	bool
OBJPROP_CHART_SCALE	Escala (valor en el intervalo 0 - 5)	int
OBJPROP_SYMBOL	Símbolo	string

El identificador obtenido a través de la propiedad OBJPROP_CHART_ID permite gestionar el objeto como un gráfico normal utilizando las funciones del capítulo Trabajar con gráficos. Sin embargo, existen algunas limitaciones:

- El objeto no puede cerrarse con *ChartClose*
- No es posible cambiar el símbolo/período en el objeto utilizando la función *CartSetSymbolPeriod*
- Las propiedades CHART_SCALE, CHART_BRING_TO_TOP,CHART_SHOW_DATE_SCALE y CHART_SHOW_PRICE_SCALE no se modifican en el objeto.

Por defecto, todas las propiedades (excepto OBJPROP_CHART_ID) son iguales a las propiedades correspondientes de la ventana actual.

La demostración de objetos gráficos se implementa como un indicador sin búfer *ObjectChart.mq5*. Crea una subventana con dos objetos gráficos para el mismo símbolo que el gráfico actual, pero con marcos temporales adyacentes por encima y por debajo del actual.

Los objetos se ajustan a la esquina superior derecha de la subventana y tienen los mismos tamaños predefinidos:

```
#define SUBCHART_HEIGHT 150
#define SUBCHART_WIDTH  200
```

Por supuesto, la altura de la subventana debe coincidir con la altura de los objetos, hasta que podamos responder de forma adaptativa a los eventos de cambio de tamaño.

```
#property indicator_separate_window
#property indicator_height SUBCHART_HEIGHT
#property indicator_buffers 0
#property indicator_plots   0
```

Un minigráfico se configura en la función *SetupSubChart*, que toma como entradas el número del objeto, sus dimensiones y el marco temporal requerido. El resultado de *SetupSubChart* es el identificador del objeto gráfico, que acabamos de introducir en el registro como referencia.

```
void OnInit()
{
   Print(SetupSubChart(0, SUBCHART_WIDTH, SUBCHART_HEIGHT, PeriodUp(_Period)));
   Print(SetupSubChart(1, SUBCHART_WIDTH, SUBCHART_HEIGHT, PeriodDown(_Period)));
}
```

Las macros *PeriodUp* y *PeriodDown* utilizan la función de ayuda *PeriodRelative*.

```
#define PeriodUp(P) PeriodRelative(P, +1)
#define PeriodDown(P) PeriodRelative(P, -1)

ENUM_TIMEFRAMES PeriodRelative(const ENUM_TIMEFRAMES tf, const int step)
{
   static const ENUM_TIMEFRAMES stdtfs[] =
   {
      PERIOD_M1,  // =1 (1)
      PERIOD_M2,  // =2 (2)
      ...
      PERIOD_W1,  // =32769 (8001)
      PERIOD_MN1, // =49153 (C001)
   };
   const int x = ArrayBsearch(stdtfs, tf == PERIOD_CURRENT ? _Period : tf);
   const int needle = x + step;
   if(needle >= 0 && needle < ArraySize(stdtfs))
   {
      return stdtfs[needle];
   }
   return tf;
}
```

Esta es la principal función de trabajo *SetupSubChart*.

```
long SetupSubChart(const int n, const int dx, const int dy,
   ENUM_TIMEFRAMES tf = PERIOD_CURRENT, const string symbol = NULL)
{
   // create an object
   const string name = Prefix + "Chart-"
      + (symbol == NULL ? _Symbol : symbol) + PeriodToString(tf);
   ObjectCreate(0, name, OBJ_CHART, ChartWindowFind(), 0, 0);

   // anchor to the top right corner of the subwindow
   ObjectSetInteger(0, name, OBJPROP_CORNER, CORNER_RIGHT_UPPER);
   // position and size
   ObjectSetInteger(0, name, OBJPROP_XSIZE, dx);
   ObjectSetInteger(0, name, OBJPROP_YSIZE, dy);
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, (n + 1) * dx);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, 0);

   // specific chart settings
   if(symbol != NULL)
   {
      ObjectSetString(0, name, OBJPROP_SYMBOL, symbol);
   }

   if(tf != PERIOD_CURRENT)
   {
      ObjectSetInteger(0, name, OBJPROP_PERIOD, tf);
   }
   // disable the display of lines
   ObjectSetInteger(0, name, OBJPROP_DATE_SCALE, false);
   ObjectSetInteger(0, name, OBJPROP_PRICE_SCALE, false);
   // add the MA indicator to the object by its id just for demo
   const long id = ObjectGetInteger(0, name, OBJPROP_CHART_ID);
   ChartIndicatorAdd(id, 0, iMA(NULL, tf, 10, 0, MODE_EMA, PRICE_CLOSE));
   return id;
}
```

Para un objeto gráfico, el punto de anclaje siempre se fija en la esquina superior izquierda del objeto, por lo que cuando se ancla a la esquina derecha de la ventana, es necesario añadir la anchura del objeto (esto se hace mediante +1 en la expresión*(n+1)*dx* para OBJPROP_XDISTANCE).

En la siguiente captura de pantalla se muestra el resultado del indicador en el gráfico XAUUSD,H1:

Dos objetos gráficos en la subventana del indicador

Como podemos ver, los minigráficos muestran los marcos temporales M30 y H2.

Es importante tener en cuenta que se pueden añadir indicadores a objetos gráficos y aplicar plantillas tpl, incluyendo aquellos con Asesores Expertos. No obstante, no se pueden crear objetos dentro de objetos gráficos.

> Cuando el objeto gráfico está oculto debido a una visualización deshabilitada en el marco temporal actual o en todos los marcos temporales, la propiedad CHART_WINDOW_IS_VISIBLE para el gráfico interno sigue devolviendo *true*.

5.8.29 Mover objetos

Para mover objetos en coordenadas de tiempo/precio, puede utilizar no sólo las funciones *ObjectSet* de cambio de propiedades, sino también la función especial *ObjectMove*, que cambia las coordenadas del punto de anclaje especificado del objeto.

bool ObjectMove(long chartId, const string name, int index, datetime time, double price)

El parámetro *chartId* establece el ID del gráfico (0 es para el gráfico actual). El nombre del objeto se pasa en el parámetro *name*. Las coordenadas y el índice del punto de anclaje se especifican en los parámetros *index*, *time* y *price*, respectivamente.

La función utiliza una llamada asíncrona, es decir, envía una orden a la cola de eventos del gráfico y no espera al movimiento en sí.

La función devuelve una indicación de si la orden se ha puesto en cola correctamente (en este caso, el resultado es *true*). La posición real del objeto debe conocerse mediante llamadas a las funciones de *ObjectGet*.

En el indicador *ObjectHighLowFibo.mq5*modificamos la función *DrawFibo* de manera que permita *ObjectMove*. En lugar de dos llamadas a las funciones *ObjectSet* en el bucle a través de los puntos de anclaje, ahora tenemos una llamada a *ObjectMove*:

```
bool DrawFibo(const string name, const datetime &t[], const double &p[],
   const color clr)
{
   ...
   for(int i = 0; i < ArraySize(t); ++i)
   {
      // was:
      // ObjectSetInteger(0, name, OBJPROP_TIME, i, t[i]);
      // ObjectSetDouble(0, name, OBJPROP_PRICE, i, p[i]);
      // became:
      ObjectMove(0, name, i, t[i], p[i]);
   }
   ...
}
```

Tiene sentido aplicar la función *ObjectMove* cuando cambian las dos coordenadas del punto de anclaje. En algunos casos, sólo una coordenada tiene efecto (por ejemplo, en los canales de desviación estándar y regresión lineal en los puntos de anclaje, sólo son importantes las fechas/horas de inicio y fin, y los canales calculan automáticamente el valor del precio en estos puntos). En estos casos, una única llamada a la función *ObjectSet* es más adecuada que *ObjectMove*.

5.8.30 Obtener hora o precio en puntos de línea especificados

Muchos objetos gráficos incluyen una o varias líneas rectas. MQL5 permite interpolar y extrapolar puntos en estas líneas y obtener otra coordenada a partir de una coordenada; por ejemplo, precio por tiempo o tiempo por precio.

La interpolación está siempre disponible: funciona «dentro» del objeto, es decir, entre puntos de anclaje. La extrapolación fuera de un objeto sólo es posible si la propiedad de rayo en la dirección correspondiente está activada para él (véase Propiedades de los rayos para objetos con líneas rectas).

La función *ObjectGetValueByTime* devuelve el valor del precio para el tiempo especificado. La función *ObjectGetTimeByValue* devuelve el valor temporal del precio especificado

double ObjectGetValueByTime(long chartId, const string name, datetime time, int line)
datetime ObjectGetTimeByValue(long chartId, const string name, double value, int line)

Los cálculos se realizan para un objeto denominado *name* en el gráfico con *chartId*. Los parámetros *time* y *value* especifican una coordenada conocida para la que debe calcularse la incógnita. Dado que un objeto puede tener varias líneas, varios valores corresponderán a una coordenada, por lo que es necesario especificar el número de línea en el parámetro *line*.

La función devuelve el precio o valor temporal para la proyección del punto con la coordenada inicial especificada respecto a la línea.

En caso de error, se devolverá 0 y el código de error se escribirá en *_LastError*. Por ejemplo, intentar extrapolar un valor de línea con la propiedad de haz desactivada genera un error OBJECT_GETVALUE_FAILED (4205).

Las funciones son aplicables a los siguientes objetos:

- Línea de tendencia (OBJ_TREND)
- Línea de tendencia por ángulo (OBJ_TRENDBYANGLE)
- Línea de Gann (OBJ_GANNLINE)
- Canal equidistante (OBJ_CHANNEL), 2 líneas
- Canal de regresión lineal (OBJ_REGRESSION); 3 líneas
- Canal de desviación estándar (OBJ_STDDEVCHANNEL); 3 líneas
- Línea de flecha (OBJ_ARROWED_LINE)

Comprobemos el funcionamiento de la función utilizando un indicador sin búfer *ObjectChannels.mq5*. Crea dos objetos con canales de desviación estándar y regresión lineal, tras lo cual solicita y muestra en el comentario el precio de las líneas superior e inferior en barras futuras. Para el canal de desviación estándar, la propiedad OBJPROP_RAY_RIGHT está activada, pero para el canal de regresión no lo está (intencionadamente). En este sentido, no se recibirán valores del segundo canal y siempre se mostrarán ceros en la pantalla para él.

A medida que se formen nuevas barras, los canales se desplazarán automáticamente hacia la derecha. La longitud de los canales se ajusta en el parámetro de entrada *WorkPeriod* (10 barras por defecto).

```
input int WorkPeriod = 10;

const string Prefix = "ObjChnl-";
const string ObjStdDev = Prefix + "StdDev";
const string ObjRegr = Prefix + "Regr";

void OnInit()
{
   CreateObjects();
   UpdateObjects();
}
```

La función *CreateObjects* crea 2 canales y realiza los ajustes iniciales para ellos.

```
void CreateObjects()
{
   ObjectCreate(0, ObjStdDev, OBJ_STDDEVCHANNEL, 0, 0, 0);
   ObjectCreate(0, ObjRegr, OBJ_REGRESSION, 0, 0, 0);
   ObjectSetInteger(0, ObjStdDev, OBJPROP_COLOR, clrBlue);
   ObjectSetInteger(0, ObjStdDev, OBJPROP_RAY_RIGHT, true);
   ObjectSetInteger(0, ObjRegr, OBJPROP_COLOR, clrRed);
   // NB: ray is not enabled for the regression channel (intentionally)
}
```

La función *UpdateObjects* desplaza los canales a las últimas barras de *WorkPeriod*.

```
void UpdateObjects()
{
   const datetime t0 = iTime(NULL, 0, WorkPeriod);
   const datetime t1 = iTime(NULL, 0, 0);

   // we don't use ObjectMove because channels work
   // only with time coordinate (price is calculated automatically)
   ObjectSetInteger(0, ObjStdDev, OBJPROP_TIME, 0, t0);
   ObjectSetInteger(0, ObjStdDev, OBJPROP_TIME, 1, t1);
   ObjectSetInteger(0, ObjRegr, OBJPROP_TIME, 0, t0);
   ObjectSetInteger(0, ObjRegr, OBJPROP_TIME, 1, t1);
}
```

En el manejador *OnCalculate*, actualizamos la posición de los canales en las nuevas barras, y en cada tick, llamamos a *DisplayObjectData* para obtener la extrapolación del precio y mostrarla como un comentario.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
   static datetime now = 0;
   if(now != iTime(NULL, 0, 0))
   {
      UpdateObjects();
      now = iTime(NULL, 0, 0);
   }

   DisplayObjectData();

   return rates_total;
}
```

En la función *DisplayObjectData* encontraremos los precios en los puntos de anclaje de la línea media (OBJPROP_PRICE). Además, utilizando *ObjectGetValueByTime*, solicitaremos valores de precios para las líneas superior e inferior del canal a través de las barras *WorkPeriod* en el futuro.

```
void DisplayObjectData()
{
   const double p0 = ObjectGetDouble(0, ObjStdDev, OBJPROP_PRICE, 0);
   const double p1 = ObjectGetDouble(0, ObjStdDev, OBJPROP_PRICE, 1);

   // the following equalities are always true due to the channel calculation algorit
   // - the middle lines of both channels are the same,
   // - anchor points always lie on the middle line,
   // ObjectGetValueByTime(0, ObjStdDev, iTime(NULL, 0, 0), 0) == p1
   // ObjectGetValueByTime(0, ObjRegr, iTime(NULL, 0, 0), 0) == p1

   // trying to extrapolate future prices from the upper and lower lines
   const double d1 = ObjectGetValueByTime(0, ObjStdDev, iTime(NULL, 0, 0)
      + WorkPeriod * PeriodSeconds(), 1);
   const double d2 = ObjectGetValueByTime(0, ObjStdDev, iTime(NULL, 0, 0)
      + WorkPeriod * PeriodSeconds(), 2);

   const double r1 = ObjectGetValueByTime(0, ObjRegr, iTime(NULL, 0, 0)
      + WorkPeriod * PeriodSeconds(), 1);
   const double r2 = ObjectGetValueByTime(0, ObjRegr, iTime(NULL, 0, 0)
      + WorkPeriod * PeriodSeconds(), 2);

   // display all received prices in a comment
   Comment(StringFormat("%.*f %.*f\ndev: up=%.*f dn=%.*f\nreg: up=%.*f dn=%.*f",
      _Digits, p0, _Digits, p1,
      _Digits, d1, _Digits, d2,
      _Digits, r1, _Digits, r2));
}
```

Es importante observar que, debido a que la propiedad de rayo no está habilitada para el canal de regresión, siempre da ceros en el futuro (aunque si preguntáramos por precios dentro del periodo de tiempo del canal, obtendríamos los valores correctos).

Canales y valores de precios en los puntos de sus líneas

Aquí, para los canales que tienen 10 barras, la extrapolación se hace también sobre 10 barras por delante, lo que da los valores futuros que se muestran en la línea con «dev:», que corresponde aproximadamente al borde derecho de la ventana.

5.9 Eventos interactivos en gráficos

Los gráficos de MetaTrader 5 no sólo proporcionan una representación visual de los datos y son el entorno de ejecución de los programas MQL, sino que también admiten el mecanismo de eventos interactivos, que permite a los programas responder a las acciones del usuario y de otros programas. Esto se hace mediante un tipo de evento especial, *OnChartEvent*, del que ya hablamos en la Visión general de las funciones de gestión de eventos.

Cualquier indicador o Asesor Experto puede recibir este tipo de eventos siempre que en el código se describa la función de procesamiento de eventos del mismo nombre con una firma predefinida. En algunos de los ejemplos de indicadores que hemos visto antes ya tuvimos que aprovechar esta oportunidad. En este capítulo examinaremos en detalle el sistema de eventos.

El evento *OnChartEvent* es generado por el terminal cliente durante las siguientes manipulaciones del gráfico realizadas por el usuario:

- Cambiar los ajustes o el tamaño del gráfico
- Pulsaciones de teclas cuando la ventana del gráfico está enfocada
- Movimiento del cursor del ratón
- Clics del ratón en el gráfico
- Clics del ratón en objetos gráficos

- Creación de un objeto gráfico
- Eliminación de un objeto gráfico
- Desplazamiento de un objeto gráfico con el ratón
- Finalización de la edición de la prueba en el campo de entrada del objeto OBJ_EDIT

El programa MQL recibe los eventos listados sólo del gráfico en el que se está ejecutando. Al igual que otros tipos de eventos, se añaden a una cola. A continuación, todos los eventos se procesan uno a uno por orden de llegada. Si ya existe un evento *OnChartEvent* de un tipo determinado en la cola del programa MQL o se está procesando, un nuevo evento del mismo tipo no se pone en cola (se descarta).

Algunos tipos de eventos están siempre activos, mientras que otros están desactivados por defecto y deben activarse explícitamente configurando las propiedades de gráfico adecuadas mediante la llamada *ChartSetInteger*. Tales eventos desactivados incluyen, en concreto, los movimientos del ratón y el desplazamiento de la rueda del ratón. Todos ellos se caracterizan por el hecho de que pueden generar flujos masivos de eventos y, para ahorrar recursos, se recomienda habilitarlos sólo cuando sea necesario.

Además de los eventos estándar, existe el concepto de «eventos personalizados». El significado y contenido de los parámetros para tales eventos son asignados e interpretados por el propio programa MQL (uno o varios, si hablamos de la interacción de un complejo de programas). Un programa MQL puede enviar «eventos de usuario» a un gráfico (incluso a otro) utilizando la función *EventChartCustom*. Este tipo de eventos también se gestionan mediante la función *OnChartEvent*.

Si hay varios programas MQL en el gráfico con el manejador *OnChartEvent*, todos recibirán el mismo flujo de eventos.

> Todos los programas MQL se ejecutan en hilos distintos del hilo principal de la aplicación. El hilo principal del terminal se encarga de procesar todos los mensajes del sistema Windows y, como resultado de este procesamiento, genera a su vez mensajes de Windows para su propia aplicación. Por ejemplo, arrastrar un gráfico con el ratón genera varios mensajes de sistema WM_MOUSE_MOVE (en términos de la API de Windows) para el posterior dibujo de la ventana de la aplicación, y también envía mensajes internos a los Asesores Expertos e indicadores lanzados en este gráfico. En este caso puede darse la situación de que el hilo principal de la aplicación aún no haya conseguido procesar el mensaje del sistema sobre el redibujado de la ventana WM_PAINT (y por tanto aún no haya cambiado la apariencia del gráfico), y el Asesor Experto o indicador ya haya recibido un evento sobre el movimiento del cursor del ratón. A continuación, la propiedad del gráfico CHART_FIRST_VISIBLE_BAR sólo se modificará una vez dibujado el gráfico.

Dado que de los dos tipos de programas MQL interactivos hasta ahora sólo hemos estudiado los indicadores, todos los ejemplos de este capítulo se construirán sobre la base de indicadores. El segundo tipo, los Asesores Expertos, se describirá en la siguiente Parte del libro. No obstante, los principios de trabajo con eventos en ellos coinciden completamente con los que aquí se presentan.

5.9.1 Función de gestión de eventos OnChartEvent

Un indicador o Asesor Experto puede recibir eventos interactivos desde el terminal si el código contiene la función *OnChartEvent* con el siguiente prototipo.

```
void OnChartEvent(const int event, const long &lparam, const double &dparam, const string &sparam)
```

Esta función será llamada por el terminal en respuesta a las acciones del usuario o en caso de generar un «evento de usuario» utilizando *EventChartCustom*.

En el parámetro *event*, el identificador del evento (su tipo) se pasa como uno de los valores de la enumeración ENUM_CHART_EVENT (véase la tabla).

Identificador	Descripción
CHARTEVENT_KEYDOWN	Acción del teclado
CHARTEVENT_MOUSE_MOVE	Mover el ratón y hacer clic en los botones del ratón (si la propiedad CHART_EVENT_MOUSE_MOVE está establecida para el gráfico)
CHARTEVENT_MOUSE_WHEEL	Hacer clic o desplazar la rueda del ratón (si la propiedad CHART_EVENT_MOUSE_WHEEL está establecida para el gráfico)
CHARTEVENT_CLICK	Clic con el ratón en el gráfico
CHARTEVENT_OBJECT_CREATE	Crear un objeto gráfico (si la propiedad CHART_EVENT_OBJECT_CREATE está establecida para el gráfico)
CHARTEVENT_OBJECT_CHANGE	Modificar un objeto gráfico a través del cuadro de diálogo de propiedades
CHARTEVENT_OBJECT_DELETE	Eliminar un objeto gráfico (si la propiedad CHART_EVENT_OBJECT_DELETE está establecida para el gráfico)
CHARTEVENT_OBJECT_CLICK	Clic con el ratón sobre un objeto gráfico
CHARTEVENT_OBJECT_DRAG	Arrastrar un objeto gráfico
CHARTEVENT_OBJECT_ENDEDIT	Finalizar la edición de texto en el objeto gráfico «campo de entrada»
CHARTEVENT_CHART_CHANGE	Modificar las dimensiones o propiedades del gráfico (mediante el cuadro de diálogo de propiedades, la barra de herramientas o el menú contextual)
CHARTEVENT_CUSTOM	El número inicial del evento del rango de eventos personalizados
CHARTEVENT_CUSTOM_LAST	El número final del evento del rango de eventos personalizados

Los parámetros *lparam*, *dparam* y *sparam* se utilizan de forma diferente en función del tipo de evento. En general, podemos decir que contienen datos adicionales necesarios para procesar un evento concreto. En las secciones siguientes se ofrece información detallada sobre cada tipo.

¡Atención! La función *OnChartEvent* sólo se utiliza para los indicadores y Asesores Expertos que se representan directamente en el gráfico. Si cualquier indicador se crea mediante programación utilizando *iCustom* o *IndicatorCreate*, los eventos de *OnChartEvent* no se trasladarán a él.

Además, el manejador *OnChartEvent* no se llama en el probador, ni siquiera en modo visual.

Para la primera demostración del manejador *OnChartEvent*, consideremos un indicador sin búfer *EventAll.mq5* que intercepta y registra todos los eventos.

```
void OnChartEvent(const int id,
   const long &lparam, const double &dparam, const string &sparam)
{
   ENUM_CHART_EVENT evt = (ENUM_CHART_EVENT)id;
   PrintFormat("%s %lld %f '%s'", EnumToString(evt), lparam, dparam, sparam);
}
```

De manera predeterminada, se pueden generar todo tipo de eventos en el gráfico, excepto cuatro eventos masivos que, como se indica en la tabla anterior, están habilitados por las propiedades especiales del gráfico. En la próxima sección complementaremos el indicador con ajustes para incluir determinados tipos según las preferencias.

Ejecute el indicador en un gráfico con objetos existentes o cree objetos mientras se ejecuta el indicador.

Cambie el tamaño o la configuración del gráfico, haga clic con el ratón y edite las propiedades de los objetos. En el registro aparecerán las siguientes entradas:

```
CHARTEVENT_CHART_CHANGE 0 0.000000 ''
CHARTEVENT_CLICK 149 144.000000 ''
CHARTEVENT_OBJECT_CLICK 112 105.000000 'Daily Rectangle 53404'
CHARTEVENT_CLICK 112 105.000000 ''
CHARTEVENT_KEYDOWN 46 1.000000 '339'
CHARTEVENT_CLICK 13 252.000000 ''
CHARTEVENT_OBJECT_DRAG 0 0.000000 'Daily Button 61349'
CHARTEVENT_OBJECT_CLICK 145 104.000000 'Daily Button 61349'
CHARTEVENT_CLICK 145 104.000000 ''
CHARTEVENT_CHART_CHANGE 0 0.000000 ''
CHARTEVENT_OBJECT_DRAG 0 0.000000 'Daily Vertical Line 22641'
CHARTEVENT_OBJECT_DRAG 0 0.000000 'Daily Vertical Line 22641'
CHARTEVENT_OBJECT_CLICK 177 206.000000 'Daily Vertical Line 22641'
CHARTEVENT_CLICK 177 206.000000 ''
CHARTEVENT_OBJECT_CHANGE 0 0.000000 'Daily Rectangle 37930'
CHARTEVENT_CHART_CHANGE 0 0.000000 ''
CHARTEVENT_CLICK 152 118.000000 ''
```

Aquí vemos eventos de varios tipos, el significado de sus parámetros quedará claro después de leer las siguientes secciones.

5.9.2 Propiedades de gráficos relacionados con eventos

Cuatro tipos de eventos son capaces de generar muchos mensajes y, por tanto, están desactivados por defecto. Para activarlos o desactivarlos más adelante, establezca las propiedades del gráfico adecuadas mediante la función *ChartSetInteger*. Todas las propiedades son de tipo booleano: *true* significa activado, y *false* significa desactivado.

Identificador	Descripción
CHART_EVENT_MOUSE_WHEEL	Envío de mensajes CHARTEVENT_MOUSE_WHEEL sobre eventos de rueda de ratón al gráfico
CHART_EVENT_MOUSE_MOVE	Envío de mensajes CHARTEVENT_MOUSE_MOVE sobre movimientos del ratón al gráfico
CHART_EVENT_OBJECT_CREATE	Envío de mensajes CHARTEVENT_OBJECT_CREATE sobre la creación de objetos gráficos al gráfico
CHART_EVENT_OBJECT_DELETE	Envío de mensajes CHARTEVENT_OBJECT_DELETE sobre la eliminación de objetos gráficos al gráfico

Si cualquier programa MQL cambia una de estas propiedades, afecta a todos los demás programas que se ejecutan en el mismo gráfico y permanece en vigor incluso después de que el programa original termine.

De manera predeterminada, todas las propiedades tienen el valor *false*.

Vamos a complementar el indicador *EventAll.mq5* del apartado anterior con cuatro variables de entrada que permiten habilitar cualquiera de estos tipos de eventos (además del resto que no se pueden deshabilitar). Además, describiremos cuatro variables auxiliares para poder restablecer la configuración del gráfico después de borrar el indicador.

```
input bool ShowMouseMove = false;
input bool ShowMouseWheel = false;
input bool ShowObjectCreate = false;
input bool ShowObjectDelete = false;

bool mouseMove, mouseWheel, objectCreate, objectDelete;
```

Al iniciarse, recuerde los valores actuales de las propiedades y, a continuación, aplique los ajustes seleccionados por el usuario.

```
void OnInit()
{
   mouseMove = PRTF(ChartGetInteger(0, CHART_EVENT_MOUSE_MOVE));
   mouseWheel = PRTF(ChartGetInteger(0, CHART_EVENT_MOUSE_WHEEL));
   objectCreate = PRTF(ChartGetInteger(0, CHART_EVENT_OBJECT_CREATE));
   objectDelete = PRTF(ChartGetInteger(0, CHART_EVENT_OBJECT_DELETE));

   ChartSetInteger(0, CHART_EVENT_MOUSE_MOVE, ShowMouseMove);
   ChartSetInteger(0, CHART_EVENT_MOUSE_WHEEL, ShowMouseWheel);
   ChartSetInteger(0, CHART_EVENT_OBJECT_CREATE, ShowObjectCreate);
   ChartSetInteger(0, CHART_EVENT_OBJECT_DELETE, ShowObjectDelete);
}
```

Las propiedades se restauran en el manejador *OnDeinit*.

```
void OnDeinit(const int)
{
   ChartSetInteger(0, CHART_EVENT_MOUSE_MOVE, mouseMove);
   ChartSetInteger(0, CHART_EVENT_MOUSE_WHEEL, mouseWheel);
   ChartSetInteger(0, CHART_EVENT_OBJECT_CREATE, objectCreate);
   ChartSetInteger(0, CHART_EVENT_OBJECT_DELETE, objectDelete);
}
```

Ejecute el indicador con los nuevos tipos de eventos activados. Prepárese para recibir muchos mensajes de movimiento del ratón. A continuación se muestra un fragmento del registro:

```
CHARTEVENT_MOUSE_WHEEL 5308557 -120.000000 ''
CHARTEVENT_CHART_CHANGE 0 0.000000 ''
CHARTEVENT_MOUSE_WHEEL 5308557 -120.000000 ''
CHARTEVENT_CHART_CHANGE 0 0.000000 ''
CHARTEVENT_MOUSE_MOVE 141 81.000000 '2'
CHARTEVENT_MOUSE_MOVE 141 81.000000 '0'
...
CHARTEVENT_OBJECT_CREATE 0 0.000000 'Daily Rectangle 37664'
CHARTEVENT_MOUSE_MOVE 323 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 322 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 321 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 320 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 318 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 316 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 314 146.000000 '0'
CHARTEVENT_MOUSE_MOVE 314 145.000000 '0'
...
CHARTEVENT_OBJECT_DELETE 0 0.000000 'Daily Rectangle 37664'
CHARTEVENT_KEYDOWN 46 1.000000 '339
```

A continuación, en las secciones correspondientes, revelaremos los datos específicos de cada tipo de evento.

5.9.3 Evento de cambio de gráfico

Al cambiar el tamaño del gráfico, los modos de visualización de precios, la escala u otros parámetros, el terminal envía el evento CHARTEVENT_CHART_CHANGE, que no tiene parámetros. El programa MQL debe averiguar los cambios por sí mismo utilizando llamadas a la función *ChartGet*.

Ya hemos utilizado este evento en el ejemplo de *ChartModeMonitor.mq5* en la sección sobre Modos de visualización de gráficos. Veamos ahora otro ejemplo:

Como sabe, MetaTrader 5 permite guardar la captura de pantalla del gráfico actual en un archivo de un tamaño especificado (el comando *Guardar como imagen* del menú contextual). No obstante, este método de obtener una captura de pantalla no es adecuado para todos los casos. En concreto, si necesita una imagen con una información sobre herramientas, o cuando un objeto del tipo campo de entrada está activo (cuando el texto está seleccionado dentro del campo y el cursor de texto está visible), el comando estándar no le ayudará, ya que vuelve a formar la imagen del gráfico sin tener en cuenta estos y algunos otros matices del estado actual de la ventana.

La única alternativa para obtener una copia exacta de la ventana es utilizar medios externos al terminal (por ejemplo, la tecla *PrtSc* a través del portapapeles de Windows), pero este método no garantiza el

tamaño de ventana requerido. Para no tener que seleccionar el tamaño por el método de ensayo y error o con algunos programas adicionales, crearemos un indicador *EventWindowSizer.mq5*, que seguirá la configuración de tamaño del usuario sobre la marcha y mostrará el valor actual en un comentario.

Todo el trabajo se realiza en el manejador *OnChartEvent*, comenzando con la comprobación del ID del evento para CHARTEVENT_CHART_CHANGE. Las dimensiones de la ventana en píxeles pueden obtenerse utilizando las propiedades CHART_WIDTH_IN_PIXELS y CHART_HEIGHT_IN_PIXELS. Sin embargo, éstas devuelven las dimensiones sin tener en cuenta los bordes, y normalmente se desean bordes para una captura de pantalla. Por lo tanto, mostraremos en el comentario no sólo los valores de la propiedad (marcados con la palabra «Pantalla»), sino también los valores corregidos (marcados con la palabra «Imagen»): deben añadirse 2 píxeles en anchura y 1 píxel en vertical (estas son las características de la renderización de ventanas en el terminal).

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   if(id == CHARTEVENT_CHART_CHANGE)
   {
      const int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
      const int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS);
      // "Raw" sizes "as is" are displayed with the "Screen" mark,
      // correction for (-2,-1) is needed to include frames - it is displayed with th
      // correction for (-54,-22) is needed to include scales - it is displayed with
      Comment(StringFormat("Screen: %d x %d\nPicture: %d x %d\nIncluding scales: %d x
         w, h, w + 2, h + 1, w + 2 + 54, h + 1 + 22));
   }
}
```

Además, los valores obtenidos no tienen en cuenta las escalas de hora y precio. Si éstas también deben tenerse en cuenta en el tamaño de la captura de pantalla, entonces también debe hacerse un ajuste para su tamaño. Por desgracia, la API de MQL5 no proporciona ninguna forma de averiguar estos tamaños, por lo que sólo podemos determinarlos empíricamente: para la configuración estándar de fuentes de Windows, la anchura de la escala de precios es de 54 píxeles, y la altura de la escala de tiempo es de 22 píxeles. Estas constantes pueden diferir para su versión de Windows, por lo que debe editarlas, o establecerlas utilizando parámetros de entrada.

Después de ejecutar el indicador en un gráfico, pruebe a cambiar el tamaño de la ventana y vea cómo cambian los números en el comentario.

Ventana s

Captura de pantalla de la ventana con información sobre herramientas y tamaños actuales en el comentario

5.9.4 Eventos de teclado

Los programas MQL pueden recibir mensajes sobre pulsaciones de teclas desde el terminal mediante el procesamiento en la función *OnChartEvent* de los eventos CHARTEVENT_KEYDOWN.

Es importante tener en cuenta que los eventos sólo se generan en el gráfico activo, y sólo cuando éste tiene el foco de entrada.

En Windows, el foco es la selección lógica y visual de una ventana concreta con la que el usuario está interactuando en ese momento. Por regla general, el foco se desplaza mediante un clic del ratón o mediante atajos de teclado especiales (*Tab*, *Ctrl+Tab*), lo que hace que se resalte la ventana seleccionada. Por ejemplo, aparecerá un cursor de texto en el campo de entrada, la línea actual se coloreará en la lista con un color alternativo, etc.

Efectos visuales similares se aprecian en el terminal, en concreto cuando se pone el foco en alguna de las ventanas *Market Watch* o *Data Window*, o en el registro de Expertos. Sin embargo, la situación es algo diferente con las ventanas de gráficos. No siempre es posible distinguir por signos externos si el gráfico visible en primer plano tiene o no el foco de entrada. Se garantiza que pueda cambiar el foco, como ya se ha mencionado, haciendo clic en el gráfico deseado (en el gráfico, y no en el título de la ventana o en su marco) o utilizando teclas de acceso rápido:

- Alt+W abre una ventana con una lista de gráficos, en la que puede seleccionar uno.
- Ctrl+F6 cambia al gráfico siguiente (en la lista de ventanas, donde el orden corresponde, por regla general, al orden de las pestañas).
- Crtl+Mayús+F6 cambia al gráfico anterior.

La lista completa de las teclas de acceso rápido de MetaTrader 5 se encuentra en la documentación. Tenga en cuenta que algunas combinaciones no se ajustan a las recomendaciones generales de Microsoft (por ejemplo, F10 abre la ventana de cotizaciones, pero no activa el menú principal).

Los parámetros del evento CHARTEVENT_KEYDOWN contienen la siguiente información:

- *lparam* - código de la tecla pulsada
- *dparam* - el número de pulsaciones generadas durante el tiempo que se mantuvo pulsada
- *sparam* - una máscara de bits que describe el estado de las teclas del teclado, convertida en una cadena

Bits	Descripción
0–7	Código de escaneado de la tecla (depende del hardware, OEM)
8	Atributo de tecla de teclado ampliado
9–12	Para fines de servicio de Windows (no utilizar)
13	Estado de la tecla *Alt* (1 - pulsada, 0 - liberada), no disponible (véase más abajo)
14	Estado anterior de la tecla (1 - pulsada, 0 - liberada)
15	Estado de tecla cambiado (1 si está liberada, 0 si está pulsada)

En realidad, el estado de la clave *Alt* no está disponible, ya que está interceptado por el terminal y este bit es siempre 0. El bit 15 también es siempre igual a 0 debido al contexto de activación de este evento: sólo las pulsaciones de teclas se pasan al programa MQL, no las liberaciones de teclas.

El atributo del teclado extendido (bit 8) se establece, por ejemplo, para las teclas del bloque numérico (en los portátiles suele activarse con *Fn*), teclas como *NumLock*, *ScrollLock*, *Ctrl* derecha (en contraposición a la izquierda, *Ctrl* principal), etc. Obtenga más información al respecto en la documentación de Windows.

La primera vez que se pulse cualquier tecla que no sea del sistema, el bit 14 será 0. Si mantiene pulsada la tecla, las siguientes repeticiones del evento generadas automáticamente tendrán 1 en este bit.

La siguiente estructura ayudará a garantizar que la descripción de los bits es correcta:

```
struct KeyState
{
   uchar scancode;
   bool extended;
   bool altPressed;
   bool previousState;
   bool transitionState;

   KeyState() { }
   KeyState(const ushort keymask)
   {
      this = keymask; // use operator overload=
   }
   void operator=(const ushort keymask)
   {
      scancode = (uchar)(0xFF & keymask);
      extended = 0x100 & keymask;
      altPressed = 0x2000 & keymask;
      previousState = 0x4000 & keymask;
      transitionState = 0x8000 & keymask;
   }
};
```

En un programa MQL, se puede utilizar así:

```
void OnChartEvent(const int id,
                  const long &lparam,
                  const double &dparam,
                  const string &sparam)
{
   if(id == CHARTEVENT_KEYDOWN)
   {
      PrintFormat("%lld %lld %4llX", lparam, (ulong)dparam, (ushort)sparam);
      KeyState state[1];
      state[0] =(ushort)sparam;
      ArrayPrint(state);
   }
}
```

A efectos prácticos, es más cómodo extraer atributos de bits de la máscara de claves mediante macros.

```
#define KEY_SCANCODE(SPARAM) ((uchar)(((ushort)SPARAM) & 0xFF))
#define KEY_EXTENDED(SPARAM) ((bool)(((ushort)SPARAM) & 0x100))
#define KEY_PREVIOUS(SPARAM) ((bool)(((ushort)SPARAM) & 0x4000))
```

Puede ejecutar el indicador *EventAll.mq5* de la sección Propiedades de gráficos relacionados con eventos en el gráfico y ver qué valores de los parámetros se mostrarán en el registro cuando se pulsen determinadas teclas.

Es importante tener en cuenta que el código que aparece en *lparam* es uno de los códigos de las teclas del teclado virtual. Su lista se puede ver en el archivo *MQL5/Include/VirtualKeys.mqh*, que viene con MetaTrader 5. Por ejemplo, he aquí algunas de ellas:

```
#define VK_SPACE          0x20
#define VK_PRIOR          0x21
#define VK_NEXT           0x22
#define VK_END            0x23
#define VK_HOME           0x24
#define VK_LEFT           0x25
#define VK_UP             0x26
#define VK_RIGHT          0x27
#define VK_DOWN           0x28
...
#define VK_INSERT         0x2D
#define VK_DELETE         0x2E
...
// VK_0 - VK_9 ASCII codes of characters '0' - '9' (0x30 - 0x39)
// VK_A - VK_Z ASCII codes of characters 'A' - 'Z' (0x41 - 0x5A)
```

Los códigos se denominan virtuales porque las teclas correspondientes pueden estar situadas de forma diferente en los distintos teclados, o incluso implementarse mediante la pulsación conjunta de teclas auxiliares (como *Fn* en portátiles). Además, la virtualidad tiene otra cara: una misma clave puede generar símbolos o acciones de control diferentes. Por ejemplo, la misma tecla puede denotar letras diferentes en disposiciones lingüísticas distintas. Además, cada una de las teclas de letras puede generar una letra mayúscula o minúscula, según el modo de *CapsLock* y el estado de las teclas *Shift*.

A este respecto, para obtener un carácter a partir de un código de tecla virtual, la API de MQL5 dispone de la función especial *TranslateKey*.

short TranslateKey(int key)

La función devuelve un carácter Unicode basado en el código de tecla virtual pasado, dado el idioma de entrada actual y el estado de las teclas de control.

En caso de error, se devolverá el valor -1. Puede producirse un error si el código no coincide con el carácter correcto; por ejemplo, al intentar obtener un carácter para la tecla *Shift*.

Recordemos que además del código recibido de la tecla pulsada, un programa MQL puede además Comprobar el estado del teclado en términos de modos y teclas de control. Por cierto: las constantes de la forma TERMINAL_KEYSTATE_XXX, pasadas como parámetro a la función *TerminalInfoInteger*, se basan en el principio de 1000 + código de clave virtual. Por ejemplo, TERMINAL_KEYSTATE_UP es 1038 porque VK_UP es 38 (0x26).

A la hora de planificar algoritmos que reaccionen a las pulsaciones de teclas, hay que tener en cuenta que el terminal puede interceptar muchas combinaciones de teclas, ya que están reservadas para realizar determinadas acciones (más arriba se ha facilitado el enlace a la documentación). En concreto, al pulsar la barra espaciadora se abre un campo para navegar rápidamente por el eje temporal. La API de MQL5 permite controlar en parte ese procesamiento de teclado integrado y desactivarlo si es necesario. Véase la sección sobre Control del ratón y del teclado.

El sencillo indicador sin búfer *EventTranslateKey.mq5* sirve para demostrar esta función. En su manejador *OnChartEvent* para los eventos CHARTEVENT_KEYDOWN se llama a *TranslateKey* para obtener un caracter *Unicode* válido. Si tiene éxito, el símbolo se añade a la cadena de mensajes que se muestra en el comentario del trazado. Al pulsar *Enter*, se inserta una nueva línea en el texto, y al pulsar *Backspace,* se borra el último carácter del final.

```
#include <VirtualKeys.mqh>

string message = "";

void OnChartEvent(const int id,
   const long &lparam, const double &dparam, const string &sparam)
{
   if(id == CHARTEVENT_KEYDOWN)
   {
      if(lparam == VK_RETURN)
      {
         message += "\n";
      }
      else if(lparam == VK_BACK)
      {
         StringSetLength(message, StringLen(message) - 1);
      }
      else
      {
         ResetLastError();
         const ushort c = TranslateKey((int)lparam);
         if(_LastError == 0)
         {
            message += ShortToString(c);
         }
      }
      Comment(message);
   }
}
```

Puede probar a introducir caracteres en distintos casos y en distintos idiomas.

Tenga cuidado. La función devuelve el valor con signo *short*, principalmente para poder devolver un código de error de -1. Sin embargo, el tipo de un carácter «ancho» de dos bytes se considera un entero sin signo, *ushort*. Si la variable receptora se declara como *ushort*, una comprobación utilizando -1 (por ejemplo, *c!=-1*) emitirá una advertencia del compilador de «desacuerdo de signo» (se requiere una conversión de tipos explícita), mientras que la otra (*c >= 0*) suele ser errónea, ya que siempre es igual a *true*.

Para poder insertar espacios entre las palabras del mensaje, la navegación rápida activada por la barra espaciadora está predesactivada en el manejador *OnInit*.

```
void OnInit()
{
   ChartSetInteger(0, CHART_QUICK_NAVIGATION, false);
}
```

Como ejemplo plenamente desarrollado del uso de eventos de teclado, considere la siguiente tarea de aplicación. Los usuarios de terminales saben que la escala de la ventana principal del gráfico puede modificarse de forma interactiva sin abrir el cuadro de diálogo de ajustes con el ratón: basta con pulsar el botón del ratón en la escala de precios y, sin soltarlo, desplazarse hacia arriba/abajo. Por desgracia, este método no funciona en las subventanas.

Las subventanas siempre se escalan automáticamente para ajustarse a todo el contenido, y para cambiar la escala hay que abrir un cuadro de diálogo e introducir los valores manualmente. A veces es necesario hacerlo si los indicadores de la subventana muestran «valores atípicos», es decir, lecturas individuales demasiado grandes que interfieren con el análisis del resto de los datos de tamaño normal (medio). Además, a veces conviene simplemente ampliar la imagen para abordar detalles más finos.

Para resolver este problema y permitir al usuario ajustar la escala de la subventana mediante pulsaciones de teclas, hemos implementado el indicador *SubScalermq5*. No tiene búferes y no muestra nada.

SubScaler debe ser el primer indicador de la subventana o, para decirlo en términos más estrictos, debe añadirse a la subventana antes de que se añada allí el indicador de trabajo que le interesa y cuya escala desea controlar. Para que *SubScaler* sea el primer indicador debe colocarse en el gráfico (en la ventana principal) y crear así una nueva subventana, en la que podrá añadir un indicador subordinado.

En el cuadro de diálogo de configuración del indicador de trabajo, es importante activar la opción *Inherit scale* (en la pestaña *Scale*).

Cuando ambos indicadores están funcionando en una subventana, puede utilizar las teclas de flecha *Up/Down* para acercar/alejar el zoom. Si se pulsa la tecla *Shift*, el rango de valores visible actualmente en el eje vertical se desplaza hacia arriba o hacia abajo.

Ampliar significa acercarse a los detalles («zoom de cámara»), por lo que algunos datos pueden quedar fuera de la ventana. Alejar significa que la imagen global se hace más pequeña («alejamiento de la cámara»).

Los parámetros de entrada establecidos son los siguientes:

- Máximo inicial - el límite superior de los datos durante la colocación inicial en el gráfico, +1000 por defecto.
- Mínimo inicial - el límite inferior de datos durante la colocación inicial en el gráfico, por defecto -1000.
- Factor de escala - paso con el que cambiará la escala al pulsar las teclas, valor en el rango [0.01 ... 0.5], por defecto 0.1.

Nos vemos obligados a pedirle al usuario el mínimo y el máximo porque *SubScaler* no puede conocer de antemano el rango de valores de trabajo de un indicador arbitrario de terceros, que se añadirá a continuación a la subventana.

Cuando se restaura el gráfico tras iniciar una nueva sesión de terminal o cuando se carga una plantilla tpl, *SubScaler* recoge la escala del estado anterior (guardado).

Veamos ahora la implementación de *SubScaler*.

Los ajustes anteriores se establecen en las variables de entrada correspondientes:

```
input double FixedMaximum = 1000;  // Initial Maximum
input double FixedMinimum = -1000; // Initial Minimum
input double _ScaleFactor = 0.1;   // Scale Factor [0.01 ... 0.5]
input bool Disabled = false;
```

Además, la variable *Disabled* permite desactivar temporalmente la respuesta del teclado para una instancia específica del indicador con el fin de configurar varias escalas diferentes en distintas subventanas (una por una).

Dado que las variables de entrada son de sólo lectura en MQL5, nos vemos obligados a declarar una variable más *ScaleFactor* para corregir el valor introducido dentro del rango permitido [0.01 ... 0.5].

```
double ScaleFactor;
```

El número de la subventana actual (*w*) y el número de indicadores en la misma (*n*) se almacenan en variables globales: todas ellas se rellenan en el manejador *OnInit*.

```
int w = -1, n = -1;

void OnInit()
{
  ScaleFactor = _ScaleFactor;
  if(ScaleFactor < 0.01 || ScaleFactor > 0.5)
  {
    PrintFormat("ScaleFactor %f is adjusted to default value 0.1,"
       " valid range is [0.01, 0.5]", ScaleFactor);
    ScaleFactor = 0.1;
  }
  w = ChartWindowFind();
  n = ChartIndicatorsTotal(0, w);
}
```

En la función *OnChartEvent* procesamos dos tipos de eventos: cambios en el gráfico y eventos de teclado. El evento CHARTEVENT_CHART_CHANGE es necesario para realizar un seguimiento de la adición del siguiente indicador a la subventana (indicador de trabajo que se va a escalar). Al mismo tiempo, solicitamos el rango actual de valores de la subventana (CHART_PRICE_MIN, CHART_PRICE_MAX) y determinamos si es degenerado, es decir, cuando tanto el máximo como el mínimo son iguales a cero. En este caso es necesario aplicar los límites iniciales especificados en los parámetros de entrada (*FixedMinimum*,*FixedMaximum*).

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   switch(id)
   {
   case CHARTEVENT_CHART_CHANGE:
      if(ChartIndicatorsTotal(0, w) > n)
      {
         n = ChartIndicatorsTotal(0, w);
         const double min = ChartGetDouble(0, CHART_PRICE_MIN, w);
         const double max = ChartGetDouble(0, CHART_PRICE_MAX, w);
         PrintFormat("Change: %f %f %d", min, max, n);
         if(min == 0 && max == 0)
         {
            IndicatorSetDouble(INDICATOR_MINIMUM, FixedMinimum);
            IndicatorSetDouble(INDICATOR_MAXIMUM, FixedMaximum);
         }
      }
      break;
   ...
   }
}
```

Cuando se recibe un evento de pulsación de teclado se llama a la función principal *Scale*, que recibe no sólo *lparam*, sino también el estado de la tecla *Shift* obtenido mediante referencia a *TerminalInfoInteger(TERMINAL_KEYSTATE_SHIFT)*.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
  switch(id)
  {
    case CHARTEVENT_KEYDOWN:
      if(!Disabled)
         Scale(lparam, TerminalInfoInteger(TERMINAL_KEYSTATE_SHIFT));
      break;
    ...
  }
}
```

Dentro de la función *Scale*, lo primero que hacemos es obtener el rango actual de valores en las variables *min* y *max*.

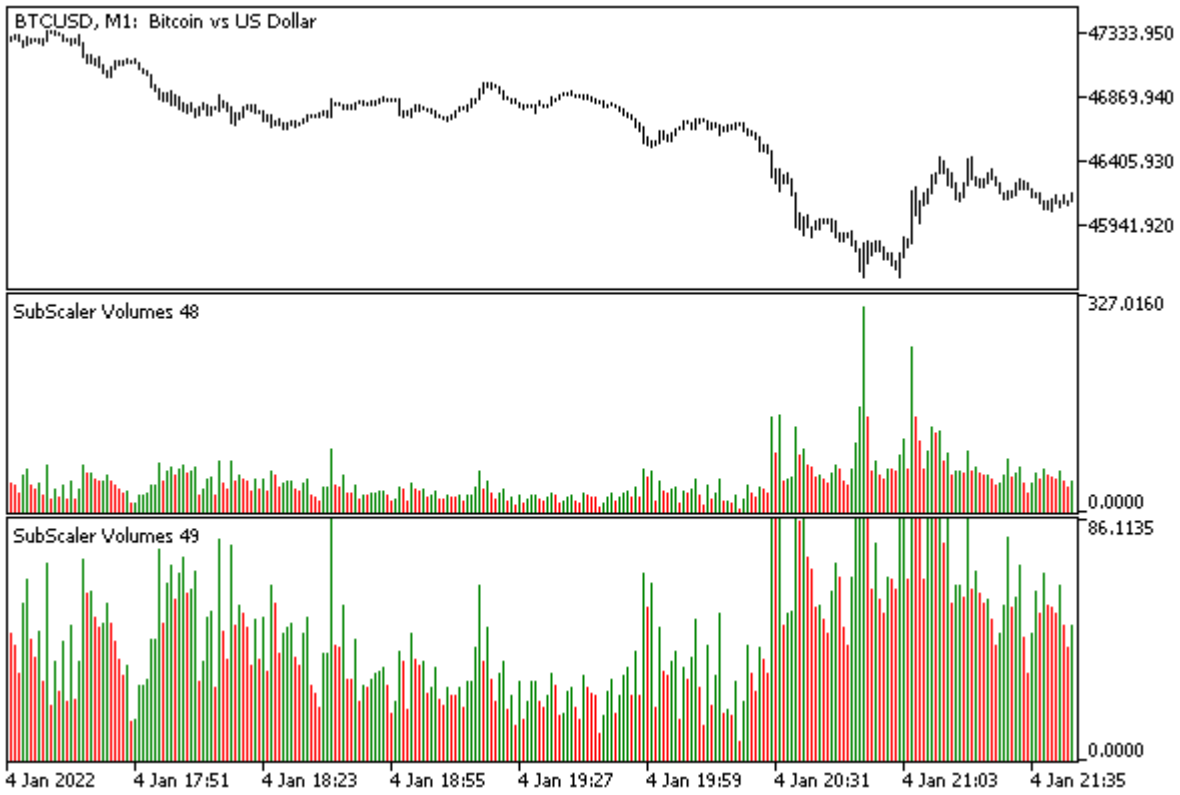
```
void Scale(const long cmd, const int shift)
{
   const double min = ChartGetDouble(0, CHART_PRICE_MIN, w);
   const double max = ChartGetDouble(0, CHART_PRICE_MAX, w);
   ...
```

A continuación, en función de si se pulsa la tecla *Shift*, se realiza el zoom o la panorámica, es decir, se desplaza el rango visible de valores hacia arriba o hacia abajo. En ambos casos, la modificación se lleva a cabo con un determinado paso (multiplicador) *ScaleFactor*, relativo a los límites *min* y *max*, y se asignan a las propiedades del indicador INDICATOR_MINIMUM e INDICATOR_MAXIMUM, respectivamente. Debido al hecho de que el indicador subordinado tiene el ajuste Heredar escala , éste se convierte en un ajuste de trabajo para él también.

```
   if((shift &0x10000000) ==0)// Shift is not pressed - scalechange
     {
        if(cmd == VK_UP) // enlarge (zoom in)
        {
           IndicatorSetDouble(INDICATOR_MINIMUM, min / (1.0 + ScaleFactor));
           IndicatorSetDouble(INDICATOR_MAXIMUM, max / (1.0 + ScaleFactor));
           ChartRedraw();
        }
        else if(cmd == VK_DOWN) // shrink (zoom out)
        {
           IndicatorSetDouble(INDICATOR_MINIMUM, min * (1.0 + ScaleFactor));
           IndicatorSetDouble(INDICATOR_MAXIMUM, max * (1.0 + ScaleFactor));
           ChartRedraw();
        }
     }
     else // Shift pressed - pan/shift range
     {
        if(cmd == VK_UP) // shifting charts up
        {
           const double d = (max - min) * ScaleFactor;
           IndicatorSetDouble(INDICATOR_MINIMUM, min - d);
           IndicatorSetDouble(INDICATOR_MAXIMUM, max - d);
           ChartRedraw();
        }
        else if(cmd == VK_DOWN) // shifting charts down
        {
           const double d = (max - min) * ScaleFactor;
           IndicatorSetDouble(INDICATOR_MINIMUM, min + d);
           IndicatorSetDouble(INDICATOR_MAXIMUM, max + d);
           ChartRedraw();
        }
     }
  }
```

Para cualquier cambio, se llama a *ChartRedraw* para actualizar el gráfico.

Veamos cómo funciona *SubScaler* con el indicador estándar de volúmenes (cualquier otro indicador, incluidos los personalizados, se controla de la misma manera).



Una escala diferente establecida por los indicadores SubScaler en dos subventanas

Aquí, en dos subventanas, dos instancias de *SubScaler* aplican diferentes escalas verticales a los volúmenes.

5.9.5 Eventos de ratón

Ya tuvimos la oportunidad de asegurarnos de que recibimos eventos del ratón utilizando el indicador *EventAll.mq5* de la sección Propiedades de gráficos relacionados con eventos. El evento CHARTEVENT_CLICK se envía al programa MQL en cada clic del botón del ratón en la ventana, y los eventos CHARTEVENT_MOUSE_MOVE de movimiento del cursor y CHARTEVENT_MOUSE_WHEEL de desplazamiento de la rueda requieren una activación previa en la configuración del gráfico, para lo que sirven las propiedades CHART_EVENT_MOUSE_MOVE y CHART_EVENT_MOUSE_WHEEL, respectivamente (ambas están desactivadas por defecto).

Si hay un objeto gráfico bajo el ratón, al pulsar el botón no sólo se genera el evento CHARTEVENT_CLICK sino también CHARTEVENT_OBJECT_CLICK.

Para los eventos CHARTEVENT_CLICK y CHARTEVENT_MOUSE_MOVE, los parámetros del manejador *OnChartEvent* contienen la siguiente información:

- lparam - coordenada X
- dparam - coordenada Y

Además, para el evento CHARTEVENT_MOUSE_MOVE, el parámetro *sparam* contiene una representación de cadena de una máscara de bits que describe el estado de los botones del ratón y las teclas de control (*Ctrl*, *Shift*). Establecer en 1 un bit determinado significa pulsar el botón o la tecla correspondiente.

Bits	Descripción
0	Estado del botón izquierdo del ratón
1	Estado del botón derecho del ratón
2	Estado de la tecla MAYÚS
3	Estado de la tecla CTRL
4	Estado del botón central del ratón
5	Estado del primer botón adicional del ratón
6	Estado del segundo botón adicional del ratón

Por ejemplo, si el bit 0 está activado, dará el número 1 (1 << 0), y si el bit 4º está activado, dará el número 16 (1 << 4). La pulsación simultánea de botones o teclas se indica mediante una superposición de bits.

Para el evento CHARTEVENT_MOUSE_WHEEL, las coordenadas X e Y, así como las banderas de estado de los botones del ratón y las teclas de control, se codifican de una manera especial dentro del parámetro *lparam*, y el parámetro *dparam* informa de la dirección (más/menos) y la cantidad de desplazamiento de la rueda (múltiplos de ±120).

El entero de 8 bytes *lparam* combina varios de los campos de información mencionados.

Bytes	Descripción
0	Valor de tipo *short* con la coordenada X
1	
2	Valor de tipo *short* con la coordenada Y
3	
4	Máscara de bits de los estados de botones y teclas
5	No se utiliza
6	
7	

Independientemente del tipo de evento, las coordenadas del ratón se transmiten en relación con toda la ventana, incluidas las subventanas, por lo que deben recalcularse para una subventana específica si es necesario.

Para comprender mejor CHARTEVENT_MOUSE_WHEEL, utilice el indicador *EventMouseWheel.mq5*. Recibe y descodifica los mensajes y, a continuación, envía su descripción al registro.

```
#define KEY_FLAG_NUMBER 7

const string keyNameByBit[KEY_FLAG_NUMBER] =
{
   "[Left Mouse] ",
   "[Right Mouse] ",
   "(Shift) ",
   "(Ctrl) ",
   "[Middle Mouse] ",
   "[Ext1 Mouse] ",
   "[Ext2 Mouse] ",
};

void OnChartEvent(const int id,
   const long &lparam, const double &dparam, const string &sparam)
{
   if(id == CHARTEVENT_MOUSE_WHEEL)
   {
      const int keymask = (int)(lparam >> 32);
      const short x = (short)lparam;
      const short y = (short)(lparam >> 16);
      const short delta = (short)dparam;
      string message = "";

      for(int i = 0; i < KEY_FLAG_NUMBER; ++i)
      {
         if(((1 << i) & keymask) != 0)
         {
            message += keyNameByBit[i];
         }
      }

      PrintFormat("X=%d Y=%d D=%d %s", x, y, delta, message);
   }
}
```

Ejecute el indicador en el gráfico y desplace la rueda del ratón pulsando sucesivamente varios botones y teclas. He aquí un ejemplo de resultado:

```
X=186 Y=303 D=-120
X=186 Y=312 D=120
X=230 Y=135 D=-120
X=230 Y=135 D=-120 (Ctrl)
X=230 Y=135 D=-120 (Shift) (Ctrl)
X=230 Y=135 D=-120 (Shift)
X=230 Y=135 D=120
X=230 Y=135 D=-120 [Middle Mouse]
X=230 Y=135 D=120 [Middle Mouse]
X=236 Y=210 D=-240
X=236 Y=210 D=-360
```

5.9.6 Eventos de objetos gráficos

Para los objetos gráficos situados en el gráfico, el terminal genera varios eventos especializados. La mayoría de ellos se aplican a objetos de cualquier tipo. El evento de fin de edición de texto en el campo de entrada (CHARTEVENT_OBJECT_ENDEDIT) se genera sólo para objetos del tipo OBJ_EDIT.

Los eventos de clic de objeto (CHARTEVENT_OBJECT_CLICK), arrastre de ratón (CHARTEVENT_OBJECT_DRAG) y cambio de propiedades de objeto (CHARTEVENT_OBJECT_CHANGE) están siempre activos, mientras que los eventos de creación de objetos CHARTEVENT_OBJECT_CREATE y CHARTEVENT_OBJECT_DELETE requieren una habilitación explícita mediante la configuración de las propiedades relevantes del gráfico: CHART_EVENT_OBJECT_CREATE y CH ART_EVENT_OBJECT_DELETE.

Al renombrar un objeto manualmente (desde el cuadro de diálogo de propiedades), el terminal genera una secuencia de eventos CHARTEVENT_OBJECT_DELETE, CHARTEVENT_OBJECT_CREATE, CHARTEVENT_OBJECT_CHANGE. Cuando se renombra un objeto mediante programación, estos eventos no se generan.

Todos los eventos en objetos llevan el nombre del objeto asociado en el parámetro *sparam* de la función *OnChartEvent*.

Además, las coordenadas de clic se pasan para CHARTEVENT_OBJECT_CLICK: X en el parámetro *lparam* e Y en el parámetro *dparam*. Las coordenadas son comunes a todo el gráfico, incluidas las subventanas.

Hacer clic en los objetos funciona de forma diferente dependiendo del tipo de objeto. Para algunos, como la elipse, el cursor debe estar sobre cualquier punto de anclaje. Para otros (triángulo, rectángulo, líneas), el cursor puede estar sobre el perímetro del objeto, no sólo sobre un punto. En todos estos casos, al pasar el cursor del ratón por encima de la zona interactiva del objeto aparece una información sobre herramientas con el nombre del objeto.

Los objetos vinculados a coordenadas de pantalla, que permiten formar la interfaz gráfica del programa, en concreto, un botón, un campo de entrada y un panel rectangular, generan eventos cuando se hace clic con el ratón en cualquier parte del objeto.

Si hay varios objetos bajo el cursor, se genera un evento para el objeto con la Prioridad Z más alta. Si las prioridades de los objetos son iguales, el suceso se asigna al que se creó más tarde (esto se corresponde con su presentación visual, es decir, el posterior se superpone al anterior).

La nueva versión del indicador le ayudará a comprobar los eventos en los objetos *EventAllObjects.mq5*. Lo crearemos y configuraremos utilizando la clase ya conocida *Object Selector* de varios objetos, para luego interceptar en el manejador *OnChartEvent* sus eventos característicos.

```
#include <MQL5Book/ObjectMonitor.mqh>

class ObjectBuilder: public ObjectSelector
{
protected:
   const ENUM_OBJECT type;
   const int window;
public:
   ObjectBuilder(const string _id, const ENUM_OBJECT _type,
      const long _chart = 0, const int _win = 0):
      ObjectSelector(_id, _chart), type(_type), window(_win)
   {
      ObjectCreate(host, id, type, window, 0, 0);
   }
};
```

Inicialmente, en *OnInit* creamos un objeto botón y una línea vertical. Para la línea, realizaremos el seguimiento del evento de movimiento (arrastre) y, al pulsar el botón, crearemos un campo de entrada para el que comprobaremos el texto introducido.

```
const string ObjNamePrefix = "EventShow-";
const string ButtonName = ObjNamePrefix + "Button";
const string EditBoxName = ObjNamePrefix + "EditBox";
const string VLineName = ObjNamePrefix + "VLine";

bool objectCreate, objectDelete;

void OnInit()
{
   // remember the original settings to restore in OnDeinit
   objectCreate = ChartGetInteger(0, CHART_EVENT_OBJECT_CREATE);
   objectDelete = ChartGetInteger(0, CHART_EVENT_OBJECT_DELETE);

   // set new properties
   ChartSetInteger(0, CHART_EVENT_OBJECT_CREATE, true);
   ChartSetInteger(0, CHART_EVENT_OBJECT_DELETE, true);

   ObjectBuilder button(ButtonName, OBJ_BUTTON);
   button.set(OBJPROP_XDISTANCE, 100).set(OBJPROP_YDISTANCE, 100)
   .set(OBJPROP_XSIZE, 200).set(OBJPROP_TEXT, "Click Me");

   ObjectBuilder line(VLineName, OBJ_VLINE);
   line.set(OBJPROP_TIME, iTime(NULL, 0, 0))
   .set(OBJPROP_SELECTABLE, true).set(OBJPROP_SELECTED, true)
   .set(OBJPROP_TEXT, "Drag Me").set(OBJPROP_TOOLTIP, "Drag Me");

   ChartRedraw();
}
```

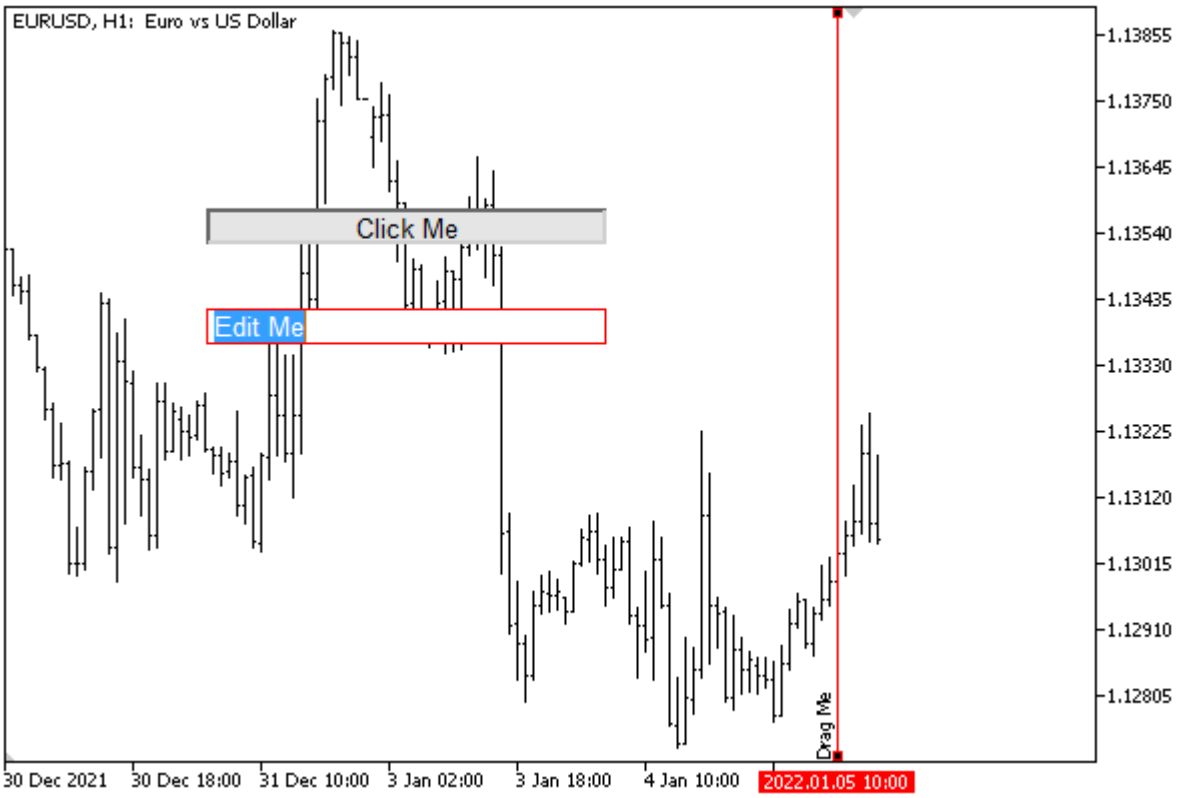
Por el camino, no olvide establecer las propiedades del gráfico CHART_EVENT_OBJECT_CREATE y CHART_EVENT_OBJECT_DELETE en *true* para ser notificado cuando un conjunto de objetos cambie.

En la función *OnChartEvent*, proporcionaremos una respuesta adicional a los eventos requeridos: una vez finalizado el arrastre, mostraremos la nueva posición de la línea en el registro y, tras editar el texto del campo de entrada, su contenido.

```
void OnChartEvent(const int id,
   const long &lparam, const double &dparam, const string &sparam)
{
   ENUM_CHART_EVENT evt = (ENUM_CHART_EVENT)id;
   PrintFormat("%s %lld %f '%s'", EnumToString(evt), lparam, dparam, sparam);
   if(id == CHARTEVENT_OBJECT_CLICK && sparam == ButtonName)
   {
      if(ObjectGetInteger(0, ButtonName, OBJPROP_STATE))
      {
         ObjectBuilder edit(EditBoxName, OBJ_EDIT);
         edit.set(OBJPROP_XDISTANCE, 100).set(OBJPROP_YDISTANCE, 150)
         .set(OBJPROP_BGCOLOR, clrWhite)
         .set(OBJPROP_XSIZE, 200).set(OBJPROP_TEXT, "Edit Me");
      }
      else
      {
         ObjectDelete(0, EditBoxName);
      }

      ChartRedraw();
   }
   else if(id == CHARTEVENT_OBJECT_ENDEDIT && sparam == EditBoxName)
   {
      Print(ObjectGetString(0, EditBoxName, OBJPROP_TEXT));
   }
   else if(id == CHARTEVENT_OBJECT_DRAG && sparam == VLineName)
   {
      Print(TimeToString((datetime)ObjectGetInteger(0, VLineName, OBJPROP_TIME)));
   }
}
```

Tenga en cuenta que, cuando se pulsa el botón por primera vez, su estado cambia de liberado a pulsado, y en respuesta a esto, creamos un campo de entrada. Si vuelve a pulsar el botón, volverá a cambiar su estado, con lo que el campo de entrada desaparecerá del gráfico.

A continuación se muestra una imagen del gráfico durante el funcionamiento del indicador.



Objetos controlados por el manejador de eventos OnChartEvent

Inmediatamente después de lanzar el indicador, aparecen las siguientes líneas en el registro:

```
CHARTEVENT_OBJECT_CREATE 0 0.000000 'EventShow-Button'
CHARTEVENT_OBJECT_CREATE 0 0.000000 'EventShow-VLine'
CHARTEVENT_CHART_CHANGE 0 0.000000 ''
```

Si a continuación arrastramos la línea con el ratón, veremos algo parecido a esto:

```
CHARTEVENT_OBJECT_DRAG 0 0.000000 'EventShow-VLine'
2022.01.05 10:00
```

A continuación, puede hacer clic en el botón y editar el texto en el campo de entrada recién creado (cuando termine la edición, haga clic en *Enter* o fuera del campo de entrada). Esto dará lugar a las siguientes entradas en el registro (las coordenadas y el texto del mensaje pueden diferir; aquí se ha introducido el texto «nuevo mensaje»):

```
CHARTEVENT_OBJECT_CLICK 181 113.000000 'EventShow-Button'
CHARTEVENT_CLICK 181 113.000000 ''
CHARTEVENT_OBJECT_CREATE 0 0.000000 'EventShow-EditBox'
CHARTEVENT_OBJECT_CLICK 152 160.000000 'EventShow-EditBox'
CHARTEVENT_CLICK 152 160.000000 ''
CHARTEVENT_OBJECT_ENDEDIT 0 0.000000 'EventShow-EditBox'
new message
```

Si suelta el botón, el campo de entrada se borrará.

```
CHARTEVENT_OBJECT_CLICK 162 109.000000 'EventShow-Button'
CHARTEVENT_CLICK 162 109.000000 ''
CHARTEVENT_OBJECT_DELETE 0 0.000000 'EventShow-EditBox'
```

Cabe señalar que el botón funciona por defecto como un interruptor de dos posiciones, es decir, se mantiene alternativamente en el estado pulsado o liberado como resultado de un clic del ratón. Para un botón normal, este comportamiento es redundante: para realizar un simple seguimiento de las pulsaciones de botón, debe devolverlo al estado liberado al procesar el evento llamando a *ObjectSetInteger(0, ButtonName, OBJPROP_STATE, false)*.

5.9.7 Generación de eventos personalizados

Además de los eventos estándar, el terminal admite la capacidad de generar mediante programación eventos personalizados, cuya esencia y contenido vienen determinados por el programa MQL. Dichos eventos se añaden a la cola general de eventos del gráfico y pueden ser procesados en la función *OnChartEvent* por todos los programas interesados.

Se reserva un rango especial de 65536 identificadores enteros para los eventos personalizados: de CHARTEVENT_CUSTOM a CHARTEVENT_CUSTOM_LAST, ambos inclusive. En otras palabras: el evento personalizado debe tener el ID CHARTEVENT_CUSTOM + n, donde n está entre 0 y 65535. CHARTEVENT_CUSTOM_LAST es exactamente igual a CHARTEVENT_CUSTOM + 65535.

Los eventos personalizados se envían al gráfico mediante la función *EventChartCustom*.

```
bool EventChartCustom(long chartId, ushort customEventId,
 long lparam, double dparam, string sparam)
```

chartId es el identificador del gráfico receptor del evento, mientras que 0 indica el gráfico actual; *customEventId* es el identificador del evento (seleccionado por el desarrollador del programa MQL). Este identificador se añade automáticamente al valor CHARTEVENT_CUSTOM y se convierte a un tipo entero. Este valor se pasará al manejador de *OnChartEvent* como primer argumento. Otros parámetros de *EventChartCustom* corresponden a los parámetros de eventos estándar de *OnChartEvent* con los tipos *long*, *double* y *string*, y pueden contener información arbitraria.

La función devuelve *true* en caso de que el evento de usuario se haya puesto en cola correctamente, o *false* en caso de error (el código de error estará disponible en *_LastError*).

A medida que nos acerquemos a la parte más compleja e importante de nuestro libro, dedicada directamente a la automatización del trading, empezaremos a resolver problemas aplicados que serán útiles en el desarrollo de robots de trading. Ahora, en el contexto de la demostración de las capacidades de los eventos personalizados, vamos a pasar al análisis multidivisa (o, más en general, multisímbolo) del entorno de trading.

Un poco antes, en el capítulo sobre indicadores, hemos visto los indicadores multidivisa, pero no prestamos atención a un punto importante: a pesar de que los indicadores procesaban cotizaciones de diferentes símbolos, el cálculo en sí se lanzó en el manejador *OnCalculate*, que se activó con la llegada de un nuevo tick de un único símbolo: el símbolo de trabajo del gráfico. Resulta que los ticks de otros instrumentos, en esencia, se omiten. Por ejemplo, si el indicador trabaja sobre el símbolo A, cuando llega su tick simplemente tomamos los últimos ticks conocidos de otros símbolos (B, C, D), pero es probable que otros ticks hayan conseguido colarse entre cada uno de ellos.

Si coloca un indicador multidivisa en el instrumento más líquido (donde se reciben ticks con más frecuencia), esto no es tan crítico. Sin embargo, diferentes instrumentos pueden ser más rápidos que

otros en diferentes momentos del día, y si un algoritmo analítico o de negociación requiere la respuesta más rápida posible a las nuevas cotizaciones de todos los instrumentos de la cartera, nos enfrentamos al hecho de que la solución actual no nos conviene.

Por desgracia, el evento estándar de llegada de un nuevo tick funciona en MQL5 solo para un símbolo, que es el símbolo de trabajo del gráfico actual. En los indicadores, se llama al manejador *OnCalculate* en esos momentos, y el manejador *OnTick* se llama en los Asesores Expertos.

Por lo tanto, es necesario inventar algún mecanismo para que el programa MQL pueda recibir notificaciones sobre ticks en todos los instrumentos de interés. Aquí es donde los eventos personalizados nos ayudarán. Por supuesto, esto no es necesario para los programas que analizan un solo instrumento.

A continuación desarrollaremos un ejemplo del indicador *EventTickSpy.mq5* que, siendo lanzado sobre un símbolo X específico, podrá enviar notificaciones de tick desde su función *OnCalculate* utilizando *EventChartCustom*. Como resultado, en el manejador *OnChartEvent*, que está especialmente preparado para recibir tales notificaciones, será posible recoger notificaciones de diferentes instancias del indicador desde diferentes símbolos.

Este ejemplo se ofrece a título ilustrativo. Posteriormente, cuando estudiemos el trading automatizado multidivisa, adaptaremos esta técnica para un uso más cómodo en los Asesores Expertos.

En primer lugar, pensemos en un número de evento personalizado para el indicador. Dado que vamos a enviar notificaciones de tick para muchos símbolos diferentes de una lista dada, podemos elegir aquí diferentes tácticas. Por ejemplo, puede seleccionar un identificador de evento, y pasar el número del símbolo en la lista y/o el nombre del símbolo en los parámetros *lparam* y *sparam*, respectivamente. O puede tomar alguna constante (mayor que e igual a CHARTEVENT_CUSTOM) y obtener números de eventos añadiendo el número de símbolo a esta constante (entonces tenemos todos los parámetros libres, en particular, *lparam* y *dparam*, y pueden utilizarse para transferir precios *Ask*, *Bid* o algún otro).

Nos centraremos en la opción cuando hay un código de evento. Declarémoslo en la macro TICKSPY. Este será el valor por defecto, que el usuario puede cambiar para evitar colisiones (aunque poco probables) con otros programas si es necesario.

```
#define TICKSPY 0xFEED // 65261
```

Este valor se toma a propósito por estar bastante alejado del primer CHARTEVENT_CUSTOM permitido.

Durante el lanzamiento inicial (interactivo) del indicador, el usuario debe especificar la lista de instrumentos cuyos ticks debe seguir el indicador. Para ello, describiremos la variable de cadena de entrada *SymbolList* con una lista de símbolos separados por comas.

El identificador del evento de usuario se establece en el parámetro *message*.

Por último, necesitamos el identificador del gráfico receptor para pasar el evento. Para ello proporcionaremos el parámetro *Chart*. El usuario no debe editarlo: en la primera instancia del indicador lanzada manualmente, el gráfico se conoce implícitamente al adjuntarlo al gráfico. En otras copias del indicador que nuestra primera instancia ejecutará programáticamente, este parámetro rellenará el algoritmo con una llamada a la función *ChartID* (véase más adelante).

```
input string SymbolList = "EURUSD,GBPUSD,XAUUSD,USDJPY"; // List of symbols separated
input ushort message = TICKSPY;                          // Custom message
input longchart = 0;                                     // Receiving chart (do not e
```

En el parámetro *SymbolList*, por ejemplo, se indica una lista con cuatro herramientas comunes. Edítelo según sea necesario para adaptarlo a su *Market Watch*.

En el manejador *OnInit*, convertimos la lista al array de símbolos *Symbols*, y luego en un bucle ejecutamos el mismo indicador para todos los símbolos del array, excepto para el actual (por norma, existe tal coincidencia porque el símbolo actual ya está siendo procesado por esta copia inicial del indicador).

```
string Symbols[];

void OnInit()
{
   PrintFormat("Starting for chart %lld, msg=0x%X [%s]", Chart, Message, SymbolList);
   if(Chart == 0)
   {
      if(StringLen(SymbolList) > 0)
      {
         const int n = StringSplit(SymbolList, ',', Symbols);
         for(int i = 0; i < n; ++i)
         {
            if(Symbols[i] != _Symbol)
            {
               ResetLastError();
               // run the same indicator on another symbol with different settings,
               // in particular, we pass our ChartID to receive notifications back
               iCustom(Symbols[i], PERIOD_CURRENT, MQLInfoString(MQL_PROGRAM_NAME),
                  "", Message, ChartID());
               if(_LastError != 0)
               {
                  PrintFormat("The symbol '%s' seems incorrect", Symbols[i]);
               }
            }
         }
      }
      else
      {
         Print("SymbolList is empty: tracking current symbol only!");
         Print("To monitor other symbols, fill in SymbolList, i.e."
            " 'EURUSD,GBPUSD,XAUUSD,USDJPY'");
      }
   }
}
```

Al principio de *OnInit* se muestra en el registro información sobre la instancia del indicador lanzada para que quede claro lo que está sucediendo.

Si eligiéramos la opción con códigos de evento separados para cada carácter, tendríamos que llamar a *iCustom* de la siguiente manera (añadir *i* a *message*):

```
      iCustom(Symbols[i], PERIOD_CURRENT, MQLInfoString(MQL_PROGRAM_NAME), "",
         Message + i, ChartID());
```

Tenga en cuenta que el valor distinto de cero del parámetro *Chart* implica que esta copia se lanza mediante programación y que debe supervisar un único símbolo, es decir, el símbolo de trabajo del gráfico. Por lo tanto, no necesitamos pasar una lista de símbolos al ejecutar las copias esclavas.

En la función *OnCalculate*, a la que se llama cuando se recibe un nuevo tick, enviamos el evento personalizado *Message* al gráfico *Chart* llamando a *EventChartCustom*. En este caso no se utiliza el parámetro *lparam* (igual a 0). En el parámetro *dparam*, pasamos el precio actual (último) *price[0]* (esto es, *Bid* o *Last*, según el tipo de precio en el que se base el gráfico: también es el precio del último tick procesado por el gráfico), y pasamos el nombre del símbolo en el parámetro *sparam*.

```
int OnCalculate(const int rates_total, const int prev_calculated,
   const int, const double &price[])
{
   if(prev_calculated)
   {
      ArraySetAsSeries(price, true);
      if(Chart > 0)
      {
         // send a tick notification to the parent chart
         EventChartCustom(Chart, Message, 0, price[0], _Symbol);
      }
      else
      {
         OnSymbolTick(_Symbol, price[0]);
      }
   }

   return rates_total;
}
```

En la instancia original del indicador, en la que el parámetro *Chart* es 0, llamamos directamente a una función especial, una especie de manejador de ticks multiactivos *OnSymbolTick*. En este caso, no es necesario llamar a *EventChartCustom*: aunque dicho mensaje seguirá llegando al gráfico y a esta copia del indicador, la transmisión tarda varios milisegundos y carga la cola en vano.

El único propósito de *OnSymbolTick* en esta demostración es imprimir el nombre del símbolo y el nuevo precio en el registro.

```
void OnSymbolTick(const string &symbol, const double price)
{
   Print(symbol, " ", DoubleToString(price,
      (int)SymbolInfoInteger(symbol, SYMBOL_DIGITS)));
}
```

Por supuesto, la misma función es llamada desde el manejador *OnChartEvent* en la copia receptora (fuente) del indicador, siempre que nuestro mensaje haya sido recibido. Recordemos que el terminal llama a *OnChartEvent* sólo en la copia interactiva del indicador (aplicada al gráfico) y no aparece en aquellas copias que hemos creado «invisibles» mediante *iCustom*.

```
void OnChartEvent(const int id,
   const long &lparam, const double &dparam, const string &sparam)
{
   if(id >= CHARTEVENT_CUSTOM + Message)
   {
      OnSymbolTick(sparam, dparam);
      // OR (if using custom event range):
      // OnSymbolTick(Symbols[id - CHARTEVENT_CUSTOM - Message], dparam);
   }
}
```

Podríamos evitar enviar, bien el precio o bien el nombre del símbolo en nuestro evento, ya que en el indicador inicial (que inició el proceso) se conoce la lista general de símbolos (que inició el proceso), y por lo tanto podríamos indicarle de alguna manera el número del símbolo de la lista. Esto podría hacerse en el parámetro *lparam* o, como se ha mencionado anteriormente, añadiendo un número a la constante de base del evento de usuario. A continuación, el indicador original, mientras recibe eventos, podría tomar un símbolo por índice del array y obtener toda la información sobre el último tick usando *SymbolInfoTick*, incluidos los distintos tipos de precios.

Vamos a ejecutar el indicador en el gráfico EURUSD con la configuración por defecto, incluyendo la lista de prueba «EURUSD,GBPUSD,XAUUSD,USDJPY». He aquí el registro:

```
16:45:48.745 (EURUSD,H1) Starting for chart 0, msg=0xFEED [EURUSD,GBPUSD,XAUUSD,USDJP
16:45:48.761 (GBPUSD,H1) Starting for chart 132358585987782873, msg=0xFEED []
16:45:48.761 (USDJPY,H1) Starting for chart 132358585987782873, msg=0xFEED []
16:45:48.761 (XAUUSD,H1) Starting for chart 132358585987782873, msg=0xFEED []
16:45:48.777 (EURUSD,H1) XAUUSD 1791.00
16:45:49.120 (EURUSD,H1) EURUSD 1.13068 *
16:45:49.135 (EURUSD,H1) USDJPY 115.797
16:45:49.167 (EURUSD,H1) XAUUSD 1790.95
16:45:49.167 (EURUSD,H1) USDJPY 115.796
16:45:49.229 (EURUSD,H1) USDJPY 115.797
16:45:49.229 (EURUSD,H1) XAUUSD 1790.74
16:45:49.369 (EURUSD,H1) XAUUSD 1790.77
16:45:49.572 (EURUSD,H1) GBPUSD 1.35332
16:45:49.572 (EURUSD,H1) XAUUSD 1790.80
16:45:49.791 (EURUSD,H1) XAUUSD 1790.80
16:45:49.791 (EURUSD,H1) USDJPY 115.796
16:45:49.931 (EURUSD,H1) EURUSD 1.13069 *
16:45:49.931 (EURUSD,H1) XAUUSD 1790.86
16:45:49.931 (EURUSD,H1) USDJPY 115.795
16:45:50.056 (EURUSD,H1) USDJPY 115.793
16:45:50.181 (EURUSD,H1) XAUUSD 1790.88
16:45:50.321 (EURUSD,H1) XAUUSD 1790.90
16:45:50.399 (EURUSD,H1) EURUSD 1.13066 *
16:45:50.727 (EURUSD,H1) EURUSD 1.13067 *
16:45:50.773 (EURUSD,H1) GBPUSD 1.35334
```

Observe que en la columna con (símbolo,marco temporal), que es la fuente del registro, vemos en primer lugar las instancias del indicador inicial en cuatro símbolos solicitados.

Tras el lanzamiento, el primer tick fue XAUUSD, no EURUSD. Otros símbolos aparecen con una intensidad aproximadamente igual, intercalados. Los ticks de EURUSD están marcados con asteriscos, para que pueda hacerse una idea de cuántos otros ticks se habrían perdido sin las notificaciones.

Las marcas de tiempo se han guardado en la columna de la izquierda como referencia.

Los lugares en los que coinciden dos precios de dos eventos consecutivos del mismo símbolo suelen indicar que el precio *Ask* ha cambiado (simplemente no lo mostramos aquí).

Un poco más adelante, tras estudiar la API de MQL5 de trading, aplicaremos el mismo principio para responder a los ticks multidivisa en los Asesores Expertos.

Parte 6. Automatización de trading

En esta parte estudiaremos el componente más complejo e importante de la API de MQL5 que permite automatizar las acciones de trading.

Comenzaremos por describir las entidades sin las cuales es imposible escribir un Asesor Experto adecuado. Ello incluye las opciones de configuración de símbolos financieros y cuentas de trading.

A continuación, examinaremos las funciones de trading y estructuras de datos integradas, junto con eventos específicos de robot y modos de funcionamiento. En concreto, la característica clave de los Asesores Expertos es la integración con el probador, lo que permite a los usuarios evaluar los resultados financieros y optimizar las estrategias de trading. Consideraremos los mecanismos internos de optimización y la gestión de la optimización a través de la API.

El probador de estrategias es una herramienta esencial para el desarrollo de programas MQL, ya que ofrece la posibilidad de depurar programas en varios modos, incluyendo barras y ticks, basándose en ticks modelados o reales, con o sin visualización del flujo de precios.

Ya hemos intentado probar indicadores en modo visual, pero el conjunto de parámetros de simulación es limitado para los indicadores. Al desarrollar Asesores Expertos, tendremos acceso a toda la gama de capacidades del probador.

Además, conoceremos una nueva forma de información sobre el mercado: la Profundidad de Mercado y su interfaz de software.

Programación en MQL5 para Traders: códigos fuente del libro. Parte 6

Los ejemplos del libro también están disponibles en el proyecto público \MQL5\Shared Projects\MQL5Book

6.1 Instrumentos financieros y Observación de Mercado

MetaTrader 5 permite a los usuarios analizar y operar con instrumentos financieros (también conocidos como símbolos o tickers), los cuales constituyen la base de casi todos los subsistemas del terminal. Los gráficos, los indicadores y el historial de precios de las cotizaciones existen en relación con los símbolos de trading. La principal funcionalidad del terminal se basa en instrumentos financieros como las órdenes de trading, las transacciones, el control de los requisitos de margen y el historial de la cuenta de trading.

A través del terminal, los brókers entregan a los operadores una lista específica de símbolos, de la que cada usuario elige los que prefiera, formando Observación de Mercado. La ventana Observación de mercado determina los símbolos para los que el terminal solicita cotizaciones en línea y permite abrir gráficos y consultar el historial.

La API de MQL5 proporciona herramientas de software similares que permiten ver y analizar las características de todos los símbolos, añadirlos a Observación de Mercado o excluirlos de ahí.

Además de los símbolos estándar con información proporcionada por los brókers, MetaTrader 5 permite crear símbolos personalizados: sus propiedades e historial de precios pueden cargarse desde fuentes de datos arbitrarias y calcularse mediante fórmulas o programas MQL. Los símbolos personalizados también participan en Observación de Mercado y pueden utilizarse para estrategias de simulación y análisis técnico, pero tienen también una limitación natural: no pueden negociarse en línea utilizando

las herramientas regulares de la API de MQL5, ya que estos símbolos no están disponibles en el servidor. Los símbolos personalizados se abordarán en un capítulo aparte, en la séptima y última parte del libro.

Hace poco, en los capítulos correspondientes, vimos ya las series temporales con datos de precios de símbolos individuales, incluida la paginación del historial utilizando un ejemplo con indicadores. En realidad, toda esta funcionalidad presupone que los símbolos correspondientes ya están activados en Observación de Mercado. Esto es especialmente cierto para los indicadores multidivisa y los Asesores Expertos que se refieren no sólo al símbolo de trabajo del gráfico, sino también a otros símbolos. En este capítulo descubriremos cómo se gestiona la lista de Observación de Mercado desde los programas MQL.

En el capítulo dedicado a los gráficos ya se han descrito algunas de las propiedades de los símbolos disponibles a través de funciones básicas de acceso a propiedades de un gráfico actual (*Point*, *Digits*) ya que el gráfico no puede funcionar sin el símbolo asociado. Ahora estudiaremos la mayoría de las propiedades de los símbolos, incluida su especificación. Su conjunto completo puede consultarse en la documentación de MQL5 en el sitio web.

6.1.1 Obtener símbolos disponibles y listas de Observación de Mercado

La API de MQL5 dispone de varias funciones para realizar operaciones con símbolos. Con ellas, puede encontrar el número total de símbolos disponibles y el número de símbolos seleccionados en *Observación de Mercado*, así como sus nombres. Como sabe, la lista general de símbolos disponibles en el terminal se indica en forma de estructura jerárquica en el cuadro de diálogo *Símbolos*, que el usuario puede abrir con el comando *Ver -> Símbolos* o desde el menú contextual *Observación de Mercado*. En esta lista se incluyen tanto los símbolos proporcionados por el bróker como los símbolos personalizados creados localmente. Puede utilizar la función *SymbolsTotal* para hallar el número total de símbolos.

int SymbolsTotal(bool selected)

El parámetro *selected* especifica si sólo se solicitan los símbolos de *Observación de Mercado* (*true*) o todos los símbolos disponibles (*false*).

La función *SymbolName* se utiliza a menudo junto con *SymbolsTotal*. Devuelve el nombre del símbolo por su índice (aquí no se tiene en cuenta la agrupación del almacenamiento de símbolos en carpetas lógicas, véase la propiedad SYMBOL_PATH).

string SymbolName(int index, bool selected)

El parámetro *index* especifica el índice del símbolo solicitado. *Index* debe estar comprendido entre 0 y el número de símbolos, sujeto al contexto de solicitud especificado por el segundo parámetro *selected*: *true* limita la enumeración a los símbolos elegidos en *Observación de Mercado*, mientras que *false* coincide absolutamente con todos los símbolos (por analogía con *SymbolsTotal*). Por lo tanto, al llamar a *SymbolName*, ajuste el parámetro *selected* al mismo valor que en la llamada anterior a *SymbolsTotal*, que se utiliza para definir el rango de índices.

En caso de error, en particular si el índice solicitado está fuera del rango de la lista, la función devolverá una cadena vacía y el código de error se escribirá en la variable *_LastError*.

> Es importante observar que, cuando la opción *selected* está activada, el par de funciones *SymbolsTotal* y *SymbolName* devuelve información para la lista de símbolos realmente actualizados por el terminal, es decir, los símbolos para los que se realiza una sincronización constante con el servidor y para los que el historial de cotizaciones está disponible para los programas MQL. Esta lista puede ser mayor que la lista visible en *Observación de Mercado*, donde los elementos se añaden

explícitamente: por el usuario o por un programa MQL (para saber cómo hacerlo, consulte la sección Editar la lista en *Observación de Mercado*). Estos símbolos, invisibles en la ventana, son conectados automáticamente por el terminal cuando se necesitan para calcular tipos de cambio cruzados. Entre las propiedades de los símbolos, hay dos que permiten distinguir entre selección explícita (SYMBOL_VISIBLE) y selección implícita (SYMBOL_SELECT), y que se abordarán en la sección sobre comprobación del estado de los símbolos. En términos estrictos, para las funciones *SymbolsTotal* y *SymbolName*, el ajuste de *selected* a *true* coincide con el conjunto de símbolos extendidos con SYMBOL_SELECT inclinado, no sólo con aquellos con SYMBOL_VISIBLE igual a *true*.

El orden en que se devuelven los símbolos de *Observación de Mercado* corresponde al de la ventana del terminal (teniendo en cuenta la posible reordenación realizada por el usuario, y no la ordenación por ninguna columna, si está activada). No es posible cambiar el orden de los símbolos en *Observación de Mercado* mediante programación.

El orden en la lista general de *Symbols* lo establece el propio terminal (el contenido y la ordenación de *Observación de Mercado* no le afectan).

Como ejemplo, veamos el sencillo script *SymbolList.mq5*, que imprime los símbolos disponibles en el registro. El parámetro de entrada *MarketWatchOnly* permite al usuario limitar la lista únicamente a los símbolos *Observación de Mercado* (si el parámetro es *true*) u obtener la lista completa (*false*).

```
#property script_show_inputs

#include <MQL5Book/PRTF.mqh>

input bool MarketWatchOnly = true;

void OnStart()
{
   const int n = SymbolsTotal(MarketWatchOnly);
   Print("Total symbol count: ", n);
   // write a list of symbols in the Market Watch or all available
   for(int i = 0; i < n; ++i)
   {
      PrintFormat("%4d %s", i, SymbolName(i, MarketWatchOnly));
   }
   // intentionally asking for out-of-range to show an error
   PRTF(SymbolName(n, MarketWatchOnly)); // MARKET_UNKNOWN_SYMBOL(4301)
}
```

He aquí un registro de ejemplo:

```
Total symbol count: 10
   0 EURUSD
   1 XAUUSD
   2 BTCUSD
   3 GBPUSD
   4 USDJPY
   5 USDCHF
   6 AUDUSD
   7 USDCAD
   8 NZDUSD
   9 USDRUB
SymbolName(n,MarketWatchOnly)= / MARKET_UNKNOWN_SYMBOL(4301)
```

6.1.2 Editar la lista de Observación de Mercado

Utilizando la función *SymbolSelect*, el desarrollador del programa MQL puede añadir un símbolo específico a *Observación de Mercado* o eliminarlo de allí.

bool SymbolSelect(const string name, bool select)

El parámetro *name* contiene el nombre del símbolo afectado por esta operación. Dependiendo del valor del parámetro *select*, se añade un símbolo a *Observación de Mercado* (*true*) o se elimina de él. Los nombres de los símbolos distinguen entre mayúsculas y minúsculas: por ejemplo, «EURUSD.m» no es igual a «EURUSD.M».

La función devuelve un indicador de éxito (*true*) o de error (*false*). El código de error puede consultarse en *_LastError*.

> No se puede eliminar un símbolo si hay gráficos abiertos o posiciones abiertas para este símbolo. Además, no puede eliminar un símbolo que se utilice explícitamente en la fórmula de cálculo de un instrumento sintético (personalizado) añadido a *Observación de Mercado*.

Hay que tener en cuenta que aunque no existan gráficos y posiciones abiertas para un símbolo, éste puede ser utilizado indirectamente por los programas MQL: por ejemplo, pueden leer su historial de cotizaciones o ticks. La eliminación de dicho símbolo puede causar problemas en estos programas.

El siguiente script *SymbolRemoveUnused.mq5* es capaz de ocultar todos los símbolos que no se utilizan explícitamente, por lo que se recomienda comprobarlo primero en una cuenta demo o guardar el conjunto de símbolos actual por medio del menú contextual.

```
#include <MQL5Book/MqlError.mqh>

#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1) - 1] = V)

void OnStart()
{
   // request user confirmation for deletion
   if(IDOK == MessageBox("This script will remove all unused symbols"
      " from the Market Watch. Proceed?", "Please, confirm", MB_OKCANCEL))
   {
      const int n = SymbolsTotal(true);
      ResetLastError();
      string removed[];
      // go through the symbols of the Market Watch in reverse order
      for(int i = n - 1; i >= 0; --i)
      {
         const string s = SymbolName(i, true);
         if(SymbolSelect(s, false))
         {
            // remember what was deleted
            PUSH(removed, s);
         }
         else
         {
            // in case of an error, display the reason
            PrintFormat("Can't remove '%s': %s (%d)", s, E2S(_LastError), _LastError)
         }
      }
      const int r = ArraySize(removed);
      PrintFormat("%d out of %d symbols removed", r, n);
      ArrayPrint(removed);
      ...
```

Después de que el usuario confirme el análisis de la lista de símbolos, el programa intenta ocultar cada símbolo secuencialmente llamando a *SymbolSelect(s, false)*. Esto sólo funciona para los instrumentos que no se utilizan explícitamente. La enumeración de símbolos se realiza en orden inverso para no violar la indexación. Todos los símbolos eliminados con éxito se recogen en el array *removed*. El registro muestra estadísticas y el propio array.

Si se cambia *Observación de Mercado*, el usuario tiene la oportunidad de restaurar todos los símbolos borrados llamando a *SymbolSelect(removed[i], true)* en un bucle.

```
      if(r > 0)
      {
         // it is possible to return the deleted symbols back to the Market Watch
         // (at this point, the window displays a reduced list)
         if(IDOK == MessageBox("Do you want to restore removed symbols"
            " in the Market Watch?", "Please, confirm", MB_OKCANCEL))
         {
            int restored = 0;
            for(int i = r - 1; i >= 0; --i)
            {
               restored += SymbolSelect(removed[i], true);
            }
            PrintFormat("%d symbols restored", restored);
         }
      }
   }
}
```

Este es el aspecto que podría tener la salida del registro:

```
Can't remove 'EURUSD': MARKET_SELECT_ERROR (4305)
Can't remove 'XAUUSD': MARKET_SELECT_ERROR (4305)
Can't remove 'BTCUSD': MARKET_SELECT_ERROR (4305)
Can't remove 'GBPUSD': MARKET_SELECT_ERROR (4305)
...
Can't remove 'USDRUB': MARKET_SELECT_ERROR (4305)
2 out of 10 symbols removed
"NZDUSD" "USDCAD"
2 symbols restored
```

Tenga en cuenta que, aunque los símbolos se restauran en su orden original, tal y como estaban en *Observación de Mercado* unos respecto de otros, la adición se produce al final de la lista, después de los símbolos restantes. Así, todos los símbolos «ocupados» estarán al principio de la lista, y todos los restaurados les seguirán. Tal es el funcionamiento específico de *SymbolSelect*: un símbolo se añade siempre al final de la lista, es decir, es imposible insertar un símbolo en una posición determinada. Por lo tanto, la reordenación de los elementos de la lista sólo está disponible para edición manual.

6.1.3 Comprobar la existencia de un símbolo

En lugar de buscar en toda la lista de símbolos, un programa MQL puede comprobar la presencia de un símbolo concreto por su nombre. Para ello existe la función *SymbolExist*.

bool SymbolExist(const string name, bool &isCustom)

En el parámetro *name* debe pasar el nombre del símbolo deseado. El parámetro *isCustom* pasado por referencia será establecido por la función dependiendo de si el símbolo especificado es estándar (*false*) o personalizado (*true*).

La función devuelve *false* si el símbolo no se encuentra ni en los símbolos estándar ni en los personalizados.

Un análogo parcial de esta función es la consulta de la propiedad SYMBOL_EXIST.

Vamos a analizar el sencillo script *SymbolExists.mq5* para probar esta función. En su parámetro, el usuario puede especificar el nombre, que luego se pasa a *SymbolExist*, y se registra el resultado. Si se introduce una cadena vacía, se comprobará el símbolo de trabajo del gráfico actual. Por defecto, el parámetro se establece en «XYZ», que presumiblemente no coincide con ninguno de los símbolos disponibles.

```
#property script_show_inputs

input string SymbolToCheck = "XYZ";

void OnStart()
{
   const string _SymbolToCheck = SymbolToCheck == "" ? _Symbol : SymbolToCheck;
   bool custom = false;
   PrintFormat("Symbol '%s' is %s", _SymbolToCheck,
      (SymbolExist(_SymbolToCheck, custom) ? (custom ? "custom" : "standard") : "miss
}
```

Cuando el script se ejecuta dos veces, primero con el valor por defecto y luego con una línea vacía en el gráfico EURUSD, obtendremos las siguientes entradas en el registro.

```
Symbol 'XYZ' is missing
Symbol 'EURUSD' is standard
```

Si ya tiene símbolos personalizados o crea uno nuevo con una fórmula de cálculo sencilla, puede asegurarse de que la variable personalizada está poblada. Por ejemplo, si abre la ventana *Symbols* en el terminal y pulsa el botón *Create symbol*, puede introducir «SP500/FTSE100» (los nombres de los índices pueden diferir según su bróker) en el campo *Synthetic tool formula* y «GBPUSD.INDEX» en el campo con el nombre *Symbol*. Un clic en *OK* creará un instrumento personalizado para el que puede abrir un gráfico, y nuestro script debería mostrar en él lo siguiente:

```
Symbol 'GBPUSD.INDEX' is custom
```

Cuando configure su propio símbolo, no olvide establecer no sólo la fórmula, sino también valores suficientemente «pequeños» para el tamaño del punto y el paso de cambio de precio (tick). De lo contrario, la serie de cotizaciones sintéticas puede resultar «escalonada», o incluso degenerar en una línea recta.

6.1.4 Comprobar la pertinencia de los datos de los símbolos

Debido a la arquitectura distribuida cliente-servidor, los datos del cliente y del servidor pueden ser ocasionalmente diferentes. Por ejemplo, esto puede ocurrir inmediatamente después del inicio de la sesión de terminal, cuando se pierde la conexión o cuando los recursos del ordenador están muy cargados. Además, lo más probable es que el símbolo permanezca no sincronizado durante algún tiempo inmediatamente después de ser añadido a Observación de Mercado. La API de MQL5 permite comprobar la relevancia de los datos de cotización de un símbolo concreto mediante la función *SymbolIsSynchronized*.

bool SymbolIsSynchronized(const string name)

La función devuelve *true* si los datos locales del símbolo denominado *name* están sincronizados con los datos del servidor de trading.

En la sección Obtención de características de arrays de precios, entre otras propiedades de series temporales, se introdujo la propiedad SERIES_SYNCHRONIZED, que devuelve un atributo de sincronización más limitado en su significado: se aplica a una combinación específica de símbolo y marco temporal. En contraste con esta propiedad, la función *SymbolIsSynchronized* devuelve un atributo de sincronización del historial general para un símbolo.

La construcción de todos los marcos temporales comienza sólo después de la finalización de la descarga del historial. Debido a la arquitectura multihilo y a la computación paralela en el terminal, puede ocurrir que *SymbolIsSynchronized* devuelva *true*, y para un marco temporal en el mismo símbolo, la propiedad SERIES_SYNCHRONIZED sea temporalmente igual a *false*.

Veamos cómo trabaja la nueva función en el indicador *SymbolListSync.mq5*. Está diseñada para comprobar periódicamente la sincronización de todos los símbolos de *Observación de Mercado*. El periodo de comprobación lo establece el usuario en segundos en el parámetro *SyncCheckupPeriod*. Hace que el temporizador se inicie en *OnInit*.

```
#property indicator_chart_window
#property indicator_plots 0

input int SyncCheckupPeriod = 1; // SyncCheckupPeriod (seconds)

void OnInit()
{
   EventSetTimer(SyncCheckupPeriod);
}
```

En el manejador *OnTimer*, en un bucle, llamamos a *SymbolIsSynchronized* y recopilamos todos los símbolos no sincronizados en una cadena común, tras lo cual se muestran en el comentario y en el registro.

```
void OnTimer()
{
   string unsynced;
   const int n = SymbolsTotal(true);
   // check all symbols in the Market Watch
   for(int i = 0; i < n; ++i)
   {
      const string s = SymbolName(i, true);
      if(!SymbolIsSynchronized(s))
      {
         unsynced += s + "\n";
      }
   }

   if(StringLen(unsynced) > 0)
   {
      Comment("Unsynced symbols:\n" + unsynced);
      Print("Unsynced symbols:\n" + unsynced);
   }
   else
   {
      Comment("All Market Watch is in sync");
   }
}
```

Por ejemplo, si añadimos algún símbolo que faltaba (Brent) a *Observación de Mercado*, obtendremos una entrada como ésta:

```
Unsynced symbols:
Brent
```

En condiciones normales, la mayor parte del tiempo (mientras el indicador está funcionando) no debería haber mensajes de este tipo en el registro. No obstante, en caso de problemas de comunicación puede generarse una avalancha de alertas.

6.1.5 Obtener el último tick de un símbolo

En el capítulo sobre series temporales, en la sección Trabajar con arrays de ticks reales, introdujimos la estructura *MqlTick* integrada que contiene campos con valores de precio y volumen para un símbolo concreto, conocidos en el momento de cada cambio en las cotizaciones. En modo online, un programa MQL puede consultar los últimos precios y volúmenes recibidos utilizando la función *SymbolInfoTick* que adopta la misma estructura.

bool SymbolInfoTick(const string symbol, MqlTick &tick)

Para un símbolo con un nombre dado *symbol*, la función rellena la estructura *tick* pasada por referencia. Si tiene éxito, devuelve *true*.

Como sabe, los indicadores y Asesores Expertos son llamados automáticamente por el terminal a la llegada de un nuevo tick si contienen la descripción de los manejadores correspondientes *OnCalculate* y *OnTick*. No obstante, la información sobre el significado de los cambios de precios, el volumen de la última operación y la hora de generación del tick no se transfieren directamente a los manejadores. Se puede obtener información más detallada con la función *SymbolInfoTick*.

Los eventos de ticks se generan sólo para un símbolo del gráfico, por lo que ya hemos considerado la opción de obtener nuestro propio evento multisímbolo para ticks basado en eventos personalizados. En este caso, *SymbolInfoTick* permite leer información sobre ticks en símbolos de terceros acerca de la recepción de notificaciones.

Tomemos el indicador *EventTickSpy.mq5* y convirtámoslo en *SymbolTickSpy.mq5*, que solicitará la estructura *MqlTick* para el símbolo correspondiente en cada tick «multidivisa» y luego calculará y mostrará todos los diferenciales en el gráfico.

Vamos a añadir un nuevo parámetro de entrada *Index*. Será necesario para una nueva forma de enviar notificaciones: enviaremos sólo el índice del símbolo cambiado en el evento de usuario (ver más adelante).

```
#define TICKSPY 0xFEED // 65261

input string SymbolList =
   "EURUSD,GBPUSD,XAUUSD,USDJPY,USDCHF"; // List of symbols, comma separated (example
input ushort Message = TICKSPY;          // Custom message id
input long Chart = 0;                    // Receiving chart id (do not edit)
input int Index = 0;                     // Index in symbol list (do not edit)
```

Además, añadimos el array *Spreads* para almacenar diferenciales por símbolos y la variable *SelfIndex* para recordar la posición del símbolo del gráfico actual en la lista (si está incluido en la lista, que suele ser así). Esto último es necesario para llamar a nuestra nueva función de manejo de ticks desde *OnCalculate* en la copia original del indicador. Es más fácil y más correcto tomar un índice ya hecho para *_Symbol* de forma explícita y no devolvérnoslo en un evento a nosotros mismos.

```
int Spreads[];
int SelfIndex = -1;
```

Las estructuras de datos introducidas se inicializan en *OnInit*. Por lo demás, *OnInit* se ha mantenido sin cambios, incluido el lanzamiento de instancias subordinadas del indicador en símbolos de terceros (estas líneas se omiten aquí).

```
void OnInit()
{
   ...
         const int n = StringSplit(SymbolList, ',', Symbols);
         ArrayResize(Spreads, n);
         for(int i = 0; i < n; ++i)
         {
            if(Symbols[i] != _Symbol)
            {
               ...
            }
            else
            {
               SelfIndex = i;
            }
            Spreads[i] = 0;
         }
   ...
}
```

En el manejador *OnCalculate* generamos un evento personalizado en cada tick si la copia del indicador funciona en el otro símbolo (al mismo tiempo, el ID del gráfico *Chart* al que deben enviarse las notificaciones no es igual a 0). Tenga en cuenta que el único parámetro rellenado en el evento es *lparam*, que es igual a *Index* (*dparam* es 0, y *sparam* es NULL). Si *Chart* es igual a 0, significa que estamos en la copia principal del indicador trabajando sobre el símbolo gráfico *_Symbol*, y si se encuentra en la lista de símbolos de entrada, llamamos directamente a *OnSymbolTick* con el índice *SelfIndex* correspondiente.

```
int OnCalculate(const int rates_total, const int prev_calculated, const int, const do
{
   if(prev_calculated)
   {
      if(Chart > 0)
      {
         EventChartCustom(Chart, Message, Index, 0, NULL);
      }
      else if(SelfIndex > -1)
      {
         OnSymbolTick(SelfIndex);
      }
   }

   return rates_total;
}
```

En la parte receptora del algoritmo de eventos en *OnChartEvent*, también llamamos a *OnSymbolTick*, pero esta vez obtenemos el número de símbolo de la lista en *lparam* (lo que se envió como el parámetro *Index* desde otra copia del indicador).

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   if(id == CHARTEVENT_CUSTOM + Message)
   {
      OnSymbolTick((int)lparam);
   }
}
```

La función *OnSymbolTick* solicita la información completa de los ticks utilizando *SymbolInfoTick* y calcula el diferencial como la diferencia entre los precios de *Ask* y *Bid* dividida por el tamaño del punto (la propiedad SYMBOL_POINT se abordará más adelante).

```
void OnSymbolTick(const int index)
{
   const string symbol = Symbols[index];

   MqlTick tick;
   if(SymbolInfoTick(symbol, tick))
   {
      Spreads[index] = (int)MathRound((tick.ask - tick.bid)
         / SymbolInfoDouble(symbol, SYMBOL_POINT));
      string message = "";
      for(int i = 0; i < ArraySize(Spreads); ++i)
      {
         message += Symbols[i] + "=" + (string)Spreads[i] + "\n";
      }

      Comment(message);
   }
}
```

El nuevo diferencial actualiza la celda correspondiente del array *Spreads*, tras lo cual el array entero se muestra en el gráfico del comentario. He aquí el aspecto que tiene:

Diferenciales actuales para la lista de símbolos solicitada

Puede comparar en tiempo real la correspondencia de la información en el comentario y en la ventana *Observación de Mercado*.

6.1.6 Horarios de sesiones de trading y cotización

Un poco más adelante, en capítulos posteriores, hablaremos de las funciones de la API de MQL5 que nos permiten automatizar las operaciones de trading. Pero antes debemos estudiar las características técnicas de la plataforma, que determinan el éxito de la llamada a estas API. En concreto, las especificaciones de los instrumentos financieros imponen algunas restricciones. En este capítulo examinaremos gradualmente su análisis programático por entero, y empezaremos por un tema como las sesiones.

A la hora de negociar instrumentos financieros hay que tener en cuenta que muchos mercados internacionales, como las bolsas de valores, tienen horarios de apertura predeterminados, y la información y el trading sólo están disponibles durante ese horario. A pesar de que el terminal está constantemente conectado al servidor del bróker, los intentos de realizar una operación fuera del horario de trabajo fracasarán. En este sentido, el terminal almacena, para cada símbolo, un calendario de sesiones, es decir, los periodos de tiempo dentro de un día en los que se pueden realizar determinadas acciones.

Como sabe, hay dos tipos principales de sesiones: de cotización y de trading. Durante la sesión de cotización, el terminal recibe (puede recibir) cotizaciones actuales. Durante la sesión de trading está permitido enviar órdenes de trading y realizar transacciones. Durante el día puede haber varias sesiones de cada tipo, con descansos (por ejemplo, mañana y tarde). Es evidente que la duración de las sesiones de cotización es superior o igual a la de las sesiones de trading.

En cualquier caso, las horas de sesión, es decir, las horas de apertura y cierre, son traducidas por el terminal de la zona horaria local de la bolsa a la zona horaria del bróker (hora del servidor).

La API de MQL5 permite conocer las sesiones de cotización y trading de cada instrumento mediante las funciones *SymbolInfoSessionQuote* y *SymbolInfoSessionTrade*. En concreto, esta importante información permite al programa comprobar si el mercado está abierto en ese momento antes de enviar una solicitud de operación al servidor. Así, prevenimos el inevitable resultado erróneo y evitamos cargas innecesarias del servidor. Tenga en cuenta que, en el caso de solicitudes erróneas masivas al servidor debido a un programa MQL incorrectamente implementado, el servidor puede empezar a «ignorar» su terminal, negándose a ejecutar comandos posteriores (incluso los correctos) durante algún tiempo.

```
bool SymbolInfoSessionQuote(const string symbol, ENUM_DAY_OF_WEEK dayOfWeek, uint
sessionIndex, datetime &from, datetime &to)
bool SymbolInfoSessionTrade(const string symbol, ENUM_DAY_OF_WEEK dayOfWeek, uint
sessionIndex, datetime &from, datetime &to)
```

Las funciones funcionan de la misma manera. Para un *symbol* y día de la semana *dayOfWeek* determinados, rellenan los parámetros *from* y *to* pasados por referencia con las horas de apertura y cierre de la sesión con *sessionIndex*. La indexación de la sesión comienza en 0. La estructura ENUM_DAY_OF_WEEK se describió en la sección Enumeraciones.

No existen funciones separadas para consultar el número de sesiones: en lugar de ello, debemos ir llamando a *SymbolInfoSessionQuote* y*SymbolInfoSessionTrade* con índice creciente *sessionIndex*, hasta que la función devuelva una bandera de error (*false*). Cuando existe una sesión con el número especificado y los argumentos de salida *from* y *to* reciben valores correctos, las funciones devuelven un indicador de éxito (*true*).

Según la documentación de MQL5, en los valores recibidos de *from* y *to* de tipo *datetime*, la fecha debe ignorarse y sólo debe considerarse la hora. Esto se debe a que la información es un horario intradiario. No obstante, existe una importante excepción a esta regla.

Dado que el mercado está potencialmente abierto las 24 horas del día, como en el caso de Forex, o de una bolsa al otro lado del mundo, donde el horario comercial diurno coincide con el cambio de fechas en la «zona horaria» de su bróker, el final de las sesiones puede tener una hora igual o superior a 24 horas. Por ejemplo, si el inicio de las sesiones de Forex es a las 00:00, el final es a las 24:00. Sin embargo, desde el punto de vista del tipo *datetime*, 24 horas son las 00 horas 00 minutos del día siguiente.

La situación se vuelve más confusa para aquellas bolsas en las que el horario se desplaza varias horas con respecto a la zona horaria de su bróker, de manera que la sesión comienza en un día y termina en otro. Por ello, la variable *to* registra no sólo la hora, sino también un día extra que no se puede ignorar, porque de lo contrario la hora intradiaria *from* sería mayor que la hora intradiaria *to* (por ejemplo, una sesión puede durar desde las 21:00 de hoy hasta las 8:00 de mañana, es decir, 21 > 8). En este caso, la comprobación de la ocurrencia de la hora actual dentro de la sesión («la hora x es mayor que el inicio y menor que el final») resultará ser incorrecta (por ejemplo, la condición x >= 21 && x < 8 no se cumple para x = 23, aunque la sesión esté, de hecho, activa).

Así, llegamos a la conclusión de que es imposible ignorar la fecha en los parámetros de *from/to*, y este punto debe tenerse en cuenta en los algoritmos (véase el ejemplo).

Para demostrar las capacidades de las funciones, volvamos a un ejemplo de la secuencia de comandos *EnvPermissions.mq5* que se presentó en la sección Permisos. Uno de los tipos de permisos (o restricciones, si lo prefiere) se refiere específicamente a la disponibilidad de trading. Anteriormente, el script tenía en cuenta la configuración del terminal (TERMINAL_TRADE_ALLOWED) y la configuración de un programa MQL específico (MQL_TRADE_ALLOWED). Ahora podemos añadirle comprobaciones de

sesión para determinar los permisos de trading que son válidos en un momento dado para un símbolo concreto.

La nueva versión del script se llama *SymbolPermissions.mq5*. Tampoco es definitiva: en uno de los capítulos siguientes estudiaremos las limitaciones impuestas por la configuración de la cuenta de trading.

Recordemos que el script implementa la clase *Permissions*, que proporciona una descripción centralizada de todos los tipos de permisos/restricciones aplicables a los programas MQL. Entre otras cosas, la clase dispone de métodos para comprobar la disponibilidad del trading: *isTradeEnabled* y *isTradeOnSymbolEnabled*. El primero de ellos se refiere a los permisos globales y permanecerá prácticamente inalterado:

```
class Permissions
{
public:
   static bool isTradeEnabled(const string symbol = NULL, const datetime now = 0)
   {
      return TerminalInfoInteger(TERMINAL_TRADE_ALLOWED)
          && MQLInfoInteger(MQL_TRADE_ALLOWED)
          && isTradeOnSymbolEnabled(symbol == NULL ? _Symbol : symbol, now);
   }
   ...
```

Tras comprobar las propiedades del terminal y del programa MQL, el script pasa a *isTradeOnSymbolEnabled*, donde se analiza la especificación del símbolo. Antes, este método estaba prácticamente vacío.

Además del símbolo de trabajo pasado en el parámetro del símbolo, la función *isTradeOnSymbolEnabled* recibe la hora actual (*now*) y el modo de trading requerido (*mode*). Analizaremos este último aspecto con más detalle en las secciones siguientes (véase Permisos de trading). Por ahora, observemos que el valor por defecto de SYMBOL_TRADE_MODE_FULL ofrece la máxima libertad (se permiten todas las operaciones de trading).

```
static bool isTradeOnSymbolEnabled(string symbol, const datetime now = 0,
   const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
   // checking sessions
   bool found = now == 0;
   if(!found)
   {
      const static ulong day = 60 * 60 * 24;
      const ulong time = (ulong)now % day;
      datetime from, to;
      int i = 0;

      ENUM_DAY_OF_WEEK d = TimeDayOfWeek(now);

      while(!found && SymbolInfoSessionTrade(symbol, d, i++, from, to))
      {
         found = time >= (ulong)from && time < (ulong)to;
      }
   }
   // checking the trading mode for the symbol
   return found && (SymbolInfoInteger(symbol, SYMBOL_TRADE_MODE) == mode);
}
```

Si no se especifica el tiempo *now* (por defecto es igual a 0), consideramos que no nos interesan las sesiones. Esto significa que la variable *found* con una indicación de que se ha encontrado una sesión adecuada (es decir, una sesión que contiene la hora dada) se establece inmediatamente en *true*. Pero si se especifica el parámetro *now*, la función entra en el bloque de análisis de la sesión de trading.

Para extraer la hora sin tener en cuenta la fecha a partir de los valores del tipo *datetime*, describimos la constante *day* igual al número de segundos de un día. Una expresión como *now % day* devolverá el resto de dividir la fecha y hora completas por la duración de un día, lo que dará sólo la hora (los dígitos más significativos en *datetime* serán nulos).

La función *TimeDayOfWeek* devuelve el día de la semana para el valor *datetime* dado. Se encuentra en el archivo de encabezado *MQL5Book/DateTime.mqh* que ya hemos utilizado anteriormente (véase Fecha y hora).

Más adelante, en el bucle *while*, llamamos a la función *SymbolInfoSessionTrade* mientras incrementamos constantemente el índice de sesión *i* hasta que se encuentra una sesión adecuada o la función devuelve *false* (no hay más sesiones). De este modo, el programa puede obtener una lista completa de sesiones por día de la semana, similar a lo que se muestra en el terminal en la ventana del símbolo *Specifications*.

Obviamente, una sesión adecuada es la que contiene el valor *time* especificado entre las horas de inicio *from* y fin *to* de la sesión. Es aquí donde tenemos en cuenta el problema asociado al posible trading durante las veinticuatro horas del día: *from* y *to* se comparan con *time* «tal cual», sin descartar el día (*from % day* o *to % day*).

Una vez que *found* se hace igual a *true*, salimos del bucle. De lo contrario, el bucle finalizará cuando se supere el número de sesiones permitido (la función *SymbolInfoSessionTrade* devolverá *false*) y nunca se encontrará una sesión adecuada.

Si, según el horario de la sesión, el trading está permitido, comprobamos además el modo de trading del símbolo (SYMBOL_TRADE_MODE). Por ejemplo, el trading de símbolos puede estar completamente prohibido («indicativo») o estar en el modo «sólo posiciones de cierre».

El código anterior tiene algunas simplificaciones en comparación con la versión final del archivo *SymbolPermissions.mq5*. Además, implementa un mecanismo para marcar el origen de la restricción que provocó la desactivación del trading. Todas estas fuentes se resumen en la enumeración TRADE_RESTRICTIONS.

```
enum TRADE_RESTRICTIONS
{
   TERMINAL_RESTRICTION = 1,
   PROGRAM_RESTRICTION = 2,
   SYMBOL_RESTRICTION = 4,
   SESSION_RESTRICTION = 8,
};
```

Actualmente, la restricción puede proceder de 4 instancias: el terminal, el programa, el símbolo y el horario de la sesión. Más adelante añadiremos más opciones.

Para registrar el hecho de que se ha encontrado una restricción en la clase *Permissions*, tenemos la variable *lastFailReasonBitMask*, que permite recopilar una máscara de bits de los elementos de la enumeración mediante un método auxiliar *pass* (el bit se pone en marcha cuando la condición comprobada *value* es falsa, y el bit es igual a *false*).

```
static uint lastFailReasonBitMask;
static bool pass(const bool value, const uint bitflag)
{
   if(!value) lastFailReasonBitMask |= bitflag;
   return value;
}
```

La llamada al método *pass* con una bandera específica se realiza en los pasos de validación adecuados. Por ejemplo, el método *isTradeEnabled* completo tiene el siguiente aspecto:

```
static bool isTradeEnabled(const string symbol = NULL, const datetime now = 0)
{
   lastFailReasonBitMask = 0;
   return pass(TerminalInfoInteger(TERMINAL_TRADE_ALLOWED), TERMINAL_RESTRICTION)
       && pass(MQLInfoInteger(MQL_TRADE_ALLOWED), PROGRAM_RESTRICTION)
       && isTradeOnSymbolEnabled(symbol == NULL ? _Symbol : symbol, now);
}
```

Debido a esto, con un resultado negativo de la llamada *TerminalInfoInteger(TERMINAL_TRADE_ALLOWED)* o *MQLInfoInteger(MQL_TRADE_ALLOWED)*, se activará la bandera TERMINAL_RESTRICTION o PROGRAM_RESTRICTION, respectivamente.

El método *isTradeOnSymbolEnabled* también establece sus propias banderas cuando se detectan problemas, incluidas banderas de sesión.

```
static bool isTradeOnSymbolEnabled(string symbol, const datetime now = 0,
   const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
   ...
   return pass(found, SESSION_RESTRICTION)
      && pass(SymbolInfoInteger(symbol, SYMBOL_TRADE_MODE) == mode, SYMBOL_RESTRIC
}
```

Como resultado, el programa MQL que está utilizando la consulta *Permissions::isTradeEnabled*, después de recibir una restricción, puede aclarar su significado utilizando los métodos *getFailReasonBitMask* y *explainBitMask*: el primero devuelve la máscara de banderas de prohibición establecidas «tal cual», y el segundo forma una descripción de texto fácil de usar de las restricciones.

```
static uint getFailReasonBitMask()
{
   return lastFailReasonBitMask;
}

static string explainBitMask()
{
   string result = "";
   for(int i = 0; i < 4; ++i)
   {
      if(((1 << i) & lastFailReasonBitMask) != 0)
      {
         result += EnumToString((TRADE_RESTRICTIONS)(1 << i));
      }
   }
   return result;
}
```

Con la clase *Permissions* anterior en el manejador *OnStart* se comprueba la disponibilidad de trading de todos los símbolos de *Observación de Mercado* (actualmente, *TimeCurrent*).

```
void OnStart()
{
   string disabled = "";

   const int n = SymbolsTotal(true);
   for(int i = 0; i < n; ++i)
   {
      const string s = SymbolName(i, true);
      if(!Permissions::isTradeEnabled(s, TimeCurrent()))
      {
         disabled += s + "=" + Permissions::explainBitMask() +"\n";
      }
   }
   if(disabled != "")
   {
      Print("Trade is disabled for the following symbols and origins:");
      Print(disabled);
   }
}
```

Si la negociación está prohibida para un símbolo determinado, veremos una explicación en el registro.

```
Trade is disabled for following symbols and origins:
USDRUB=SESSION_RESTRICTION
SP500m=SYMBOL_RESTRICTION
```

En este caso, el mercado está cerrado para «USDRUB» y el trading está desactivado para el símbolo «SP500m» (más estrictamente, no corresponde al modo SYMBOL_TRADE_MODE_FULL).

Se da por sentado que, al ejecutar el script, el trading algorítmico estaba activado globalmente en el terminal. De lo contrario, veremos adicionalmente las prohibiciones TERMINAL_RESTRICTION y PROGRAM_RESTRICTION en el registro.

6.1.7 Coeficientes de margen de los símbolos

Entre las características de la especificación de símbolos disponibles en la API de MQL5, que trataremos en detalle en secciones posteriores, hay varias características relacionadas con los requisitos en materia de márgenes, que se aplican al abrir y mantener posiciones de trading. Debido al hecho de que el terminal ofrece trading en diferentes mercados y diferentes tipos de instrumentos, estos requisitos pueden variar significativamente. De forma generalizada, esto se expresa en la aplicación de tasas de corrección de los márgenes que se fijan individualmente para los símbolos y los distintos tipos de operaciones de trading. Para el usuario, las tasas se muestran en el terminal en la ventana *Specifications*.

Como veremos a continuación, el multiplicador (si se aplica) se multiplica por el valor del margen de las propiedades del símbolo. El coeficiente de margen puede obtenerse mediante programación utilizando la función *SymbolInfoMarginRate*.

bool SymbolInfoMarginRate(const string symbol, ENUM_ORDER_TYPE orderType, double &initial, double &maintenance)

Para el símbolo y el tipo de orden especificados (ENUM_ORDER_TYPE), la función rellena los parámetros pasados por referencia *initial* y *maintenance* con los coeficientes de margen inicial y de

mantenimiento, respectivamente. Las tasas resultantes deben multiplicarse por el valor del margen del tipo correspondiente (la forma de solicitarlo se describe en la sección sobre requisitos de margen) para obtener el importe que se reservará en la cuenta al realizar una orden como *orderType*.

La función devuelve *true* en caso de éxito.

Utilicemos como ejemplo un sencillo script *SymbolMarginRate.mq5*, que muestra los coeficientes de margen de *Observación de Mercado* o de todos los símbolos disponibles, en función del parámetro *MarketWatchOnly*. El tipo de operación puede especificarse en el parámetro *OrderType*.

```
#include <MQL5Book/MqlError.mqh>

input bool MarketWatchOnly = true;
input ENUM_ORDER_TYPE OrderType = ORDER_TYPE_BUY;

void OnStart()
{
   const int n = SymbolsTotal(MarketWatchOnly);
   PrintFormat("Margin rates per symbol for %s:", EnumToString(OrderType));
   for(int i = 0; i < n; ++i)
   {
      const string s = SymbolName(i, MarketWatchOnly);
      double initial = 1.0, maintenance = 1.0;
      if(!SymbolInfoMarginRate(s, OrderType, initial, maintenance))
      {
         PrintFormat("Error: %s(%d)", E2S(_LastError), _LastError);
      }
      PrintFormat("%4d %s = %f %f", i, s, initial, maintenance);
   }
}
```

A continuación se muestra el registro:

```
Margin rates per symbol for ORDER_TYPE_BUY:
   0 EURUSD = 1.000000 0.000000
   1 XAUUSD = 1.000000 0.000000
   2 BTCUSD = 0.330000 0.330000
   3 USDCHF = 1.000000 0.000000
   4 USDJPY = 1.000000 0.000000
   5 AUDUSD = 1.000000 0.000000
   6 USDRUB = 1.000000 1.000000
```

Puede comparar los valores recibidos con las especificaciones de símbolos del terminal.

6.1.8 Visión general de las funciones para obtener las propiedades de los símbolos

La especificación completa de cada símbolo puede obtenerse consultando sus propiedades: para ello, la API de MQL5 proporciona tres funciones, a saber, *SymbolInfoInteger*, *SymbolInfoDouble* y *SymbolInfoString*, cada una de las cuales se encarga de las propiedades de un tipo concreto. Las propiedades se describen como miembros de tres enumeraciones: ENUM_SYMBOL_INFO_INTEGER,

ENUM_SYMBOL_INFO_DOUBLE, and ENUM_SYMBOL_INFO_STRING, respectively. Una técnica similar se utiliza en las API de gráficos y objetos que ya conocemos.

El nombre del símbolo y el identificador de la propiedad solicitada se pasan a cualquiera de las funciones.

Cada una de las funciones se presenta de dos formas: abreviada y completa. La versión abreviada devuelve directamente la propiedad solicitada, mientras que la completa la escribe en el parámetro out pasado por referencia. Por ejemplo, para las propiedades compatibles con un tipo entero, las funciones tienen prototipos como éste:

```
long SymbolInfoInteger(const string symbol, ENUM_SYMBOL_INFO_INTEGER property)
bool SymbolInfoInteger(const string symbol, ENUM_SYMBOL_INFO_INTEGER property, long &value)
```

La segunda forma devuelve un indicador booleano de éxito (*true*) o error (*false*). Las razones más plausibles por las que una función puede devolver *false* incluyen un nombre de símbolo no válido (MARKET_UNKNOWN_SYMBOL, 4301) o un identificador no válido para la propiedad solicitada (MARKET_WRONG_PROPERTY, 4303). Los detalles figuran en *_LastError*.

Como antes, las propiedades de la enumeración ENUM_SYMBOL_INFO_INTEGER son de varios tipos compatibles con enteros: *bool*, *int*, *long*, *color*, *datetime* y enumeraciones especiales (todas ellas se abordarán en secciones separadas).

Para las propiedades con un tipo de número real, se definen las siguientes dos formas de la función *SymbolInfoDouble*:

```
double SymbolInfoDouble(const string symbol, ENUM_SYMBOL_INFO_DOUBLE property)
bool SymbolInfoDouble(const string symbol, ENUM_SYMBOL_INFO_DOUBLE property, double &value)
```

Por último, para las propiedades de cadena, las funciones similares tienen el siguiente aspecto:

```
string SymbolInfoString(const string symbol, ENUM_SYMBOL_INFO_STRING property)
bool SymbolInfoString(const string symbol, ENUM_SYMBOL_INFO_STRING property, string &value)
```

Las propiedades de varios tipos que se utilizarán a menudo más adelante al desarrollar Asesores Expertos se agrupan lógicamente en las descripciones de las siguientes secciones de este capítulo.

Basándonos en las funciones anteriores, crearemos una clase universal *SymbolMonitor* (archivo *SymbolMonitor.mqh*) para obtener las propiedades de cualquier símbolo. Se basará en un conjunto de métodos sobrecargados *get* para tres enumeraciones.

```
class SymbolMonitor
{
public:
   const string name;
   SymbolMonitor(): name(_Symbol) { }
   SymbolMonitor(const string s): name(s) { }

   long get(const ENUM_SYMBOL_INFO_INTEGER property) const
   {
      return SymbolInfoInteger(name, property);
   }

   double get(const ENUM_SYMBOL_INFO_DOUBLE property) const
   {
      return SymbolInfoDouble(name, property);
   }

   string get(const ENUM_SYMBOL_INFO_STRING property) const
   {
      return SymbolInfoString(name, property);
   }
   ...
```

Los otros tres métodos similares permiten eliminar el tipo de enumeración en el primer parámetro y seleccionar la sobrecarga necesaria por parte del compilador gracias al segundo parámetro ficticio (su tipo aquí siempre coincide con el tipo de resultado). Lo utilizaremos en futuras clases de plantillas.

```
   long get(const int property, const long) const
   {
      return SymbolInfoInteger(name, (ENUM_SYMBOL_INFO_INTEGER)property);
   }

   double get(const int property, const double) const
   {
      return SymbolInfoDouble(name, (ENUM_SYMBOL_INFO_DOUBLE)property);
   }

   string get(const int property, const string) const
   {
      return SymbolInfoString(name, (ENUM_SYMBOL_INFO_STRING)property);
   }
   ...
```

Así, mediante la creación de un objeto con el nombre de símbolo deseado, se pueden consultar uniformemente sus propiedades de cualquier tipo. Para consultar y registrar todas las propiedades del mismo tipo podríamos implementar algo como esto:

```
// project (draft)
template<typename E,typename R>
void list2log()
{
   E e = (E)0;
   int array[];
   const int n = EnumToArray(e, array, 0, USHORT_MAX);
   for(int i = 0; i < n; ++i)
   {
      e = (E)array[i];
      R r = get(e);
      PrintFormat("% 3d %s=%s", i, EnumToString(e), (string)r);
   }
}
```

Sin embargo, debido al hecho de que en las propiedades del tipo *long* en realidad se «ocultan» valores de otros tipos, que deben mostrarse de una forma específica (por ejemplo, llamando a *EnumToString* para enumeraciones, *imeToString* para fecha y hora, etc.), tiene sentido definir otros tres métodos sobrecargados que devuelvan una representación de cadena de la propiedad. Los llamaremos *stringify*. Entonces, en el proyecto *list2log* anterior, es posible utilizar *stringify* en lugar de valores de conversión a *(string)*, y el propio método eliminará un parámetro de plantilla.

```
template<typename E>
void list2log()
{
   E e = (E)0;
   int array[];
   const int n = EnumToArray(e, array, 0, USHORT_MAX);
   for(int i = 0; i < n; ++i)
   {
      e = (E)array[i];
      PrintFormat("% 3d %s=%s", i, EnumToString(e), stringify(e));
   }
}
```

Para los tipos real y cadena, la implementación de *stringify* parece bastante sencilla.

```
string stringify(const ENUM_SYMBOL_INFO_DOUBLE property, const string format = NUL
{
   if(format == NULL) return (string)SymbolInfoDouble(name, property);
   return StringFormat(format, SymbolInfoDouble(name, property));
}

string stringify(const ENUM_SYMBOL_INFO_STRING property) const
{
   return SymbolInfoString(name, property);
}
```

Sin embargo, para ENUM_SYMBOL_INFO_INTEGER, todo es un poco más complicado. Por supuesto, cuando la propiedad es del tipo *long* o *int*, basta con convertirla a *(string)*. Todos los demás casos deben analizarse y convertirse individualmente en el operador *switch*.

```
string stringify(const ENUM_SYMBOL_INFO_INTEGER property) const
{
   const long v = SymbolInfoInteger(name, property);
   switch(property)
   {
      ...
   }

   return (string)v;
}
```

Por ejemplo, si una propiedad es de tipo booleano, es conveniente representarla con la cadena «true» o «false» (así será visualmente diferente de los simples números 1 y 0). De cara al futuro, y por poner un ejemplo, digamos que entre las propiedades se encuentra SYMBOL_EXIST, que equivale a la función *SymbolExist*, es decir, devuelve una indicación booleana de si el carácter especificado existe. Para su procesamiento y otras propiedades lógicas tiene sentido implementar un método auxiliar *boolean*.

```
static string boolean(const long v)
{
   return v ? "true" : "false";
}

string stringify(const ENUM_SYMBOL_INFO_INTEGER property) const
{
   const long v = SymbolInfoInteger(name, property);
   switch(property)
   {
      case SYMBOL_EXIST:
         return boolean(v);
      ...
   }

   return (string)v;
}
```

Para las propiedades que son enumeraciones, la solución más adecuada sería un método de plantilla que utilice la función *EnumToString*.

```
template<typename E>
static string enumstr(const long v)
{
   return EnumToString((E)v);
}
```

Por ejemplo, la propiedad SYMBOL_SWAP_ROLLOVER3DAYS determina en qué día de la semana se carga un swap triple en las posiciones abiertas de un símbolo, y esta propiedad tiene el tipo ENUM_DAY_OF_WEEK. Así, para procesarlo, podemos escribir lo siguiente dentro de *switch*:

```
      case SYMBOL_SWAP_ROLLOVER3DAYS:
         return enumstr<ENUM_DAY_OF_WEEK>(v);
```

Un caso especial lo presentan las propiedades cuyos valores son combinaciones de banderas de bits. En concreto, para cada símbolo, el bróker establece permisos para órdenes de tipos específicos, como mercado, límite, stop loss, take profit, etc. (analizaremos estos permisos por separado). Cada tipo de

orden se indica mediante una constante con un bit habilitado, por lo que su superposición (combinada mediante el operador bitwise OR '|') se almacena en la propiedad SYMBOL_ORDER_MODE, y en ausencia de restricciones, todos los bits se habilitan al mismo tiempo. Para tales propiedades definiremos nuestras propias enumeraciones en nuestro archivo de encabezado; por ejemplo:

```
enum SYMBOL_ORDER
{
   _SYMBOL_ORDER_MARKET = 1,
   _SYMBOL_ORDER_LIMIT = 2,
   _SYMBOL_ORDER_STOP = 4,
   _SYMBOL_ORDER_STOP_LIMIT = 8,
   _SYMBOL_ORDER_SL = 16,
   _SYMBOL_ORDER_TP = 32,
   _SYMBOL_ORDER_CLOSEBY = 64,
};
```

Aquí, para cada constante integrada, como SYMBOL_ORDER_MARKET, se declara un elemento correspondiente, cuyo identificador es el mismo que la constante pero va precedido de un guion bajo para evitar conflictos de nombres.

Para representar combinaciones de banderas de tales enumeraciones en forma de cadena, implementamos otro método de plantilla, *maskstr*.

```
template<typename E>
static string maskstr(const long v)
{
   string text = "";
   for(int i = 0; ; ++i)
   {
      ResetLastError();
      const string s = EnumToString((E)(1 << i));
      if(_LastError != 0)
      {
         break;
      }
      if((v & (1 << i)) != 0)
      {
         text += s + " ";
      }
   }
   return text;
}
```

Su significado es como *enumstr*, pero se llama a la función *EnumToString* por cada bit habilitado en el valor de la propiedad, tras lo cual se «pegan» las cadenas resultantes.

Ahora es posible procesar SYMBOL_ORDER_MODE en la sentencia *switch* de forma similar:

```
      case SYMBOL_ORDER_MODE:
         return maskstr<SYMBOL_ORDER>(v);
```

Aquí está el código completo del método *stringify* para ENUM_SYMBOL_INFO_INTEGER. Nos iremos familiarizando poco a poco con todas las propiedades y enumeraciones en las siguientes secciones.

```
string stringify(const ENUM_SYMBOL_INFO_INTEGER property) const
{
   const long v = SymbolInfoInteger(name, property);
   switch(property)
   {
      case SYMBOL_SELECT:
      case SYMBOL_SPREAD_FLOAT:
      case SYMBOL_VISIBLE:
      case SYMBOL_CUSTOM:
      case SYMBOL_MARGIN_HEDGED_USE_LEG:
      case SYMBOL_EXIST:
         return boolean(v);
      case SYMBOL_TIME:
         return TimeToString(v, TIME_DATE|TIME_SECONDS);
      case SYMBOL_TRADE_CALC_MODE:
         return enumstr<ENUM_SYMBOL_CALC_MODE>(v);
      case SYMBOL_TRADE_MODE:
         return enumstr<ENUM_SYMBOL_TRADE_MODE>(v);
      case SYMBOL_TRADE_EXEMODE:
         return enumstr<ENUM_SYMBOL_TRADE_EXECUTION>(v);
      case SYMBOL_SWAP_MODE:
         return enumstr<ENUM_SYMBOL_SWAP_MODE>(v);
      case SYMBOL_SWAP_ROLLOVER3DAYS:
         return enumstr<ENUM_DAY_OF_WEEK>(v);
      case SYMBOL_EXPIRATION_MODE:
         return maskstr<SYMBOL_EXPIRATION>(v);
      case SYMBOL_FILLING_MODE:
         return maskstr<SYMBOL_FILLING>(v);
      case SYMBOL_START_TIME:
      case SYMBOL_EXPIRATION_TIME:
         return TimeToString(v);
      case SYMBOL_ORDER_MODE:
         return maskstr<SYMBOL_ORDER>(v);
      case SYMBOL_OPTION_RIGHT:
         return enumstr<ENUM_SYMBOL_OPTION_RIGHT>(v);
      case SYMBOL_OPTION_MODE:
         return enumstr<ENUM_SYMBOL_OPTION_MODE>(v);
      case SYMBOL_CHART_MODE:
         return enumstr<ENUM_SYMBOL_CHART_MODE>(v);
      case SYMBOL_ORDER_GTC_MODE:
         return enumstr<ENUM_SYMBOL_ORDER_GTC_MODE>(v);
      case SYMBOL_SECTOR:
         return enumstr<ENUM_SYMBOL_SECTOR>(v);
      case SYMBOL_INDUSTRY:
         return enumstr<ENUM_SYMBOL_INDUSTRY>(v);
      case SYMBOL_BACKGROUND_COLOR: // Bytes: Transparency Blue Green Red
         return StringFormat("TBGR(0x%08X)", v);
   }

   return (string)v;
}
```

Para probar la clase *SymbolMonitor* hemos creado un sencillo script *SymbolMonitor.mq5*. Registra todas las propiedades del símbolo del gráfico de trabajo.

```
#include <MQL5Book/SymbolMonitor.mqh>

void OnStart()
{
   SymbolMonitor m;
   m.list2log<ENUM_SYMBOL_INFO_INTEGER>();
   m.list2log<ENUM_SYMBOL_INFO_DOUBLE>();
   m.list2log<ENUM_SYMBOL_INFO_STRING>();
}
```

Por ejemplo, si ejecutamos el script en el gráfico EURUSD, podemos obtener los siguientes registros (facilitados de forma abreviada):

```
ENUM_SYMBOL_INFO_INTEGER Count=36
  0 SYMBOL_SELECT=true
  ...
  4 SYMBOL_TIME=2022.01.12 10:52:22
  5 SYMBOL_DIGITS=5
  6 SYMBOL_SPREAD=0
  7 SYMBOL_TICKS_BOOKDEPTH=10
  8 SYMBOL_TRADE_CALC_MODE=SYMBOL_CALC_MODE_FOREX
  9 SYMBOL_TRADE_MODE=SYMBOL_TRADE_MODE_FULL
 10 SYMBOL_TRADE_STOPS_LEVEL=0
 11 SYMBOL_TRADE_FREEZE_LEVEL=0
 12 SYMBOL_TRADE_EXEMODE=SYMBOL_TRADE_EXECUTION_INSTANT
 13 SYMBOL_SWAP_MODE=SYMBOL_SWAP_MODE_POINTS
 14 SYMBOL_SWAP_ROLLOVER3DAYS=WEDNESDAY
 15 SYMBOL_SPREAD_FLOAT=true
 16 SYMBOL_EXPIRATION_MODE=_SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY »
    _SYMBOL_EXPIRATION_SPECIFIED _SYMBOL_EXPIRATION_SPECIFIED_DAY
 17 SYMBOL_FILLING_MODE=_SYMBOL_FILLING_FOK
 ...
 23 SYMBOL_ORDER_MODE=_SYMBOL_ORDER_MARKET _SYMBOL_ORDER_LIMIT _SYMBOL_ORDER_STOP »
    _SYMBOL_ORDER_STOP_LIMIT _SYMBOL_ORDER_SL _SYMBOL_ORDER_TP _SYMBOL_ORDER_CLOSEBY
 ...
 26 SYMBOL_VISIBLE=true
 27 SYMBOL_CUSTOM=false
 28 SYMBOL_BACKGROUND_COLOR=TBGR(0xFF000000)
 29 SYMBOL_CHART_MODE=SYMBOL_CHART_MODE_BID
 30 SYMBOL_ORDER_GTC_MODE=SYMBOL_ORDERS_GTC
 31 SYMBOL_MARGIN_HEDGED_USE_LEG=false
 32 SYMBOL_EXIST=true
 33 SYMBOL_TIME_MSC=1641984742149
 34 SYMBOL_SECTOR=SECTOR_CURRENCY
 35 SYMBOL_INDUSTRY=INDUSTRY_UNDEFINED
ENUM_SYMBOL_INFO_DOUBLE Count=57
  0 SYMBOL_BID=1.13681
  1 SYMBOL_BIDHIGH=1.13781
  2 SYMBOL_BIDLOW=1.13552
  3 SYMBOL_ASK=1.13681
  4 SYMBOL_ASKHIGH=1.13781
  5 SYMBOL_ASKLOW=1.13552
 ...
 12 SYMBOL_POINT=1e-05
 13 SYMBOL_TRADE_TICK_VALUE=1.0
 14 SYMBOL_TRADE_TICK_SIZE=1e-05
 15 SYMBOL_TRADE_CONTRACT_SIZE=100000.0
 16 SYMBOL_VOLUME_MIN=0.01
 17 SYMBOL_VOLUME_MAX=500.0
 18 SYMBOL_VOLUME_STEP=0.01
 19 SYMBOL_SWAP_LONG=-0.7
 20 SYMBOL_SWAP_SHORT=-1.0
 21 SYMBOL_MARGIN_INITIAL=0.0
 22 SYMBOL_MARGIN_MAINTENANCE=0.0
```

```
  ...
  28 SYMBOL_TRADE_TICK_VALUE_PROFIT=1.0
  29 SYMBOL_TRADE_TICK_VALUE_LOSS=1.0
  ...
  43 SYMBOL_MARGIN_HEDGED=100000.0
  ...
  47 SYMBOL_PRICE_CHANGE=0.0132
ENUM_SYMBOL_INFO_STRING Count=15
   0 SYMBOL_BANK=
   1 SYMBOL_DESCRIPTION=Euro vs US Dollar
   2 SYMBOL_PATH=Forex\EURUSD
   3 SYMBOL_CURRENCY_BASE=EUR
   4 SYMBOL_CURRENCY_PROFIT=USD
   5 SYMBOL_CURRENCY_MARGIN=EUR
  ...
  13 SYMBOL_SECTOR_NAME=Currency
```

En concreto, puede ver que los precios de los símbolos se emiten con 5 dígitos (SYMBOL_DIGITS), el símbolo existe (SYMBOL_EXIST), el tamaño del contrato es 100000.0 (SYMBOL_TRADE_CONTRACT_SIZE), etc. Toda la información corresponde a la especificación.

6.1.9 Comprobar el estado de los símbolos

Antes hemos visto varias funciones relacionadas con el estado de un símbolo. Recordemos que *SymbolExist* se utiliza para comprobar la existencia de un símbolo, y *SymbolSelect* se utiliza para comprobar la inclusión o exclusión de la lista *Observación de Mercado*. Entre las propiedades del símbolo, hay varias banderas de propósito similar, cuyo uso tiene ventajas y desventajas en comparación con las funciones anteriores.

En concreto, la propiedad SYMBOL_SELECT permite averiguar si el símbolo especificado está seleccionado en *Observación de Mercado*, mientras que la función *SymbolSelect* modifica esta propiedad.

La función *SymbolExist*, a diferencia de la propiedad similar SYMBOL_EXIST, rellena adicionalmente la variable de salida con una indicación de que el símbolo es un símbolo definido por el usuario. Al consultar las propiedades, sería necesario analizar estos dos atributos por separado, ya que el atributo del símbolo personalizado se almacena en otra propiedad, SYMBOL_CUSTOM. Sin embargo, en algunos casos, el programa puede necesitar sólo una propiedad, y entonces la posibilidad de una consulta independiente se convierte en una ventaja.

Todas las banderas son valores booleanos obtenidos a través de la función *SymbolInfoInteger*.

Identificador	Descripción
SYMBOL_EXIST	Indica que existe un símbolo con el nombre dado.
SYMBOL_SELECT	Indica que el símbolo está seleccionado en *Observación de Mercado*
SYMBOL_VISIBLE	Indica que el símbolo especificado se muestra en *Observación de Mercado*

Especialmente interesante es SYMBOL_VISIBLE. El hecho es que algunos símbolos (por regla general, se trata de tipos de cambio cruzados que son necesarios para calcular los requisitos de margen y el beneficio en la divisa de depósito) se seleccionan en *Observación de Mercado* automáticamente y no aparecen en la lista visible para el usuario. Tales símbolos deben ser elegidos explícitamente (por el usuario o mediante programación) para que se muestren. Así, es la propiedad SYMBOL_VISIBLE la que permite determinar si un símbolo es visible en la ventana: puede ser igual a *false* para algunos elementos de la lista, obtenidos mediante un par de funciones *SymbolsTotal* y *SymbolName* con el parámetro *selected* igual a *true*.

Consideremos un sencillo script (*SymbolInvisible.mq5*) que busca en el terminal los símbolos seleccionados implícitamente, es decir, aquellos que no se muestran en Observación de Mercado (se restablece SYMBOL_VISIBLE) mientras SYMBOL_SELECT para ellos es igual a *true*.

```
#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1) - 1] = V)

void OnStart()
{
   const int n = SymbolsTotal(false);
   int selected = 0;
   string invisible[];
   // loop through all available symbols
   for(int i = 0; i < n; ++i)
   {
      const string s = SymbolName(i, false);
      if(SymbolInfoInteger(s, SYMBOL_SELECT))
      {
         selected++;
         if(!SymbolInfoInteger(s, SYMBOL_VISIBLE))
         {
            // collect selected but invisible symbols into an array
            PUSH(invisible, s);
         }
      }
   }
   PrintFormat("Symbols: total=%d, selected=%d, implicit=%d",
      n, selected, ArraySize(invisible));
   if(ArraySize(invisible))
   {
      ArrayPrint(invisible);
   }
}
```

Pruebe a compilar y ejecutar el script en diferentes cuentas. No siempre se da la situación de que un símbolo se seleccione implícitamente. Por ejemplo, si en *Observación de Mercado* se seleccionan los tickers de los blue chips rusos que cotizan en rublos y la cuenta de trading está en otra divisa (por ejemplo, dólares o euros, pero no rublos), se seleccionará automáticamente el símbolo USDRUB. Por supuesto, esto supone que no se ha añadido previamente a *Observación de Mercado* de forma explícita. Obtenemos el siguiente resultado en el registro:

```
Symbols: total=50681, selected=49, implicit=1
"USDRUB"
```

6.1.10 Tipo de precio para construir gráficos de símbolos

Las barras en los gráficos de precios de MetaTrader 5 se pueden trazar basándose en los precios *Bid* o *Last*, y el tipo de trazado se indica en la especificación de cada instrumento. Un programa MQL puede encontrar esta característica llamando a la función *SymbolInfoInteger* para la propiedad SYMBOL_CHART_MODE. El valor devuelto es un miembro de la enumeración ENUM_SYMBOL_CHART_MODE.

Identificador	Descripción
SYMBOL_CHART_MODE_BID	Las barras se construyen a los precios de oferta.
SYMBOL_CHART_MODE_LAST	Las barras se construyen a los precios de la última transacción.

El modo con precios *Last* se utiliza para los símbolos negociados en bolsas (a diferencia del mercado de divisas descentralizado), y Profundidad de Mercado está disponible para estos símbolos. La profundidad del mercado puede determinarse en función de la propiedad SYMBOL_TICKS_BOOKDEPTH.

La propiedad SYMBOL_CHART_MODE es útil para ajustar las señales de indicadores o estrategias que se construyen, por ejemplo, a los precios *Last* del gráfico, mientras que las órdenes se ejecutarán «al precio de mercado», es decir, a los precios *Ask* o *Bid* dependiendo de la dirección.

Además, el tipo de precio es necesario para calcular las barras del instrumento personalizado: si depende de símbolos estándar, puede tener sentido considerar su configuración por tipo de precio. Cuando el usuario introduce la fórmula del instrumento sintético en la ventana *Custom Symbol* (que se abre seleccionando *Create Symbol* en el cuadro de diálogo *Symbols*), es posible seleccionar los tipos de precios según las especificaciones de los respectivos símbolos estándar utilizados. Sin embargo, cuando el algoritmo de cálculo se forma en un programa MQL, es responsable precisamente de la correcta elección del tipo de precio.

En primer lugar, vamos a recopilar estadísticas sobre el uso de los precios *Bid* y *Last* para construir gráficos en una cuenta específica. Esto es lo que hará el script *SymbolStatsByPriceType.mq5*:

```
const bool MarketWatchOnly = false;

void OnStart()
{
   const int n = SymbolsTotal(MarketWatchOnly);
   int k = 0;
   // loop through all available characters
   for(int i = 0; i < n; ++i)
   {
      if(SymbolInfoInteger(SymbolName(i, MarketWatchOnly), SYMBOL_CHART_MODE)
          == SYMBOL_CHART_MODE_LAST)
      {
         k++;
      }
   }
   PrintFormat("Symbols in total: %d", n);
   PrintFormat("Symbols using price types: Bid=%d, Last=%d", n - k, k);
}
```

Pruébelo en diferentes cuentas (algunas pueden no tener símbolos bursátiles). He aquí cuál podría ser el resultado:

```
Symbols in total: 52304
Symbols using price types: Bid=229, Last=52075
```

Un ejemplo más práctico es el indicador *SymbolBidAskChart.mq5*, diseñado para dibujar un diagrama en forma de barras generadas a partir de los precios del tipo especificado. Esto le permitirá comparar las velas de un gráfico que utiliza precios de la propiedad SYMBOL_CHART_MODE para su construcción con barras de un tipo de precio alternativo. Por ejemplo, puede ver barras al precio *Bid* en el gráfico del instrumento al precio *Last* u obtener barras para el precio *Ask*, que los gráficos estándar del terminal no admiten.

Como base para un nuevo indicador, tomaremos un indicador *IndDeltaVolume.mq5* ya elaborado que se presentó en la sección Esperar datos y gestionar la visibilidad. En ese indicador, descargamos un historial de ticks para un determinado número de barras *BarCount* y calculamos el delta de volúmenes, es decir, los volúmenes de compra y de venta por separado. En el nuevo indicador sólo tenemos que reemplazar el algoritmo de cálculo con la búsqueda de los precios *Open*, *High*, *Low* y *Close* basados en ticks dentro de cada barra.

La configuración del indicador incluye cuatro búferes y un gráfico de barras (DRAW_BARS) que se muestra en la ventana principal.

```
#property indicator_chart_window
#property indicator_buffers 4
#property indicator_plots   1

#property indicator_type1   DRAW_BARS
#property indicator_color1  clrDodgerBlue
#property indicator_width1  2
#property indicator_label1  "Open;High;Low;Close;"
```

La visualización en forma de barras se elige para facilitar su lectura cuando se ejecutan sobre las velas del gráfico principal, de modo que ambas versiones de cada barra sean visibles.

El nuevo parámetro de entrada *ChartMode* permite al usuario seleccionar uno de los tres tipos de precio (tenga en cuenta que *Ask* es nuestra adición respecto al conjunto estándar de elementos de ENUM_SYMBOL_CHART_MODE).

```
enum ENUM_SYMBOL_CHART_MODE_EXTENDED
{
   _SYMBOL_CHART_MODE_BID,  // SYMBOL_CHART_MODE_BID
   _SYMBOL_CHART_MODE_LAST, // SYMBOL_CHART_MODE_LAST
   _SYMBOL_CHART_MODE_ASK,  // SYMBOL_CHART_MODE_ASK*
};

input int BarCount = 100;
input COPY_TICKS TickType = INFO_TICKS;
input ENUM_SYMBOL_CHART_MODE_EXTENDED ChartMode = _SYMBOL_CHART_MODE_BID;
```

La antigua clase *CalcDeltaVolume* ha cambiado su nombre por el de *CalcCustomBars*, pero ha permanecido casi inalterada. Las diferencias incluyen un nuevo conjunto de cuatro búferes y el campo *chartMode*, que se inicializa en el constructor a partir de la variable de entrada *ChartMode*.

```
class CalcCustomBars
{
   const int limit;
   const COPY_TICKS tickType;
   const ENUM_SYMBOL_CHART_MODE_EXTENDED chartMode;

   double open[];
   double high[];
   double low[];
   double close[];
   ...
public:
   CalcCustomBars(
      const int bars,
      const COPY_TICKS type,
      const ENUM_SYMBOL_CHART_MODE_EXTENDED mode)
      : limit(bars), tickType(type), chartMode(mode) ...
   {
      // register arrays as indicator buffers
      SetIndexBuffer(0, open);
      SetIndexBuffer(1, high);
      SetIndexBuffer(2, low);
      SetIndexBuffer(3, close);
      const static string defTitle[] = {"Open;High;Low;Close;"};
      const static string types[] = {"Bid", "Last", "Ask"};
      string name = defTitle[0];
      StringReplace(name, ";", types[chartMode] + ";");
      PlotIndexSetString(0, PLOT_LABEL, name);
      IndicatorSetInteger(INDICATOR_DIGITS, _Digits);
   }
   ...
```

Dependiendo del modo de *chartMode*, el método auxiliar *price* devuelve un tipo de precio específico de cada tick.

```
protected:
   double price(const MqlTick &t) const
   {
      switch(chartMode)
      {
      case _SYMBOL_CHART_MODE_BID:
         return t.bid;
      case _SYMBOL_CHART_MODE_LAST:
         return t.last;
      case _SYMBOL_CHART_MODE_ASK:
         return t.ask;
      }
      return 0; // error
   }
   ...
```

Usando el método *price* podemos implementar fácilmente la modificación del método de cálculo principal *calc* que llena los búferes para la barra numerada *i* basándose en un array de *ticks* para esta barra.

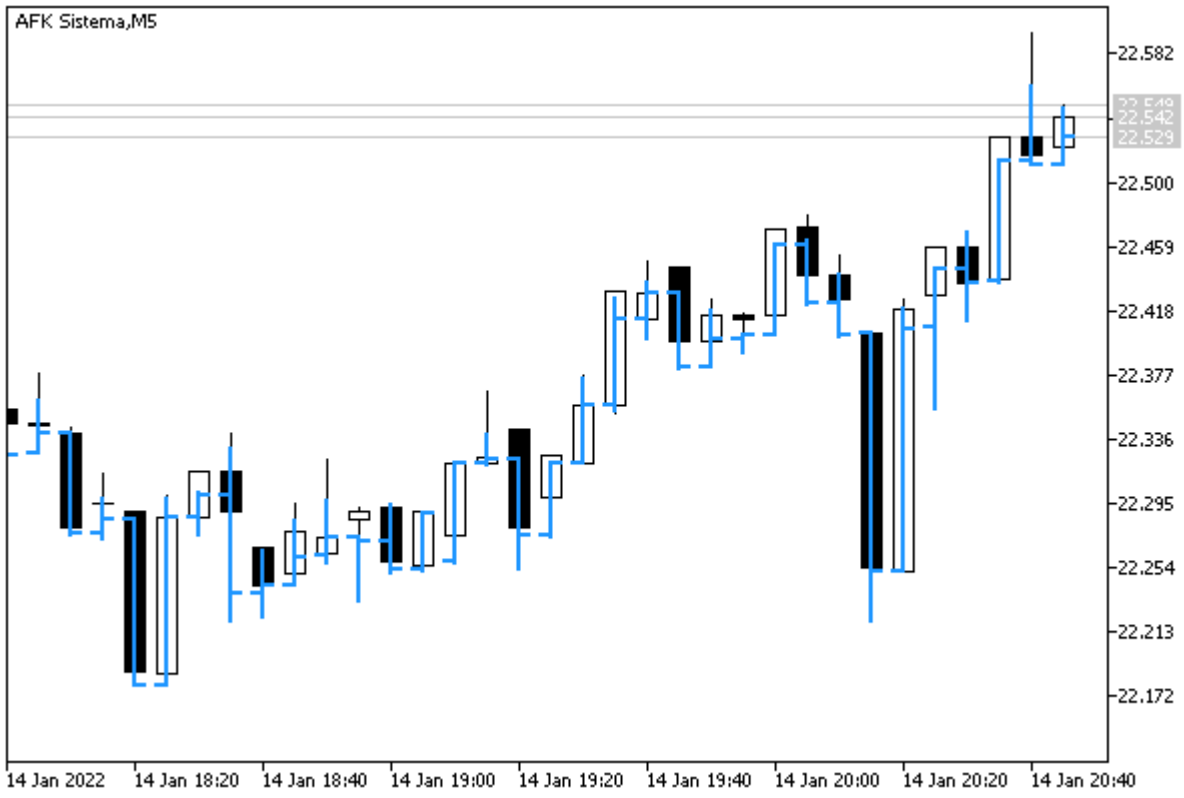
```
   void calc(const int i, const MqlTick &ticks[], const int skip = 0)
   {
      const int n = ArraySize(ticks);
      for(int j = skip; j < n; ++j)
      {
         const double p = price(ticks[j]);
         if(open[i] == EMPTY_VALUE)
         {
            open[i] = p;
         }

         if(p > high[i] || high[i] == EMPTY_VALUE)
         {
            high[i] = p;
         }

         if(p < low[i])
         {
            low[i] = p;
         }

         close[i] = p;
      }
   }
```

Los restantes fragmentos del código fuente y los principios de su trabajo corresponden a la descripción de *IndDeltaVolume.mq5*.

En el manejador *OnInit* mostramos de forma adicional el tipo de precio actual del gráfico y devolvemos una advertencia si el usuario decide construir un indicador basado en el tipo de precio *Last* para el instrumento donde *Last* está ausente.

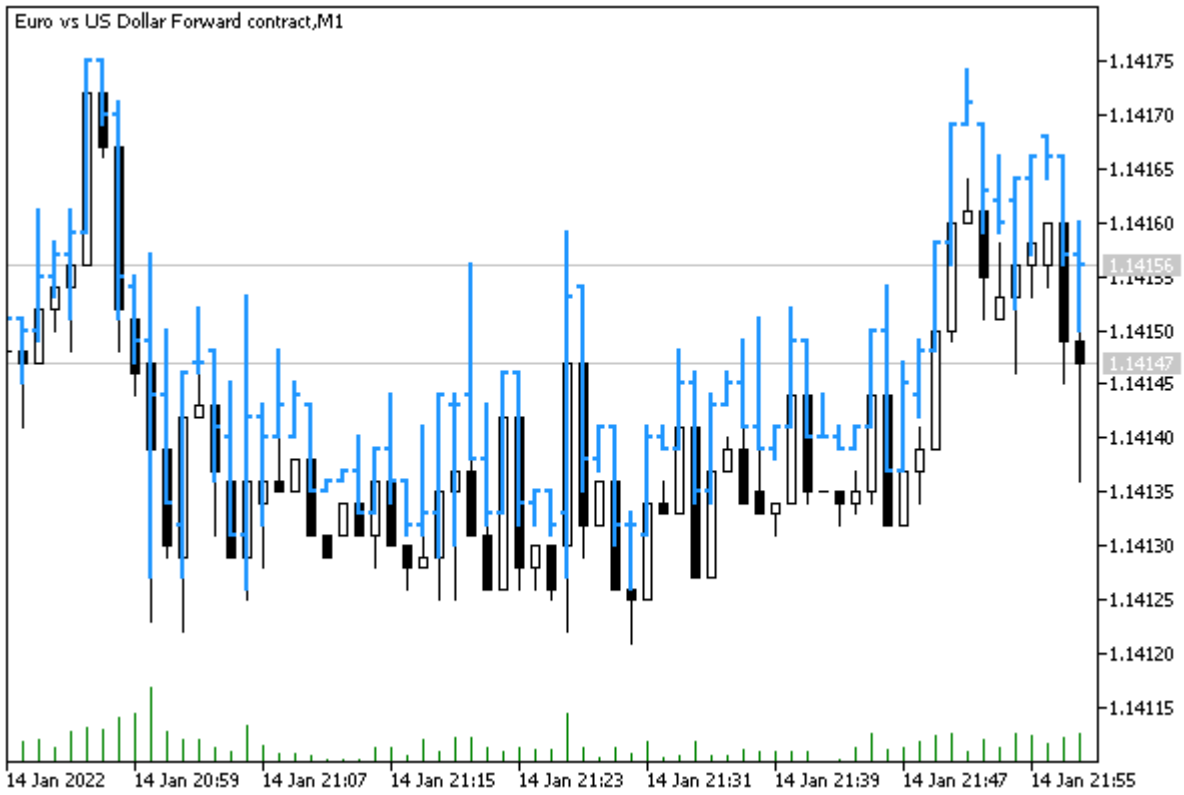
```
int OnInit()
{
   ...
   ENUM_SYMBOL_CHART_MODE mode =
      (ENUM_SYMBOL_CHART_MODE)SymbolInfoInteger(_Symbol, SYMBOL_CHART_MODE);
   Print("Chart mode: ", EnumToString(mode));

   if(mode == SYMBOL_CHART_MODE_BID
      && ChartMode == _SYMBOL_CHART_MODE_LAST)
   {
      Alert("Last price is not available for ", _Symbol);
   }

   return INIT_SUCCEEDED;
}
```

A continuación se muestra una captura de pantalla de un instrumento con el modo de trazado del gráfico basado en el precio *Last*; sobre el gráfico se coloca un indicador con el tipo de precio *Bid*.



Indicador con barras a precios Bid en el gráfico a precios Last

También es interesante observar las barras del precio *Ask* sobre un gráfico de precios normal *Bid*.



Indicador con barras a precios Ask en el gráfico a precios Bid

Durante las horas de baja liquidez, cuando el diferencial se amplía, se puede observar una diferencia significativa entre los gráficos de *Bid* y *Ask*.

6.1.11 Divisas base, de cotización y de margen del instrumento

Una de las propiedades más importantes de cada instrumento financiero son sus divisas de trabajo:

- La divisa base en la que está expresado el activo comprado o vendido (para instrumentos Forex).
- La divisa de cálculo de los beneficios (cotización)
- La divisa de cálculo del margen

Un programa MQL puede obtener los nombres de estas divisas utilizando la función *SymbolInfoString* y tres propiedades de la siguiente tabla:

Identificador	Descripción
SYMBOL_CURRENCY_BASE	Divisa base
SYMBOL_CURRENCY_PROFIT	Divisa de los beneficios
SYMBOL_CURRENCY_MARGIN	Divisa de margen

Estas propiedades ayudan a analizar los instrumentos de Forex, en cuyos nombres muchos brókers añaden diversos prefijos y sufijos, así como instrumentos bursátiles. En concreto, el algoritmo podrá encontrar un símbolo para obtener un tipo de cambio cruzado de dos divisas dadas o seleccionar una cartera de índices con una divisa de cotización común dada.

Dado que la búsqueda de herramientas de acuerdo con ciertos requisitos es una tarea muy común, vamos a crear una clase *SymbolFilter* (*SymbolFilter.mqh*) para construir una lista de símbolos adecuados y sus propiedades seleccionadas. En el futuro, utilizaremos esta clase para analizar no sólo divisas, sino también otras características.

En primer lugar, consideraremos una versión simplificada y, a continuación, la completaremos con una funcionalidad conveniente.

En el desarrollo, utilizaremos herramientas auxiliares ya preparadas: un array de mapas asociativo (*MapArray.mqh*) para almacenar pares clave-valor de los tipos seleccionados y un monitor de propiedades de símbolos (*SymbolMonitor.mqh*).

```
#include <MQL5Book/MapArray.mqh>
#include <MQL5Book/SymbolMonitor.mqh>
```

Para simplificar las sentencias de acumulación de los resultados del trabajo en arrays, utilizamos una versión mejorada de la macro PUSH, que ya hemos visto en ejemplos anteriores, así como su versión EXPAND para arrays multidimensionales (la asignación simple es imposible en este caso).

```
#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1, ArraySize(A) * 2) - 1] = V)
#define EXPAND(A) (ArrayResize(A, ArrayRange(A, 0) + 1, ArrayRange(A, 0) * 2) - 1)
```

Un objeto de la clase *SymbolFilter* debe tener un almacenamiento para los valores de las propiedades que se utilizarán para filtrar los símbolos. Por lo tanto, describiremos tres arrays *MapArray* en la clase para las propiedades de entero, real y cadena.

```
class SymbolFilter
{
   MapArray<ENUM_SYMBOL_INFO_INTEGER,long> longs;
   MapArray<ENUM_SYMBOL_INFO_DOUBLE,double> doubles;
   MapArray<ENUM_SYMBOL_INFO_STRING,string> strings;
   ...
```

La configuración de las propiedades de filtro necesarias se realiza mediante los métodos *let* sobrecargados.

```
public:
   SymbolFilter *let(const ENUM_SYMBOL_INFO_INTEGER property, const long value)
   {
      longs.put(property, value);
      return &this;
   }

   SymbolFilter *let(const ENUM_SYMBOL_INFO_DOUBLE property, const double value)
   {
      doubles.put(property, value);
      return &this;
   }

   SymbolFilter *let(const ENUM_SYMBOL_INFO_STRING property, const string value)
   {
      strings.put(property, value);
      return &this;
   }
   ...
```

Tenga en cuenta que los métodos devuelven un puntero al filtro, lo que le permite escribir condiciones en forma de cadena: por ejemplo, si anteriormente en el código se describió un objeto *f* de tipo *SymbolFilter*, entonces puede imponer dos condiciones en el tipo de precio y el nombre de la divisa de beneficio de la siguiente manera:

```
f.let(SYMBOL_CHART_MODE, SYMBOL_CHART_MODE_LAST).let(SYMBOL_CURRENCY_PROFIT, "USD");
```

La formación de un array de símbolos que satisfagan las condiciones la realiza el objeto filtro en diversas variantes del método *select*, la más sencilla de las cuales se presenta a continuación (más adelante se comentarán otras opciones).

El parámetro *watch* define el contexto de búsqueda de símbolos: entre los seleccionados en *Observación de Mercado* (*true*) o todos los disponibles (*false*). El array de salida *symbols* se rellenará con los nombres de los símbolos coincidentes. Ya conocemos la estructura del código dentro del método: tiene un bucle a través de los símbolos para cada uno de los cuales se crea un objeto monitor *m*.

```
void select(const bool watch, string &symbols[]) const
{
   const int n = SymbolsTotal(watch);
   for(int i = 0; i < n; ++i)
   {
      const string s = SymbolName(i, watch);
      SymbolMonitor m(s);
      if(match<ENUM_SYMBOL_INFO_INTEGER,long>(m, longs)
      && match<ENUM_SYMBOL_INFO_DOUBLE,double>(m, doubles)
      && match<ENUM_SYMBOL_INFO_STRING,string>(m, strings))
      {
         PUSH(symbols, s);
      }
   }
}
```

Es con la ayuda del monitor que podemos obtener el valor de cualquier propiedad de forma unificada. La comprobación de si las propiedades del símbolo actual coinciden con el conjunto de condiciones almacenadas en los arrays *longs*, *doubles* y *strings* se implementa mediante un método auxiliar *match*. Sólo si todas las propiedades solicitadas coinciden, el nombre del símbolo se guardará en el array de salida *symbols*.

En el caso más sencillo, la implementación del método *match* es la siguiente (posteriormente se modificará):

```
protected:
   template<typename K,typename V>
   bool match(const SymbolMonitor &m, const MapArray<K,V> &data) const
   {
      for(int i = 0; i < data.getSize(); ++i)
      {
         const K key = data.getKey(i);
         if(!equal(m.get(key), data.getValue(i)))
         {
            return false;
         }
      }
      return true;
   }
```

Si al menos uno de los valores del array *data* no coincide con la propiedad de caracteres correspondiente, el método devuelve *false*. Si todas las propiedades coinciden (o no hay condiciones para las propiedades de este tipo), el método devuelve *true*.

La comparación de dos valores se realiza mediante *equal*. Dado que entre las propiedades puede haber propiedades de tipo *double*, la implementación no es tan sencilla como cabría pensar.

```
template<typename V>
static bool equal(const V v1, const V v2)
{
   return v1 == v2 || eps(v1, v2);
}
```

Para el tipo *double*, la expresión *v1 == v2* puede no funcionar para números cercanos, por lo que debe tenerse en cuenta la precisión del tipo real DBL_EPSILON. Esto se hace en un método separado *eps*, sobrecargado por separado para el tipo *double* y todos los demás tipos debido a la plantilla.

```
static bool eps(const double v1, const double v2)
{
   return fabs(v1 - v2) < DBL_EPSILON * fmax(v1, v2);
}

template<typename V>
static bool eps(const V v1, const V v2)
{
   return false;
}
```

Cuando los valores de cualquier tipo excepto *double* son iguales, el método de plantilla *eps* simplemente no será llamado, y en todos los demás casos (incluyendo cuando los valores difieren), devuelve *false* según lo requerido (por lo tanto, sólo la condición *v1 == v2*).

La opción de filtro descrita anteriormente sólo permite comprobar la igualdad de las propiedades. Sin embargo, en la práctica a menudo es necesario analizar las condiciones para la desigualdad, así como para mayor/menor. Por este motivo, la clase *SymbolFilter* dispone de la enumeración *IS* con operaciones básicas de comparación (si se desea, puede completarse).

```
class SymbolFilter
{
   ...
   enum IS
   {
      EQUAL,
      GREATER,
      NOT_EQUAL,
      LESS
   };
   ...
```

Para cada propiedad de las enumeraciones ENUM_SYMBOL_INFO_INTEGER, ENUM_SYMBOL_INFO_DOUBLE y ENUM_SYMBOL_INFO_STRING se requiere guardar no sólo el valor deseado de la propiedad (recuerde los arrays asociativos *longs*, *doubles*, *strings*), sino también el método de comparación de la nueva enumeración *IS*.

Dado que los elementos de las enumeraciones estándar tienen valores que no se solapan (hay una excepción relacionada con los volúmenes, pero no es crítica), tiene sentido reservar un array de mapa común *conditions* para el método de comparación. Esto plantea la cuestión de qué tipo elegir para la clave del mapa con el fin de «combinar» técnicamente diferentes enumeraciones. Para ello, hemos tenido que describir la enumeración ficticia ENUM_ANY, que sólo denota un determinado tipo de enumeración genérica. Recordemos que todas las enumeraciones tienen una representación interna equivalente a un entero *int*, y por tanto pueden reducirse entre sí.

```
enum ENUM_ANY
{
};

MapArray<ENUM_ANY,IS> conditions;
MapArray<ENUM_ANY,long> longs;
MapArray<ENUM_ANY,double> doubles;
MapArray<ENUM_ANY,string> strings;
...
```

Ahora podemos completar todos los métodos *let* que establecen el valor deseado de la propiedad añadiendo el parámetro de entrada *cmp* que especifica el método de comparación. Por defecto, establece la comprobación de igualdad (EQUAL).

```
SymbolFilter *let(const ENUM_SYMBOL_INFO_INTEGER property, const long value,
   const IS cmp = EQUAL)
{
   longs.put((ENUM_ANY)property, value);
   conditions.put((ENUM_ANY)property, cmp);
   return &this;
}
```

He aquí una variante para propiedades de enteros. Las otras dos sobrecargas cambian de la misma manera.

Teniendo en cuenta la nueva información sobre distintas formas de comparar y eliminar simultáneamente distintos tipos de claves en arrays de mapas, modificamos el método *match*. En él, para cada propiedad especificada, recuperamos una condición del array *conditions* basándonos en la clave del array del mapa *data*, y se realizan las comprobaciones apropiadas utilizando el operador *switch*.

```
template<typename V>
bool match(const SymbolMonitor &m, const MapArray<ENUM_ANY,V> &data) const
{
   // dummy variable to select m.get method overload below
   static const V type = (V)NULL;
   // cycle by conditions imposed on the properties of the symbol
   for(int i = 0; i < data.getSize(); ++i)
   {
      const ENUM_ANY key = data.getKey(i);
      // choice of comparison method in the condition
      switch(conditions[key])
      {
      case EQUAL:
         if(!equal(m.get(key, type), data.getValue(i))) return false;
         break;
      case NOT_EQUAL:
         if(equal(m.get(key, type), data.getValue(i))) return false;
         break;
      case GREATER:
         if(!greater(m.get(key, type), data.getValue(i))) return false;
         break;
      case LESS:
         if(greater(m.get(key, type), data.getValue(i))) return false;
         break;
      }
   }
   return true;
}
```

El nuevo método de plantilla *greater* se aplica de forma simplista.

```
template<typename V>
static bool greater(const V v1, const V v2)
{
   return v1 > v2;
}
```

Ahora la llamada al método *match* puede escribirse de forma más corta, ya que el único tipo que queda de la plantilla *V* se determina automáticamente mediante el argumento *data* pasado (y éste es uno de los arrays *longs*, *doubles* o *strings*).

```
void select(const bool watch, string &symbols[]) const
{
   const int n = SymbolsTotal(watch);
   for(int i = 0; i < n; ++i)
   {
      const string s = SymbolName(i, watch);
      SymbolMonitor m(s);
      if(match(m, longs)
         && match(m, doubles)
         && match(m, strings))
      {
         PUSH(symbols, s);
      }
   }
}
```

Aún no es la versión final de la clase *SymbolFilter*, pero ya podemos probarla en acción.

Vamos a crear un script *SymbolFilterCurrency.mq5* que pueda filtrar símbolos basándose en las propiedades de la divisa base y la divisa de beneficio; en este caso, es USD. Por defecto, el parámetro *MarketWatchOnly* sólo busca en *Observación de Mercado*.

```
#include <MQL5Book/SymbolFilter.mqh>

input bool MarketWatchOnly = true;

void OnStart()
{
   SymbolFilter f;   // filter object
   string symbols[]; // array for results
   ...
```

Digamos que queremos encontrar instrumentos de Forex que tengan cotizaciones directas, es decir, que «USD» aparezca en sus nombres al principio. Para no depender de las particularidades de la formación de nombres para un bróker en particular utilizaremos la propiedad SYMBOL_CURRENCY_BASE, que contiene la primera divisa.

Vamos a escribir la condición de que la divisa base del símbolo sea igual a USD y apliquemos el filtro.

```
f.let(SYMBOL_CURRENCY_BASE, "USD")
.select(MarketWatchOnly, symbols);
Print("===== Base is USD =====");
ArrayPrint(symbols);
...
```

El array resultante se envía al registro.

```
===== Base is USD =====
"USDCHF" "USDJPY" "USDCNH" "USDRUB" "USDCAD" "USDSEK" "SP500m" "Brent"
```

Como puede ver, el array incluye no sólo símbolos Forex con USD al principio del ticker, sino también el índice S&P500 y la materia prima (petróleo). Los dos últimos símbolos se cotizan en dólares, pero tienen también la misma divisa base. Al mismo tiempo, la divisa de cotización de los símbolos Forex (es

también la divisa de beneficio) es la segunda y difiere del USD. Esto le permite complementar el filtro de manera que los símbolos que no sean de Forex ya no coincidan con él.

Vaciemos el array, añadamos una condición de que la divisa de beneficio no sea igual a «USD», y volvamos a solicitar símbolos adecuados (la condición anterior se guardó en el objeto *f*).

```
    ...
    ArrayResize(symbols, 0);

    f.let(SYMBOL_CURRENCY_PROFIT, "USD", SymbolFilter::IS::NOT_EQUAL)
    .select(MarketWatchOnly, symbols);
    Print("===== Base is USD and Profit is not USD =====");
    ArrayPrint(symbols);
}
```

Esta vez, en el registro se muestran únicamente los símbolos que busca.

```
===== Base is USD and Profit is not USD =====
"USDCHF" "USDJPY" "USDCNH" "USDRUB" "USDCAD" "USDSEK"
```

6.1.12 Precisión de la representación de precios y pasos de cambio

Hemos visto ya dos propiedades interrelacionadas del símbolo de trabajo del gráfico: el paso mínimo de cambio de precio (*Point*) y la precisión de la presentación del precio, que se expresa en el número de decimales (*Digits*). También están disponibles en las Variables predefinidas. Para obtener propiedades similares de un símbolo arbitrario, debe consultar las propiedades SYMBOL_POINT y SYMBOL_DIGITS, respectivamente. La propiedad SYMBOL_POINT está estrechamente relacionada con el cambio mínimo de precio (conocido en un programa MQL como la propiedad SYMBOL_TRADE_TICK_SIZE) y su valor (SYMBOL_TRADE_TICK_VALUE), normalmente en la divisa de la cuenta de trading (pero algunos símbolos pueden ser configurados para usar la divisa base; puede contactar con su bróker para obtener más detalles si es necesario). En la tabla siguiente se muestra el grupo completo de estas propiedades:

Identificador	Descripción
SYMBOL_DIGITS	El número de decimales
SYMBOL_POINT	El valor de un punto en la divisa de cotización
SYMBOL_TRADE_TICK_VALUE	SYMBOL_TRADE_TICK_VALUE_PROFIT value
SYMBOL_TRADE_TICK_VALUE_PROFIT	Valor actual del tick para una posición rentable
SYMBOL_TRADE_TICK_VALUE_LOSS	Valor actual del tick para una posición perdedora
SYMBOL_TRADE_TICK_SIZE	Variación mínima del precio en la divisa de cotización

Todas las propiedades excepto SYMBOL_DIGITS son números reales y se solicitan utilizando la función *SymbolInfoDouble*. La propiedad SYMBOL_DIGITS está disponible a través de *SymbolInfoInteger*. Para probar el trabajo con estas propiedades, utilizaremos las clases ya creadas *SymbolFilter* y *SymbolMonitor*, que llamarán automáticamente a la función deseada para cualquier propiedad.

También mejoraremos la clase *SymbolFilter* añadiendo una nueva sobrecarga del método *select*, que podrá rellenar no sólo un array con los nombres de los símbolos adecuados, sino también otro array con los valores de su propiedad específica.

En un caso más general, puede que nos interesen varias propiedades para cada símbolo al mismo tiempo, por lo que es aconsejable utilizar no uno de los tipos de datos integrados para el array de salida, sino un tipo compuesto especial con diferentes campos.

En programación, estos tipos se denominan tuplas y son en cierto modo equivalentes a las estructuras MQL5.

```
template<typename T1,typename T2,typename T3> // we can describe up to 64 fields
struct Tuple3                                 // MQL5 allows 64 template parameters
{
   T1 _1;
   T2 _2;
   T3 _3;
};
```

No obstante, las estructuras requieren una descripción preliminar con todos los campos, mientras que no conocemos de antemano el número y la lista de propiedades de los símbolos solicitados. Por lo tanto, para simplificar el código, representaremos nuestra tupla como un vector en la segunda dimensión de un array dinámico que recibe los resultados de la consulta.

```
T array[][S];
```

Como tipo de datos *T* podemos utilizar cualquiera de las enumeraciones y tipos integrados utilizados para las propiedades. El tamaño *S* debe coincidir con el número de propiedades solicitadas.

A decir verdad, tal simplificación nos limita en una consulta a valores de los mismos tipos, es decir, sólo enteros, sólo reales o sólo cadenas. Sin embargo, las condiciones de filtrado pueden incluir cualquier propiedad. Implementaremos el enfoque con tuplas un poco más adelante, utilizando el ejemplo de filtros de otras entidades de trading: órdenes, transacciones y posiciones.

Así, la nueva versión del método *SymbolFilter::select* toma como entrada una referencia al array *property* con identificadores de propiedades para leer desde los símbolos filtrados. Los nombres de los símbolos en sí y los valores de estas propiedades se escribirán en los arrays de salida *symbols* y *data*.

```
template<typename E,typename V>
bool select(const bool watch, const E &property[], string &symbols[],
   V &data[][], const bool sort = false) const
{
   // the size of the array of requested properties must match the output tuple
   const int q = ArrayRange(data, 1);
   if(ArraySize(property) != q) return false;

   const int n = SymbolsTotal(watch);
   // iterate over characters
   for(int i = 0; i < n; ++i)
   {
      const string s = SymbolName(i, watch);
      // access to the symbol properties is provided by the monitor
      SymbolMonitor m(s);
      // check all filter conditions
      if(match(m, longs)
      && match(m, doubles)
      && match(m, strings))
      {
         // properties of a suitable symbol are written to arrays
         const int k = EXPAND(data);
         for(int j = 0; j < q; ++j)
         {
            data[k][j] = m.get(property[j]);
         }
         PUSH(symbols, s);
      }
   }

   if(sort)
   {
      ...
   }
   return true;
}
```

Además, el nuevo método puede ordenar el array de salida por la primera dimensión (la primera propiedad solicitada): esta funcionalidad se deja para su estudio por separado mediante códigos fuente. Para activar la clasificación, ajuste el parámetro *sort* en *true*. Los arrays con nombres de símbolos y datos se ordenan de forma coherente.

Para evitar tuplas en el código de llamada cuando sólo se necesita solicitar una propiedad de los caracteres filtrados, se implementa la opción *select* siguiente en *SymbolFilter*: en su interior se definen arrays intermedios de propiedades (*properties*) y valores (*tuples*) con tamaño 1 en la segunda dimensión, que se utilizan para llamar a la versión completa anterior de *select*.

```
   template<typename E,typename V>
   bool select(const bool watch, const E property, string &symbols[], V &data[],
      const bool sort = false) const
   {
      E properties[1] = {property};
      V tuples[][1];

      const bool result = select(watch, properties, symbols, tuples, sort);
      ArrayCopy(data, tuples);
      return result;
   }
```

Utilizando el filtro avanzado, vamos a intentar construir una lista de símbolos ordenados por el valor de tick SYMBOL_TRADE_TICK_VALUE (véase el archivo *SymbolFilterTickValue.mq5*). Suponiendo que la divisa de depósito sea USD, deberíamos obtener un valor igual a 1.0 para los instrumentos de Forex cotizados en USD (del tipo XXXUSD). Para otros activos, veremos valores no triviales.

```
#include <MQL5Book/SymbolFilter.mqh>

input bool MarketWatchOnly = true;

void OnStart()
{
   SymbolFilter f;      // filter object
   string symbols[];    // array with symbol names
   double tickValues[]; // array for results

   // apply the filter without conditions, fill and sort the array
   f.select(MarketWatchOnly, SYMBOL_TRADE_TICK_VALUE, symbols, tickValues, true);

   PrintFormat("===== Tick values of the symbols (%d) =====",
      ArraySize(tickValues));
   ArrayPrint(symbols);
   ArrayPrint(tickValues, 5);
}
```

He aquí el resultado de ejecutar el script:

```
===== Tick values of the symbols (13) =====
"BTCUSD" "USDRUB" "XAUUSD" "USDSEK" "USDCNH" "USDCAD" "USDJPY" "NZDUSD" "AUDUSD" "EUR
 0.00100  0.01309  0.10000  0.10955  0.15744  0.80163  0.87319  1.00000  1.00000  1.0
```

6.1.13 Volúmenes permitidos de operaciones de trading

En capítulos posteriores, donde aprenderemos a programar Asesores Expertos, necesitaremos controlar muchas de las características de los símbolos que determinan el éxito del envío de órdenes de trading. En particular, esto se aplica a la parte de la especificación de símbolos en la que se indica el ámbito permitido de las operaciones. Las propiedades correspondientes también están disponibles en MQL5. Toda ellas son del tipo *double* y son solicitadas por la función *SymbolInfoDouble*.

Identificador	Descripción
SYMBOL_VOLUME_MIN	Volumen mínimo de la transacción en lotes
SYMBOL_VOLUME_MAX	Volumen máximo de la transacción en lotes
SYMBOL_VOLUME_STEP	Paso mínimo para modificar el volumen de la transacción en lotes
SYMBOL_VOLUME_LIMIT	Volumen total máximo permitido de una posición abierta y órdenes pendientes en una dirección (compra o venta).
SYMBOL_TRADE_CONTRACT_SIZE	Tamaño del contrato de trading = 1 lote

Los intentos de comprar o vender un instrumento financiero con un volumen inferior al mínimo, superior al máximo o que no sea múltiplo de un paso darán lugar a un error. En el capítulo relativo a API de trading implementaremos un código para unificar las comprobaciones necesarias y normalizar los volúmenes antes de llamar a las funciones de trading de la API de MQL5.

Entre otras cosas, el programa MQL también debe comprobar SYMBOL_VOLUME_LIMIT. Por ejemplo, con un límite de 5 lotes, puede tener una posición de compra abierta con un volumen de 5 lotes y colocar una orden pendiente *Sell Limit* con un volumen de 5 lotes. No obstante, no puede colocar una orden pendiente *Buy Limit* (porque el volumen acumulado en una dirección superará el límite) ni establecer un *Sell Limit* de más de 5 lotes.

Como ejemplo introductorio, considere el script *SymbolFilterVolumes.mq5*, que registra los valores de las propiedades anteriores para los símbolos seleccionados. Añadamos la variable *MinimalContractSize* a los parámetros de entrada para poder filtrar los símbolos por la propiedad SYMBOL_TRADE_CONTRACT_SIZE: mostramos sólo aquellos cuyo tamaño de contrato sea mayor que el especificado (por defecto, 0, es decir, todos los símbolos cumplen la condición).

```
#include <MQL5Book/SymbolFilter.mqh>

input bool MarketWatchOnly = true;
input double MinimalContractSize = 0;
```

Al principio de *OnStart* vamos a definir un objeto filtro y arrays de salida para obtener listas de nombres de propiedades y valores como vectores *double* para cuatro campos. La lista de las cuatro propiedades requeridas se indica en el array *volumeIds*.

```
void OnStart()
{
   SymbolFilter f;                        // filter object
   string symbols[];                      // receiving array with names
   double volumeLimits[][4];              // receiving array with data vectors

   // requested symbol properties
   ENUM_SYMBOL_INFO_DOUBLE volumeIds[] =
   {
      SYMBOL_VOLUME_MIN,
      SYMBOL_VOLUME_STEP,
      SYMBOL_VOLUME_MAX,
      SYMBOL_VOLUME_LIMIT
   };
   ...
```

A continuación, aplicamos un filtro por tamaño de contrato (debe ser mayor que el especificado) y obtenemos los campos de especificación asociados a volúmenes para los símbolos coincidentes.

```
   f.let(SYMBOL_TRADE_CONTRACT_SIZE, MinimalContractSize, SymbolFilter::IS::GREATER)
   .select(MarketWatchOnly, volumeIds, symbols, volumeLimits);

   const int n = ArraySize(volumeLimits);
   PrintFormat("===== Volume limits of the symbols (%d) =====", n);
   string title = "";
   for(int i = 0; i < ArraySize(volumeIds); ++i)
   {
      title += "\t" + EnumToString(volumeIds[i]);
   }
   Print(title);
   for(int i = 0; i < n; ++i)
   {
      Print(symbols[i]);
      ArrayPrint(volumeLimits, 3, NULL, i, 1, 0);
   }
}
```

Para la configuración por defecto, el script podría mostrar resultados como los siguientes (con abreviaturas):

```
===== Volume limits of the symbols (13) =====
SYMBOL_VOLUME_MIN SYMBOL_VOLUME_STEP SYMBOL_VOLUME_MAX SYMBOL_VOLUME_LIMIT
EURUSD
  0.010   0.010 500.000   0.000
GBPUSD
  0.010   0.010 500.000   0.000
USDCHF
  0.010   0.010 500.000   0.000
USDJPY
  0.010   0.010 500.000   0.000
USDCNH
   0.010    0.010 1000.000    0.000
USDRUB
   0.010    0.010 1000.000    0.000
...
XAUUSD
  0.010   0.010 100.000   0.000
BTCUSD
   0.010    0.010 1000.000    0.000
SP500m
 0.100  0.100  5.000 15.000
```

Algunos símbolos pueden no estar limitados por SYMBOL_VOLUME_LIMIT (el valor es 0). Puede comparar los resultados con las especificaciones del símbolo: deben coincidir.

6.1.14 Permiso de trading

Como continuación del tema relacionado con la correcta preparación de las órdenes de trading que iniciamos en la sección anterior, vamos a pasar al siguiente par de propiedades que desempeñan un papel muy importante en el desarrollo de Asesores Expertos.

Identificador	Descripción
SYMBOL_TRADE_MODE	Permisos para los distintos modos de trading del símbolo (véase ENUM_SYMBOL_TRADE_MODE)
SYMBOL_ORDER_MODE	Banderas de los tipos de orden permitidos, máscara de bits (véase más adelante)

Ambas propiedades son de tipo entero y están disponibles a través de la función *SymbolInfoInteger*.

Ya hemos utilizado la propiedad SYMBOL_TRADE_MODE en el script *SymbolPermissions.mq5*. Su valor es uno de los elementos de la enumeración ENUM_SYMBOL_TRADE_MODE.

Identificador	Valor	Descripción
SYMBOL_TRADE_MODE_DISABLED	0	El trading está desactivado para el símbolo
SYMBOL_TRADE_MODE_LONGONLY	1	Sólo se permiten operaciones de compra
SYMBOL_TRADE_MODE_SHORTONLY	2	Sólo se permiten operaciones de venta
SYMBOL_TRADE_MODE_CLOSEONLY	3	Sólo se permiten operaciones de cierre
SYMBOL_TRADE_MODE_FULL	4	Sin restricciones en las operaciones comerciales

Recordemos que la clase *Permissions* contiene el método *isTradeOnSymbolEnabled*, que comprueba varios aspectos que afectan a la disponibilidad de trading de símbolos, y uno de ellos es la propiedad SYMBOL_TRADE_MODE. Por defecto, consideramos que nos interesa el pleno acceso al trading, es decir, vender y comprar: SYMBOL_TRADE_MODE_FULL. Dependiendo de la estrategia de trading, el programa MQL puede considerar suficientes, por ejemplo, permisos sólo para comprar, sólo para vender o sólo para cerrar operaciones.

```
static bool isTradeOnSymbolEnabled(string symbol, const datetime now = 0,
   const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
   // checking sessions
   bool found = now == 0;
   ...
   // checking the trading mode for the symbol
   return found && (SymbolInfoInteger(symbol, SYMBOL_TRADE_MODE) == mode);
}
```

Además del modo de trading, en el futuro tendremos que analizar los permisos para órdenes de distintos tipos: se indican mediante bits separados en la propiedad SYMBOL_ORDER_MODE y pueden combinarse arbitrariamente con un OR lógico ('|'). Por ejemplo, el valor 127 (0x7F) corresponde a todos los bits activados, es decir, la disponibilidad de todos los tipos de órdenes.

Identificador	Valor	Descripción
SYMBOL_ORDER_MARKET	1	Se permiten órdenes de mercado (compra y venta)
LIMITE_ORDEN_SIMBOLO	2	Se permiten órdenes Limit (Buy Limit y Sell Limit)
SYMBOL_ORDER_STOP	4	Se permiten órdenes Stop (Buy Stop y Sell Stop)
SYMBOL_ORDER_STOP_LIMIT	8	Se permiten las órdenes Stop Limit (Buy Stop Limit y Sell Stop Limit)
SYMBOL_ORDER_SL	16	Se permite fijar niveles de Stop Loss
SYMBOL_ORDER_TP	32	Se permite establecer niveles de Take Profit
SYMBOL_ORDER_CLOSEBY	64	Permiso para cerrar una posición por otra opuesta para el mismo símbolo, operación Close By.

La propiedad SYMBOL_ORDER_CLOSEBY sólo se establece para las cuentas con contabilidad de cobertura (ACCOUNT_MARGIN_MODE_RETAIL_HEDGING, véase Tipo de cuenta).

En el script de prueba *SymbolFilterTradeMode.mq5*, solicitaremos un par de propiedades descritas para símbolos visibles en *Market Watch*. La salida de bits y sus combinaciones como números no es muy informativa, así que utilizaremos el hecho de que en la clase *SymbolMonitor* tenemos un método *stringify* que resulta práctico para imprimir los miembros de la enumeración y las máscaras de bits de todas las propiedades.

```
void OnStart()
{
   SymbolFilter f;                        // filter object
   string symbols[];                      // array for names
   long permissions[][2];                 // array for data (property values)

   // list of requested symbol properties
   ENUM_SYMBOL_INFO_INTEGER modes[] =
   {
      SYMBOL_TRADE_MODE,
      SYMBOL_ORDER_MODE
   };

   // apply the filter, get arrays with results
   f.let(SYMBOL_VISIBLE, true).select(true, modes, symbols, permissions);

   const int n = ArraySize(symbols);
   PrintFormat("===== Trade permissions for the symbols (%d) =====", n);
   for(int i = 0; i < n; ++i)
   {
      Print(symbols[i] + ":");
      for(int j = 0; j < ArraySize(modes); ++j)
      {
         // display bit and number descriptions "as is"
         PrintFormat("  %s (%d)",
            SymbolMonitor::stringify(permissions[i][j], modes[j]),
            permissions[i][j]);
      }
   }
}
```

A continuación se muestra parte del registro resultante de ejecutar el script.

```
===== Trade permissions for the symbols (13) =====
EURUSD:
  SYMBOL_TRADE_MODE_FULL (4)
  [ _SYMBOL_ORDER_MARKET _SYMBOL_ORDER_LIMIT _SYMBOL_ORDER_STOP
  _SYMBOL_ORDER_STOP_LIMIT _SYMBOL_ORDER_SL _SYMBOL_ORDER_TP
  _SYMBOL_ORDER_CLOSEBY ] (127)
GBPUSD:
  SYMBOL_TRADE_MODE_FULL (4)
  [ _SYMBOL_ORDER_MARKET _SYMBOL_ORDER_LIMIT _SYMBOL_ORDER_STOP
  _SYMBOL_ORDER_STOP_LIMIT _SYMBOL_ORDER_SL _SYMBOL_ORDER_TP
  _SYMBOL_ORDER_CLOSEBY ] (127)
...
SP500m:
  SYMBOL_TRADE_MODE_DISABLED (0)
  [ _SYMBOL_ORDER_MARKET _SYMBOL_ORDER_LIMIT _SYMBOL_ORDER_STOP
  _SYMBOL_ORDER_STOP_LIMIT _SYMBOL_ORDER_SL _SYMBOL_ORDER_TP ] (63)
```

Tenga en cuenta que la negociación para el último símbolo SP500m está completamente desactivada (sus cotizaciones se proporcionan únicamente a título «indicativo»). Al mismo tiempo, su conjunto de banderas por tipos de orden no es 0, pero no hace ninguna diferencia.

Dependiendo de los acontecimientos en el mercado, el bróker puede cambiar las propiedades del símbolo a su discreción; por ejemplo, dejando sólo la oportunidad de cerrar posiciones durante algún tiempo, por lo que un robot de trading correcto debe controlar estas propiedades antes de cada operación.

6.1.15 Condiciones de trading de símbolos y modos de ejecución de órdenes

En esta sección profundizaremos en los aspectos de la automatización del trading que dependen de la configuración de los instrumentos financieros. Por ahora estudiaremos únicamente las propiedades, mientras que su aplicación práctica se presentará en capítulos posteriores. Se da por sentado que el lector ya está familiarizado con la terminología básica, como orden de mercado y pendiente, operación y posición.

Al enviar una solicitud de ejecución de una operación, debe tenerse en cuenta que en los mercados financieros no hay garantía de que, en un momento determinado, todo el volumen solicitado esté disponible para este instrumento financiero al precio deseado. Por lo tanto, el trading en tiempo real está regulada por los modos de ejecución de precio y volumen. Los modos o, en otras palabras, las políticas de ejecución, definen las reglas para los casos en que el precio ha cambiado o el volumen solicitado no puede ejecutarse completamente en el momento actual.

En la API de MQL5, estos modos están disponibles para cada símbolo como las siguientes propiedades que se pueden obtener a través de la función *SymbolInfoInteger*.

Identificador	Descripción
SYMBOL_TRADE_EXEMODE	Modalidades de ejecución de las operaciones relacionadas con el precio
SYMBOL_FILLING_MODE	Banderas de los modos de rellenado de órdenes permitidos en relación con el volumen (máscara de bits, véase más adelante)

El valor de la propiedad SYMBOL_TRADE_EXEMODE es un miembro de la enumeración ENUM_SYMBOL_TRADE_EXECUTION.

Identificador	Descripción
SYMBOL_TRADE_EXECUTION_REQUEST	Operación al precio solicitado
SYMBOL_TRADE_EXECUTION_INSTANT	Ejecución instantánea (trading a precios recibidos de forma continua)
SYMBOL_TRADE_EXECUTION_MARKET	Ejecución de mercado
SYMBOL_TRADE_EXECUTION_EXCHANGE	Ejecución bursátil

Los usuarios de terminales deberían conocer todos o la mayoría de estos modos de la lista desplegable *Type* del cuadro de diálogo *New order* (F9). Recordemos brevemente lo que significan. Para obtener más detalles, consulte la documentación del terminal.

- Ejecución a petición (SYMBOL_TRADE_EXECUTION_REQUEST): ejecución de una orden de mercado a un precio previamente recibido del bróker. Antes de enviar una orden de mercado, el operador solicita el precio actual al bróker. La ejecución posterior de la orden a este precio puede confirmarse o rechazarse.
- Ejecución instantánea (SYMBOL_TRADE_EXECUTION_INSTANT):- ejecución de una orden de mercado al precio actual. Al enviar una solicitud de operación para su ejecución, el terminal inserta automáticamente los precios actuales en la orden. Si el bróker acepta el precio, la orden se ejecuta. Si el bróker no acepta el precio solicitado, devuelve los precios a los que se puede ejecutar esta orden, lo que se denomina recotización.
- Ejecución de mercado (SYMBOL_TRADE_EXECUTION_MARKET): el bróker inserta el precio de ejecución en la orden sin confirmación adicional del operador. El envío de una orden de mercado en este modo implica un acuerdo anticipado con el precio al que se ejecutará.
- Ejecución bursátil (SYMBOL_TRADE_EXECUTION_EXCHANGE): las operaciones de trading se realizan a los precios de las ofertas actuales del mercado.

En cuanto a los bits de SYMBOL_FILLING_MODE que pueden combinarse con el operador lógico OR ('|'), su presencia o ausencia indica las siguientes acciones:

Identificador	Valor	Política de relleno
SYMBOL_FILLING_FOK	1	Todo o Nada (Fill Or Kill, FOK): la orden debe ejecutarse exclusivamente en el volumen especificado o cancelarse.
SYMBOL_FILLING_IOC	2	Inmediata o Cancelar (IOC): negociar el volumen máximo disponible en el mercado dentro de los límites especificados en la orden o cancelar.
(Falta el identificador)	(cualquiera, incluido 0)	Devolución: en caso de ejecución parcial, la orden Limit o de mercado con el volumen restante no se cancela, sino que sigue siendo válida.

La posibilidad de utilizar los modos FOK e IOC viene determinada por el servidor de trading.

Si el modo SYMBOL_FILLING_FOK está activado, mientras se envía una orden con la función *OrderSend*, el programa MQL podrá utilizar el tipo de relleno de orden correspondiente en la estructura *MqlTradeRequest*: ORDER_FILLING_FOK. Si al mismo tiempo no hay un volumen suficiente del instrumento financiero en el mercado, la orden no se ejecutará. Hay que tener en cuenta que el volumen requerido puede estar formado por varias ofertas disponibles actualmente en el mercado, lo que da lugar a varias transacciones.

Si el modo SYMBOL_FILLING_IOC está activado, el programa MQL tendrá acceso al método de llenado de órdenes ORDER_FILLING_IOC del mismo nombre (también se especifica en el campo especial de «llenado» (*type_filling*) de la estructura *MqlTradeRequest* antes de enviar la orden a la función *OrderSend*). Al utilizar este modo, en caso de imposibilidad de ejecución completa, la orden se ejecutará en el volumen disponible, y el volumen restante de la orden se cancelará.

La última política sin identificador es el modo por defecto y está disponible independientemente de los demás modos (por eso coincide con cero o cualquier otro valor). En otras palabras: aunque obtengamos el valor 1 (SYMBOL_FILLING_FOK), 2 (SYMBOL_FILLING_IOC) o 3 (SYMBOL_FILLING_FOK | SYMBOL_FILLING_IOC) para la propiedad SYMBOL_FILLING_MODE, el modo de retorno estará implícito. Para utilizar esta política, al formar una orden (rellenando la estructura *MqlTradeRequest*) debemos especificar el tipo de relleno ORDER_FILLING_RETURN.

> Entre todos los modos SYMBOL_TRADE_EXEMODE existe una especificidad relativa a la ejecución de mercado (SYMBOL_TRADE_EXECUTION_MARKET): las órdenes de devolución siempre están prohibidas en el modo de ejecución de mercado.

Dado que ORDER_FILLING_FOK corresponde a la constante 0, la ausencia de una indicación explícita del tipo de llenado en una solicitud de operación implicará este modo particular.

Analizaremos todos estos matices en la práctica cuando desarrollemos Asesores Expertos, pero por ahora vamos a comprobar la lectura de propiedades en un sencillo script *SymbolFilterExecMode.mq5*.

```
#include <MQL5Book/SymbolFilter.mqh>

void OnStart()
{
   SymbolFilter f;                        // filter object
   string symbols[];                      // array of symbol names
   long permissions[][2];                 // array with property value vectors

   // properties to read
   ENUM_SYMBOL_INFO_INTEGER modes[] =
   {
      SYMBOL_TRADE_EXEMODE,
      SYMBOL_FILLING_MODE
   };
   // apply filter - fill arrays
   f.select(true, modes, symbols, permissions);

   const int n = ArraySize(symbols);
   PrintFormat("===== Trade execution and filling modes for the symbols (%d) =====",
   for(int i = 0; i < n; ++i)
   {
      Print(symbols[i] + ":");
      for(int j = 0; j < ArraySize(modes); ++j)
      {
         // output properties as descriptions and numbers
         PrintFormat("  %s (%d)",
            SymbolMonitor::stringify(permissions[i][j], modes[j]),
            permissions[i][j]);
      }
   }
}
```

A continuación se muestra un fragmento del registro con los resultados del script. Casi todos los símbolos tienen aquí un modo de ejecución inmediata a precios (SYMBOL_TRADE_EXECUTION_INSTANT) excepto el último SP500m (SYMBOL_TRADE_EXECUTION_MARKET). Aquí podemos encontrar varios modos de llenado de volumen, tanto SYMBOL_FILLING_FOK por separado, SYMBOL_FILLING_IOC, como su combinación. Sólo BTCUSD tiene especificado SYMBOL_FILLING_RETURN, es decir, se recibió un valor de 0 (sin bits FOK y IOC).

```
===== Trade execution and filling modes for the symbols (13) =====
EURUSD:
  SYMBOL_TRADE_EXECUTION_INSTANT (1)
  [ _SYMBOL_FILLING_FOK ] (1)
GBPUSD:
  SYMBOL_TRADE_EXECUTION_INSTANT (1)
  [ _SYMBOL_FILLING_FOK ] (1)
...
USDCNH:
  SYMBOL_TRADE_EXECUTION_INSTANT (1)
  [ _SYMBOL_FILLING_FOK _SYMBOL_FILLING_IOC ] (3)
USDRUB:
  SYMBOL_TRADE_EXECUTION_INSTANT (1)
  [ _SYMBOL_FILLING_IOC ] (2)
AUDUSD:
  SYMBOL_TRADE_EXECUTION_INSTANT (1)
  [ _SYMBOL_FILLING_FOK ] (1)
NZDUSD:
  SYMBOL_TRADE_EXECUTION_INSTANT (1)
  [ _SYMBOL_FILLING_FOK _SYMBOL_FILLING_IOC ] (3)
...
XAUUSD:
  SYMBOL_TRADE_EXECUTION_INSTANT (1)
  [ _SYMBOL_FILLING_FOK _SYMBOL_FILLING_IOC ] (3)
BTCUSD:
  SYMBOL_TRADE_EXECUTION_INSTANT (1)
  [(_SYMBOL_FILLING_RETURN)] (0)
SP500m:
  SYMBOL_TRADE_EXECUTION_MARKET (2)
  [ _SYMBOL_FILLING_FOK ] (1)
```

Recordemos que los guiones bajos en los identificadores del modo de relleno aparecen debido a que tuvimos que definir nuestra propia enumeración SYMBOL_FILLING (*SymbolMonitor.mqh*) con elementos con valores constantes. Esto se hizo porque MQL5 no tiene tal enumeración integrada, pero al mismo tiempo, no podemos nombrar los elementos de nuestra enumeración exactamente como constantes integradas, ya que esto causaría un conflicto de nombres.

6.1.16 Requisitos de margen

Una de las informaciones más importantes sobre un instrumento financiero para un operador es la cantidad de fondos necesarios para abrir una posición. Sin saber cuánto dinero se necesita para comprar o vender un número determinado de lotes, es imposible implementar un sistema de gestión de dinero dentro de un Asesor Experto y controlar el saldo de la cuenta.

Dado que MetaTrader 5 se utiliza para operar diversos instrumentos (divisas, materias primas, acciones, bonos, opciones y futuros), los principios de cálculo del margen difieren significativamente. La documentación proporciona detalles, en particular para el mercado de divisas y futuros, así como mercados bursátiles.

Varias propiedades de la API de MQL5 permiten definir el tipo de mercado y el método de cálculo del margen para un instrumento concreto.

Supongamos que, para una determinada combinación de parámetros, como el tipo de operación de trading, el instrumento, el volumen y el precio, MQL5 le permite calcular el margen utilizando la función *OrderCalcMargin*. Este es el método más sencillo, pero tiene una limitación importante: la función no tiene en cuenta las posiciones abiertas actuales ni las órdenes pendientes. Esto, en particular, ignora los posibles ajustes por solapamiento de volúmenes cuando se permiten posiciones opuestas en la cuenta.

Así, para obtener un desglose de los fondos de la cuenta utilizados actualmente como margen para las órdenes y posiciones abiertas, un programa MQL puede necesitar analizar las siguientes propiedades y cálculos mediante fórmulas. Además, está prohibido utilizar la función *OrderCalcMargin* en los indicadores. Puede estimar el margen libre por adelantado después de que se complete la transacción propuesta utilizando *OrderCheck*.

Identificador	Descripción
SYMBOL_TRADE_CALC_MODE	El método para calcular el margen y el beneficio (véase ENUM_SYMBOL_CALC_MODE)
SYMBOL_MARGIN_HEDGED_USE_LEG	Bandera booleana para activar (true) o desactivar (false) el modo de cálculo del margen de cobertura para la mayor de las posiciones solapadas (compra y venta).
SYMBOL_MARGIN_INITIAL	Margen inicial de un instrumento bursátil
SYMBOL_MARGIN_MAINTENANCE	Margen de mantenimiento de un instrumento bursátil
SYMBOL_MARGIN_HEDGED	Tamaño del contrato o margen para un lote de posiciones cubiertas (posiciones opuestas para un símbolo)

Las dos primeras propiedades están incluidas en la enumeración ENUM_SYMBOL_INFO_INTEGER, y las tres últimas en ENUM_SYMBOL_INFO_DOUBLE, y pueden ser leídas, respectivamente, por las funciones *SymbolInfoInteger* y *SymbolInfoDouble*.

Las fórmulas específicas de cálculo de márgenes dependen de la propiedad SYMBOL_TRADE_CALC_MODE y se muestran en la tabla que aparece más abajo. Encontrará información más completa en Documentación MQL5.

Tenga en cuenta que los márgenes inicial y de mantenimiento no se utilizan para los instrumentos Forex, y estas propiedades son siempre 0 para ellos.

El margen inicial indica el importe del depósito de garantía exigido en la divisa de margen para abrir una posición con un volumen de un lote. Se utiliza cuando se comprueba la suficiencia de los fondos del cliente antes de entrar en el mercado. Para obtener el importe final del margen cobrado en función del tipo y la dirección de la orden, compruebe los coeficientes de margen utilizando la función *SymbolInfoMarginRate*. Así, el bróker puede establecer un apalancamiento o descuento individual para cada instrumento.

El margen de mantenimiento indica el valor mínimo de fondos en la divisa del margen del instrumento para mantener una posición abierta de un lote. Se utiliza al comprobar la suficiencia de fondos del cliente cuando cambia el estado de la cuenta (condiciones de trading). Si el nivel de fondos cae por debajo del importe del margen de mantenimiento de todas las posiciones, el bróker comenzará a cerrarlas a la fuerza.

Si la propiedad de margen de mantenimiento es 0, se utiliza el margen inicial. Al igual que en el caso del margen inicial, para obtener el importe final del margen cobrado en función del tipo y la dirección, deberá comprobar los coeficientes de margen utilizando la función *SymbolInfoMarginRate*.

Las posiciones de cobertura, es decir, las posiciones multidireccionales para un mismo símbolo, sólo pueden existir en cuentas de cobertura en trading. Obviamente, el cálculo del margen de cobertura junto con las propiedades SYMBOL_MARGIN_HEDGED_USE_LEG, SYMBOL_MARGIN_HEDGED sólo tiene sentido en dichas cuentas. El margen de cobertura se aplica al volumen cubierto.

El bróker puede elegir para cada instrumento uno de los dos métodos existentes para calcular el margen de las posiciones cubiertas:

- El cálculo base se aplica cuando el modo de cálculo del lado más largo está desactivado, es decir, la propiedad SYMBOL_MARGIN_HEDGED_USE_LEG es igual a *false*. En este caso, el margen consta de tres componentes: el margen para el volumen descubierto de la posición existente, el margen para el volumen cubierto (si hay posiciones opuestas y la propiedad SYMBOL_MARGIN_HEDGED es distinta de cero), el margen para órdenes pendientes. Si se establece el margen inicial para el instrumento (la propiedad SYMBOL_MARGIN_INITIAL es distinta de cero), entonces el margen de cobertura se especifica como un valor absoluto (en dinero). Si no se fija el margen inicial (igual a 0), entonces SYMBOL_MARGIN_HEDGED especifica el tamaño del contrato que se utilizará al calcular el margen según la fórmula correspondiente al tipo de instrumento de trading (SYMBOL_TRADE_CALC_MODE).
- El cálculo de la posición más alta se aplica cuando la propiedad SYMBOL_MARGIN_HEDGED_USE_LEG es igual a *true*. El valor de SYMBOL_MARGIN_HEDGED se ignora en este caso. En lugar de ello se calcula el volumen de todas las posiciones cortas y largas en el instrumento, y se calcula el precio medio ponderado de apertura para cada lado. Además, utilizando las fórmulas correspondientes al tipo de instrumento (SYMBOL_TRADE_CALC_MODE), se calcula el margen para el lado corto y el lado largo. El valor mayor se utiliza como valor final.

En la siguiente tabla se enumeran los elementos ENUM_SYMBOL_CALC_MODE y sus respectivos métodos de cálculo de márgenes. La misma propiedad (SYMBOL_TRADE_CALC_MODE) es también responsable de calcular el beneficio/pérdida de una posición, pero consideraremos este aspecto más adelante, en el capítulo sobre las funciones de trading de MQL5.

Identificador	Fórmula
SYMBOL_CALC_MODE_FOREX *Forex*	Lots * ContractSize * MarginRate / Leverage
SYMBOL_CALC_MODE_FOREX_NO_LEVERAGE *Forex without leverage*	Lots * ContractSize * MarginRate
SYMBOL_CALC_MODE_CFD *CFD*	Lots * ContractSize * MarketPrice * MarginRate
SYMBOL_CALC_MODE_CFDLEVERAGE *CFD with leverage*	Lots * ContractSize * MarketPrice * MarginRate / Leverage
SYMBOL_CALC_MODE_CFDINDEX *CFDs on indices*	Lots * ContractSize * MarketPrice * TickPrice / TickSize * MarginRate
SYMBOL_CALC_MODE_EXCH_STOCKS *Securities on the stock exchange*	Lots * ContractSize * LastPrice * MarginRate

Identificador	Fórmula
SYMBOL_CALC_MODE_EXCH_STOCKS_MOEX *Securities on MOEX*	Lots * ContractSize * LastPrice * MarginRate
SYMBOL_CALC_MODE_FUTURES *Futures*	Lots * InitialMargin * MarginRate
SYMBOL_CALC_MODE_EXCH_FUTURES *Futures on the stock exchange*	Lots * InitialMargin * MarginRate *or* Lots * MaintenanceMargin * MarginRate
SYMBOL_CALC_MODE_EXCH_FUTURES_FORTS *Futures on FORTS*	Lots * InitialMargin * MarginRate *or* Lots * MaintenanceMargin * MarginRate
SYMBOL_CALC_MODE_EXCH_BONDS *Bonds on the stock exchange*	Lots * ContractSize * FaceValue * OpenPrice / 100
SYMBOL_CALC_MODE_EXCH_BONDS_MOEX *Bonds on MOEX*	Lots * ContractSize * FaceValue * OpenPrice / 100
SYMBOL_CALC_MODE_SERV_COLLATERAL	Activo no negociable (margen no aplicable)

En las fórmulas se utiliza la siguiente notación:

- Lots: posición o volumen de la orden en lotes (acciones del contrato).
- ContractSize: tamaño del contrato (un lote, SYMBOL_TRADE_CONTRACT_SIZE)
- Leverage: apalancamiento de la cuenta de trading (ACCOUNT_LEVERAGE)
- InitialMargin: margen inicial (SYMBOL_MARGIN_INITIAL)
- MaintenanceMargin: margen de mantenimiento (SYMBOL_MARGIN_MAINTENANCE)
- TickPrice: precio del tick (SYMBOL_TRADE_TICK_VALUE)
- TickSize: tamaño del tick (SYMBOL_TRADE_TICK_SIZE)
- MarketPrice: el último precio *Bid/Ask* conocido en función del tipo de transacción
- LastPrice: el último precio *Last* conocido
- OpenPrice: precio medio ponderado de apertura de una posición u orden
- FaceValue: valor nominal del bono
- MarginRate: coeficiente de margen según la función *SymbolInfoMarginRate*; puede tener también dos valores diferentes: para el margen inicial y para el margen de mantenimiento.

Encontrará una implementación alternativa de los cálculos de la fórmula para la mayoría de los tipos de símbolos en el archivo *MarginProfitMeter.mqh* (véase la sección Estimación del beneficio de una operación de trading) que también puede utilizarse en indicadores.

Formulemos un par de comentarios sobre algunos modos.

En la tabla anterior, sólo tres de las fórmulas de futuros utilizan el margen inicial (SYMBOL_MARGIN_INITIAL). Sin embargo, si esta propiedad tiene un valor distinto de cero en la especificación de cualquier otro símbolo, entonces determina el margen.

Algunas bolsas pueden imponer sus propias especificidades en el ajuste de márgenes, como el sistema de descuento para FORTS (SYMBOL_CALC_MODE_EXCH_FUTURES_FORTS). Consulte la documentación de MQL5 y a su bróker para obtener más detalles.

En el modo SYMBOL_CALC_MODE_SERV_COLLATERAL, el valor de un instrumento se tiene en cuenta en los Activos, que se suman al Capital (Equity). Por lo tanto, las posiciones abiertas en dicho instrumento aumentan el importe del Margen Libre y sirven como garantía adicional para las posiciones abiertas en instrumentos negociados. El valor de mercado de una posición abierta se calcula en función del volumen, el tamaño del contrato, el precio actual de mercado y el coeficiente de liquidez: *Lots * ContractSize * MarketPrice * LiquidityRate* (este último valor puede obtenerse como la propiedad SYMBOL_TRADE_LIQUIDITY_RATE).

Como ejemplo de trabajo con propiedades relacionadas con los márgenes, vea el script *SymbolFilterMarginStats.mq5*. Su objetivo será calcular estadísticas sobre los métodos de cálculo de márgenes en la lista de símbolos seleccionados, así como, opcionalmente, registrar estas propiedades para cada símbolo. Seleccionaremos los símbolos para el análisis utilizando la clase de filtro ya conocida *SymbolFilter* y las condiciones para ello suministradas a partir de las variables de entrada.

```
#include <MQL5Book/SymbolFilter.mqh>

input bool UseMarketWatch = false;
input bool ShowPerSymbolDetails = false;
input bool ExcludeZeroInitMargin = false;
input bool ExcludeZeroMainMargin = false;
input bool ExcludeZeroHedgeMargin = false;
```

Por defecto, se solicita información para todos los símbolos disponibles. Para limitar el contexto únicamente a la visión general del mercado, debemos fijar *UseMarketWatch* en *true*.

El parámetro *ShowPerSymbolDetails* permite activar la salida de información detallada sobre cada símbolo (por defecto, el parámetro es *false*, y sólo se muestran estadísticas).

Los tres últimos parámetros sirven para filtrar los símbolos según las condiciones de los valores de margen cero (inicial, de mantenimiento y de cobertura, respectivamente).

Para recoger y mostrar convenientemente en el registro un conjunto completo de propiedades para cada símbolo (cuando *ShowPerSymbolDetails* está activada), se define en el código la estructura *MarginSettings*.

```
struct MarginSettings
{
   string name;
   ENUM_SYMBOL_CALC_MODE calcMode;
   bool hedgeLeg;
   double initial;
   double maintenance;
   double hedged;
};
```

Dado que algunas de las propiedades son de enteros (SYMBOL_TRADE_CALC_MODE, SYMBOL_MARGIN_HEDGED_USE_LEG), y algunas de reales (SYMBOL_MARGIN_INITIAL, SYMBOL_MARGIN_MAINTENANCE, SYMBOL_MARGIN_HEDGED), el objeto de filtro tendrá que solicitarlas por separado.

Ahora vamos directamente al código de trabajo en *OnStart*. Aquí, como de costumbre, definimos el objeto de filtro (*f*), los arrays de salida para los nombres de los caracteres (*symbols*) y los valores de las propiedades solicitadas (*flags*,*values*). Además de ellas, añadimos un array de estructuras *MarginSettings*.

```
void OnStart()
{
   SymbolFilter f;                // filter object
   string symbols[];              // array for names
   long flags[][2];               // array for integer vectors
   double values[][3];            // array for real vectors
   MarginSettings margins[];      // composite output array
   ...
```

Se ha introducido el mapa del array *stats* para calcular estadísticas con una clave como ENUM_SYMBOL_CALC_MODE y el valor entero *int* para el número de veces que se ha encontrado cada método. Además, todos los casos de margen cero y el modo de cálculo habilitado en el tramo más largo deben registrarse en las variables del contador correspondientes.

```
   MapArray<ENUM_SYMBOL_CALC_MODE,int> stats; // counters for each method/mode
   int hedgeLeg = 0;                          // and other counters
   int zeroInit = 0;                          // ...
   int zeroMaintenance = 0;
   int zeroHedged = 0;
   ...
```

A continuación, especificamos las propiedades que nos interesan relacionadas con el margen, que se leerán de la configuración del símbolo. Primero, los enteros del array *ints* y luego, los reales del array *doubles*.

```
   ENUM_SYMBOL_INFO_INTEGER ints[] =
   {
      SYMBOL_TRADE_CALC_MODE,
      SYMBOL_MARGIN_HEDGED_USE_LEG
   };

   ENUM_SYMBOL_INFO_DOUBLE doubles[] =
   {
      SYMBOL_MARGIN_INITIAL,
      SYMBOL_MARGIN_MAINTENANCE,
      SYMBOL_MARGIN_HEDGED
   };
   ...
```

En función de los parámetros de entrada, estableceremos las condiciones de filtrado.

```
   if(ExcludeZeroInitMargin) f.let(SYMBOL_MARGIN_INITIAL, 0, SymbolFilter::IS::GREATE
   if(ExcludeZeroMainMargin) f.let(SYMBOL_MARGIN_MAINTENANCE, 0, SymbolFilter::IS::GR
   if(ExcludeZeroHedgeMargin) f.let(SYMBOL_MARGIN_HEDGED, 0, SymbolFilter::IS::GREATE
   ...
```

Ahora todo está listo para seleccionar símbolos por condiciones y obtener sus propiedades en arrays. Lo hacemos dos veces, por un lado para las propiedades de enteros y por otro, para las de reales.

```
f.select(UseMarketWatch, ints, symbols, flags);
const int n = ArraySize(symbols);
ArrayResize(symbols, 0, n);
f.select(UseMarketWatch, doubles, symbols, values);
...
```

Un array con símbolos debe ponerse a cero después de la primera aplicación del filtro para que los nombres no se dupliquen. A pesar de tratarse de dos consultas distintas, el orden de los elementos en todos los arrays de salida (*ints* y *doubles*) es el mismo, ya que las condiciones de filtrado no cambian.

Si el usuario activa un registro detallado, asignamos memoria para el array de estructuras *margins*.

```
if(ShowPerSymbolDetails) ArrayResize(margins, n);
```

Por último, calculamos las estadísticas iterando sobre todos los elementos de los arrays resultantes y, opcionalmente, rellenamos el array de estructuras.

```
for(int i = 0; i < n; ++i)
{
   stats.inc((ENUM_SYMBOL_CALC_MODE)flags[i].value[0]);
   hedgeLeg += (int)flags[i].value[1];
   if(values[i].value[0] == 0) zeroInit++;
   if(values[i].value[1] == 0) zeroMaintenance++;
   if(values[i].value[2] == 0) zeroHedged++;

   if(ShowPerSymbolDetails)
   {
      margins[i].name = symbols[i];
      margins[i].calcMode = (ENUM_SYMBOL_CALC_MODE)flags[i][0];
      margins[i].hedgeLeg = (bool)flags[i][1];
      margins[i].initial = values[i][0];
      margins[i].maintenance = values[i][1];
      margins[i].hedged = values[i][2];
   }
}
...
```

Ahora mostramos las estadísticas en el registro.

```
   PrintFormat("===== Margin calculation modes for %s symbols %s=====",
      (UseMarketWatch ? "Market Watch" : "all available"),
      (ExcludeZeroInitMargin || ExcludeZeroMainMargin || ExcludeZeroHedgeMargin
         ? "(with conditions) " : ""));
   PrintFormat("Total symbols: %d", n);
   PrintFormat("Hedge leg used in: %d", hedgeLeg);
   PrintFormat("Zero margin counts: initial=%d, maintenance=%d, hedged=%d",
      zeroInit, zeroMaintenance, zeroHedged);

   Print("Stats per calculation mode:");
   stats.print();
   ...
```

Dado que los miembros de la enumeración ENUM_SYMBOL_CALC_MODE se muestran como números enteros (lo cual no es muy informativo), mostramos también un texto en el que cada valor tiene un nombre (de *EnumToString*).

```
   Print("Legend: key=calculation mode, value=count");
   for(int i = 0; i < stats.getSize(); ++i)
   {
      PrintFormat("%d -> %s", stats.getKey(i), EnumToString(stats.getKey(i)));
   }
   ...
```

Si se necesita información detallada sobre los caracteres seleccionados, se emite el array de estructuras *margins*.

```
   if(ShowPerSymbolDetails)
   {
      Print("Settings per symbol:");
      ArrayPrint(margins);
   }
}
```

Vamos a ejecutar el script un par de veces con diferentes configuraciones. Empecemos con la configuración por defecto.

```
===== Margin calculation modes for all available symbols =====
Total symbols: 131
Hedge leg used in: 14
Zero margin counts: initial=123, maintenance=130, hedged=32
Stats per calculation mode:
    [key] [value]
[0]     0     101
[1]     4      16
[2]     1       1
[3]     2      11
[4]     5       2
Legend: key=calculation mode, value=count
0 -> SYMBOL_CALC_MODE_FOREX
4 -> SYMBOL_CALC_MODE_CFDLEVERAGE
1 -> SYMBOL_CALC_MODE_FUTURES
2 -> SYMBOL_CALC_MODE_CFD
```

```
5 -> SYMBOL_CALC_MODE_FOREX_NO_LEVERAGE
```

Para la segunda ejecución, establezcamos *ShowPerSymbolDetails* y *ExcludeZeroInitMargin* en *true*. Así se solicita información detallada sobre todos los símbolos que tienen un valor distinto de cero del margen inicial.

```
===== Margin calculation modes for all available symbols (with conditions) =====
Total symbols: 8
Hedge leg used in: 0
Zero margin counts: initial=0, maintenance=7, hedged=0
Stats per calculation mode:
    [key] [value]
[0]     0       5
[1]     1       1
[2]     5       2
Legend: key=calculation mode, value=count
0 -> SYMBOL_CALC_MODE_FOREX
1 -> SYMBOL_CALC_MODE_FUTURES
5 -> SYMBOL_CALC_MODE_FOREX_NO_LEVERAGE
Settings per symbol:
      [name] [calcMode] [hedgeLeg]    [initial] [maintenance]    [hedged]
[0] "XAUEUR"          0      false    100.00000       0.00000    50.00000
[1] "XAUAUD"          0      false    100.00000       0.00000   100.00000
[2] "XAGEUR"          0      false   1000.00000       0.00000  1000.00000
[3] "USDGEL"          0      false 100000.00000  100000.00000 50000.00000
[4] "SP500m"          1      false   6600.00000       0.00000  6600.00000
[5] "XBRUSD"          5      false    100.00000       0.00000    50.00000
[6] "XNGUSD"          0      false  10000.00000       0.00000 10000.00000
[7] "XTIUSD"          5      false    100.00000       0.00000    50.00000
```

6.1.17 Reglas de vencimiento de órdenes pendientes

Cuando se trabaja con órdenes pendientes (incluyendo los niveles *Stop Loss* y *Take Profit*), un programa MQL debe comprobar un par de propiedades que definen las reglas para su vencimiento. Ambas propiedades están disponibles como miembros de la enumeración ENUM_SYMBOL_INFO_INTEGER para la llamada a la función *SymbolInfoInteger*.

Identificador	Descripción
SYMBOL_EXPIRATION_MODE	Banderas de los modos de vencimiento de órdenes permitidos (máscara de bits)
SYMBOL_ORDER_GTC_MODE	El período de validez se define mediante uno de los elementos de la enumeración ENUM_SYMBOL_ORDER_GTC_MODE

La propiedad SYMBOL_ORDER_GTC_MODE sólo se tiene en cuenta si SYMBOL_EXPIRATION_MODE contiene SYMBOL_EXPIRATION_GTC (véase más adelante). GTC es el acrónimo de Good Till Canceled (válida hasta que se cancele).

Para cada instrumento financiero, la propiedad SYMBOL_EXPIRATION_MODE puede especificar varios modos de validez (vencimiento) de las órdenes pendientes. Cada modo tiene asociada una bandera (bit).

Identificador (Valor)	Descripción
SYMBOL_EXPIRATION_GTC (1)	La orden es válida según la propiedad ENUM_SYMBOL_ORDER_GTC_MODE.
SYMBOL_EXPIRATION_DAY (2)	La orden es válida hasta el final del día en curso.
SYMBOL_EXPIRATION_SPECIFIED (4)	La fecha y hora de caducidad se especifican en la orden.
SYMBOL_EXPIRATION_SPECIFIED_DAY (8)	La fecha de vencimiento se especifica en la orden.

Las banderas pueden combinarse con una operación lógica OR ('|'); por ejemplo, SYMBOL_EXPIRATION_GTC | SYMBOL_EXPIRATION_SPECIFIED, equivalente a 1 | 4, que es el número 5. Para comprobar si un modo concreto está activado para una herramienta, realice una operación lógica AND ('&') sobre el resultado de la función y el bit de modo deseado: un valor distinto de cero significa que el modo está disponible.

En el caso de SYMBOL_EXPIRATION_SPECIFIED_DAY, la orden es válida hasta las 23:59:59 del día especificado. Si esta hora no coincide con la sesión de trading, el vencimiento se producirá en la siguiente hora de trading más próxima.

La enumeración ENUM_SYMBOL_ORDER_GTC_MODE contiene los siguientes miembros:

Identificador	Descripción
SYMBOL_ORDERS_GTC	Las órdenes pendientes y los niveles Stop Loss/Take Profit son válidos de forma indefinida hasta que se cancelen explícitamente
SYMBOL_ORDERS_DAILY	Las órdenes sólo son válidas durante un día de trading: una vez finalizado, se eliminan todas las órdenes pendientes, así como los niveles Stop Loss y Take Profit.
SYMBOL_ORDERS_DAILY_EXCLUDING_STOPS	Al cambiar el día de trading sólo se eliminan las órdenes pendientes, pero se guardan los niveles de Stop Loss y Take Profit.

Dependiendo de los bits establecidos en la propiedad SYMBOL_EXPIRATION_MODE, al preparar una orden para su envío, un programa MQL puede seleccionar uno de los modos correspondientes a estos bits. Técnicamente, esto se hace rellenando el campo type_time en una estructura especial *MqlTradeRequest* antes de llamar a la función *OrderSend*. El valor del campo debe ser un elemento de la enumeración ENUM_ORDER_TYPE_TIME (véase Fechas de vencimiento de órdenes pendientes): como veremos más adelante, tiene algo en común con el conjunto de banderas anterior, es decir, cada bandera establece el modo correspondiente en la orden ORDER_TIME_GTC, ORDER_TIME_DAY, ORDER_TIME_SPECIFIED, ORDER_TIME_SPECIFIED_DAY. La hora o el día de vencimiento deben especificarse en otro campo de la misma estructura.

El script *SymbolFilterExpiration.mq5* permite conocer las estadísticas de uso de cada una de las banderas en los símbolos disponibles (en la visión general del mercado o en general, dependiendo del parámetro de entrada *UseMarketWatch*). El segundo parámetro de *ShowPerSymbolDetails*, al ser igual a *true*, hará que se registren todas las banderas de cada carácter, así que tenga cuidado: si al mismo

tiempo, el modo *UseMarketWatch* es igual a *false*, se generará un número muy grande de entradas de registro.

```
#property script_show_inputs

#include <MQL5Book/SymbolFilter.mqh>

input bool UseMarketWatch = false;
input bool ShowPerSymbolDetails = false;
```

En la función *OnStart*, además del objeto filtro y los arrays receptores para los nombres de los símbolos y los valores de las propiedades, describimos *MapArray* para calcular las estadísticas por separado para cada una de las propiedades SYMBOL_EXPIRATION_MODE y SYMBOL_ORDER_GTC_MODE.

```
void OnStart()
{
   SymbolFilter f;                // filter object
   string symbols[];              // receiving array for symbol names
   long flags[][2];               // receiving array for property values

   MapArray<SYMBOL_EXPIRATION,int> stats;        // mode counters
   MapArray<ENUM_SYMBOL_ORDER_GTC_MODE,int> gtc; // GTC counters

   ENUM_SYMBOL_INFO_INTEGER ints[] =
   {
      SYMBOL_EXPIRATION_MODE,
      SYMBOL_ORDER_GTC_MODE
   };
   ...
```

A continuación, aplique el filtro y calcule las estadísticas.

```
   f.select(UseMarketWatch, ints, symbols, flags);
   const int n = ArraySize(symbols);

   for(int i = 0; i < n; ++i)
   {
      if(ShowPerSymbolDetails)
      {
         Print(symbols[i] + ":");
         for(int j = 0; j < ArraySize(ints); ++j)
         {
            // properties in the form of descriptions and numbers
            PrintFormat("  %s (%d)",
               SymbolMonitor::stringify(flags[i][j], ints[j]),
               flags[i][j]);
         }
      }

      const SYMBOL_EXPIRATION mode = (SYMBOL_EXPIRATION)flags[i][0];
      for(int j = 0; j < 4; ++j)
      {
         const SYMBOL_EXPIRATION bit = (SYMBOL_EXPIRATION)(1 << j);
         if((mode & bit) != 0)
         {
            stats.inc(bit);
         }

         if(bit == SYMBOL_EXPIRATION_GTC)
         {
            gtc.inc((ENUM_SYMBOL_ORDER_GTC_MODE)flags[i][1]);
         }
      }
   }
   ...
```

Por último, enviamos los números recibidos al registro.

```
      PrintFormat("===== Expiration modes for %s symbols =====",
         (UseMarketWatch ? "Market Watch" : "all available"));
      PrintFormat("Total symbols: %d", n);

      Print("Stats per expiration mode:");
      stats.print();
      Print("Legend: key=expiration mode, value=count");
      for(int i = 0; i < stats.getSize(); ++i)
      {
         PrintFormat("%d -> %s", stats.getKey(i), EnumToString(stats.getKey(i)));
      }
      Print("Stats per GTC mode:");
      gtc.print();
      Print("Legend: key=GTC mode, value=count");
      for(int i = 0; i < gtc.getSize(); ++i)
      {
         PrintFormat("%d -> %s", gtc.getKey(i), EnumToString(gtc.getKey(i)));
      }
   }
```

Vamos a ejecutar el script dos veces. La primera vez, con la configuración por defecto, podemos obtener algo como lo que se muestra en la siguiente imagen:

```
===== Expiration modes for all available symbols =====
Total symbols: 52357
Stats per expiration mode:
    [key] [value]
[0]     1   52357
[1]     2   52357
[2]     4   52357
[3]     8   52303
Legend: key=expiration mode, value=count
1 -> _SYMBOL_EXPIRATION_GTC
2 -> _SYMBOL_EXPIRATION_DAY
4 -> _SYMBOL_EXPIRATION_SPECIFIED
8 -> _SYMBOL_EXPIRATION_SPECIFIED_DAY
Stats per GTC mode:
    [key] [value]
[0]     0   52357
Legend: key=GTC mode, value=count
0 -> SYMBOL_ORDERS_GTC
```

Aquí puede ver que casi todas las banderas están permitidas para la mayoría de los símbolos, y para el modo SYMBOL_EXPIRATION_GTC, se utiliza la única variante SYMBOL_ORDERS_GTC.

Ejecute el script por segunda vez ajustando *UseMarketWatch* y *ShowPerSymbolDetails* a *true* (se supone que en *Market Watch* se selecciona un número limitado de símbolos).

```
  GBPUSD:
    [ _SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY _SYMBOL_EXPIRATION_SPECIFIED ] (7)
    SYMBOL_ORDERS_GTC (0)
  USDCHF:
    [ _SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY _SYMBOL_EXPIRATION_SPECIFIED ] (7)
    SYMBOL_ORDERS_GTC (0)
  USDJPY:
    [ _SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY _SYMBOL_EXPIRATION_SPECIFIED ] (7)
    SYMBOL_ORDERS_GTC (0)
  ...
  XAUUSD:
    [ _SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY _SYMBOL_EXPIRATION_SPECIFIED
    _SYMBOL_EXPIRATION_SPECIFIED_DAY ] (15)
    SYMBOL_ORDERS_GTC (0)
  SP500m:
    [ _SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY _SYMBOL_EXPIRATION_SPECIFIED
    _SYMBOL_EXPIRATION_SPECIFIED_DAY ] (15)
    SYMBOL_ORDERS_GTC (0)
  UK100:
    [ _SYMBOL_EXPIRATION_GTC _SYMBOL_EXPIRATION_DAY _SYMBOL_EXPIRATION_SPECIFIED
    _SYMBOL_EXPIRATION_SPECIFIED_DAY ] (15)
    SYMBOL_ORDERS_GTC (0)
  ===== Expiration modes for Market Watch symbols =====
  Total symbols: 15
  Stats per expiration mode:
      [key] [value]
  [0]     1      15
  [1]     2      15
  [2]     4      15
  [3]     8       6
  Legend: key=expiration mode, value=count
  1 -> _SYMBOL_EXPIRATION_GTC
  2 -> _SYMBOL_EXPIRATION_DAY
  4 -> _SYMBOL_EXPIRATION_SPECIFIED
  8 -> _SYMBOL_EXPIRATION_SPECIFIED_DAY
  Stats per GTC mode:
      [key] [value]
  [0]     0      15
  Legend: key=GTC mode, value=count
  0 -> SYMBOL_ORDERS_GTC
```

De los 15 símbolos seleccionados, sólo 6 tienen activada la bandera SYMBOL_EXPIRATION_SPECIFIED_DAY. Más arriba encontrará información detallada sobre las banderas de cada símbolo.

6.1.18 Diferenciales y distancia de orden con respecto al precio actual

Para muchas estrategias de trading, especialmente las que se basan en operaciones a corto plazo, es importante disponer de información sobre el diferencial y la distancia con respecto al precio actual que permita la instalación o modificación de órdenes. Todas estas propiedades forman parte de la

enumeración ENUM_SYMBOL_INFO_INTEGER y están disponibles a través de la función *SymbolInfoInteger*.

Identificador	Descripción
SYMBOL_SPREAD	Tamaño del diferencial (en puntos)
SYMBOL_SPREAD_FLOAT	Signo booleano de un diferencial flotante
SYMBOL_TRADE_STOPS_LEVEL	Distancia mínima permitida desde el precio actual (en puntos) para establecer Stop Loss, Take Profit y órdenes pendientes.
SYMBOL_TRADE_FREEZE_LEVEL	Distancia desde el precio actual (en puntos) para congelar órdenes y posiciones

En el cuadro anterior, el precio actual se refiere al precio de *Ask* o *Bid*, según la naturaleza de la operación que se esté realizando.

Los niveles de protección *Stop Loss* y *Take Profit* indican que debe cerrarse una posición. Esto se realiza mediante una operación opuesta a la apertura. Por lo tanto, para las órdenes de compra abiertas al precio *Ask*, los niveles de protección indican *Bid*, y para las órdenes de venta abiertas en *Bid*, los niveles de protección indican *Ask*. Al colocar órdenes pendientes, el tipo de precio abierto se selecciona de forma estándar: las órdenes de compra (*Buy Stop*, *Buy Limit*, *Buy Stop Limit*) se basan en *Ask*, y las órdenes de venta (*Sell Stop*, *Sell Limit*, *Sell Stop Limit*) se basan en *Bid*. Teniendo en cuenta estos tipos de precios en el contexto de las operaciones de trading mencionadas, se calcula la distancia en puntos para las propiedades SYMBOL_TRADE_STOPS_LEVEL y SYMBOL_TRADE_FREEZE_LEVEL.

La propiedad SYMBOL_TRADE_STOPS_LEVEL, si es distinta de cero, desactiva la modificación de los niveles *Stop Loss* y *Take Profit* para una posición abierta si el nuevo nivel estuviera más cerca del precio actual que la distancia especificada. Del mismo modo, es imposible mover el precio de apertura de una orden pendiente a menos puntos que SYMBOL_TRADE_STOPS_LEVEL del precio actual.

La propiedad SYMBOL_TRADE_FREEZE_LEVEL, si es distinta de cero, restringe cualquier operación de trading para una orden pendiente o una posición abierta dentro de la distancia especificada desde el precio actual. Para una orden pendiente, la congelación se produce cuando el precio de apertura especificado está a una distancia inferior a los puntos SYMBOL_TRADE_FREEZE_LEVEL con respecto del precio actual (de nuevo, el tipo del precio actual es *Ask* o *Bid*, dependiendo de si es de compra o de venta). Para una posición, la congelación ocurre para los niveles *Stop Loss* y *Take Profit*, que resultaron estar cerca del precio actual, y por lo tanto la medición para ellos se realiza para tipos de precios «opuestos».

Si la propiedad SYMBOL_SPREAD_FLOAT es *true*, la propiedad SYMBOL_SPREAD no forma parte de la especificación del símbolo, sino que contiene el diferencial real, que cambia dinámicamente con cada llamada según las condiciones del mercado. También puede encontrarse como la diferencia entre los precios de *Ask* y *Bid* en la estructura *MqlTick* llamando a *SymbolInfoTick*.

El script *SymbolFilterSpread.mq5* le permitirá analizar las propiedades especificadas. Define una enumeración personalizada ENUM_SYMBOL_INFO_INTEGER_PART, que incluye sólo las propiedades que nos interesan en este contexto de ENUM_SYMBOL_INFO_INTEGER.

```
enum ENUM_SYMBOL_INFO_INTEGER_PART
{
   SPREAD_FIXED = SYMBOL_SPREAD,
   SPREAD_FLOAT = SYMBOL_SPREAD_FLOAT,
   STOPS_LEVEL = SYMBOL_TRADE_STOPS_LEVEL,
   FREEZE_LEVEL = SYMBOL_TRADE_FREEZE_LEVEL
};
```

La nueva enumeración define el parámetro de entrada *Property*, que especifica cuál de las cuatro propiedades se analizará. Los parámetros *UseMarketWatch* y *ShowPerSymbolDetails* controlan el proceso de la forma ya conocida, como en los scripts de prueba anteriores.

```
input bool UseMarketWatch = true;
input ENUM_SYMBOL_INFO_INTEGER_PART Property = SPREAD_FIXED;
input bool ShowPerSymbolDetails = true;
```

Para la cómoda visualización de la información de cada símbolo (nombre de la propiedad y valor en cada línea) mediante la función *ArrayPrint*, se define una estructura auxiliar *SymbolDistance* (utilizada sólo cuando *ShowPerSymbolDetails* es igual a *true*).

```
struct SymbolDistance
{
   string name;
   int value;
};
```

En el manejador *OnStart*, describimos los objetos y arrays necesarios.

```
void OnStart()
{
   SymbolFilter f;                // filter object
   string symbols[];              // receiving array for names
   long values[];                 // receiving array for values
   SymbolDistance distances[];    // array to print
   MapArray<long,int> stats;      // counters of specific values of the selected prop
   ...
```

A continuación, aplicamos el filtro y rellenamos los arrays receptores con los valores de la dirección *Property* especificada, al tiempo que aplicamos la ordenación.

```
   f.select(UseMarketWatch, (ENUM_SYMBOL_INFO_INTEGER)Property, symbols, values, true
   const int n = ArraySize(symbols);
   if(ShowPerSymbolDetails) ArrayResize(distances, n);
   ...
```

En un bucle, contamos las estadísticas y rellenamos las estructuras *SymbolDistance*, si es necesario.

```
      for(int i = 0; i < n; ++i)
      {
         stats.inc(values[i]);
         if(ShowPerSymbolDetails)
         {
            distances[i].name = symbols[i];
            distances[i].value = (int)values[i];
         }
      }
      ...
```

Por último, enviamos los resultados al registro.

```
      PrintFormat("===== Distances for %s symbols =====",
         (UseMarketWatch ? "Market Watch" : "all available"));
      PrintFormat("Total symbols: %d", n);

      PrintFormat("Stats per %s:", EnumToString((ENUM_SYMBOL_INFO_INTEGER)Property));
      stats.print();

      if(ShowPerSymbolDetails)
      {
         Print("Details per symbol:");
         ArrayPrint(distances);
      }
   }
```

Esto es lo que se obtiene al ejecutar el script con la configuración por defecto, que es coherente con el análisis de diferenciales.

```
===== Distances for Market Watch symbols =====
Total symbols: 13
Stats per SYMBOL_SPREAD:
    [key] [value]
[0]     0       2
[1]     2       3
[2]     3       1
[3]     6       1
[4]     7       1
[5]     9       1
[6]   151       1
[7]   319       1
[8]  3356       1
[9]  3400       1
Details per symbol:
       [name] [value]
[ 0] "USDJPY"       0
[ 1] "EURUSD"       0
[ 2] "USDCHF"       2
[ 3] "USDCAD"       2
[ 4] "GBPUSD"       2
[ 5] "AUDUSD"       3
[ 6] "XAUUSD"       6
[ 7] "SP500m"       7
[ 8] "NZDUSD"       9
[ 9] "USDCNH"     151
[10] "USDSEK"     319
[11] "BTCUSD"    3356
[12] "USDRUB"    3400
```

Para entender si los diferenciales son flotantes (cambian dinámicamente) o fijos, vamos a ejecutar el script con diferentes configuraciones: Propiedad = SPREAD_FLOAT, ShowPerSymbolDetails = false.

```
===== Distances for Market Watch symbols =====
Total symbols: 13
Stats per SYMBOL_SPREAD_FLOAT:
    [key] [value]
[0]     1      13
```

Según estos datos, todos los símbolos de Observación de Mercado tienen un diferencial flotante (el valor 1 de la clave *key* es *true* en SYMBOL_SPREAD_FLOAT). Por lo tanto, si ejecutamos el script una y otra vez con la configuración por defecto, recibiremos nuevos valores (con un mercado abierto).

6.1.19 Obtener tamaños de swap

Para la aplicación de estrategias a medio y largo plazo, los tamaños de los swaps cobran importancia, ya que pueden tener un impacto significativo, normalmente negativo, en el resultado financiero. Sin embargo, algunos lectores probablemente sean entusiastas de la estrategia «Carry Trade», que originalmente se basaba en beneficiarse de los swaps positivos. MQL5 tiene varias propiedades de símbolo que proporcionan acceso a las cadenas de especificación que están asociadas a los swaps.

Identificador	Descripción
SYMBOL_SWAP_MODE	Modelo de cálculo de swap ENUM_SYMBOL_SWAP_MODE
SYMBOL_SWAP_ROLLOVER3DAYS	Día de la semana para swap triple ENUM_DAY_OF_WEEK
SYMBOL_SWAP_LONG	Tamaño de swap para una posición larga
SYMBOL_SWAP_SHORT	Tamaño de swap para una posición corta

La enumeración ENUM_SYMBOL_SWAP_MODE contiene elementos que especifican opciones para las unidades de medida y los principios de cálculo de los swaps. Al igual que SYMBOL_SWAP_ROLLOVER3DAYS, hacen referencia a las propiedades de enteros de ENUM_SYMBOL_INFO_INTEGER.

Los tamaños de swap se especifican directamente en las propiedades SYMBOL_SWAP_LONG y SYMBOL_SWAP_SHORT como parte de ENUM_SYMBOL_INFO_DOUBLE, es decir, de tipo *double*.

A continuación se indican los elementos de ENUM_SYMBOL_SWAP_MODE.

Identificador	Descripción
SYMBOL_SWAP_MODE_DISABLED	sin swaps
SYMBOL_SWAP_MODE_POINTS	puntos
SYMBOL_SWAP_MODE_CURRENCY_SYMBOL	la divisa base del símbolo
SYMBOL_SWAP_MODE_CURRENCY_MARGIN	divisa de margen de símbolo
SYMBOL_SWAP_MODE_CURRENCY_DEPOSIT	divisa de depósito
SYMBOL_SWAP_MODE_INTEREST_CURRENT	porcentaje anual del precio del instrumento en el momento del cálculo del swap
SYMBOL_SWAP_MODE_INTEREST_OPEN	porcentaje anual del precio de apertura de la posición del símbolo
SYMBOL_SWAP_MODE_REOPEN_CURRENT	puntos (con reapertura de la posición al precio de cierre)
SYMBOL_SWAP_MODE_REOPEN_BID	puntos (con reapertura de la posición al precio Bid del nuevo día). (en los parámetros SYMBOL_SWAP_LONG y SYMBOL_SWAP_SHORT)

Para las opciones SYMBOL_SWAP_MODE_INTEREST_CURRENT y SYMBOL_SWAP_MODE_INTEREST_OPEN, se asumen 360 días hábiles bancarios por año.

Para las opciones SYMBOL_SWAP_MODE_REOPEN_CURRENT y SYMBOL_SWAP_MODE_REOPEN_BID, la posición se cierra de manera forzosa al final del día de trading, y entonces su comportamiento es diferente.

Con SYMBOL_SWAP_MODE_REOPEN_CURRENT, la posición se reabre al día siguiente al precio de cierre de ayer +/- el número de puntos especificado. Con SYMBOL_SWAP_MODE_REOPEN_BID, la posición se

reabre al día siguiente al precio Bid actual +/- el número de puntos especificado. En ambos casos, el número de puntos está en los parámetros SYMBOL_SWAP_LONG y SYMBOL_SWAP_SHORT.

Vamos a comprobar el funcionamiento de las propiedades utilizando el script *SymbolFilterSwap.mq5*. En los parámetros de entrada, proporcionamos la elección del contexto de análisis: *Market Watch* o todos los símbolos en función de *UseMarketWatch*. Cuando el parámetro *ShowPerSymbolDetails* es *false*, calcularemos estadísticas, cuántas veces se utiliza en símbolos un modo u otro de ENUM_SYMBOL_SWAP_MODE. Cuando el parámetro *ShowPerSymbolDetails* es *true*, obtendremos un array de todos los símbolos con el modo especificado en *mode*, y ordenaremos el array en orden descendente de valores en los campos SYMBOL_SWAP_LONG y SYMBOL_SWAP_SHORT.

```
input bool UseMarketWatch = true;
input bool ShowPerSymbolDetails = false;
input ENUM_SYMBOL_SWAP_MODE Mode = SYMBOL_SWAP_MODE_POINTS;
```

Para los elementos del array de swaps combinados, describimos la estructura *SymbolSwap* con el nombre del símbolo y el valor del swap. La dirección del swap se indicará mediante un prefijo en el campo de nombre: «+» para los swaps de posiciones largas, «-» para los swaps de posiciones cortas.

```
struct SymbolSwap
{
   string name;
   double value;
};
```

Por tradición, describimos el objeto de filtro al principio de *OnStart*. No obstante, el siguiente código difiere significativamente en función del valor de la variable *ShowPerSymbolDetails*.

```
void OnStart()
{
   SymbolFilter f;                // filter object
   PrintFormat("===== Swap modes for %s symbols =====",
      (UseMarketWatch ? "Market Watch" : "all available"));

   if(ShowPerSymbolDetails)
   {
      // summary table of swaps of the selected Mode
      ...
   }
   else
   {
      // calculation of mode statistics
      ...
   }
}
```

Vamos a presentar en primer lugar la segunda rama. Aquí rellenamos arrays con nombres de símbolos utilizando los modos de filtro (*symbols*) y swap (*values*) que se toman de la propiedad SYMBOL_SWAP_MODE. Los valores resultantes se acumulan en un mapa de array *MapArray<ENUM_SYMBOL_SWAP_MODE,int> stats*.

```
// calculation of mode statistics
string symbols[];
long values[];
MapArray<ENUM_SYMBOL_SWAP_MODE,int> stats; // counters for each mode
// apply filter and collect mode values
f.select(UseMarketWatch, SYMBOL_SWAP_MODE, symbols, values);
const int n = ArraySize(symbols);
for(int i = 0; i < n; ++i)
{
   stats.inc((ENUM_SYMBOL_SWAP_MODE)values[i]);
}
...
```

A continuación mostramos las estadísticas recopiladas:

```
PrintFormat("Total symbols: %d", n);
Print("Stats per swap mode:");
stats.print();
Print("Legend: key=swap mode, value=count");
for(int i = 0; i < stats.getSize(); ++i)
{
   PrintFormat("%d -> %s", stats.getKey(i), EnumToString(stats.getKey(i)));
}
```

Para el caso de la construcción de una tabla con valores de swap, el algoritmo es el siguiente. Los swaps para posiciones largas y cortas se solicitan por separado, por lo que definimos arrays pareadas para nombres y valores que se juntarán en el conjunto de estructuras de array *swaps*.

```
// summary table of swaps of the selected Mode
string buyers[], sellers[];    // arrays for names
double longs[], shorts[];      // arrays for swap values
SymbolSwap swaps[];            // total array to print
```

Establezca la condición para el modo de swap seleccionado en el filtro. Esto es necesario para poder comparar y ordenar los elementos de un array.

```
f.let(SYMBOL_SWAP_MODE, Mode);
```

A continuación aplicamos el filtro dos veces para distintas propiedades (SYMBOL_SWAP_LONG, SYMBOL_SWAP_SHORT) y rellenamos distintos arrays con sus valores (*longs*, *shorts*). Dentro de cada llamada, los arrays se colocan en orden ascendente.

```
f.select(UseMarketWatch, SYMBOL_SWAP_LONG, buyers, longs, true);
f.select(UseMarketWatch, SYMBOL_SWAP_SHORT, sellers, shorts, true);
```

En teoría, los tamaños de los arrays deberían ser los mismos, ya que la condición de filtrado es la misma, pero para mayor claridad, vamos a asignar una variable para cada tamaño. Dado que cada símbolo aparecerá dos veces en la tabla resultante, para los lados largo y corto, proporcionamos un tamaño doble para el array *swaps*.

```
const int l = ArraySize(longs);
const int s = ArraySize(shorts);
const int n = ArrayResize(swaps, l + s); // should be l == s
PrintFormat("Total symbols with %s: %d", EnumToString(Mode), l);
```

A continuación, unimos los dos arrays *longs* y *shorts*, procesándolos en orden inverso, ya que necesitamos ordenar desde los valores positivos a los negativos.

```
if(n > 0)
{
   int i = l - 1, j = s - 1, k = 0;
   while(k < n)
   {
      const double swapLong = i >= 0 ? longs[i] : -DBL_MAX;
      const double swapShort = j >= 0 ? shorts[j] : -DBL_MAX;

      if(swapLong >= swapShort)
      {
         swaps[k].name = "+" + buyers[i];
         swaps[k].value = longs[i];
         --i;
         ++k;
      }
      else
      {
         swaps[k].name = "-" + sellers[j];
         swaps[k].value = shorts[j];
         --j;
         ++k;
      }
   }
   Print("Swaps per symbols (ordered):");
   ArrayPrint(swaps);
}
```

Es interesante ejecutar el script varias veces con diferentes configuraciones. Por ejemplo, por defecto, podemos obtener los siguientes resultados:

```
===== Swap modes for Market Watch symbols =====
Total symbols: 13
Stats per swap mode:
    [key] [value]
[0]     1      10
[1]     0       2
[2]     2       1
Legend: key=swap mode, value=count
1 -> SYMBOL_SWAP_MODE_POINTS
0 -> SYMBOL_SWAP_MODE_DISABLED
2 -> SYMBOL_SWAP_MODE_CURRENCY_SYMBOL
```

Estas estadísticas muestran que 10 símbolos tienen el modo swap SYMBOL_SWAP_MODE_POINTS, para dos los swaps están desactivados, SYMBOL_SWAP_MODE_DISABLED, y para uno está en la divisa base SYMBOL_SWAP_MODE_CURRENCY_SYMBOL.

Averigüemos qué tipo de símbolos tienen SYMBOL_SWAP_MODE_POINTS y descubramos sus swaps. Para ello, ajustaremos *ShowPerSymbolDetails* a *true* (el parámetro *mode* ya está ajustado a SYMBOL_SWAP_MODE_POINTS).

```
===== Swap modes for Market Watch symbols =====
Total symbols with SYMBOL_SWAP_MODE_POINTS: 10
Swaps per symbols (ordered):
        [name]   [value]
[ 0] "+AUDUSD"   6.30000
[ 1] "+NZDUSD"   2.80000
[ 2] "+USDCHF"   0.10000
[ 3] "+USDRUB"   0.00000
[ 4] "-USDRUB"   0.00000
[ 5] "+USDJPY"  -0.10000
[ 6] "+GBPUSD"  -0.20000
[ 7] "-USDCAD"  -0.40000
[ 8] "-USDJPY"  -0.60000
[ 9] "+EURUSD"  -0.70000
[10] "+USDCAD"  -0.80000
[11] "-EURUSD"  -1.00000
[12] "-USDCHF"  -1.00000
[13] "-GBPUSD"  -2.20000
[14] "+USDSEK"  -4.50000
[15] "-XAUUSD"  -4.60000
[16] "-USDSEK"  -4.90000
[17] "-NZDUSD"  -6.70000
[18] "+XAUUSD" -12.60000
[19] "-AUDUSD" -14.80000
```

Puede comparar los valores con las especificaciones de los símbolos.

Por último, cambiamos el modo a SYMBOL_SWAP_MODE_CURRENCY_SYMBOL. En nuestro caso, deberíamos obtener un símbolo, pero espaciado en dos líneas: con un más y con un menos en el nombre.

```
===== Swap modes for Market Watch symbols =====
Total symbols with SYMBOL_SWAP_MODE_CURRENCY_SYMBOL: 1
Swaps per symbols (ordered):
       [name]   [value]
[0] "-SP500m" -35.00000
[1] "+SP500m" -41.41000
```

Según la tabla, ambos swaps son negativos.

6.1.20 Información actual sobre el mercado (tick)

En la sección Obtener el último tick de un símbolo hemos visto ya la función *SymbolInfoTick*, que proporciona información completa sobre el último tick (evento de cambio de precio) en la forma de la estructura *MqlTick*. Si es necesario, el programa MQL puede solicitar por separado los valores de precios y volúmenes correspondientes a los campos de esta estructura. Todos ellos se indican mediante propiedades de distintos tipos que forman parte de las enumeraciones ENUM_SYMBOL_INFO_INTEGER y ENUM_SYMBOL_INFO_DOUBLE.

Identificador	Descripción	Tipo de propiedad
SYMBOL_TIME	Hora de la última cotización	datetime
SYMBOL_BID	Precio de venta (bid); la mejor oferta de venta	double
SYMBOL_ASK	Precio de compra (ask); la mejor oferta de compra	double
SYMBOL_LAST	Último; el precio de la última transacción	double
SYMBOL_VOLUME	El volumen de la última transacción	long
SYMBOL_TIME_MSC	La hora de la última cotización en milisegundos desde 01/01/1970	long
SYMBOL_VOLUME_REAL	El volumen de la última transacción con mayor precisión	double

Observe que el código de las dos propiedades relacionadas con el volumen, SYMBOL_VOLUME y SYMBOL_VOLUME_REAL, es el mismo en ambas enumeraciones. Este es el único caso en el que los ID de elementos de distintas enumeraciones se solapan. La cuestión es que devuelven esencialmente la misma propiedad tick, pero con diferente precisión de representación.

A diferencia de una estructura, las propiedades no proporcionan un análogo al campo *uint flags*, que indica qué tipo de cambios en el mercado provocaron la generación del tick. Este campo sólo tiene sentido dentro de una estructura.

Vamos a intentar solicitar las propiedades de los ticks por separado y comparémoslas con el resultado de la llamada a *SymbolInfoTick*. En un mercado rápido existe la posibilidad de que los resultados difieran. Puede producirse un nuevo tick (o incluso varios) entre llamadas a funciones.

```
void OnStart()
{
   PRTF(TimeToString(SymbolInfoInteger(_Symbol, SYMBOL_TIME), TIME_DATE | TIME_SECOND
   PRTF(SymbolInfoDouble(_Symbol, SYMBOL_BID));
   PRTF(SymbolInfoDouble(_Symbol, SYMBOL_ASK));
   PRTF(SymbolInfoDouble(_Symbol, SYMBOL_LAST));
   PRTF(SymbolInfoInteger(_Symbol, SYMBOL_VOLUME));
   PRTF(SymbolInfoInteger(_Symbol, SYMBOL_TIME_MSC));
   PRTF(SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_REAL));

   MqlTick tick[1];
   SymbolInfoTick(_Symbol, tick[0]);
   ArrayPrint(tick);
}
```

Es fácil comprobar que, en un caso concreto, la información coincidía.

```
TimeToString(SymbolInfoInteger(_Symbol,SYMBOL_TIME),TIME_DATE|TIME_SECONDS)
   =2022.01.25 13:52:51 / ok
SymbolInfoDouble(_Symbol,SYMBOL_BID)=1838.44 / ok
SymbolInfoDouble(_Symbol,SYMBOL_ASK)=1838.49 / ok
SymbolInfoDouble(_Symbol,SYMBOL_LAST)=0.0 / ok
SymbolInfoInteger(_Symbol,SYMBOL_VOLUME)=0 / ok
SymbolInfoInteger(_Symbol,SYMBOL_TIME_MSC)=1643118771166 / ok
SymbolInfoDouble(_Symbol,SYMBOL_VOLUME_REAL)=0.0 / ok
                 [time]   [bid]   [ask] [last] [volume]    [time_msc] [flags] [volume
[0] 2022.01.25 13:52:51 1838.44 1838.49   0.00        0 1643118771166       6
```

6.1.21 Propiedades descriptivas de los símbolos

La plataforma proporciona un grupo de propiedades de texto para los programas MQL que describen características cualitativas importantes. Por ejemplo, al desarrollar indicadores o estrategias de trading basados en una cesta de instrumentos financieros, puede ser necesario seleccionar símbolos por país de origen, sector económico o nombre del activo subyacente (si el instrumento es un derivado).

Identificador	Descripción
SYMBOL_BASIS	Nombre del activo subyacente del derivado
SYMBOL_CATEGORY	Nombre de la categoría a la que pertenece el instrumento financiero.
SYMBOL_COUNTRY	País al que está asignado el instrumento financiero.
SYMBOL_SECTOR_NAME	Sector de la economía al que pertenece el instrumento financiero.
SYMBOL_INDUSTRY_NAME	La rama de la economía o el tipo de sector a la que pertenece el instrumento financiero.
SYMBOL_BANK	Fuente de cotización actual
SYMBOL_DESCRIPTION	Cadena descriptiva del símbolo
SYMBOL_EXCHANGE	Nombre de la bolsa o mercado donde se negocia el símbolo.
SYMBOL_ISIN	Un código alfanumérico único de 12 dígitos en el sistema de códigos internacionales de identificación de valores ISIN (International Securities Identification Number)
SYMBOL_PAGE	Dirección de la página de Internet con información sobre el símbolo
SYMBOL_PATH	Ruta en el árbol de símbolos

Otro caso en el que el programa puede aplicar el análisis de estas propiedades se produce cuando se busca un tipo de conversión de una moneda a otra. Ya sabemos cómo encontrar un símbolo con la combinación correcta de divisa base y de cotización, pero la dificultad estriba en que puede haber varios símbolos de este tipo. La lectura de propiedades como SYMBOL_SECTOR_NAME (debe buscar «Divisa» o un sinónimo; consulte las especificaciones de su bróker) o SYMBOL_PATH puede ayudar en estos casos.

SYMBOL_PATH contiene toda la jerarquía de carpetas del directorio de símbolos que contienen el símbolo específico: los nombres de las carpetas están separados por barras invertidas ('\\\') del mismo modo que en el sistema de archivos. El último elemento de la ruta es el nombre del propio símbolo.

Algunas propiedades de cadena tienen homólogos enteros. En particular, en lugar de SYMBOL_SECTOR_NAME, puede utilizar la propiedad SYMBOL_SECTOR, que devuelve un miembro de la enumeración ENUM_SYMBOL_SECTOR con todos los sectores admitidos. Por analogía, para SYMBOL_INDUSTRY_NAME existe una propiedad similar SYMBOL_INDUSTRY con el tipo de enumeración ENUM_SYMBOL_INDUSTRY.

Si es necesario, un programa MQL puede incluso encontrar el color de fondo utilizado al mostrar un símbolo en la Observación de Mercado simplemente leyendo la propiedad SYMBOL_BACKGROUND_COLOR. Esto permitirá a aquellos programas que creen su propia interfaz en el gráfico mediante objetos gráficos (cuadros de diálogo, listas, etc.) que la unifiquen con los controles nativos del terminal.

Veamos el script de ejemplo *SymbolFilterDescription.mq5*, que genera cuatro propiedades de texto predefinidas para los símbolos *Observación de Mercado*. La primera de ellas es SYMBOL_DESCRIPTION (no confundir con el nombre del propio símbolo), y conforme a ella se ordenará la lista resultante. Las otras tres son puramente de referencia: SYMBOL_SECTOR_NAME, SYMBOL_COUNTRY, SYMBOL_PATH. Todos los valores se rellenan de forma específica para cada bróker (puede haber discrepancias para el mismo ticker).

No lo hemos mencionado, pero nuestra clase *SymbolFilter* implementa una sobrecarga especial del método *equal* para comparar cadenas. Permite buscar la aparición de una subcadena con un patrón en el que el carácter comodín '*' representa 0 o más caracteres arbitrarios. Por ejemplo, «*ian*» buscará todos los caracteres que contengan la subcadena «ian» (en cualquier lugar), e «*Index» sólo encontrará las cadenas que terminen en «Index».

Esta función se asemeja a una búsqueda por subcadenas en el cuadro de diálogo *Symbols* disponible para los usuarios. Sin embargo, no es necesario especificar un carácter comodín, ya que siempre se busca una subcadena. En el algoritmo que puede encontrarse en los códigos fuente (*SymbolFilter.mqh*), dejamos la posibilidad de buscar una coincidencia completa (no hay caracteres «*») o una subcadena (hay al menos un asterisco).

La comparación distingue entre mayúsculas y minúsculas. En caso necesario, es fácil adaptar el código para compararlo sin distinguir entre minúsculas y mayúsculas.

Dada la nueva característica, vamos a definir una variable de entrada para la cadena de búsqueda en la descripción de los símbolos. Si la variable está vacía, se mostrarán todos los símbolos de la ventana *Observación de Mercado*.

```
input string SearchPattern = "";
```

Además, todo es como de costumbre.

```
void OnStart()
{
   SymbolFilter f;                        // filter object
   string symbols[];                      // array of names
   string text[][4];                      // array of vectors with data

   // properties to read
   ENUM_SYMBOL_INFO_STRING fields[] =
   {
      SYMBOL_DESCRIPTION,
      SYMBOL_SECTOR_NAME,
      SYMBOL_COUNTRY,
      SYMBOL_PATH
   };

   if(SearchPattern != "")
   {
      f.let(SYMBOL_DESCRIPTION, SearchPattern);
   }

   // apply the filter and get arrays sorted by description
   f.select(true, fields, symbols, text, true);

   const int n = ArraySize(symbols);
   PrintFormat("===== Text fields for symbols (%d) =====", n);
   for(int i = 0; i < n; ++i)
   {
      Print(symbols[i] + ":");
      ArrayPrint(text, 0, NULL, i, 1, 0);
   }
}
```

He aquí una posible versión de la lista (con abreviaturas):

```
===== Text fields for symbols (16) =====
AUDUSD:
"Australian Dollar vs US Dollar" "Currency"   ""   "Forex\AUDUSD"
EURUSD:
"Euro vs US Dollar" "Currency"   ""   "Forex\EURUSD"
UK100:
"FTSE 100 Index" "Undefined"   ""   "Indexes\UK100"
XAUUSD:
"Gold vs US Dollar" "Commodities"   ""   "Metals\XAUUSD"
JAGG:
    "JPMorgan U.S. Aggregate Bond ETF"  "Financial"
    "USA"   "ETF\United States\NYSE\JPMorgan\JAGG"
NZDUSD:
"New Zealand Dollar vs US Dollar" "Currency"   ""   "Forex\NZDUSD"
GBPUSD:
"Pound Sterling vs US Dollar" "Currency"   ""   "Forex\GBPUSD"
SP500m:
"Standard & Poor's 500" "Undefined"   ""   "Indexes\SP500m"
FIHD:
    "UBS AG FI Enhanced Global High Yield ETN" "Financial"
    "USA"   "ETF\United States\NYSE\UBS\FIHD"
...
```

Si introducimos la cadena de búsqueda «*ian*» en la variable de entrada *SearchPattern*, obtendremos el siguiente resultado.

```
===== Text fields for symbols (3) =====
AUDUSD:
"Australian Dollar vs US Dollar" "Currency"   ""   "Forex\AUDUSD"
USDCAD:
"US Dollar vs Canadian Dollar" "Currency"   ""   "Forex\USDCAD"
USDRUB:
"US Dollar vs Russian Ruble" "Currency"   ""   "Forex\USDRUB"
```

6.1.22 Profundidad de Mercado

Cuando se trata de instrumentos bursátiles, MetaTrader 5 le permite obtener no sólo información sobre precios y volúmenes empaquetada en ticks, sino también la Profundidad de Mercado (libro de órdenes, precios de nivel II), es decir, la distribución de los volúmenes de las órdenes de compra y venta colocadas en los niveles más próximos al precio actual. Una de las propiedades enteras del símbolo SYMBOL_TICKS_BOOKDEPTH contiene el número máximo de niveles mostrados en la Profundidad de Mercado. Esta cantidad está permitida para cada una de las partes, es decir, el tamaño total del libro de órdenes puede ser dos veces mayor (y esto no tiene en cuenta los niveles de precios con volúmenes cero que no se emiten).

Dependiendo de la situación del mercado, el tamaño real del libro de órdenes transmitido puede ser inferior al indicado en esta propiedad. Para los instrumentos no bursátiles, esta propiedad suele ser igual a 0, aunque algunos brókers pueden difundir el libro de órdenes para los símbolos de Forex, limitado únicamente por las órdenes de sus clientes.

El propio libro de órdenes y las notificaciones sobre su actualización deben ser solicitados por el programa MQL interesado mediante una API especial, de la que hablaremos en el capítulo siguiente.

Cabe señalar que, debido a las características arquitectónicas de la plataforma, esta propiedad no está directamente relacionada con la traducción del libro de órdenes, es decir, es sólo un campo de especificación rellenado por el bróker. En otras palabras: un valor distinto de cero de la propiedad no significa que el libro de órdenes vaya a llegar necesariamente al terminal en un mercado abierto. Esto depende de otros ajustes del servidor y de si tiene una conexión activa con el proveedor de datos.

Vamos a intentar obtener estadísticas sobre la profundidad del mercado para todos los símbolos o los símbolos seleccionados utilizando el script *SymbolFilterBookDepth.mq5*.

```
input bool UseMarketWatch = false;
input int ShowSymbolsWithDepth = -1;
```

El parámetro *ShowSymbolsWithDepth*, que es igual a -1 por defecto, indica que se recopilen estadísticas sobre diferentes configuraciones de Profundidad de Mercado entre todos los símbolos. Si ajusta el parámetro a un valor diferente, el programa intentará encontrar todos los símbolos con la profundidad de libro de órdenes especificada.

```
void OnStart()
{
   SymbolFilter f;                // filter object
   string symbols[];              // array for symbol names
   long depths[];                 // array of property values
   MapArray<long,int> stats;      // counters of occurrences of each depth

   if(ShowSymbolsWithDepth > -1)
   {
      f.let(SYMBOL_TICKS_BOOKDEPTH, ShowSymbolsWithDepth);
   }

   // apply filter and fill arrays
   f.select(UseMarketWatch, SYMBOL_TICKS_BOOKDEPTH, symbols, depths, true);
   const int n = ArraySize(symbols);

   PrintFormat("===== Book depths for %s symbols %s=====",
      (UseMarketWatch ? "Market Watch" : "all available"),
      (ShowSymbolsWithDepth > -1 ? "(filtered by depth="
      + (string)ShowSymbolsWithDepth + ") " : ""));
   PrintFormat("Total symbols: %d", n);
   ...
```

Si se da una profundidad específica, simplemente se obtiene un array de símbolos (todos satisfacen la condición del filtro) y se sale.

```
   if(ShowSymbolsWithDepth > -1)
   {
      ArrayPrint(symbols);
      return;
   }
   ...
```

En caso contrario, calculamos las estadísticas y las mostramos.

```
      for(int i = 0; i < n; ++i)
      {
         stats.inc(depths[i]);
      }

      Print("Stats per depth:");
      stats.print();
      Print("Legend: key=depth, value=count");
   }
```

Con la configuración por defecto podemos obtener la siguiente imagen:

```
===== Book depths for all available symbols =====
Total symbols: 52357
Stats per depth:
    [key] [value]
[0]     0   52244
[1]     5       3
[2]    10      67
[3]    16       5
[4]    20      13
[5]    32      25
Legend: key=depth, value=count
```

Si establecemos *ShowSymbolsWithDepth* en uno de los valores detectados, por ejemplo, 32, obtendremos una lista de símbolos con esta profundidad de libro de órdenes.

```
===== Book depths for all available symbols (filtered by depth=32) =====
Total symbols: 25
[ 0] "USDCNH" "USDZAR" "USDHUF" "USDPLN" "EURHUF" "EURNOK" "EURPLN" "EURSEK" "EURZAR"
[13] "NZDCAD" "NZDCHF" "USDMXN" "EURMXN" "GBPMXN" "CADMXN" "CHFMXN" "MXNJPY" "NZDMXN"
```

6.1.23 Propiedades de símbolos personalizados

En la introducción de este capítulo hemos mencionado los símbolos personalizados. Estos son los símbolos con las cotizaciones creados directamente en el terminal a la orden del usuario o mediante programación.

Los símbolos personalizados pueden utilizarse, por ejemplo, para crear un instrumento sintético basado en una fórmula que incluya otros símbolos de Observación de Mercado. Esto está a disposición del usuario directamente en la interfaz del terminal.

Un programa MQL puede implementar escenarios más complejos en MQL5, como fusionar diferentes instrumentos para diferentes periodos, generar series según una distribución aleatoria determinada o recibir datos (cotizaciones, barras o ticks) de fuentes externas.

Para poder distinguir un símbolo estándar de un símbolo personalizado en los algoritmos, MQL5 proporciona la propiedad SYMBOL_CUSTOM, que es una señal lógica de que un símbolo es personalizado.

Si el símbolo tiene una fórmula, ésta está disponible a través de la propiedad de cadena SYMBOL_FORMULA. En las fórmulas, como sabe, puede utilizar los nombres de otros símbolos, así como operadores y funciones matemáticas. He aquí algunos ejemplos:

- Símbolo sintético: «@ESU19»/EURCAD
- Spread de calendario: «Si-9.13»-«Si-6.13»
- Índice del euro: 34.38805726 * pow(EURUSD,0.3155) * pow(EURGBP,0.3056) * pow(EURJPY,0.1891) * pow(EURCHF,0.1113) * pow(EURSEK,0.0785)

Especificar una fórmula es conveniente para el usuario, pero normalmente no se utiliza desde programas MQL ya que pueden calcular fórmulas directamente en el código, con funciones no estándar y con más control, en concreto, en cada tick y no en un temporizador 1 vez cada 100 ms.

Vamos a comprobar el trabajo con las propiedades en el script *SymbolFilterCustom.mq5*: registra todos los símbolos personalizados y sus fórmulas (si las hay).

```
input bool UseMarketWatch = false;

void OnStart()
{
   SymbolFilter f;                  // filter object
   string symbols[];                // array for symbol names
   string formulae[];               // array for formulas

 // apply filter and fill arrays
   f.let(SYMBOL_CUSTOM, true)
   .select(UseMarketWatch, SYMBOL_FORMULA, symbols, formulae);
   const int n = ArraySize(symbols);

   PrintFormat("===== %s custom symbols =====",
      (UseMarketWatch ? "Market Watch" : "All available"));
   PrintFormat("Total symbols: %d", n);

   for(int i = 0; i < n; ++i)
   {
      Print(symbols[i], " ", formulae[i]);
   }
}
```

A continuación se muestra el resultado con el único carácter personalizado encontrado.

```
===== All available custom symbols =====
Total symbols: 1
synthEURUSD SP500m/UK100
```

6.1.24 Propiedades específicas (bolsa, derivados, bonos)

En esta sección final del capítulo revisaremos brevemente otras propiedades de los símbolos que están fuera del ámbito del libro pero que pueden ser útiles para implementar estrategias avanzadas de trading. Encontrará información detallada sobre estas propiedades en la Documentación MQL5.

Como sabe, MetaTrader 5 le permite operar con instrumentos del mercado de derivados, incluyendo opciones, futuros y bonos. Esto también se refleja en la interfaz del software. La API de MQL5 proporciona muchas propiedades específicas de los símbolos relacionadas con las categorías de instrumentos mencionadas.

En concreto, en el caso de las opciones, se trata del período de circulación (la fecha de inicio SYMBOL_START_TIME y la fecha de finalización SYMBOL_EXPIRATION_TIME del trading); el precio de ejercicio (SYMBOL_OPTION_STRIKE); el derecho a comprar o vender (SYMBOL_OPTION_RIGHT, Call/Put); el tipo europeo o americano (SYMBOL_OPTION_MODE) en función de la posibilidad de ejercicio anticipado; la variación diaria de los precios de cierre (SYMBOL_PRICE_CHANGE), y la volatilidad (SYMBOL_PRICE_VOLATILITY), así como los coeficientes estimados (los griegos) que caracterizan la dinámica del comportamiento de los precios.

En el caso de los bonos, resultan especialmente interesantes los ingresos acumulados por cupones (SYMBOL_TRADE_ACCRUED_INTEREST), el valor nominal (SYMBOL_TRADE_FACE_VALUE) y el coeficiente de liquidez (SYMBOL_TRADE_LIQUIDITY_RATE).

Para futuros, el interés abierto (SYMBOL_SESSION_INTEREST) y los volúmenes totales de órdenes por compra (SYMBOL_SESSION_BUY_ORDERS_VOLUME) y venta (SYMBOL_SESSION_SELL_ORDERS_VOLUME), el precio de compensación al cierre de la sesión de trading (SYMBOL_SESSION_PRICE_SETTLEMENT).

Aparte de los datos actuales del mercado que componen un tick, MQL5 permite conocer su rango diario: los valores máximo y mínimo de cada uno de los campos del tick. Por ejemplo, SYMBOL_BIDHIGH es el máximo *Bid* por día, y SYMBOL_BIDLOW es el mínimo. Tenga en cuenta que las propiedades SYMBOL_VOLUMEHIGH, SYMBOL_VOLUMELOW (del tipo *long*) en realidad duplican, sólo que con menos precisión, los volúmenes en SYMBOL_VOLUMEHIGH_REAL y SYMBOL_VOLUMELOW_REAL (*double*).

La información sobre los volúmenes y precios *Last* sólo está disponible, por regla general, para los símbolos bursátiles.

Tenga en cuenta que llenar las propiedades depende de la configuración del servidor implementado por el bróker.

6.2 Profundidad de Mercado

Además de varios tipos de datos actualizados sobre precios de mercado (*Ask/Bid/Last*) y los últimos volúmenes negociados que se reciben en el terminal en forma de ticks, MetaTrader 5 admite también Profundidad de Mercado (libro de órdenes), que es un array de registros sobre los volúmenes de órdenes de compra y venta colocadas en torno al precio actual de mercado. Los volúmenes se agregan a varios niveles por encima y por debajo del precio actual, con el menor incremento de movimiento de precios según la especificación del símbolo. Como hemos visto, el tamaño máximo del libro de órdenes (número de niveles de precios) se fija en la propiedad del símbolo SYMBOL_TICKS_BOOKDEPTH.

Los usuarios de terminales conocen la herramienta Profundidad de Mercado de la interfaz y sus principios de funcionamiento. Si necesita más información, consulte la documentación.

El libro de órdenes contiene información ampliada sobre el mercado, lo que suele denominarse «profundidad de mercado», y conocerlo le permite crear sistemas de negociación más sofisticados.

De hecho, la información sobre un tick es sólo una pequeña porción del libro de órdenes. En un sentido algo simplificado, un tick es un libro de órdenes de 2 niveles con un precio *Ask* más cercano (oferta disponible) y un precio *Bid* más cercano (demanda disponible). Además, los ticks no proporcionan volúmenes de órdenes a estos precios.

Los cambios en la Profundidad de Mercado pueden producirse con mucha más frecuencia que los ticks, ya que afectan no sólo a la reacción a transacciones concluidas, sino también a los cambios en el volumen de órdenes Limit pendientes en la Profundidad de Mercado.

Normalmente, los proveedores de datos para el libro de órdenes y las cotizaciones (ticks, transacciones) son instancias diferentes, y los eventos de ticks (*OnTick* en Asesores Expertos u *OnCalculate* en indicadores) no coinciden con los eventos de Profundidad de Mercado. Ambos hilos llegan de forma asíncrona y en paralelo, pero al final terminan en la cola de eventos de un programa MQL.

Es importante señalar que, por regla general, existe un libro de órdenes para los instrumentos bursátiles, pero hay excepciones tanto en un sentido como en el otro:

- La Profundidad de Mercado puede faltar por una razón u otra para un instrumento bursátil;
- La Profundidad de Mercado puede ser proporcionada por un bróker para un instrumento OTC basándose en la información que ha recopilado sobre las órdenes de sus clientes.

En MQL5, los datos de Profundidad de Mercado están disponibles para los Asesores Expertos y los indicadores. Utilizando funciones especiales (*MarketBookAdd*, *MarketBookRelease*), los programas pueden activar o desactivar su suscripción para recibir notificaciones sobre los cambios de Profundidad de Mercado en la plataforma. Para recibir las notificaciones, el programa debe definir la función del manejador de eventos *OnBookEvent* en su código. Tras recibir una notificación, los datos del libro de órdenes pueden leerse mediante la función *MarketBookGet*.

> El terminal mantiene el historial de cotizaciones y ticks, pero no de los datos de Profundidad de Mercado. En concreto, el usuario o un programa MQL puede descargar el historial en la retrospectiva requerida (si el bróker lo tiene) y probar en él Asesores Expertos e indicadores.
>
> En cambio, la Profundidad de Mercado sólo se emite en línea y no está disponible en el probador. Un bróker no tiene un archivo de datos de Profundidad de Mercado en el servidor. Para emular el comportamiento del libro de órdenes en el probador, debe recopilar el historial de Profundidad de Mercado en línea y luego leerlo desde el programa MQL que se ejecuta en el probador. Puede encontrar productos ya preparados en el Mercado de MQL5.

6.2.1 Gestionar suscripciones a eventos de Profundidad de Mercado

El terminal recibe la información de Profundidad de Mercado por suscripción: un programa MQL debe expresar su intención de recibir eventos de Profundidad de Mercado (libro de órdenes) o, por el contrario, finalizar su suscripción llamando a las funciones apropiadas, *MarketBookAdd* y *MarketBookRelease*.

La función *MarketBookAdd* se suscribe para recibir notificaciones sobre cambios en el libro de órdenes del instrumento especificado. De este modo, puede suscribirse a libros de órdenes para muchos instrumentos, y no sólo para el instrumento de trabajo del gráfico actual.

bool MarketBookAdd(const string symbol)

Normalmente, se llama a esta función desde *OnInit* o en el constructor de clase de un objeto de larga duración. Las notificaciones sobre la modificación del libro de órdenes se envían al programa en forma de eventos *OnBookEvent*, por lo que, para procesarlos, el programa debe disponer de una función de manejador del mismo nombre.

Si el símbolo especificado no estaba seleccionado en Observación de Mercado antes de llamar a la función, se añadirá a la ventana automáticamente.

La función *MarketBookRelease* cancela la suscripción a notificaciones sobre cambios en el libro de órdenes especificado.

bool MarketBookRelease(const string symbol)

Por regla general, esta función debe llamarse desde *OnDeinit* o desde el destructor de clase de un objeto de larga duración.

Ambas funciones devuelven el valor *true* en caso de éxito y *false* en caso contrario.

Para todas las aplicaciones que se ejecutan en el mismo gráfico se mantienen contadores de suscripción separados por símbolos. En otras palabras: puede haber varias suscripciones a distintos símbolos en el gráfico, y cada una de ellas tiene su propio contador.

La suscripción o la anulación de la suscripción mediante una única llamada a cualquiera de las funciones modifica el contador de suscripciones sólo para un símbolo concreto, en un gráfico concreto en el que se esté ejecutando el programa. Esto significa que dos gráficos pueden tener suscripciones a *OnBookEvent* eventos del mismo símbolo, pero con diferentes valores de contadores de suscripción.

El valor inicial del contador de suscripciones es cero. En cada llamada de *MarketBookAdd*, el contador de suscripciones para el símbolo especificado en el gráfico dado se incrementa en 1 (el símbolo del gráfico y el símbolo en *MarketBookAdd* no tienen por qué coincidir). Al llamar a *MarketBookRelease*, el contador de suscripciones al símbolo especificado dentro del gráfico disminuye en 1.

OnBookEvent para cualquier símbolo dentro del gráfico se generan siempre que el contador de suscripciones para este símbolo sea mayor que cero. Por lo tanto, es importante que todo programa MQL que contenga llamadas a *MarketBookAdd*, al finalizar su trabajo, se dé de baja correctamente de la recepción de eventos para cada símbolo utilizando *MarketBookRelease*. Para ello, debe asegurarse de que el número de llamadas *MarketBookAdd* y *MarketBookRelease* coinciden. MQL5 no permite averiguar el valor del contador.

El primer ejemplo es un simple indicador *MarketBookAddRelease.mq5* sin búfer que habilita una suscripción al libro de órdenes en el momento de su lanzamiento y la deshabilita cuando se descarga. En el parámetro de entrada *WorkSymbol* puede especificar un símbolo para suscribirse. Si se deja vacío (valor por defecto), la suscripción se iniciará para el símbolo de trabajo del gráfico actual.

```
input string WorkSymbol = ""; // WorkSymbol (empty means current chart symbol)

const string _WorkSymbol = StringLen(WorkSymbol) == 0 ? _Symbol : WorkSymbol;
string symbols[];

void OnInit()
{
   const int n = StringSplit(_WorkSymbol, ',', symbols);
   for(int i = 0; i < n; ++i)
   {
      if(!PRTF(MarketBookAdd(symbols[i])))
         PrintFormat("MarketBookAdd(%s) failed", symbols[i]);
   }
}

int OnCalculate(const int rates_total, const int prev_calculated, const int, const do
{
   return rates_total;
}

void OnDeinit(const int)
{
   for(int i = 0; i < ArraySize(symbols); ++i)
   {
      if(!PRTF(MarketBookRelease(symbols[i])))
         PrintFormat("MarketBookRelease(%s) failed", symbols[i]);
   }
}
```

Como característica adicional se permite especificar varios instrumentos separados por comas. En este caso, se solicitará la suscripción a todos.

Cuando se lanza el indicador aparece en el registro una señal de éxito de la suscripción o un código de error. A continuación, el indicador intenta darse de baja de los eventos en el manejador *OnDeinit*.

Con la configuración por defecto, en el gráfico con el símbolo para el que está disponible el libro de órdenes obtendremos las siguientes entradas en el registro:

```
MarketBookAdd(symbols[i])=true / ok
MarketBookRelease(symbols[i])=true / ok
```

Si ponemos el indicador en un gráfico con un símbolo sin el libro de órdenes, veremos códigos de error.

```
MarketBookAdd(symbols[i])=false / BOOKS_CANNOT_ADD(4901)
MarketBookAdd(XPDUSD) failed
MarketBookRelease(symbols[i])=false / BOOKS_CANNOT_DELETE(4902)
MarketBookRelease(XPDUSD) failed
```

Puede experimentar especificando en el parámetro de entrada *WorkSymbol* los caracteres existentes o los que faltan. Analizaremos el caso de la suscripción a libros de órdenes de varios símbolos en la siguiente sección.

6.2. Recibir eventos sobre cambios en la Profundidad de Mercado

El evento *OnBookEvent* es generado por el terminal cuando cambia el estado del libro de órdenes. El evento es procesado por la función *OnBookEvent* definida en el código fuente. Para que el terminal comience a enviar notificaciones *OnBookEvent* al programa MQL para un símbolo específico, primero debe suscribirse para recibirlas utilizando la función *MarketBookAdd*.

Para dejar de recibir el evento *OnBookEvent* de un símbolo, llame a la función *MarketBookRelease*.

El evento *OnBookEvent* es de difusión, lo que significa que es suficiente para un programa MQL en el gráfico para suscribirse a eventos *OnBookEvent*, y todos los demás programas en el mismo gráfico también comenzarán a recibir los eventos siempre que tengan el manejador *OnBookEvent* en el código. Por lo tanto, es necesario analizar el nombre del símbolo, que se pasa al manejador como parámetro.

El prototipo del manejador *OnBookEvent* es el siguiente:

```
void OnBookEvent(const string &symbol)
```

OnBookEvent se ponen en cola incluso si el procesamiento del evento *OnBookEvent* anterior aún no ha finalizado.

Es importante que los eventos *OnBookEvent* sean sólo notificaciones y no proporcionen el estado del libro de órdenes. Para obtener los datos de Profundidad de Mercado, llame a la función *MarketBookGet*.

Hay que tener en cuenta, sin embargo, que la llamada a *MarketBookGet*, aunque se haga directamente desde el manejador *OnBookEvent*, recibirá el estado actual del libro de órdenes en el momento en que se llame a *MarketBookGet*, que no tiene por qué coincidir con el estado del libro de órdenes que desencadenó el envío del evento *OnBookEvent*. Esto puede ocurrir cuando llega al terminal una secuencia de cambios muy rápidos en el libro de órdenes.

En este sentido, para obtener la cronología más completa de los cambios de Profundidad de Mercado, necesitamos escribir una implementación de *OnBookEvent* y priorizar la optimización por la velocidad de ejecución.

Al mismo tiempo, no hay forma garantizada de obtener todos los estados únicos de Profundidad de Mercado en MQL5.

> Si su programa empezó a recibir notificaciones correctamente y luego desaparecieron cuando se abrió el mercado (y los ticks siguen llegando), esto puede indicar problemas en la suscripción. En concreto, otro programa MQL mal diseñado podría darse de baja más veces de las necesarias. En tales casos, se recomienda volver a suscribirse con una nueva llamada a *MarketBookAdd* tras un tiempo de espera predefinido (por ejemplo, varias decenas de segundos o un minuto).

Un ejemplo de indicador *MarketBookEvent.mq5* sin búfer permite realizar un seguimiento de la llegada de eventos *OnBookEvent* e imprime el nombre del símbolo y la hora actual (contador de milisegundos del sistema) en un comentario. Para mayor claridad, utilizamos la función de comentario multilínea del archivo *Comments.mqh*, sección Visualización de mensajes en la ventana de gráficos.

Curiosamente, si deja vacío el parámetro de entrada *WorkSymbol* (valor por defecto), el propio indicador no iniciará una suscripción al libro de órdenes, pero podrá interceptar los mensajes solicitados por otros programas MQL en el mismo gráfico. Vamos a comprobarlo.

```
#include <MQL5Book/Comments.mqh>

input string WorkSymbol = ""; // WorkSymbol (if empty, intercept events initiated by

void OnInit()
{
   if(StringLen(WorkSymbol))
   {
      PRTF(MarketBookAdd(WorkSymbol));
   }
   else
   {
      Print("Start listening to OnBookEvent initiated by other programs");
   }
}

void OnBookEvent(const string &symbol)
{
   ChronoComment(symbol + " " + (string)GetTickCount());
}

void OnDeinit(const int)
{
   Comment("");
   if(StringLen(WorkSymbol))
   {
      PRTF(MarketBookRelease(WorkSymbol));
   }
}
```

Vamos a ejecutar *MarketBookEvent* con la configuración por defecto (sin suscripción propia) y luego añadir el indicador *MarketBookAddRelease* de la sección anterior, y especificar para ello una lista de varios símbolos con libros de órdenes disponibles (en el ejemplo siguiente, se trata de «XAUUSD,BTCUSD,USDCNH»). No importa en qué gráfico ejecutar los indicadores: puede ser un símbolo completamente diferente, como EURUSD.

Inmediatamente después de lanzar *MarketBookEvent*, el gráfico estará vacío (sin comentarios) porque aún no hay suscripciones. Una vez que se inicia *MarketBookAddRelease* (deben aparecer tres líneas en el registro con el estado de una suscripción correcta igual a *true*), los nombres de los símbolos comenzarán a aparecer en los comentarios alternativamente a medida que se actualicen sus libros de órdenes (aún no hemos aprendido a leer el libro de órdenes; esto se abordará en la próxima sección).

Este es su aspecto en la pantalla:

Notificaciones sobre cambios en los libros de órdenes de símbolos de «terceros»

Si ahora eliminamos el indicador *MarketBookAddRelease*, cancelará sus suscripciones y el comentario dejará de actualizarse. La eliminación posterior de *MarketBookEvent* borrará el comentario.

Tenga en cuenta que transcurre cierto tiempo (uno o dos segundos) entre la solicitud de baja y el momento en que los eventos de Profundidad de Mercado dejan realmente de actualizar el comentario.

Puede ejecutar el indicador *MarketBookEvent* solo en el gráfico, especificando algún símbolo en su parámetro *WorkSymbol* para asegurarse de que las notificaciones funcionan dentro de la misma aplicación. *MarketBookAddRelease* se utilizaba anteriormente sólo para demostrar la naturaleza de difusión de las notificaciones. En otras palabras: habilitar una suscripción a los cambios en el libro de órdenes en un programa sí afecta a la recepción de notificaciones en otro.

6.2.3 Leer los datos actuales de Profundidad de Mercado

Una vez ejecutado con éxito la función *MarketBookAdd*, un programa MQL puede consultar los estados del libro de órdenes utilizando la función *MarketBookGet* a la llegada de eventos OnBookEvent. La función *MarketBookGet* rellena el array *MqlBookInfo* de estructuras pasadas por referencia con los valores de Profundidad de Mercado del símbolo especificado.

bool MarketBookGet(string symbol, MqlBookInfo &book[])

Para el array receptor puede preasignar memoria para un número suficiente de registros. Si el array dinámico tiene un tamaño cero o insuficiente, el propio terminal le asignará memoria.

La función devuelve un indicador de éxito (*true*) o de error (*false*).

MarketBookGet suele utilizarse directamente en el código de manejador *OnBookEvent* o en funciones llamadas desde él.

En la estructura *MqlBookInfo* se almacena un registro independiente sobre el nivel de precios de Profundidad de Mercado.

```
struct MqlBookInfo
{
   ENUM_BOOK_TYPE type;          // request type
   double         price;         // price
   long           volume;        // volume
   double         volume_real;   // volume with increased accuracy
};
```

La enumeración ENUM_BOOK_TYPE contiene los siguientes miembros:

Identificador	Descripción
BOOK_TYPE_SELL	Solicitud de venta
BOOK_TYPE_BUY	Solicitud de compra
BOOK_TYPE_SELL_MARKET	Solicitud de venta a precio de mercado
BOOK_TYPE_BUY_MARKET	Solicitud de compra a precio de mercado

En el libro de órdenes, las órdenes de venta se sitúan en su mitad superior, y las de compra, en la inferior. Por regla general, esto conduce a una secuencia de elementos que van de los precios altos a los bajos. En otras palabras: por debajo del índice 0 se encuentra el precio más alto, y la última entrada es la más baja, mientras que entre ellas los precios disminuyen gradualmente. En este caso, el escalón de precio mínimo entre los niveles es SYMBOL_TRADE_TICK_SIZE; sin embargo, los niveles con volúmenes nulos no se trasladan, es decir, los elementos adyacentes pueden estar separados por un importe grande.

En la interfaz de usuario del terminal, la ventana del libro de órdenes ofrece una opción para activar o desactivar *Advanced Mode*, en la que también se muestran los niveles con volúmenes cero, pero por defecto, en el modo estándar, estos niveles están ocultos (omitidos en la tabla).

En la práctica, el contenido del libro de órdenes puede a veces contradecir las normas anunciadas. En concreto, algunas solicitudes de compra o venta pueden caer en la mitad opuesta del libro de órdenes (probablemente, alguien realizó una compra a un precio desfavorablemente alto o una venta a un precio desfavorablemente bajo, pero el proveedor también puede tener errores de agregación de datos). Como resultado, debido a la observancia de la prioridad «todas las órdenes de venta desde arriba, todas las órdenes de compra desde abajo», se violará la secuencia de precios en el libro de órdenes (véase el ejemplo a continuación). Además, pueden encontrarse valores repetidos de precios (niveles) tanto en una mitad del libro de órdenes como en la opuesta.

En teoría, la coincidencia de precios de compra y venta en el centro del libro de órdenes es correcta. Significa diferencial cero. Sin embargo, por desgracia, los niveles duplicados también se producen a mayor profundidad del libro de órdenes.

Cuando decimos «la mitad» del libro de órdenes no hay que tomarlo al pie de la letra. Dependiendo de la liquidez, el número de niveles de oferta y demanda puede no coincidir. En general, el libro no es simétrico.

El programa MQL debe comprobar que el libro de órdenes es correcto (en particular, el orden de clasificación de precios) y estar preparado para gestionar posibles desviaciones.

Entre las situaciones anómalas menos graves (que, no obstante, deben tenerse en cuenta en el algoritmo) figuran:

- Libros de órdenes idénticos consecutivos, sin cambios
- Libro de órdenes vacío
- Libro de órdenes con un nivel

A continuación se muestra un fragmento de una Profundidad de Mercado real recibida de un bróker. Las letras «S» y «B» marcan, respectivamente, los precios de las solicitudes de venta y de compra.

Observe que los niveles de compra y venta se solapan: visualmente no se aprecia mucho, ya que todos los registros «S» del libro de órdenes están colocados especialmente hacia arriba (el principio del array de recepción), y los registros «B» están hacia abajo (el final del array). Sin embargo, fíjese bien: los precios de compra en los elementos 20 y 21 son 143.23 y 138.86, respectivamente, y esto es más que todas las ofertas de venta. Y, al mismo tiempo, los precios de venta en los elementos 18 y 19 son 134.62 y 133.55, lo cual es inferior a todas las ofertas de compra.

```
...
10 S 138.48 652
11 S 138.47 754
12 S 138.45 2256
13 S 138.43 300
14 S 138.42 14
15 S 138.40 1761
16 S 138.39 670    // Duplicate
17 S 138.11 200
18 S 134.62 420    // Low
19 S 133.55 10627  // Low

20 B 143.23 9564   // High
21 B 138.86 533    // High
22 B 138.39 739    // Duplicate
23 B 138.38 106
24 B 138.31 100
25 B 138.25 29
26 B 138.24 6072
27 B 138.23 571
28 B 138.21 17
29 B 138.20 201
30 B 138.19 1
...
```

Además, el precio de 138.39 se encuentra tanto en la mitad superior, en el número 16, como en la mitad inferior, en el número 22.

Los errores en el libro de órdenes son más probables en condiciones extremas: con fuerte volatilidad o falta de liquidez.

Comprobemos la recepción del libro de órdenes mediante el indicador *MarketBookDisplay.mq5*. Se suscribirá a eventos de Profundidad de Mercado para el símbolo especificado en el parámetro *WorkSymbol* (si deja una línea vacía ahí se asume el símbolo de trabajo del gráfico actual).

```
input string WorkSymbol = ""; // WorkSymbol (if empty, use current chart symbol)

const string _WorkSymbol = StringLen(WorkSymbol) == 0 ? _Symbol : WorkSymbol;
int digits;

void OnInit()
{
   PRTF(MarketBookAdd(_WorkSymbol));
   digits = (int)SymbolInfoInteger(_WorkSymbol, SYMBOL_DIGITS);
   ...
}

void OnDeinit(const int)
{
   Comment("");
   PRTF(MarketBookRelease(_WorkSymbol));
}
```

El manejador *OnBookEvent* se define en el código para el manejo de eventos, en el que se llama a *MarketBookGet*, y todos los elementos del array *MqlBookInfo* resultante salen como un comentario multilínea.

```
void OnBookEvent(const string &symbol)
{
   if(symbol == _WorkSymbol) // take only order books of the requested symbol
   {
      MqlBookInfo mbi[];
      if(MarketBookGet(symbol, mbi)) // getting the current order book
      {
         ...
         int half = ArraySize(mbi) / 2; // estimate of the middle of the order book
         bool correct = true;
         // collect information about levels and volumes in one line (with hyphens)
         string s = "";
         for(int i = 0; i < ArraySize(mbi); ++i)
         {
            s += StringFormat("%02d %s %s %d %g\n", i,
               (mbi[i].type == BOOK_TYPE_BUY ? "B" :
               (mbi[i].type == BOOK_TYPE_SELL ? "S" : "?")),
               DoubleToString(mbi[i].price, digits),
               mbi[i].volume, mbi[i].volume_real);

            if(i > 0) // look for the middle of the order book as a change in request
            {
               if(mbi[i - 1].type == BOOK_TYPE_SELL
                  && mbi[i].type == BOOK_TYPE_BUY)
               {
                  half = i; // this is the middle, because there has been a type chan
               }

               if(mbi[i - 1].price <= mbi[i].price)
               {
                  correct = false; // reverse order = data problem
               }
            }
         }
         Comment(s + (!correct ? "\nINCORRECT BOOK" : ""));
         ...
      }
   }
}
```

Como el libro de órdenes cambia con bastante rapidez, no es muy conveniente seguir el comentario. Por lo tanto, añadiremos un par de búferes al indicador, en los que mostraremos el contenido de dos mitades del libro de órdenes como histogramas: venta y compra por separado. La barra cero corresponderá a los niveles centrales que forman el diferencial. Al aumentar el número de barras, aumenta la «profundidad del mercado», es decir, aparecen niveles de precios cada vez más lejanos: en el histograma superior, esto significa precios más bajos con órdenes de compra, y en el inferior, precios más altos con órdenes de venta.

```
#property indicator_separate_window
#property indicator_plots 2
#property indicator_buffers 2

#property indicator_type1   DRAW_HISTOGRAM
#property indicator_color1  clrDodgerBlue
#property indicator_width1  2
#property indicator_label1  "Buys"

#property indicator_type2   DRAW_HISTOGRAM
#property indicator_color2  clrOrangeRed
#property indicator_width2  2
#property indicator_label2  "Sells"

double buys[], sells[];
```

Vamos a ofrecer la posibilidad de visualizar el libro de órdenes en los modos estándar y ampliado (es decir, omitir o mostrar los niveles con volúmenes nulos), así como mostrar los propios volúmenes en fracciones de lotes o unidades. Ambas opciones tienen análogos en la ventana de Profundidad de Mercado integrada.

```
input bool AdvancedMode = false;
input bool ShowVolumeInLots = false;
```

Vamos a configurar los búferes y la obtención de algunas propiedades de los símbolos (que necesitaremos más adelante) en *OnInit*.

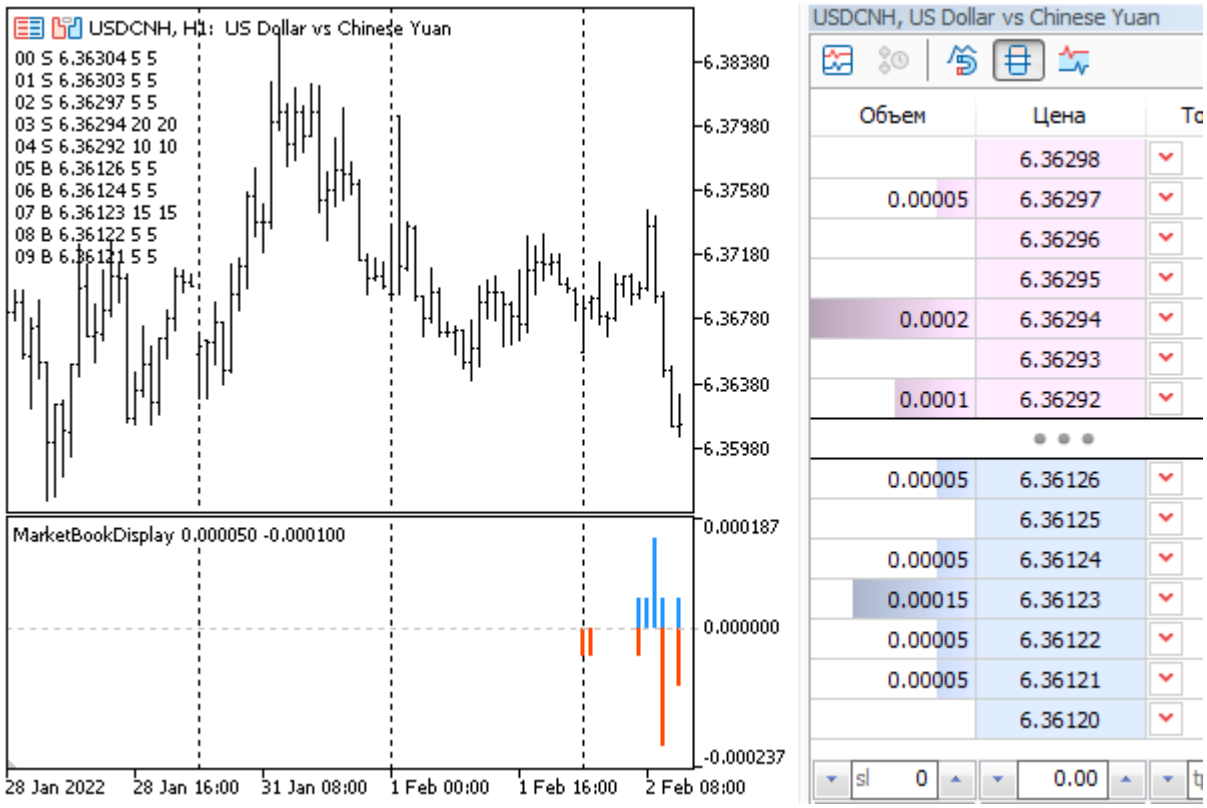
```
int depth, digits;
double tick, contract;

void OnInit()
{
   ...
   // setting indicator buffers
   SetIndexBuffer(0, buys);
   SetIndexBuffer(1, sells);
   ArraySetAsSeries(buys, true);
   ArraySetAsSeries(sells, true);
   // getting the necessary symbol properties
   depth = (int)PRTF(SymbolInfoInteger(_WorkSymbol, SYMBOL_TICKS_BOOKDEPTH));
   tick = SymbolInfoDouble(_WorkSymbol, SYMBOL_TRADE_TICK_SIZE);
   contract = SymbolInfoDouble(_WorkSymbol, SYMBOL_TRADE_CONTRACT_SIZE);
}
```

Añadamos el llenado del búfer al manejador *OnBookEvent*.

```
#define VOL(V) (ShowVolumeInLots ? V / contract : V)

void OnBookEvent(const string &symbol)
{
   if(symbol == _WorkSymbol) // take only order books of the requested symbol
   {
      MqlBookInfo mbi[];
      if(MarketBookGet(symbol, mbi)) // getting the current order book
      {
         // clear the buffers to the depth with 10 times the margin of the maximum de
         // because extended mode can have a lot of empty elements
         for(int i = 0; i <= depth * 10; ++i)
         {
            buys[i] = EMPTY_VALUE;
            sells[i] = EMPTY_VALUE;
         }
         ...// further along we form and display the comment as before
         if(!correct) return;

         // filling buffers with data
         if(AdvancedMode) // show skips enabled
         {
            for(int i = 0; i < ArraySize(mbi); ++i)
            {
               if(i < half)
               {
                  int x = (int)MathRound((mbi[i].price - mbi[half - 1].price) / tick)
                  sells[x] = -VOL(mbi[i].volume_real);
               }
               else
               {
                  int x = (int)MathRound((mbi[half].price - mbi[i].price) / tick);
                  buys[x] = VOL(mbi[i].volume_real);
               }
            }
         }
         else // standard mode: show only significant elements
         {
            for(int i = 0; i < ArraySize(mbi); ++i)
            {
               if(i < half)
               {
                  sells[half - i - 1] = -VOL(mbi[i].volume_real);
               }
               else
               {
                  buys[i - half] = VOL(mbi[i].volume_real);
               }
            }
         }
      }
```

```
      }
   }
```

En la siguiente imagen se muestra cómo funciona el indicador con los ajustes *AdvancedMode=true*, *ShowVolumeInLots=true*.



Contenido del libro de órdenes en el indicador MarketBookDisplay.mq5 en el gráfico USDCNH

Las compras se muestran como valores positivos (barra azul en la parte superior), y las ventas como valores negativos (barra roja en la parte inferior). Para mayor claridad, a la derecha hay una ventana estándar de Profundidad de Mercado con la misma configuración (en modo avanzado, volúmenes en lotes), para que pueda asegurarse de que los valores coinciden.

Debe tenerse en cuenta que es posible que el indicador no tenga tiempo de redibujarse con la suficiente rapidez para mantener la sincronización con el libro de órdenes integrado. Esto no significa que el programa MQL no haya recibido el evento a tiempo, sino que es un efecto secundario de la representación asíncrona del gráfico. Los algoritmos de trabajo suelen tener procesamiento analítico y colocación de órdenes con el libro de órdenes en lugar de visualización.

En este caso, la actualización del gráfico se solicita implícitamente en el momento de llamar a la función *Comment*.

6.2.4 Utilizar datos de Profundidad de Mercado en algoritmos aplicados

La Profundidad de Mercado se considera una tecnología muy útil para desarrollar sistemas avanzados de trading. En concreto, el análisis de la distribución de los volúmenes de Profundidad de Mercado en los niveles próximos al mercado permite conocer de antemano el precio medio de ejecución de la orden de un volumen específico: basta con sumar los volúmenes de los niveles (en sentido opuesto) que asegurarán su llenado. En un mercado poco activo, con volúmenes insuficientes, el algoritmo puede abstenerse de abrir una operación para evitar un deslizamiento significativo del precio.

A partir de los datos de Profundidad de Mercado pueden construirse también otras estrategias. Por ejemplo, puede ser importante conocer los niveles de precios en los que se encuentran los grandes volúmenes.

MarketBookVolumeAlert.mq5

En el siguiente indicador de prueba *MarketBookVolumeAlert.mq5* implementamos un algoritmo sencillo para rastrear los volúmenes o los cambios en los mismos que superan un valor determinado.

```
#property indicator_chart_window
#property indicator_plots 0

input string WorkSymbol = ""; // WorkSymbol (if empty, use current chart symbol)
input bool CountVolumeInLots = false;
input double VolumeLimit = 0;

const string _WorkSymbol = StringLen(WorkSymbol) == 0 ? _Symbol : WorkSymbol;
```

No hay gráficos en el indicador. El símbolo controlado se introduce en el parámetro *WorkSymbol* (si se deja en blanco quiere decirse que se trata del símbolo de trabajo del gráfico). El umbral mínimo de objetos rastreados, es decir, la sensibilidad del algoritmo, se especifica en el parámetro *VolumeLimit*. En función del parámetro *CountVolumeInLots*, los volúmenes se analizan y se muestran al usuario en lotes (*true*) o en unidades (*false*). Esto también afecta a la forma de introducir el valor *VolumeLimit*. La conversión de unidades a fracciones de lotes la proporciona la macro VOL: el tamaño del contrato utilizado en ella *contract* se inicializa en *OnInit* (véase más abajo).

```
#define VOL(V) (CountVolumeInLots ? V / contract : V)
```

Si se encuentran grandes volúmenes por encima del umbral, el programa mostrará un mensaje sobre el nivel correspondiente en el comentario. Para guardar el historial de avisos más cercano, utilizamos la clase de comentarios multilínea que ya conocemos (*Comments.mqh*).

```
#define N_LINES 25                // number of lines in the comment buffer
#include <MQL5Book/Comments.mqh>
```

En el manejador *OnInit* vamos a preparar los ajustes necesarios y a suscribirnos a los eventos de DOM.

```
double contract;
int digits;

void OnInit()
{
   MarketBookAdd(_WorkSymbol);
   contract = SymbolInfoDouble(_WorkSymbol, SYMBOL_TRADE_CONTRACT_SIZE);
   digits = (int)MathRound(MathLog10(contract));
   Print(SymbolInfoDouble(_WorkSymbol, SYMBOL_SESSION_BUY_ORDERS_VOLUME));
   Print(SymbolInfoDouble(_WorkSymbol, SYMBOL_SESSION_SELL_ORDERS_VOLUME));
}
```

Las propiedades SYMBOL_SESSION_BUY_ORDERS_VOLUME y SYMBOL_SESSION_SELL_ORDERS_VOLUME, si están rellenadas por su bróker para el símbolo seleccionado, le ayudarán a averiguar qué umbral tiene sentido elegir. Por defecto, *VolumeLimit* es 0, por lo que absolutamente todos los cambios en el libro de órdenes generarán avisos. Para filtrar las fluctuaciones insignificantes se recomienda fijar *VolumeLimit* en un valor que supere el tamaño medio

de los volúmenes en todos los niveles (busque de antemano en el libro de órdenes integrado o en el indicador *MarketBookDisplay.mq5*).

De la forma habitual, implementamos la finalización.

```
void OnDeinit(const int)
{
   MarketBookRelease(_WorkSymbol);
   Comment("");
}
```

El trabajo principal lo realiza el procesador *OnBookEvent*. Describe un array estático *MqlBookInfo mbp* para almacenar la versión anterior del libro de órdenes (desde la última llamada a la función).

```
void OnBookEvent(const string &symbol)
{
   if(symbol != _WorkSymbol) return; // process only the requested symbol

   static MqlBookInfo mbp[];      // previous table/book
   MqlBookInfo mbi[];
   if(MarketBookGet(symbol, mbi)) // read the current book
   {
      if(ArraySize(mbp) == 0) // first time we just save, because nothing to compare
      {
         ArrayCopy(mbp, mbi);
         return;
      }
      ...
```

Si hay un libro de órdenes antiguo y otro nuevo, comparamos los volúmenes de sus niveles entre sí en bucles anidados mediante *i* y *j*. Recordemos que un aumento del índice significa una disminución del precio.

```
   int j = 0;
   for(int i = 0; i < ArraySize(mbi); ++i)
   {
      bool found = false;
      for( ; j < ArraySize(mbp); ++j)
      {
         if(MathAbs(mbp[j].price - mbi[i].price) < DBL_EPSILON * mbi[i].price)
         {       // mbp[j].price == mbi[i].price
            if(VOL(mbi[i].volume_real - mbp[j].volume_real) >= VolumeLimit)
            {
               NotifyVolumeChange("Enlarged", mbp[j].price,
                  VOL(mbp[j].volume_real), VOL(mbi[i].volume_real));
            }
            else
            if(VOL(mbp[j].volume_real - mbi[i].volume_real) >= VolumeLimit)
            {
               NotifyVolumeChange("Reduced", mbp[j].price,
                  VOL(mbp[j].volume_real), VOL(mbi[i].volume_real));
            }
            found = true;
            ++j;
            break;
         }
         else if(mbp[j].price > mbi[i].price)
         {
            if(VOL(mbp[j].volume_real) >= VolumeLimit)
            {
               NotifyVolumeChange("Removed", mbp[j].price,
                  VOL(mbp[j].volume_real), 0.0);
            }
            // continue the loop increasing ++j to lower prices
         }
         else // mbp[j].price < mbi[i].price
         {
            break;
         }
      }
      if(!found) // unique (new) price
      {
         if(VOL(mbi[i].volume_real) >= VolumeLimit)
         {
            NotifyVolumeChange("Added", mbi[i].price, 0.0, VOL(mbi[i].volume_real)
         }
      }
   }
   ...
```

Aquí no se hace hincapié en el tipo de nivel, sino sólo en el valor del volumen. Sin embargo, si lo desea, puede añadir fácilmente la designación de compras o ventas a las notificaciones, dependiendo del campo *type* del nivel en el que se haya producido el cambio importante.

Por último, guardamos una nueva copia de *mbi* en un array estático *mbp* para compararla con ella en la siguiente llamada a la función.

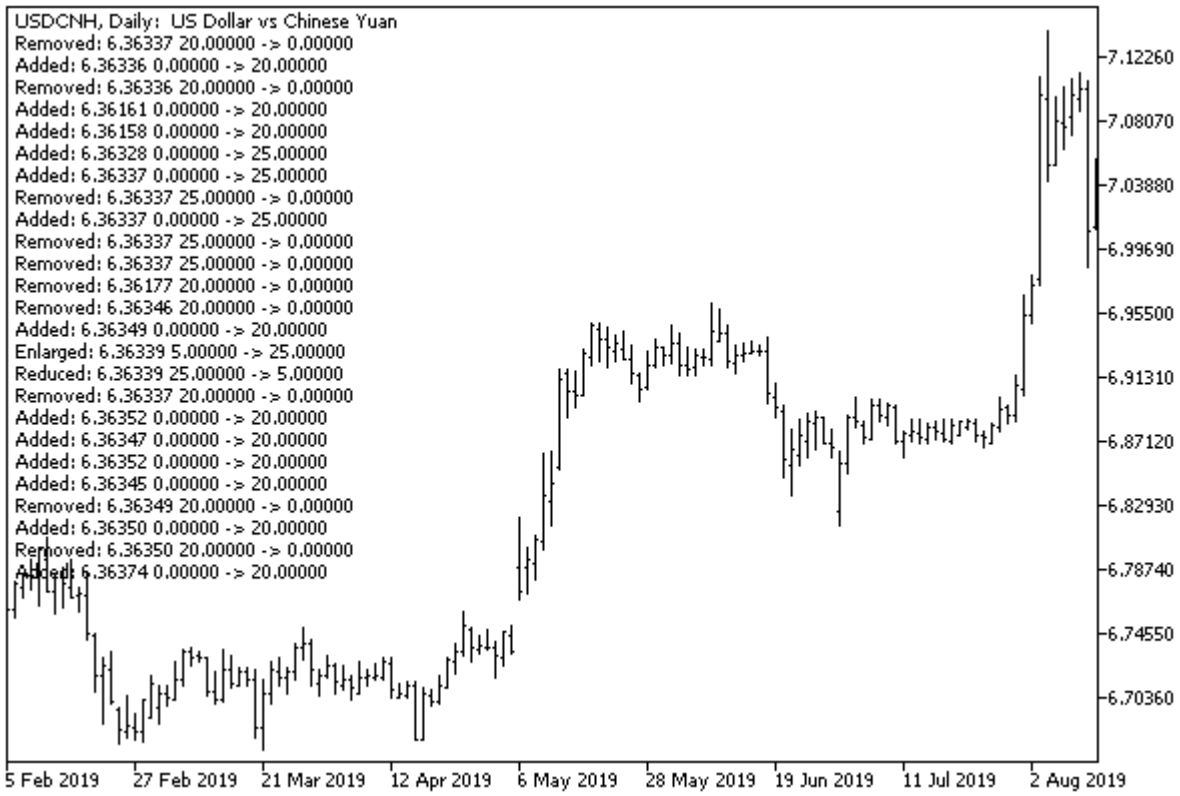
```
      if(ArrayCopy(mbp, mbi) <= 0)
      {
         Print("ArrayCopy failed:", _LastError);
      }
      if(ArrayResize(mbp, ArraySize(mbi)) <= 0) // shrink if needed
      {
         Print("ArrayResize failed:", _LastError);
      }
   }
}
```

ArrayCopy no encoge automáticamente un array de destino dinámico si resulta ser mayor que el array de origen, por lo que establecemos explícitamente su tamaño exacto con *ArrayResize*.

Una función auxiliar *NotifyVolumeChange* simplemente añade información sobre el cambio encontrado al comentario.

```
void NotifyVolumeChange(const string action, const double price,
   const double previous, const double volume)
{
   const string message = StringFormat("%s: %s %s -> %s",
      action,
      DoubleToString(price, (int)SymbolInfoInteger(_WorkSymbol, SYMBOL_DIGITS)),
      DoubleToString(previous, digits),
      DoubleToString(volume, digits));
   ChronoComment(message);
}
```

En la siguiente imagen se muestra el resultado del indicador para los ajustes *CountVolumeInLots=false*, *VolumeLimit=20*.



Notificaciones sobre cambios de volumen en el libro de órdenes

MarketBookQuasiTicks.mq5

Como segundo ejemplo del posible uso del libro de órdenes, echemos un vistazo al problema de la obtención de ticks multidivisa. Ya hemos hablado de ello en la sección Generación de eventos personalizados, donde hemos visto una de las posibles soluciones y el indicador *EventTickSpy.mq5*. Ahora, después de familiarizarnos con la API de Profundidad de Mercado, podemos implementar una alternativa.

Vamos a crear un indicador *MarketBookQuasiTicks.mq5*, que se suscribirá a los libros de órdenes de una lista dada de instrumentos y encontrará los precios de la mejor oferta y demanda en ellos, es decir, pares de precios en torno al diferencial, que no son más que los precios *Ask* y *Bid*.

Por supuesto, esta información no es un equivalente completo de los ticks estándar (recuérdese que los flujos de operaciones/ticks y del libro de órdenes pueden proceder de proveedores completamente distintos), pero proporciona una visión adecuada y oportuna del mercado.

Los nuevos valores de los precios por símbolos se mostrarán en un comentario de varias líneas.

La lista de símbolos de trabajo se especifica en el parámetro de entrada *SymbolList* como una lista separada por comas. La activación y desactivación de suscripciones a eventos de Profundidad de Mercado se realiza en los manejadores *OnInit* y *OnDeinit*.

```
#define N_LINES 25                // number of lines in the comment buffer
#include <MQL5Book/Comments.mqh>

input string SymbolList = "EURUSD,GBPUSD,XAUUSD,USDJPY"; // SymbolList (comma,separat

const string WorkSymbols = StringLen(SymbolList) == 0 ? _Symbol : SymbolList;
string symbols[];

void OnInit()
{
   const int n = StringSplit(WorkSymbols, ',', symbols);
   for(int i = 0; i < n; ++i)
   {
      if(!MarketBookAdd(symbols[i]))
      {
         PrintFormat("MarketBookAdd(%s) failed with code %d", symbols[i], _LastError)
      }
   }
}

void OnDeinit(const int)
{
   for(int i = 0; i < ArraySize(symbols); ++i)
   {
      if(!MarketBookRelease(symbols[i]))
      {
         PrintFormat("MarketBookRelease(%s) failed with code %d", symbols[i], _LastEr
      }
   }
   Comment("");
}
```

El análisis de cada nuevo libro de órdenes se realiza en *OnBookEvent*.

```
void OnBookEvent(const string &symbol)
{
   MqlBookInfo mbi[];
   if(MarketBookGet(symbol, mbi)) // getting the current order book
   {
      int half = ArraySize(mbi) / 2; // estimate the middle of the order book
      bool correct = true;
      for(int i = 0; i < ArraySize(mbi); ++i)
      {
         if(i > 0)
         {
            if(mbi[i - 1].type == BOOK_TYPE_SELL
               && mbi[i].type == BOOK_TYPE_BUY)
            {
               half = i; // specify the middle of the order book
            }

            if(mbi[i - 1].price <= mbi[i].price)
            {
               correct = false;
            }
         }
      }

      if(correct) // retrieve the best Bid/Ask prices from the correct order book
      {
         // mbi[half - 1].price // Ask
         // mbi[half].price     // Bid
         OnSymbolTick(symbol, mbi[half].price);
      }
   }
}
```

Los precios *Ask/Bid* encontrados en el mercado se pasan a la función de ayuda *OnSymbolTick* para que se muestren en un comentario.

```
void OnSymbolTick(const string &symbol, const double price)
{
   const string message = StringFormat("%s %s",
      symbol, DoubleToString(price, (int)SymbolInfoInteger(symbol, SYMBOL_DIGITS)));
   ChronoComment(message);
}
```

Si lo desea, puede asegurarse de que nuestros ticks sintetizados no difieren mucho de los ticks estándar.

Así es como se ve en el gráfico la información sobre los cuasi-ticks entrantes.

Cuasi-ticks multisímbolo basados en eventos del libro de órdenes

Al mismo tiempo, hay que señalar una vez más que los eventos del libro de órdenes sólo están disponibles en la plataforma en línea, pero no en el probador. Si el sistema de trading se construye exclusivamente a partir de cuasi-ticks del libro de órdenes, su simulación requerirá el uso de soluciones de terceros que garanticen la recopilación y reproducción de los libros de órdenes en el probador.

6.3 Información sobre la cuenta de trading

En este capítulo estudiaremos el último aspecto importante del entorno de trading de los programas MQL y, en concreto, de los Asesores Expertos, que desarrollaremos en detalle en los próximos capítulos. Veamos una cuenta de trading.

Tener una cuenta válida y una conexión activa a la misma son una condición necesaria para el funcionamiento de la mayoría de los programas MQL. Hasta ahora, no nos hemos centrado en esto, pero obtener cotizaciones, ticks y, en general, tener la capacidad de abrir un gráfico en el que poder trabajar implica una conexión satisfactoria a una cuenta de trading.

En el contexto de los Asesores Expertos, una cuenta refleja de forma adicional la condición financiera del cliente, acumula el historial de trading y determina los modos específicos permitidos para operar.

La API de MQL5 permite obtener las propiedades de una cuenta, empezando por su número y terminando por el beneficio actual. Todas ellas son de sólo lectura en el terminal y las instala el bróker en el servidor.

El terminal sólo puede conectarse a una cuenta a la vez. Todos los programas MQL funcionan con esta cuenta. Como ya hemos señalado en la sección Funciones de inicio y parada de programas de varios tipos, al cambiar de cuenta se recargan los indicadores y Asesores Expertos asociados a los gráficos. Sin embargo, en el manejador *OnDeinit*, el programa puede encontrar el motivo de la desinicialización, que, al cambiar la cuenta, será igual a REASON_ACCOUNT.

6.3.1 Visión general de las funciones para obtener las propiedades de la cuenta

El conjunto completo de propiedades de las cuentas se divide lógicamente en tres grupos en función de su tipo. Las propiedades de cadena se resumen en la enumeración ENUM_ACCOUNT_INFO_STRING y se consultan mediante la función *AccountInfoString*. Las propiedades de tipo real se combinan en la enumeración ENUM_ACCOUNT_INFO_DOUBLE, y la función que trabaja para ellas es *AccountInfoDouble*. La enumeración ENUM_ACCOUNT_INFO_INTEGER utilizada en la función *AccountInfoInteger* contiene identificadores de propiedades de enteros y booleanos (banderas), así como varias enumeraciones ENUM_ACCOUNT_INFO aplicadas.

double AccountInfoDouble(ENUM_ACCOUNT_INFO_DOUBLE property)
long AccountInfoInteger(ENUM_ACCOUNT_INFO_INTEGER property)
string AccountInfoString(ENUM_ACCOUNT_INFO_STRING property)

Hemos creado la clase *AccountMonitor* (*AccountMonitor.mqh*) para simplificar la lectura de las propiedades. Mediante la sobrecarga de los métodos *get*, la clase proporciona la llamada automática de la función API requerida dependiendo del elemento de una enumeración específica pasada en el parámetro.

```
class AccountMonitor
{
public:
   long get(const ENUM_ACCOUNT_INFO_INTEGER property) const
   {
      return AccountInfoInteger(property);
   }

   double get(const ENUM_ACCOUNT_INFO_DOUBLE property) const
   {
      return AccountInfoDouble(property);
   }

   string get(const ENUM_ACCOUNT_INFO_STRING property) const
   {
      return AccountInfoString(property);
   }

   long get(const int property, const long) const
   {
      return AccountInfoInteger((ENUM_ACCOUNT_INFO_INTEGER)property);
   }

   double get(const int property, const double) const
   {
      return AccountInfoDouble((ENUM_ACCOUNT_INFO_DOUBLE)property);
   }

   string get(const int property, const string) const
   {
      return AccountInfoString((ENUM_ACCOUNT_INFO_STRING)property);
   }
   ...
```

Además, dispone de varias sobrecargas del método *stringify*, que forman una representación de cadena fácil de usar de los valores de las propiedades (en concreto, esto es útil para las enumeraciones aplicadas, que de otro modo se mostrarían como números poco informativos). Las características de cada propiedad se analizarán en las secciones siguientes.

```
   static string boolean(const long v)
   {
      return v ? "true" : "false";
   }

   template<typename E>
   static string enumstr(const long v)
   {
      return EnumToString((E)v);
   }

   // "decode" properties according to subtype inside integer values
   static string stringify(const long v, const ENUM_ACCOUNT_INFO_INTEGER property)
   {
      switch(property)
      {
         case ACCOUNT_TRADE_ALLOWED:
         case ACCOUNT_TRADE_EXPERT:
         case ACCOUNT_FIFO_CLOSE:
            return boolean(v);
         case ACCOUNT_TRADE_MODE:
            return enumstr<ENUM_ACCOUNT_TRADE_MODE>(v);
         case ACCOUNT_MARGIN_MODE:
            return enumstr<ENUM_ACCOUNT_MARGIN_MODE>(v);
         case ACCOUNT_MARGIN_SO_MODE:
            return enumstr<ENUM_ACCOUNT_STOPOUT_MODE>(v);
      }

      return (string)v;
   }

   string stringify(const ENUM_ACCOUNT_INFO_INTEGER property) const
   {
      return stringify(AccountInfoInteger(property), property);
   }

   string stringify(const ENUM_ACCOUNT_INFO_DOUBLE property, const string format = NU
   {
      if(format == NULL) return DoubleToString(AccountInfoDouble(property),
         (int)get(ACCOUNT_CURRENCY_DIGITS));
      return StringFormat(format, AccountInfoDouble(property));
   }

   string stringify(const ENUM_ACCOUNT_INFO_STRING property) const
   {
      return AccountInfoString(property);
   }
   ...
```

Por último, existe un método de plantilla *list2log* que permite obtener información exhaustiva sobre la cuenta.

```
   // list of names and values of all properties of enum type E
   template<typename E>
   void list2log()
   {
      E e = (E)0; // suppress warning 'possible use of uninitialized variable'
      int array[];
      const int n = EnumToArray(e, array, 0, USHORT_MAX);
      Print(typename(E), " Count=", n);
      for(int i = 0; i < n; ++i)
      {
         e = (E)array[i];
         PrintFormat("% 3d %s=%s", i, EnumToString(e), stringify(e));
      }
   }
};
```

Veremos la nueva clase en acción en la siguiente sección.

6.3.2 Identificación de cuenta, cliente, servidor y bróker

Quizá las propiedades más importantes de una cuenta sean su número y los datos de identificación: el nombre del servidor y de la empresa del bróker, así como el nombre del cliente. Todas estas propiedades, excepto el número, son propiedades de cadena.

Identificador	Descripción
ACCOUNT_LOGIN	Número de cuenta (largo)
ACCOUNT_NAME	Nombre del cliente
ACCOUNT_SERVER	Nombre del servidor de trading
ACCOUNT_COMPANY	Nombre de la empresa que gestiona la cuenta

Utilicemos la clase *AccountMonitor* de la sección anterior para registrar estas y muchas otras propiedades que se abordarán en un momento. Vamos a crear el objeto correspondiente y a llamar a sus propiedades en el script *AccountInfo.mq5*.

```
#include <MQL5Book/AccountMonitor.mqh>

void OnStart()
{
   AccountMonitor m;
   m.list2log<ENUM_ACCOUNT_INFO_INTEGER>();
   m.list2log<ENUM_ACCOUNT_INFO_DOUBLE>();
   m.list2log<ENUM_ACCOUNT_INFO_STRING>();
}
```

He aquí un ejemplo de un posible resultado del script:

```
ENUM_ACCOUNT_INFO_INTEGER Count=10
  0 ACCOUNT_LOGIN=30000003
  1 ACCOUNT_TRADE_MODE=ACCOUNT_TRADE_MODE_DEMO
  2 ACCOUNT_TRADE_ALLOWED=true
  3 ACCOUNT_TRADE_EXPERT=true
  4 ACCOUNT_LEVERAGE=100
  5 ACCOUNT_MARGIN_SO_MODE=ACCOUNT_STOPOUT_MODE_PERCENT
  6 ACCOUNT_LIMIT_ORDERS=200
  7 ACCOUNT_MARGIN_MODE=ACCOUNT_MARGIN_MODE_RETAIL_HEDGING
  8 ACCOUNT_CURRENCY_DIGITS=2
  9 ACCOUNT_FIFO_CLOSE=false
ENUM_ACCOUNT_INFO_DOUBLE Count=14
  0 ACCOUNT_BALANCE=10000.00
  1 ACCOUNT_CREDIT=0.00
  2 ACCOUNT_PROFIT=-78.76
  3 ACCOUNT_EQUITY=9921.24
  4 ACCOUNT_MARGIN=1000.00
  5 ACCOUNT_MARGIN_FREE=8921.24
  6 ACCOUNT_MARGIN_LEVEL=992.12
  7 ACCOUNT_MARGIN_SO_CALL=50.00
  8 ACCOUNT_MARGIN_SO_SO=30.00
  9 ACCOUNT_MARGIN_INITIAL=0.00
 10 ACCOUNT_MARGIN_MAINTENANCE=0.00
 11 ACCOUNT_ASSETS=0.00
 12 ACCOUNT_LIABILITIES=0.00
 13 ACCOUNT_COMMISSION_BLOCKED=0.00
ENUM_ACCOUNT_INFO_STRING Count=4
  0 ACCOUNT_NAME=Vincent Silver
  1 ACCOUNT_COMPANY=MetaQuotes Software Corp.
  2 ACCOUNT_SERVER=MetaQuotes-Demo
  3 ACCOUNT_CURRENCY=USD
```

Preste atención a las propiedades de esta sección (ACCOUNT_LOGIN, ACCOUNT_NAME, ACCOUNT_COMPANY, ACCOUNT_SERVER). En este caso, el script se ejecutó en la cuenta del servidor de demostración «MetaQuotes-Demo». Obviamente, esta debe ser una cuenta demo, y esto se indica no sólo por el nombre del servidor, sino también por otra propiedad, ACCOUNT_TRADE_MODE, que se abordará en la siguiente sección.

Los identificadores de cuenta suelen utilizarse para vincular los programas MQL a un entorno de trading específico. Un ejemplo de este tipo de algoritmo se presentó en la sección Servicios.

6.3.3 Tipo de cuenta: real, demo o concurso

Son varios los tipos de cuenta que MetaTrader 5 permite abrir para un cliente. La propiedad ACCOUNT_TRADE_MODE, que forma parte de ENUM_ACCOUNT_INFO_INTEGER, permite conocer el tipo de cuenta actual. Los posibles valores de esta propiedad se describen en la enumeración ENUM_ACCOUNT_TRADE_MODE.

Identificador	Descripción
ACCOUNT_TRADE_MODE_DEMO	Cuenta de trading de demostración
ACCOUNT_TRADE_MODE_CONTEST	Cuenta de trading de concurso
ACCOUNT_TRADE_MODE_REAL	Cuenta de trading real

Esta propiedad es conveniente para construir versiones demo (gratuitas) de programas MQL. Una versión de pago con todas las funciones puede requerir la vinculación a un número de cuenta, y la cuenta debe ser real.

Como vimos en el ejemplo de ejecución del script *AccountInfo.mq5* en la sección anterior, la cuenta del servidor «MetaQuotes-Demo» es del tipo ACCOUNT_TRADE_MODE_DEMO.

6.3.4 Divisa de la cuenta

Saldo, beneficio, margen, comisiones y otros indicadores financieros al final siempre se convierten a la divisa de la cuenta, aunque las especificaciones de algunas operaciones requieran la liquidación en otras divisas, como por ejemplo, en la divisa del margen de un par de divisas.

La API de MQL5 proporciona dos propiedades que describen la divisa de la cuenta: su nombre y la precisión de la representación, es decir, el tamaño de la unidad mínima de medida (como los céntimos).

Identificador	Descripción
ACCOUNT_CURRENCY	Divisa del depósito (cadena)
ACCOUNT_CURRENCY_DIGITS	Número de decimales de la divisa de la cuenta necesarios para la visualización precisa de los resultados de trading (entero)

Por ejemplo, para la cuenta de demostración utilizada para probar el script *AccountInfo* en la sección sobre Identificación de la cuenta, la propiedad ACCOUNT_CURRENCY era «USD» y la precisión de ACCOUNT_CURRENCY_DIGITS era de 2 decimales. Hemos utilizado la propiedad ACCOUNT_CURRENCY_DIGITS de la clase *AccountMonitor* en el método *stringify* para valores de tipo *double* (en las características de la cuenta, todos están asociados a dinero).

6.3.5 Tipo de cuenta: compensación o cobertura

MetaTrader 5 admite varios tipos de cuentas, en concreto, compensación y cobertura. En el caso de la compensación, sólo se permite una posición para cada símbolo. Para la cobertura, puede abrir varias posiciones para un símbolo, incluidos los multidireccionales. Las órdenes, operaciones y posiciones se abordarán en detalle en los siguientes capítulos.

Un programa MQL determina el tipo de cuenta consultando la propiedad ACCOUNT_MARGIN_MODE mediante la función *AccountInfoInteger*. Como puede deducirse del nombre de la propiedad, no sólo describe el tipo de cuenta, sino también el modo de cálculo del margen. Sus posibles valores se especifican en la enumeración ENUM_ACCOUNT_MARGIN_MODE.

Identificador	Descripción
ACCOUNT_MARGIN_MODE_RETAIL_NETTING	Mercado OTC, considerando las posiciones en el modo de compensación. El cálculo del margen se basa en la propiedad SYMBOL_TRADE_CALC_MODE.
ACCOUNT_MARGIN_MODE_EXCHANGE	Mercado de divisas, considerando las posiciones en el modo de compensación. El margen se calcula en base a las reglas de la bolsa con la posibilidad de descuentos especificados por el bróker en la configuración del instrumento.
ACCOUNT_MARGIN_MODE_RETAIL_HEDGING	Mercado OTC con consideración independiente de las posiciones en el modo de cobertura. El cálculo del margen se basa en la propiedad del símbolo SYMBOL_TRADE_CALC_MODE al tiempo que se tiene en cuenta el tamaño del margen de cobertura SYMBOL_MARGIN_HEDGED.

Por ejemplo, la ejecución del script *AccountInfo* en la sección Identificación de la cuenta mostró que la cuenta es del tipo ACCOUNT_MARGIN_MODE_RETAIL_HEDGING.

6.3.6 Restricciones y permisos para las operaciones de la cuenta

Entre las propiedades de la cuenta existen restricciones en las operaciones de trading, entre las que se incluyen la completa deshabilitación del trading. Todas estas propiedades pertenecen a la enumeración ENUM_ACCOUNT_INFO_INTEGER y son banderas booleanas, excepto ACCOUNT_LIMIT_ORDERS.

Identificador	Descripción
ACCOUNT_TRADE_ALLOWED	Permiso para operar en una cuenta corriente
ACCOUNT_TRADE_EXPERT	Permiso para el trading algorítmico mediante Asesores Expertos y scripts
ACCOUNT_LIMIT_ORDERS	Número máximo permitido de órdenes pendientes válidas
ACCOUNT_FIFO_CLOSE	Obligación de cerrar las posiciones sólo según la regla FIFO

Dado que nuestro libro trata sobre la programación MQL5, que incluye el trading algorítmico, debe tenerse en cuenta que el permiso ACCOUNT_TRADE_EXPERT desactivado es tan crítico como la prohibición general de operar cuando ACCOUNT_TRADE_ALLOWED es igual a *false*. El bróker tiene la capacidad de prohibir el trading utilizando Asesores Expertos y scripts mientras que permite el trading manual.

La propiedad ACCOUNT_TRADE_ALLOWED suele ser igual a *false* si la conexión a la cuenta se realizó utilizando la contraseña de inversión.

Si el valor de la propiedad ACCOUNT_FIFO_CLOSE es *true*, las posiciones de cada símbolo sólo se pueden cerrar en el mismo orden en que se abrieron, es decir, primero se cierra la orden más antigua, luego la más nueva, y así sucesivamente hasta la última. Si intenta cerrar posiciones en un orden diferente, recibirá un error. Para las cuentas sin cobertura de posiciones, es decir, si la propiedad

ACCOUNT_MARGIN_MODE no es igual a ACCOUNT_MARGIN_MODE_RETAIL_HEDGING, la propiedad ACCOUNT_FIFO_CLOSE es siempre *false*.

En las secciones Permisos y Horarios de sesiones de trading y cotización ya empezamos a desarrollar una clase para detectar las operaciones de trading disponibles para el programa MQL. Ahora podemos complementarlo con comprobaciones de permisos de cuentas y llevarlo a la versión final (*Permissions.mqh*).

Los niveles de restricción se proporcionan en la enumeración TRADE_RESTRICTIONS, que, tras añadir dos nuevos elementos relacionados con las propiedades de las cuentas, adopta la siguiente forma:

```
class Permissions
{
   enum TRADE_RESTRICTIONS
   {
      NO_RESTRICTIONS = 0,
      TERMINAL_RESTRICTION = 1, // user's restriction for all programs
      PROGRAM_RESTRICTION = 2,  // user's restriction for a specific program
      SYMBOL_RESTRICTION = 4,   // the symbol is not traded according to the specific
      SESSION_RESTRICTION = 8,  // the market is closed according to the session sche
      ACCOUNT_RESTRICTION = 16, // investor password or broker restriction
      EXPERTS_RESTRICTION = 32, // broker restricted algorithmic trading
   };
   ...
```

Durante la comprobación, el programa MQL puede detectar varias restricciones por diversos motivos, por lo que los elementos se codifican mediante bits separados. El resultado final puede representar su superposición.

Las dos últimas restricciones sólo corresponden a las nuevas propiedades y se establecen en el método *getTradeRestrictionsOnAccount*. La máscara de bits general de las restricciones detectadas (si las hay) se forma en la variable *lastRestrictionBitMask*.

```
private:
   static uint lastRestrictionBitMask;
   static bool pass(const uint bitflag)
   {
      lastRestrictionBitMask |= bitflag;
      return lastRestrictionBitMask == 0;
   }

public:
   static uint getTradeRestrictionsOnAccount()
   {
      return (AccountInfoInteger(ACCOUNT_TRADE_ALLOWED) ? 0 : ACCOUNT_RESTRICTION)
         | (AccountInfoInteger(ACCOUNT_TRADE_EXPERT) ? 0 : EXPERTS_RESTRICTION);
   }

   static bool isTradeOnAccountEnabled()
   {
      lastRestrictionBitMask = 0;
      return pass(getTradeRestrictionsOnAccount());
   }
   ...
```

Si el código de llamada no está interesado en el motivo de la restricción, sino que sólo necesita determinar la posibilidad de realizar operaciones de trading, es más conveniente utilizar el método *isTradeOnAccountEnabled*, que devuelve un signo booleano (*true/false*).

Las comprobaciones de las propiedades de los símbolos y los terminales se han reorganizado siguiendo un principio similar. Por ejemplo, el método *getTradeRestrictionsOnSymbol* contiene el código fuente ya conocido de la versión anterior de la clase (comprobación de las sesiones de trading y los modos de trading del símbolo), pero devuelve una máscara de banderas. Si al menos un bit está activado, describe el origen de la restricción.

```
static uint getTradeRestrictionsOnSymbol(const string symbol, datetime now = 0,
   const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
   if(now == 0) now = TimeTradeServer();
   bool found = false;
   // checking the symbol trading sessions and setting 'found' to 'true',
   // if the 'now' time is inside one of the sessions
   ...

   // in addition to sessions, check the trading mode
   const ENUM_SYMBOL_TRADE_MODE m = (ENUM_SYMBOL_TRADE_MODE)SymbolInfoInteger(symb
   return (found ? 0 : SESSION_RESTRICTION)
      | (((m & mode) != 0) || (m == SYMBOL_TRADE_MODE_FULL) ? 0 : SYMBOL_RESTRICTI
}

static bool isTradeOnSymbolEnabled(const string symbol, const datetime now = 0,
   const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
   lastRestrictionBitMask = 0;
   return pass(getTradeRestrictionsOnSymbol(symbol, now, mode));
}
...
```

Por último, en los métodos *getTradeRestrictions* y *isTradeEnabled* se realiza una comprobación general de todas las posibles «instancias», incluyendo (además de los niveles anteriores) la configuración del terminal y del programa.

```
static uint getTradeRestrictions(const string symbol = NULL, const datetime now =
   const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
   return (TerminalInfoInteger(TERMINAL_TRADE_ALLOWED) ? 0 : TERMINAL_RESTRICTION)
       | (MQLInfoInteger(MQL_TRADE_ALLOWED) ? 0 : PROGRAM_RESTRICTION)
       | getTradeRestrictionsOnSymbol(symbol == NULL ? _Symbol : symbol, now, mode
       | getTradeRestrictionsOnAccount();
}

static bool isTradeEnabled(const string symbol = NULL, const datetime now = 0,
   const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
   lastRestrictionBitMask = 0;
   return pass(getTradeRestrictions(symbol, now, mode));
}
```

Una comprobación exhaustiva de los permisos de trading con una nueva clase se demuestra con el script *AccountPermissions.mq5*.

```
#include <MQL5Book/Permissions.mqh>

void OnStart()
{
   PrintFormat("Run on %s", _Symbol);
   if(!Permissions::isTradeEnabled()) // checking for current character, default
   {
      Print("Trade is disabled for the following reasons:");
      Print(Permissions::explainLastRestrictionBitMask());
   }
   else
   {
      Print("Trade is enabled");
   }
}
```

Si se encuentran restricciones, su máscara de bits puede mostrarse en una clara representación de cadena utilizando el método *explainLastRestrictionBitMask*.

A continuación se muestran algunos de los resultados del script. En los dos primeros casos, el trading estaba desactivado en la configuración global del terminal (las propiedades TERMINAL_TRADE_ALLOWED y MQL_TRADE_ALLOWED eran iguales a *false*, lo que corresponde a los bits TERMINAL_RESTRICTION y PROGRAM_RESTRICTION).

Cuando se ejecuta en USDRUB durante las horas en que el mercado está cerrado, recibiremos adicionalmente SESSION_RESTRICTION:

```
Trade is disabled for USDRUB following reasons:
TERMINAL_RESTRICTION PROGRAM_RESTRICTION SESSION_RESTRICTION
```

En el caso del símbolo SP500m, para el que el trading está totalmente desactivado, aparece la bandera SYMBOL_RESTRICTION.

```
Trade is disabled for SP500m following reasons:
TERMINAL_RESTRICTION PROGRAM_RESTRICTION SYMBOL_RESTRICTION SESSION_RESTRICTION
```

Por último, habiendo permitido el trading en el terminal pero habiendo accedido a la cuenta con la contraseña del inversor, veremos ACCOUNT_RESTRICTION en cualquier símbolo.

```
Run on XAUUSD
Trade is disabled for following reasons:
ACCOUNT_RESTRICTION
```

La comprobación anticipada de los permisos en el programa MQL ayuda a evitar intentos fallidos en serie de enviar órdenes de trading.

6.3.7 Configuración del margen de la cuenta

Para los robots de trading es importante controlar la cantidad de margen bloqueado y la cantidad disponible para asegurar nuevas operaciones. En concreto, si no hay suficientes fondos libres, el programa no podrá ejecutar una operación. Cuando se mantienen abiertas posiciones no rentables, primero se recibe un Margin Call, y si no se cumple, el bróker fuerza el cierre de las posiciones (Stop Out). Todas las propiedades de cuenta asociadas se incluyen en la enumeración ENUM_ACCOUNT_INFO_DOUBLE.

Identificador	Descripción
ACCOUNT_MARGIN	Margen reservado actual de la cuenta en la divisa del depósito
ACCOUNT_MARGIN_FREE	Margen libre actual en la cuenta en la divisa del depósito, disponible para abrir una posición
ACCOUNT_MARGIN_LEVEL	Nivel de margen de la cuenta en porcentaje (fondos propios/margen*100)
ACCOUNT_MARGIN_SO_CALL	El nivel de margen mínimo al que se exigirá la reposición de la cuenta (Margin Call)
ACCOUNT_MARGIN_SO_SO	El nivel de margen mínimo al que se obligará a cerrar la posición menos rentable (Stop Out)
ACCOUNT_MARGIN_INITIAL	Fondos reservados en la cuenta para proporcionar margen a todas las órdenes pendientes
ACCOUNT_MARGIN_MAINTENANCE	Fondos reservados en la cuenta para proporcionar el margen mínimo requerido para todas las posiciones abiertas

ACCOUNT_MARGIN_SO_CALL y ACCOUNT_MARGIN_SO_SO se expresan como porcentaje o en divisa de depósito en función del ACCOUNT_MARGIN_SO_MODE configurado (véase más adelante). Esta propiedad, con la posibilidad de medir umbrales de margen para Margin Call o Stop Out, está incluida en la enumeración ENUM_ACCOUNT_INFO_INTEGER. Además, aquí se indica también el apalancamiento total (utilizado para calcular el margen de determinados tipos de instrumentos).

Identificador	Descripción
ACCOUNT_LEVERAGE	El importe de apalancamiento
ACCOUNT_MARGIN_SO_MODE	El modo para establecer el nivel de margen mínimo permitido de la enumeración ENUM_ACCOUNT_STOPOUT_MODE

Y aquí están los elementos de la enumeración ENUM_ACCOUNT_STOPOUT_MODE:

Identificador	Descripción
ACCOUNT_STOPOUT_MODE_PERCENT	El nivel se establece como porcentaje
ACCOUNT_STOPOUT_MODE_MONEY	El nivel se establece en la divisa de la cuenta

Por ejemplo, para la opción ACCOUNT_STOPOUT_MODE_PERCENT, el porcentaje especificado (Margin Call o Stop Out) debe comprobarse frente a la relación entre el capital y el valor de la propiedad ACCOUNT_MARGIN:

```
AccountInfoDouble(ACCOUNT_EQUITY) / AccountInfoDouble(ACCOUNT_MARGIN) * 100
   > AccountInfoDouble(ACCOUNT_MARGIN_SO_CALL)
```

En la siguiente sección encontrará más detalles sobre la propiedad ACCOUNT_EQUITY y otros indicadores financieros de la cuenta.

Sin embargo, el nivel de margen actual en porcentaje ya se proporciona en la propiedad ACCOUNT_MARGIN_LEVEL. Esto es fácil de comprobar utilizando el script *AccountInfo.mq5* que registra todas las propiedades de la cuenta, incluidas las enumeradas anteriormente.

Ya hemos ejecutado este script en la sección Identificación de la cuenta. En ese momento se abrió una posición (1 lote USDRUB, igual a 100 000 USD), y las finanzas eran las siguientes:

```
0 ACCOUNT_BALANCE=10000.00
1 ACCOUNT_CREDIT=0.00
2 ACCOUNT_PROFIT=-78.76
3 ACCOUNT_EQUITY=9921.24
4 ACCOUNT_MARGIN=1000.00
5 ACCOUNT_MARGIN_FREE=8921.24
6 ACCOUNT_MARGIN_LEVEL=992.12
7 ACCOUNT_MARGIN_SO_CALL=50.00
8 ACCOUNT_MARGIN_SO_SO=30.00
```

Con un margen de 1000.00 USD es fácil comprobar que el apalancamiento de la cuenta, ACCOUNT_LEVERAGE, es efectivamente 100 (según la fórmula para calcular margen para Forex y coeficiente de margen que es igual a 1.0). No es necesario convertir el importe del margen al tipo de cambio vigente en la divisa de la cuenta, ya que es el mismo que el de la divisa base del instrumento.

Para obtener 992.12 en ACCOUNT_MARGIN_LEVEL, basta con dividir 9921.24 por 1000.00 y multiplicar por 100 %.

A continuación se abrió otra posición de 1 lote, y las cotizaciones tomaron una dirección desfavorable, como resultado de lo cual la situación cambió:

```
0 ACCOUNT_BALANCE=10000.00
1 ACCOUNT_CREDIT=0.00
2 ACCOUNT_PROFIT=-1486.07
3 ACCOUNT_EQUITY=8513.93
4 ACCOUNT_MARGIN=2000.00
5 ACCOUNT_MARGIN_FREE=6513.93
6 ACCOUNT_MARGIN_LEVEL=425.70
```

Podemos ver una pérdida en la columna ACCOUNT_PROFIT y una disminución correspondiente del capital ACCOUNT_EQUITY. El margen ACCOUNT_MARGIN aumentó proporcionalmente de 1000 a 2000, el margen libre y el nivel de margen disminuyeron (pero aún lejos de los límites del 50 % y el 30 %). De nuevo, el nivel 425.70 se obtiene como resultado de calcular la expresión 8513.93 / 2000.00 * 100.

Resulta más práctico utilizar esta fórmula para calcular el nivel de margen futuro antes de abrir una nueva posición. En este caso, es necesario aumentar el importe del margen existente con el margen adicional de *X*. Además, si una transacción de entrada en el mercado implica una deducción instantánea de la comisión *C*, entonces, en términos estrictos, ésta también debe tenerse en cuenta (aunque por lo general tiene un tamaño significativamente menor que el margen y se puede descuidar, además de que la API no proporciona una manera de averiguar la comisión por adelantado, antes de realizar una operación: sólo se puede estimar por las comisiones de las operaciones ya completadas en el historial de trading).

```
(AccountInfoDouble(ACCOUNT_EQUITY) - C) / (AccountInfoDouble(ACCOUNT_MARGIN) + X) * 1
   > AccountInfoDouble(ACCOUNT_MARGIN_SO_CALL)
```

Más adelante aprenderemos a obtener el valor de *X* utilizando la función *OrderCalcMargin*, pero además de ello pueden ser necesarios ajustes según las normas anunciadas en la sección Requisitos de margen, en concreto, teniendo en cuenta posibles cobertura de posiciones, descuentos y ajustes de margen.

Para la opción de establecer el límite de margen en dinero (ACCOUNT_STOPOUT_MODE_MONEY), la comprobación de fondos suficientes debe ser diferente.

```
AccountInfoDouble(ACCOUNT_EQUITY) > AccountInfoDouble(ACCOUNT_MARGIN_SO_CALL)
```

Aquí se omite la comisión. Tenga en cuenta que el margen *X* para una nueva posición que se está preparando para abrir «ahora» no afecta en modo alguno a la evaluación del margen «futuro».

En cualquier caso, no obstante, conviene no cargar el depósito hasta el punto de que apenas se cumplan las desigualdades. Los valores de ACCOUNT_MARGIN_SO_CALL y ACCOUNT_MARGIN_SO_SO son bastante cercanos, y aunque el margen en el nivel ACCOUNT_MARGIN_SO_CALL es sólo una advertencia para el operador, es fácil conseguir un cierre forzado. Por eso las fórmulas utilizan la propiedad ACCOUNT_MARGIN_SO_CALL.

6.3.8 Resultados financieros actuales de la cuenta

La API de MQL5 permite controlar varias propiedades de las cuentas a través de sus principales indicadores financieros. Todos ellos están incluidos en la enumeración ENUM_ACCOUNT_INFO_DOUBLE.

Identificador	Descripción
ACCOUNT_BALANCE	Saldo de la cuenta en la divisa del depósito
ACCOUNT_PROFIT	Importe del beneficio corriente de la cuenta en la divisa del depósito
ACCOUNT_EQUITY	Capital de la cuenta en la divisa del depósito
ACCOUNT_CREDIT	El importe del crédito proporcionado por el bróker en la divisa del depósito.
ACCOUNT_ASSETS	Importe actual de los activos en la cuenta
ACCOUNT_LIABILITIES	Importe actual del pasivo de la cuenta
ACCOUNT_COMMISSION_BLOCKED	El importe actual de las comisiones bloqueadas en la cuenta

En las secciones anteriores vimos ejemplos de los valores de estas propiedades al ejecutar el script *AccountInfo.mq5* en diferentes condiciones. Intente comparar estas propiedades para sus diferentes cuentas.

En el proceso de trading nos interesarán principalmente las tres primeras propiedades: saldo, beneficio (o pérdida si el valor es negativo) y capital, que en conjunto cubren el saldo de la cuenta, el crédito, el beneficio y los gastos generales (swap y comisión).

Las comisiones pueden considerarse de diferentes maneras, dependiendo de la configuración del bróker. Si las comisiones se deducen inmediatamente del saldo de la cuenta en el momento de las operaciones y se reflejan en las propiedades de la transacción, la propiedad de la cuenta ACCOUNT_COMMISSION_BLOCKED será igual a 0. Sin embargo, si el cálculo de la comisión se pospone hasta el final del periodo (por ejemplo, un día o un mes), el importe bloqueado para la comisión aparecerá en esta propiedad. A continuación, cuando se determine el importe final de la comisión y se deduzca del saldo al final del periodo, se restablecerá la propiedad.

Las propiedades ACCOUNT_ASSETS y ACCOUNT_LIABILITIES se rellenan, por regla general, sólo para las operaciones bursátiles. Reflejan el valor actual de las posiciones largas y cortas en valores.

6.4 Crear Asesores Expertos

En este capítulo comenzamos a estudiar la API de trading de MQL5 utilizada para implementar Asesores Expertos. Este tipo de programa es quizá el más complejo y exigente en cuanto a codificación sin errores y número y variedad de tecnologías implicadas. En concreto, tendremos que utilizar muchos de los conocimientos adquiridos en los capítulos anteriores, desde la programación orientada a objetos hasta los aspectos aplicados del trabajo con objetos gráficos, indicadores, símbolos y configuraciones del entorno de software.

En función de la estrategia de trading elegida, el desarrollador del Asesor Experto deberá prestar especial atención a los siguientes aspectos:

- Velocidad de toma de decisiones y envío de órdenes (para HFT, High-Frequency Trading)
- Selección de la cartera óptima de instrumentos basados en sus correlaciones y volatilidad (para trading de clústeres)
- Cálculo dinámico de lotes y distancia entre órdenes (para estrategias de martingala y cuadrícula)
- Análisis de noticias o fuentes de datos externas (se tratará en la 7ª parte del libro)

Todas estas características deben ser aplicadas de forma óptima por el desarrollador a los mecanismos de trading descritos proporcionados por la API de MQL5.

A continuación, consideraremos en detalle las funciones integradas para la gestión de la actividad de trading, el modelo de eventos del Asesor Experto y las estructuras de datos específicas, y recordaremos los principios básicos de la interacción entre el terminal y el servidor, así como los conceptos básicos para la negociación algorítmica en MetaTrader 5: orden, transacción y posición.

Al mismo tiempo, debido a la versatilidad del material, muchos matices importantes del desarrollo de Asesores Expertos, como la simulación y la optimización, se destacan en el capítulo siguiente.

Anteriormente vimos el Diseño de programas MQL de varios tipos, incluidos los Asesores Expertos, así como las Funciones de inicio y parada de programas. A pesar de que un Asesor Experto se lanza en un gráfico específico, para el que se define un símbolo de trabajo, no hay obstáculos para gestionar de forma centralizada el trading de un conjunto arbitrario de instrumentos financieros. Estos Asesores Expertos se denominan tradicionalmente multidivisa, aunque de hecho, su cartera puede incluir CFD, acciones, materias primas y tickers de otros mercados.

En los Asesores Expertos, al igual que en los indicadores, existen los Eventos clave *OnInit* y *OnDeinit*. No son obligatorios, pero, por regla general, están presentes en el código para la preparación y realización regular del programa: los hemos utilizado y los seguiremos utilizando en los ejemplos. En una sección separada, proporcionamos una Visión general de las funciones de gestión de eventos: a estas alturas ya hemos estudiado en detalle algunas de ellas (por ejemplo, los eventos de indicador *OnCalculate* y el

temporizador *OnTimer*). Los eventos específicos del Asesor Experto (*OnTick*, *ontrade*, *OnTradeTransaction*) se describirán en este capítulo.

Los Asesores Expertos pueden utilizar la más amplia gama de datos de origen como señales de trading: cotizaciones, ticks, Profundidad de Mercado, historial de la cuenta de trading o las lecturas de indicadores. En este último caso, los principios de creación de instancias indicadoras y lectura de valores de sus búferes no difieren de los abordados en el capítulo Utilización de indicadores preconfeccionados de programas MQL. En los ejemplos de Asesor Experto de las siguientes secciones, haremos una demostración de la mayoría de estos trucos.

Cabe señalar que las funciones de trading pueden utilizarse no sólo en los Asesores Expertos, sino también en los scripts. Veremos ejemplos de ambas opciones.

6.4.1 Evento principal de Asesores Expertos: OnTick

El evento *OnTick* es generado por el terminal para los Asesores Expertos cuando aparece un nuevo tick que contiene el precio del símbolo de trabajo del gráfico actual en el que se está ejecutando el Asesor Experto. Para manejar este evento, la función *OnTick* debe definirse en el código del Asesor Experto. Tiene el siguiente prototipo:

```
void OnTick(void)
```

Como puede ver, la función no tiene parámetros. En caso necesario, el propio valor del nuevo precio y otras características del tick deben solicitarse llamando a *SymbolInfoTick*.

Desde el punto de vista de la reacción al nuevo evento de tick, este manejador es similar a *OnCalculate* en los indicadores. No obstante, *OnCalculate* sólo puede definirse en indicadores, y *OnTick* sólo en Asesores Expertos (para ser más precisos, la función *OnTick* en el código de un indicador, script o servicio será simplemente ignorada).

Al mismo tiempo, el Asesor Experto no tiene que contener el manejador *OnTick*. Además de este evento, los Asesores Expertos pueden procesar los eventos *OnTimer*, *OnBookEvent* y *OnChartEvent* y realizar a partir de ellos todas las operaciones de trading necesarias.

Todos los eventos en los Asesores Expertos se procesan uno tras otro en el orden en que llegan, ya que los Asesores Expertos, como todos los demás programas MQL, son de un solo hilo. Si ya hay un evento *OnTick* en la cola o se está procesando un evento de este tipo, entonces los nuevos eventos *OnTick* no se ponen en cola.

Se genera un evento *OnTick* independientemente de si la negociación automática está desactivada o activada (botón *Algo trading* en la interfaz del terminal). La desactivación del trading automático significa sólo la restricción en el envío de solicitudes de negociación de los Asesores Expertos, pero no impide que el Asesor Experto se ejecute.

Hay que recordar que los eventos de tick sólo se generan para un símbolo, que es el símbolo del gráfico actual. Si el Asesor Experto es multidivisa, la obtención de ticks de otros símbolos debe organizarse de alguna forma alternativa; por ejemplo, utilizando un indicador espía *EventTickSpy.mq5* o una suscripción a eventos de libros de órdenes, como en *MarketBookQuasiTicks.mq5*.

Como ejemplo sencillo, considere el Asesor Experto *ExpertEvents.mq5*. Define manejadores para todos los eventos que suelen utilizarse para lanzar algoritmos de trading. Estudiaremos algunos otros eventos (*OnTrade*, *OnTradeTransaction* y eventos de probadores) más adelante.

Todos los manejadores llaman a la función de ayuda *display* que muestra la hora actual (etiqueta del contador de milisegundos del sistema) y el nombre del manejador en un comentario de varias líneas.

```
#define N_LINES 25
#include <MQL5Book/Comments.mqh>

void Display(const string message)
{
   ChronoComment((string)GetTickCount() + ": " + message);
}
```

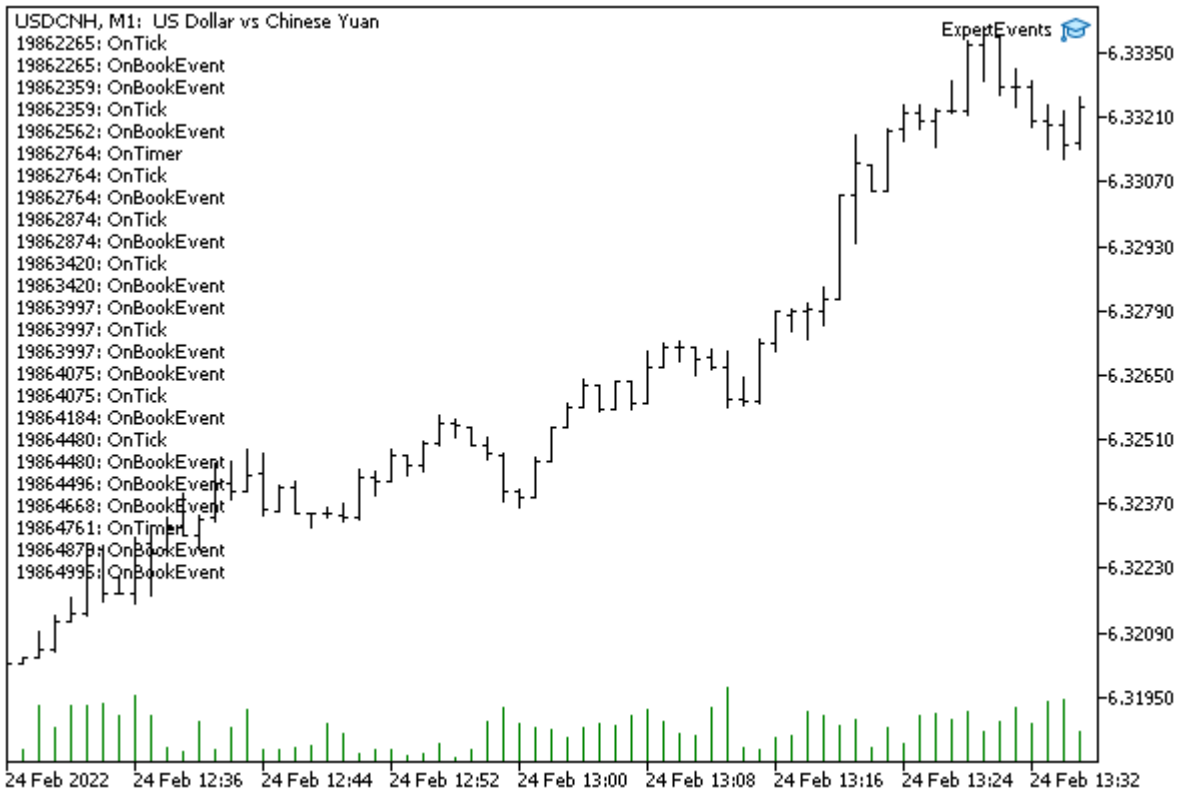
El evento *OnTick* será llamado automáticamente a la llegada de nuevos ticks. Para los eventos de temporizador y libro de órdenes, es necesario activar los manejadores correspondientes mediante llamadas a *EventSetTimer* y *MarketBookAdd* desde *OnInit*.

```
void OnInit()
{
   Print(__FUNCTION__);
   EventSetTimer(2);
   if(!MarketBookAdd(_Symbol))
   {
      Print("MarketBookAdd failed:", _LastError);
   }
}

void OnTick()
{
   Display(__FUNCTION__);
}

void OnTimer()
{
   Display(__FUNCTION__);
}

void OnBookEvent(const string &symbol)
{
   if(symbol == _Symbol) // react only to order book of "our" symbol
   {
      Display(__FUNCTION__);
   }
}
```

El evento de cambio de gráfico también está disponible: se puede utilizar para operar sobre el marcado basado en objetos gráficos, pulsando botones o teclas de acceso rápido, así como a la llegada de eventos personalizados de otros programas; por ejemplo, indicadores como *EventTickSpy.mq5*.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   Display(__FUNCTION__);
}

void OnDeinit(const int)
{
   Print(__FUNCTION__);
   MarketBookRelease(_Symbol);
   Comment("");
}
```

En la siguiente captura de pantalla se muestra el resultado de la operación del Asesor Experto en el gráfico:



Comentarios con eventos de varios tipos en el Asesor Experto

Tenga en cuenta que el evento *OnBookEvent* (si se emite para un símbolo) llega más a menudo que *OnTick*.

6.4.2 Principios y conceptos básicos: orden, transacción y posición

Antes de comenzar a estudiar el desarrollo de Asesores Expertos en MQL5, recordemos la arquitectura general de la plataforma y los conceptos básicos que formalizan la actividad de trading.

MetaTrader 5 es un terminal de cliente conectado a una parte de servidor multinivel distribuida entre los ordenadores de un bróker, operador o bolsa. Una vez que un usuario rellena una orden para ejecutar una operación, ésta pasa por varias etapas de envío y verificación, tras las cuales es registrada o rechazada por el operador o la bolsa. Entonces, una orden registrada en el mercado puede ejecutarse o

no en función de circunstancias como la liquidez, el ritmo de variación de los precios, la pausa en el trading de símbolos o cuestiones técnicas.

Régimen general de procesamiento de una solicitud de operación

Aquí, las flechas verdes indican la ejecución satisfactoria de una operación de trading a medida que pasa del terminal al mercado, y las flechas rojas indican un posible rechazo.

Las órdenes generadas por programas MQL también pasan por instancias similares. En caso de resultado desfavorable, la API de MQL5 nos permitirá conocer el motivo del fallo a través del código de error.

Todo este proceso se expresa (y documenta en informes) en tres términos fundamentales: orden, transacción y posición.

Una orden es la instrucción que un operador da a una empresa de correduría para comprar o vender un instrumento financiero. MetaTrader 5 admite varios tipos de órdenes, pero de forma simplificada se pueden dividir condicionalmente en mercado, pendientes y niveles de protección especiales *Take Profit* y *Stop Loss*.

Como resultado de la ejecución satisfactoria de una orden, se produce una transacción en el sistema de trading. Concretamente, una transacción puede cerrarse al precio actual en el caso de una orden de mercado, o cuando una orden pendiente se activa cuando el precio alcanza el valor especificado en la orden. En otras palabras: una transacción es el hecho de comprar o vender un determinado instrumento financiero.

Hay que tener en cuenta que, en algunas condiciones, la ejecución de una orden puede dar lugar a varias transacciones. Por ejemplo, si el libro de órdenes no contiene una cantidad suficiente de liquidez de símbolos, entonces una orden de compra puede ejecutarse a través de varias órdenes contrarias, incluidas aquellas a un precio ligeramente diferente.

Un instrumento financiero comprado o vendido de acuerdo con una transacción forma una posición larga o corta, respectivamente, que se refleja en el activo/pasivo de la cuenta de trading. Como resultado del cambio posterior en el precio del instrumento de la posición se forma una ganancia o pérdida flotante en la cuenta, que puede fijarse cerrando la posición mediante operaciones inversas (órdenes y transacciones). Dependiendo del tipo de cuenta de trading (compensación o cobertura), las transacciones para el mismo instrumento modifican una única posición neta o crean/borran posiciones independientes.

Encontrará más información en el manual de usuario del terminal

Todas las órdenes, transacciones y posiciones se incluyen en el historial de trading de la cuenta.

A continuación, examinaremos la API de software, que incluye funciones para enviar órdenes de trading, obtener el estado actual de la cartera en la cuenta, comprobar la carga del margen y los beneficios/pérdidas potenciales, así como analizar el historial de trading.

6.4.3 Tipos de operaciones de trading

El trading en MQL5 se lleva a cabo mediante el envío de órdenes utilizando la función *OrderSend*. Lo estudiaremos en una de las siguientes secciones porque su descripción requiere familiarizarse primero con varios conceptos.

El primer concepto nuevo será el tipo de operación de trading. Cada solicitud de operación contiene una indicación del tipo de operación solicitada y permite realizar acciones tales como abrir y cerrar posiciones, así como colocar, modificar y eliminar órdenes pendientes. Todos los tipos de operaciones de trading se describen en la enumeración ENUM_TRADE_REQUEST_ACTIONS.

Identificador	Descripción
TRADE_ACTION_DEAL	Colocar una orden de trading para una operación inmediata con los parámetros especificados (colocar una orden de mercado).
TRADE_ACTION_PENDING	Colocar una orden de trading para ejecutar una operación en las condiciones especificadas (orden pendiente).
TRADE_ACTION_SLTP	Cambiar los valores *Stop Loss* y *Take Profit* de una posición abierta
TRADE_ACTION_MODIFY	Cambiar los parámetros de una orden realizada anteriormente
TRADE_ACTION_REMOVE	Eliminar una orden pendiente previamente colocada
TRADE_ACTION_CLOSE_BY	Cerrar una posición con otra opuesta

Al solicitar TRADE_ACTION_DEAL y TRADE_ACTION_PENDING, el programa deberá especificar un tipo concreto de orden. Este es otro concepto importante que tiene su propio reflejo en la API de MQL5, y lo consideraremos en la siguiente sección.

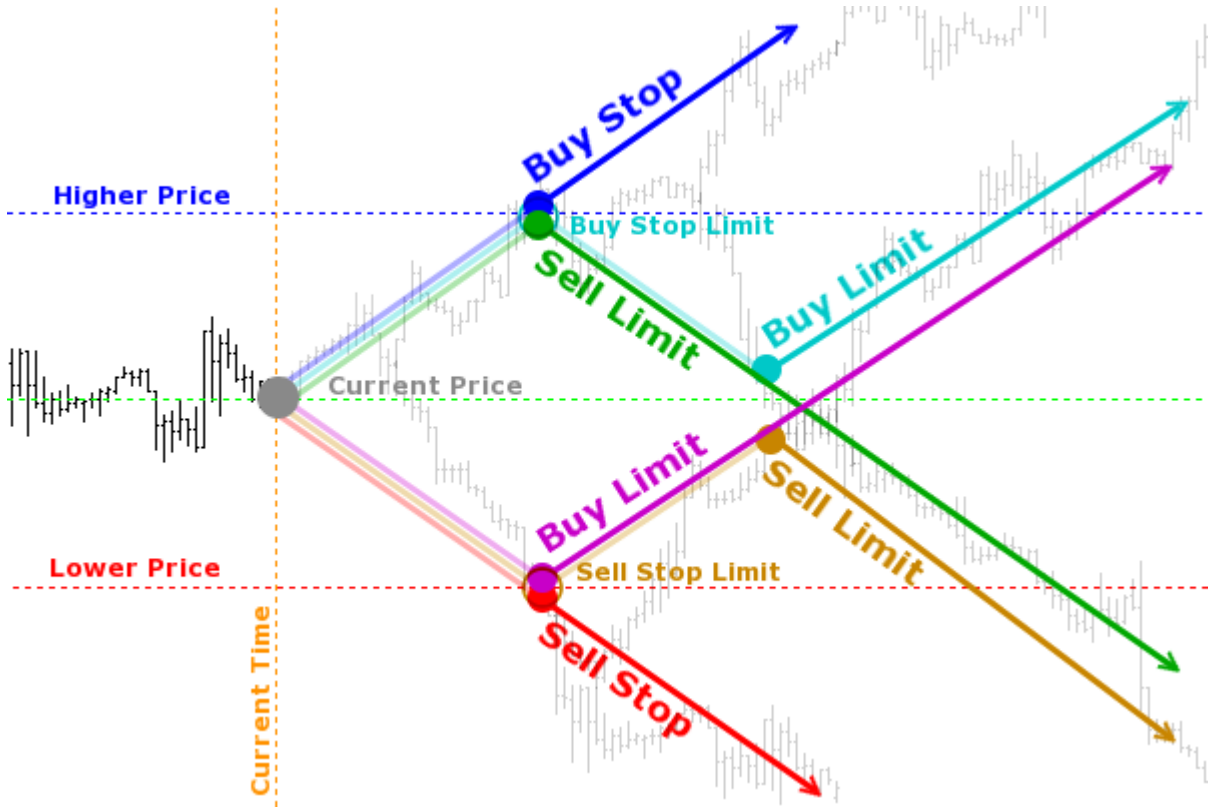
6.4.4 Tipos de órdenes

Como sabe, MetaTrader 5 admite varios tipos de órdenes: dos órdenes de mercado para comprar y vender al precio actual, y seis pendientes con niveles de activación predefinidos por encima y por debajo del mercado. Todos estos tipos están disponibles en la API de MQL5 y se describen mediante los elementos de la enumeración ENUM_ORDER_TYPE. Más adelante veremos cómo crear una orden de un tipo determinado en un programa. De momento, familiaricémonos con la enumeración.

Identificador	Descripción
ORDER_TYPE_BUY	Orden de mercado para la compra
ORDER_TYPE_SELL	Orden de mercado para la venta
ORDER_TYPE_BUY_LIMIT	*Buy Limit* es una orden pendiente.
ORDER_TYPE_SELL_LIMIT	*Sell Limit* es una orden pendiente.
ORDER_TYPE_BUY_STOP	*Buy Stop* es una orden pendiente.
ORDER_TYPE_SELL_STOP	*Sell Stop* es una orden pendiente.
ORDER_TYPE_BUY_STOP_LIMIT	*Buy Limit* es una orden pendiente que se colocará cuando el precio alcance el nivel superior especificado.
ORDER_TYPE_SELL_STOP_LIMIT	*Sell Limit* es una orden pendiente que se colocará cuando el precio alcance el nivel inferior especificado.
ORDER_TYPE_CLOSE_BY	Orden para cerrar una posición con otra opuesta

El último elemento corresponde a la acción de cerrar posiciones opuestas: esta posibilidad sólo existe en cuentas de cobertura y para los instrumentos financieros que tengan propiedades que permitan tales operaciones (SYMBOL_ORDER_CLOSEBY).

La siguiente imagen puede recordarle los principios generales de activación de órdenes pendientes. En ella se muestra en gris la evolución futura prevista de los precios. Sin embargo, en la actualidad no se sabe qué pronóstico resultará acertado.



Esquema de activación de órdenes pendientes

Buy Stop y *Sell Stop* son órdenes pendientes que siguen el principio de desglose de niveles: para *Buy Stop*, este nivel debe estar situado por encima del precio actual, y debe estar por debajo del precio actual para *Sell Stop*. En otras palabras: en un nivel determinado, queremos que se ejecute una operación de compra o venta esperando que se siga negociando en la dirección de la tendencia.

Buy Limit y *Sell Limit* aplican la estrategia de rebote desde el nivel, y en este caso, el precio de activación de la compra está por debajo del precio actual, y el precio de venta está por encima. Esto implica un cambio de tendencia o una fluctuación en el corredor. En el diagrama de arriba se utilizan los mismos niveles de activación superior (Precio superior) e inferior (Precio inferior) de las órdenes pendientes para ilustrar tanto una ruptura como un rebote.

Las órdenes pendientes pueden colocarse al precio actual, y lo más probable es que se ejecuten inmediatamente. Además, esta técnica aplicada a las órdenes Limit garantiza un precio de negociación que no es peor que el solicitado, a diferencia de una orden de mercado.

Las órdenes de los tipos *Buy Stop Limit* y *Sell Stop Limit* no se envían al mercado como resultado de su activación, sino que colocan órdenes pendientes establecidas, *Buy Limit* o *Sell Limit*, en algunos niveles adicionales especificados en la orden original.

En el caso de los instrumentos bursátiles, las órdenes Limit (*Buy Limit*, *Sell Limit*) suelen mostrarse directamente en el libro de órdenes y son visibles para los demás participantes en el mercado.

Por el contrario, las órdenes *Stop* y *Stop Limit* (*Buy Stop*, *Sell Stop*, *Buy Stop Limit* y *Sell Stop Limit*) no se envían directamente al sistema de trading externo. Hasta que se alcanza el precio de stop, este tipo de órdenes se procesan dentro de la plataforma de MetaTrader 5. Cuando se alcanza el precio stop especificado en la orden Buy Stop o Sell Stop, se ejecuta la operación de mercado correspondiente. Al alcanzar el precio de stop especificado en la orden *Buy Stop Limit* o *Sell Stop Limit* se coloca la orden Limit correspondiente.

En el modo de ejecución bursátil, el precio especificado al colocar órdenes Limit no se comprueba. Puede especificarse por encima del precio actual de *Ask* (para órdenes de compra) y por debajo del precio de *Bid* (para órdenes de venta). Al colocar una orden con ese precio, casi inmediatamente se activa y se convierte en una orden de mercado.

Tenga en cuenta que no todos los tipos de órdenes pueden estar permitidos para un instrumento financiero específico: la propiedad SYMBOL_ORDER_MODE describe las banderas de los tipos de orden permitidos.

6.4.5 Modos de ejecución de órdenes por precio y volumen

Al enviar solicitudes de negociación tendremos que especificar el precio de compra/venta y el volumen en el algoritmo de forma especial. Al mismo tiempo, hay que tener en cuenta que en los mercados financieros no hay garantía de que en ese momento todo el volumen solicitado esté disponible para el instrumento financiero al precio deseado. Por lo tanto, las operaciones de trading están reguladas por modos (o políticas) de ejecución de precios y volúmenes. Definen las reglas para los casos en que el precio ha cambiado durante el proceso de envío de la solicitud o cuando no puede satisfacerse en su totalidad.

En el capítulo sobre símbolos, en la sección Condiciones de trading de símbolos y modos de ejecución de órdenes ya hemos hablado de los ajustes para la ejecución de órdenes por precio (SYMBOL_TRADE_EXEMODE) y la ejecución de órdenes por volumen (SYMBOL_FILLING_MODE), que establece el bróker. De acuerdo con los modos SYMBOL_FILLING_MODE disponibles, el programa MQL

debe seleccionar el modo de ejecución para la orden recién formada en una estructura especial *MqlTradeRequest* (pronto lo veremos en la práctica).

Las versiones se proporcionan en la enumeración ENUM_ORDER_TYPE_FILLING: sus identificadores se hacen eco de los de SYMBOL_FILLING_MODE.

Política de ejecución (valores)	Descripción
ORDER_FILLING_FOK (0)	Fill or Kill (Todo/Nada)
ORDER_FILLING_IOC (1)	Immediate or Cancel (Todo/Parte)
ORDER_FILLING_RETURN (2)	Return (Devolver)

Con la política ORDER_FILLING_FOK, una orden sólo puede ejecutarse en el volumen especificado. Si no hay suficiente volumen del instrumento financiero en el mercado en ese momento, la orden no se ejecutará. El volumen requerido puede estar formado por varias ofertas disponibles actualmente en el mercado. La capacidad de utilizar órdenes FOK viene determinada por la presencia del permiso SYMBOL_FILLING_FOK.

Con la política ORDER_FILLING_IOC, el operador se compromete a realizar una transacción sobre el volumen máximo disponible en el mercado dentro de los límites especificados en la orden. Si no es posible la cobertura total, la orden se ejecutará sobre el volumen disponible, y el volumen que falte se cancelará. La capacidad de utilizar órdenes IOC viene determinada por la presencia del permiso SYMBOL_FILLING_IOC.

Con la política ORDER_FILLING_RETURN, en caso de ejecución parcial, la orden con el volumen restante no se cancela, sino que continúa operando. Este es el modo por defecto y siempre está disponible. Sin embargo, hay una excepción: Las órdenes Return no están permitidas en el modo de ejecución de mercado (SYMBOL_TRADE_EXECUTION_MARKET en la propiedad de símbolo SYMBOL_TRADE_EXEMODE).

Así, antes de enviar una orden de mercado (no pendiente), el programa MQL debe establecer correctamente una de las políticas ORDER_TYPE_FILLING en función de la propiedad SYMBOL_FILLING_MODE del instrumento financiero correspondiente: esta propiedad contiene una combinación de indicadores de bits de los modos permitidos.

Para las órdenes pendientes, con independencia del modo de ejecución SYMBOL_TRADE_EXEMODE, se debe utilizar la política ORDER_FILLING_RETURN, ya que dichas órdenes se ejecutarán con volumen posteriormente y según las reglas que el bróker establezca en ese momento.

> A diferencia de la política de ejecución por volumen, el modo de ejecución de la orden a un precio no se puede seleccionar, ya que está predeterminado por el bróker para cada símbolo. Esto afecta a los campos de la estructura *MqlTradeRequest* que deben ejecutarse antes de enviar una solicitud de trading.

La aplicación de las políticas de ejecución en función de los modos de ejecución puede representarse como una tabla ('+' - permitido, '-' - desactivado, '±' - depende de la configuración del símbolo):

Política de ejecución / Modo de ejecución	ORDER_FILLING _FOK	ORDER_FILLING _IOC	ORDER_FILLING RETURN
SYMBOL_TRADE_EXECUTION_INSTANT	+	+	+
SYMBOL_TRADE_EXECUTION_REQUEST	+	+	+
SYMBOL_TRADE_EXECUTION_MARKET	±	±	-
SYMBOL_TRADE_EXECUTION_EXCHANGE	±	±	+
Pendiente	-	-	+

En los modos de ejecución SYMBOL_TRADE_EXECUTION_INSTANT y SYMBOL_TRADE_EXECUTION_REQUEST se permiten todas las políticas de ejecución de volumen.

6.4.6 Fechas de vencimiento de órdenes pendientes

Para las órdenes pendientes, una característica importante es su modo de vencimiento. En la API de MQL5, el período de validez de la orden puede establecerse en el campo *type_time* de la estructura *MqlTradeRequest* al enviar una solicitud de operación a través de la función *OrderSend*. Los valores aceptables se describen en la enumeración ENUM_ORDER_TYPE_TIME.

Identificador (Valor)	Descripción
ORDER_TIME_GTC (0)	La orden permanecerá en la cola hasta que se cancele.
ORDER_TIME_DAY (1)	La orden sólo será válida durante la jornada bursátil en curso.
ORDER_TIME_SPECIFIED (2)	La orden será válida hasta la fecha de vencimiento.
ORDER_TIME_SPECIFIED_DAY (3)	La orden será válida hasta las 23:59:59 del día especificado (si esta hora no cae dentro de la sesión de trading, el vencimiento se producirá a la hora más próxima).

Cabe señalar que cada instrumento financiero tiene dos propiedades SYMBOL_EXPIRATION_MODE y SYMBOL_ORDER_GTC_MODE, que determinan las reglas de vencimiento de órdenes pendientes para este instrumento. Al formar una orden, un programa MQL puede elegir uno de los modos permitidos. Consideraremos un ejemplo después de estudiar la función *OrderSend*.

6.4.7 Cálculo del margen para una orden futura: OrderCalcMargin

Antes de enviar una solicitud de operación al servidor, un programa MQL puede calcular el margen necesario para una operación planificada utilizando la función *OrderCalcMargin*. Se recomienda hacerlo siempre así para evitar una carga excesiva de depósitos.

bool OrderCalcMargin(ENUM_ORDER_TYPE action, const string symbol, double volume, double price, double &margin)

La función calcula el margen necesario para el tipo de orden *action* especificado y el instrumento financiero *symbol* con lotes *volume*. Esto se ajusta a la configuración de la cuenta corriente, pero no tiene en cuenta las órdenes pendientes existentes ni las posiciones abiertas. La enumeración ENUM_ORDER_TYPE se introdujo en la sección Tipos de órdenes.

El valor del margen (en la divisa de la cuenta) se escribe en el parámetro *margin* pasado por referencia.

Cabe destacar que se trata de una estimación del margen para una única posición u orden nueva, y no del valor total de la fianza, en el que se convertirá tras la ejecución. Además, la evaluación se realiza como si no hubiera otras órdenes pendientes ni posiciones abiertas en la cuenta corriente. En realidad, el valor del margen depende de muchos factores, incluidas otras órdenes y posiciones, y puede variar a medida que cambia el entorno del mercado (como el apalancamiento).

La función devuelve un indicador de éxito (*true*) o de error (*false*). El código de error puede obtenerse de la forma habitual a partir de la variable *_LastError*.

> La función *OrderCalcMargin* sólo puede utilizarse en Asesores Expertos y scripts. Para calcular el margen en los indicadores es necesario implementar un método alternativo; por ejemplo, lanzar un Asesor Experto auxiliar en un objeto gráfico, pasarle parámetros y obtener el resultado a través del mecanismo de eventos, o describir de forma independiente los cálculos en MQL5 utilizando fórmulas según los tipos de instrumentos. En la sección siguiente ofrecemos un ejemplo de una implementación de este tipo, junto con una estimación de los posibles beneficios/pérdidas.

Podríamos escribir un script sencillo que llame a *OrderCalcMargin* para los símbolos de *Observación de Mercado*, y comparar los valores de los márgenes para ellos. En lugar de ello, vamos a complicar ligeramente la tarea y a considerar el archivo de encabezado *LotMarginExposure.mqh*, que permite evaluar la carga del depósito y el nivel de margen después de abrir una posición con un nivel de riesgo predeterminado. Un poco más adelante hablaremos de la función *OrderCheck* que es capaz de proporcionar información similar. No obstante, nuestro algoritmo podrá resolver además el problema inverso de elegir el tamaño del lote en función de los niveles de carga o riesgo dados.

El uso de las nuevas funciones se demuestra en un Asesor Experto no de trading *LotMarginExposureTable.mq5*.

> En teoría, el hecho de que un programa MQL se implemente como un Asesor Experto no significa que las operaciones de trading deban realizarse en él. Muy a menudo, como en nuestro caso, se crean varias utilidades en forma de Asesor Experto. Su ventaja sobre los scripts es que permanecen en el gráfico y pueden realizar sus funciones indefinidamente en respuesta a determinados eventos.

En el nuevo Asesor Experto utilizamos las habilidades de creación de una interfaz gráfica interactiva utilizando objetos. Para decirlo de una manera más sencilla, para una lista dada de símbolos, el Asesor Experto mostrará una tabla con varias columnas de indicadores de margen en el gráfico, y la tabla se puede ordenar por cada una de las columnas. Proporcionaremos la lista de columnas un poco más adelante.

Dado que el análisis de lotes, margen y carga de depósito es una tarea común, separaremos la implementación en un archivo de encabezado independiente *LotMarginExposure.mqh*.

Todas las funciones de archivo se agrupan en un espacio de nombres para evitar conflictos y en aras de la claridad (indicar el contexto antes de llamar a una función interna informa sobre el origen y la ubicación de esta función).

```
namespace LEMLR
{
   ...
};
```

La abreviatura *LEMLR* significa «Lot, Exposure, Margin Level, Risk» (lote, exposición, nivel de margen, riesgo).

Los principales cálculos se realizan en la función *Estimate*. Considerando un prototipo de la función integrada *OrderCalcMargin*, en los parámetros de la función *Estimate* necesitamos pasar el nombre del símbolo, el tipo de orden, el volumen y el precio. Pero eso no es todo lo que necesitamos.

```
bool Estimate(const ENUM_ORDER_TYPE type, const string symbol, const double lot,
   const double price,...)
```

Pretendemos evaluar varios indicadores de una operación de trading, que están interconectados y pueden calcularse en distintas direcciones, dependiendo de lo que el usuario haya introducido como datos iniciales y de lo que desee calcular. Por ejemplo, utilizando los parámetros anteriores, es fácil encontrar el nuevo nivel de margen y la carga de la cuenta. Sus fórmulas son exactamente lo contrario:

```
Ml = money / margin * 100
Ex = margin / money * 100
```

Aquí la variable *margin* indica la cantidad de margen, para lo cual basta con llamar a *OrderCalcMargin*.

No obstante, los operadores a menudo prefieren partir de un nivel de carga o margen predeterminado y calcular el volumen para ello. Además, existe un método de cálculo de lotes basado en el riesgo que también goza de gran popularidad. Por riesgo se entiende el importe de la pérdida potencial del trading en caso de un movimiento desfavorable de los precios, como consecuencia del cual disminuirá el contenido de otra variable de las fórmulas anteriores, es decir, *money*.

Para calcular la pérdida, es importante conocer la volatilidad del instrumento financiero durante el periodo de trading (la duración de la estrategia) o la distancia del Stop Loss asumida por el usuario.

Por lo tanto, la lista de parámetros de la función *Estimate* se amplía.

```
bool Estimate(const ENUM_ORDER_TYPE type, const string symbol, const double lot,
   const double price,
   const double exposure, const double riskLevel, const int riskPoints,
   const ENUM_TIMEFRAMES riskPeriod, double money,...)
```

En el parámetro *exposure* especificamos la carga de depósito deseada en porcentaje, y en el parámetro *riskLevel*, indicamos la parte del depósito (también en porcentaje) que estamos dispuestos a arriesgar. Para los cálculos basados en el riesgo, puede pasar el tamaño del Stop Loss en puntos en el parámetro *riskPoints*. Cuando es igual a 0, entra en juego el parámetro *riskPeriod*: especifica el periodo para el que el algoritmo calculará automáticamente el rango de cotizaciones de los símbolos en puntos. Por último, en el parámetro *money*, podemos especificar una cantidad arbitraria de margen libre para la evaluación de lotes. Algunos operadores dividen condicionalmente el depósito entre varios robots. Cuando *money* es 0, la función ejecutará esta variable con la propiedad *AccountInfoDouble(ACCOUNT_MARGIN_FREE)*.

Ahora tenemos que decidir cómo devolver los resultados de la función. Dado que es capaz de evaluar muchos indicadores de trading y varias opciones de volumen, tiene sentido definir la estructura *SymbolLotExposureRisk*.

```
struct SymbolLotExposureRisk
{
   double lot;                      // requested volume (or minimum)
   int atrPointsNormalized;         // price range normalized by tick size
   double atrValue;                 // range as the amount of profit/loss for 1 lo
   double lotFromExposureRaw;       // not normalized ( can be less than the minim
   double lotFromExposure;          // normalized lot from deposit loading
   double lotFromRiskOfStopLossRaw; // not normalized (can be less than the minimu
   double lotFromRiskOfStopLoss;    // normalized lot from risk
   double exposureFromLot;          // loading based on the volume of 'lot
   double marginLevelFromLot;       // margin level from 'lot' volume
   int lotDigits;                   // number of digits in normalized lots
};
```

El campo *lot* de la estructura contiene el lote pasado a la función *Exposure* si el lote no es igual a 0. Si el lote pasado es cero, se sustituye por la propiedad de símbolo SYMBOL_VOLUME_MIN.

Se asignan dos campos para los valores calculados de volúmenes basados en la carga del depósito y el riesgo: con el sufijo *Raw* (*lotFromExposureRaw*, *lotFromRiskOfStopLossRaw*) y sin él (*lotFromExposure*, *lotFromRiskOfStopLoss*). *Raw* contienen un resultado «aritmético puro», que puede no coincidir con la especificación del símbolo. En los campos sin sufijo, los lotes se normalizan considerando el mínimo, el máximo y el paso. Esta duplicación es útil, en particular, para los casos en que el cálculo da valores inferiores al lote mínimo (por ejemplo, *lotFromExposureRaw* es igual a 0.023721 con un mínimo de 0.1, debido a lo cual *lotFromExposure* se reduce a cero): entonces, a partir del contenido de los campos *Raw*, se puede evaluar cuánto dinero añadir o cuánto aumentar el riesgo para llegar al lote mínimo.

Vamos a describir el último parámetro de salida de la función *Estimate* como una referencia a esta estructura. Iremos ejecutando poco a poco todos los campos del cuerpo de la función. En primer lugar, obtenemos el margen de un lote llamando a *OrderCalcMargin* y lo guardamos en una variable local *lot1margin*.

```
bool Estimate(const ENUM_ORDER_TYPE type, const string symbol, const double lot,
   const double price, const double exposure,
   const double riskLevel, const int riskPoints, const ENUM_TIMEFRAMES riskPeriod,
   double money, SymbolLotExposureRisk &r)
{
   double lot1margin;
   if(!OrderCalcMargin(type, symbol, 1.0,
      price == 0 ? GetCurrentPrice(symbol, type) : price,
      lot1margin))
   {
      Print("OrderCalcMargin ", symbol, " failed: ", _LastError);
      return false;
   }
   if(lot1margin == 0)
   {
      Print("Margin ", symbol, " is zero, ", _LastError);
      return false;
   }
   ...
```

Si no se especifica el precio de entrada, es decir, *price* es igual a 0, la función de ayuda *GetCurrentPrice* devuelve un precio adecuado en función del tipo de orden: para las compras, se tomará

la propiedad de símbolo SYMBOL_ASK, y para las ventas será SYMBOL_BID. Esta y otras funciones de ayuda se omiten aquí, su contenido se puede encontrar en el código fuente adjunto.

Si el cálculo del margen falla, o se recibe un valor cero, la función *Estimate* devolverá *false*.

Tenga en cuenta que el margen cero puede ser la norma, pero también puede ser un error, dependiendo del instrumento y del tipo de orden. Por lo tanto, para los tickers de bolsa, las órdenes pendientes están sujetas a depósito, pero no para los tickers OTC (es decir, el depósito 0 es correcto). Este punto debe tenerse en cuenta en el código de llamada: debe solicitar margen sólo para aquellas combinaciones de símbolos y tipos de operaciones para las que tenga sentido y se suponga que es distinto de cero.

Teniendo un depósito para un lote, podemos calcular el número de lotes para asegurar una carga determinada del depósito.

```
double usedMargin = 0;
if(money == 0)
{
   money = AccountInfoDouble(ACCOUNT_MARGIN_FREE);
   usedMargin = AccountInfoDouble(ACCOUNT_MARGIN);
}

r.lotFromExposureRaw = money * exposure / 100.0 / lot1margin;
r.lotFromExposure = NormalizeLot(symbol, r.lotFromExposureRaw);
...
```

A continuación se muestra la función de ayuda *NormalizeLot*.

Con el fin de obtener un lote dependiendo del riesgo y la volatilidad, hay que calcular un poco más.

```
const double tickValue = SymbolInfoDouble(symbol, SYMBOL_TRADE_TICK_VALUE);
const int pointsInTick = (int)(SymbolInfoDouble(symbol, SYMBOL_TRADE_TICK_SIZE)
   / SymbolInfoDouble(symbol, SYMBOL_POINT));
const double pointValue = tickValue / pointsInTick;
const int atrPoints = (riskPoints > 0) ? (int)riskPoints :
   (int)(((MathMax(iHigh(symbol, riskPeriod, 1), iHigh(symbol, riskPeriod, 0))
   -  MathMin(iLow(symbol, riskPeriod, 1), iLow(symbol, riskPeriod, 0)))
   / SymbolInfoDouble(symbol, SYMBOL_POINT)));
// rounding by tick size
r.atrPointsNormalized = atrPoints / pointsInTick * pointsInTick;
r.atrValue = r.atrPointsNormalized * pointValue;

r.lotFromRiskOfStopLossRaw = money * riskLevel / 100.0
   / (pointValue * r.atrPointsNormalized);
r.lotFromRiskOfStopLoss = NormalizeLot(symbol, r.lotFromRiskOfStopLossRaw);
...
```

Aquí encontramos el coste de un pip del instrumento y el rango de sus cambios para el periodo especificado, tras el cual ya calculamos el lote.

Por último, obtenemos la carga de la cuenta y el nivel de margen para el lote dado.

```
      r.lot = lot <= 0 ? SymbolInfoDouble(symbol, SYMBOL_VOLUME_MIN) : lot;
      double margin = r.lot * lot1margin;

      r.exposureFromLot = (margin + usedMargin) / money * 100.0;
      r.marginLevelFromLot = margin > 0 ? money / (margin + usedMargin) * 100.0 : 0;
      r.lotDigits = (int)MathLog10(1.0 / SymbolInfoDouble(symbol, SYMBOL_VOLUME_MIN))

      return true;
   }
```

En caso de que el cálculo se realice correctamente, la función devolverá *true*.

Aquí está la vista abreviada de la función *NormalizeLot* (todas las comprobaciones de 0 se omiten para simplificar). Encontrará información detallada sobre las propiedades correspondientes en la sección Volúmenes permitidos de operaciones de trading.

```
double NormalizeLot(const string symbol, const double lot)
{
   const double stepLot = SymbolInfoDouble(symbol, SYMBOL_VOLUME_STEP);
   const double newLotsRounded = MathFloor(lot / stepLot) * stepLot;
   const double minLot = SymbolInfoDouble(symbol, SYMBOL_VOLUME_MIN);
   if(newLotsRounded < minLot) return 0;
   const double maxLot = SymbolInfoDouble(symbol, SYMBOL_VOLUME_MAX);
   if(newLotsRounded > maxLot) return maxLot;
   return newLotsRounded;
}
```

La implementación anterior de *Estimate* no tiene en cuenta los ajustes por solapamiento de posiciones. Por regla general, conducen a una disminución del depósito, por lo que la estimación actual de la carga de la cuenta y el nivel de margen puede ser más pesimista de lo que resulta en la realidad, pero esto proporciona una protección adicional. Los interesados pueden añadir un código para analizar la composición de los fondos de la cuenta ya congelados (su importe total figura en la propiedad de la cuenta ACCOUNT_MARGIN) desglosados por posiciones y órdenes: entonces será posible tener en cuenta el efecto potencial de una nueva orden sobre el margen (por ejemplo, sólo se tendrá en cuenta la posición más grande de las opuestas o se aplicará una tasa de margen de cobertura reducida, véanse los detalles en la sección Requisitos de margen).

Ahora es el momento de poner en práctica la estimación de márgenes y lotes en *LotMarginExposureTable.mq5*. Teniendo en cuenta que los campos *Raw* sólo se mostrarán en aquellos casos en los que la normalización de los lotes haya llevado a su puesta a cero, el número total de columnas de la tabla de indicadores resultante es de 8.

```
#include <MQL5Book/LotMarginExposure.mqh>
#define TBL_COLUMNS 8
```

En los parámetros de entrada ofreceremos la posibilidad de especificar el tipo de orden, la lista de símbolos que se van a analizar (una lista separada por comas), los fondos disponibles, así como el lote, la carga del depósito objetivo, el nivel de margen y el riesgo.

```
input ENUM_ORDER_TYPE Action = ORDER_TYPE_BUY;
input string WorkList = "";                  // Symbols (comma,separated,list)
input double Money = 0;                      // Money (0 = free margin)
input double Lot = 0;                        // Lot (0 = min lot)
input double Exposure = 5.0;                 // Exposure (%)
input double RiskLevel = 5.0;                // RiskLevel (%)
input int RiskPoints = 0;                    // RiskPoints/SL (0 = auto-range of Ris
input ENUM_TIMEFRAMES RiskPeriod = PERIOD_W1;
```

Para los tipos de órdenes pendientes es necesario seleccionar símbolos bursátiles, ya que para otros símbolos se obtendrá un margen cero, lo que provocará un error en la función *Estimate*. Si la lista de símbolos se deja vacía, el Asesor Experto procesará sólo el símbolo del gráfico actual. Los valores cero por defecto de los parámetros *Money* y *Lot* significan, respectivamente, la cantidad actual de fondos libres en la cuenta y el lote mínimo para cada símbolo.

El valor 0 en el parámetro *RiskPoints* significa obtener un rango de precios durante *RiskPeriod* (por defecto es una semana).

El parámetro de entrada *UpdateFrequency* establece la frecuencia de recálculo en segundos. Si lo deja igual a cero, el recálculo se realiza en cada nueva barra.

```
input int UpdateFrequency = 0; // UpdateFrequency (sec, 0 - once per bar)
```

En el contexto global se describen un array de símbolos (rellenada posteriormente analizando el parámetro de entrada *WorkList*) y la marca de tiempo del último cálculo realizado con éxito.

```
string symbols[];
datetime lastTime;
```

Al arrancar, encendemos el segundo temporizador.

```
void OnInit()
{
   Comment("Starting...");
   lastTime = 0;
   EventSetTimer(1);
}
```

En el manejador del temporizador proporcionamos la primera llamada al cálculo principal en *OnTick*, si aún no se ha llamado a *OnTick* al llegar un tick. Esta situación puede darse, por ejemplo, los fines de semana o durante un mercado en calma. Además, *OnTimer* es el punto de entrada para los recálculos a una frecuencia determinada.

```
void OnTimer()
{
   if(lastTime == 0) // calculation for the first time (if OnTick did have time to tr
   {
      OnTick();
      Comment("Started");
   }
   else if(lastTime != -1)
   {
      if(UpdateFrequency <= 0) // if there is no frequency, we work on new bars in On
      {
         EventKillTimer();     // and the timer is no longer needed
      }
      else if(TimeCurrent() - lastTime >= UpdateFrequency)
      {
         lastTime = LONG_MAX; // prevent re-entering this 'if' branch
         OnTick();
         if(lastTime != -1)   // completed without error
         {
 lastTime = TimeCurrent();// update timestamp
         }
      }
      Comment("");
   }
}
```

En el manejador *OnTick*, primero comprobamos los parámetros de entrada y convertimos la lista de símbolos en un array de cadenas. Si se encuentran problemas, el signo del error se escribe en *lastTime*: el valor -1, y el procesamiento de los ticks posteriores se interrumpe al principio.

```
void OnTick()
{
   if(lastTime == -1) return; // already had an error, exit

   if(UpdateFrequency <= 0)   // if the update rate is not set
   {
      if(lastTime == iTime(NULL, 0, 0)) return; // waiting for a new bar
   }
   else if(TimeCurrent() - lastTime < UpdateFrequency)
   {
      return;
   }

   const int ns = StringSplit((WorkList == "" ? _Symbol : WorkList), ',', symbols);
   if(ns <= 0)
   {
      Print("Empty symbols");
      lastTime = -1;
      return;
   }

   if(Exposure > 100 || Exposure <= 0)
   {
      Print("Percent of Exposure is incorrect: ", Exposure);
      lastTime = -1;
      return;
   }

   if(RiskLevel > 100 || RiskLevel <= 0)
   {
      Print("Percent of RiskLevel is incorrect: ", RiskLevel);
      lastTime = -1;
      return;
   }
   ...
```

En concreto, se considera un error si los valores de entrada *Exposure* y *Risk Level* están fuera del rango de 0 a 100, como debería ser para los porcentajes. En caso de datos de entrada normales, actualizamos la marca de tiempo, describimos la estructura *LEMLR::SymbolLotExposureRisk* para recibir los indicadores calculados de la función *LEMLR::Estimate* (un símbolo cada uno), así como un array bidimensional *LME* (de «Lot Margin Exposure») para recopilar los indicadores de todos los símbolos.

```
   lastTime = UpdateFrequency > 0 ? TimeCurrent() : iTime(NULL, 0, 0);

   LEMLR::SymbolLotExposureRisk r = {};

   double LME[][13];
   ArrayResize(LME, ns);
   ArrayInitialize(LME, 0);
   ...
```

En un bucle a través de símbolos, llamamos a la función *LEMLR::Estimate* y ejecutamos el array *LME*.

```
   for(int i = 0; i < ns; i++)
   {
      if(!LEMLR::Estimate(Action, symbols[i], Lot, 0,
         Exposure, RiskLevel, RiskPoints, RiskPeriod, Money, r))
      {
        Print("Calc failed (will try on the next bar, or refresh manually)");
        return;
      }

      LME[i][eLot] = r.lot;
      LME[i][eAtrPointsNormalized] = r.atrPointsNormalized;
      LME[i][eAtrValue] = r.atrValue;
      LME[i][eLotFromExposureRaw] = r.lotFromExposureRaw;
      LME[i][eLotFromExposure] = r.lotFromExposure;
      LME[i][eLotFromRiskOfStopLossRaw] = r.lotFromRiskOfStopLossRaw;
      LME[i][eLotFromRiskOfStopLoss] = r.lotFromRiskOfStopLoss;
      LME[i][eExposureFromLot] = r.exposureFromLot;
      LME[i][eMarginLevelFromLot] = r.marginLevelFromLot;
      LME[i][eLotDig] = r.lotDigits;
      LME[i][eMinLot] = SymbolInfoDouble(symbols[i], SYMBOL_VOLUME_MIN);
      LME[i][eContract] = SymbolInfoDouble(symbols[i], SYMBOL_TRADE_CONTRACT_SIZE);
      LME[i][eSymbol] = pack2double(symbols[i]);
   }
   ...
```

Los elementos de la enumeración especial LME_FIELDS se utilizan como índices de array, que proporcionan simultáneamente nombres y números para los indicadores de la estructura.

```
enum LME_FIELDS // 10 fields + 3 additional symbol properties
{
   eLot,
   eAtrPointsNormalized,
   eAtrValue,
   eLotFromExposureRaw,
   eLotFromExposure,
   eLotFromRiskOfStopLossRaw,
   eLotFromRiskOfStopLoss,
   eExposureFromLot,
   eMarginLevelFromLot,
   eLotDig,
   eMinLot,
   eContract,
   eSymbol
};
```

Se añaden como referencia las propiedades SYMBOL_VOLUME_MIN y SYMBOL_TRADE_CONTRACT_SIZE. El nombre del símbolo se «empaqueta» en un valor aproximado del tipo *double* utilizando la función *pack2double*, para posteriormente implementar una ordenación unificada por cualquiera de los campos, incluidos los nombres.

```
double pack2double(const string s)
{
   double r = 0;
   for(int i = 0; i < StringLen(s); i++)
   {
      r = (r * 255) + (StringGetCharacter(s, i) % 255);
   }
   return r;
}
```

En esta etapa podríamos ejecutar ya el Asesor Experto e imprimir los resultados en un registro, algo como lo siguiente:

```
ArrayPrint(LME);
```

No obstante, mirar un registro todo el tiempo no es conveniente. Además, el formateo unificado de valores de distintas columnas, y más aún la presentación de filas «empaquetadas» en *double*, no puede calificarse de fácil de usar. Por lo tanto, se ha desarrollado la clase scoreboard (*Tableau.mqh*) para mostrar una tabla arbitraria en el gráfico. Además de que al preparar una tabla podemos controlar nosotros mismos el formato de cada campo (en el futuro, resaltarlo en un color diferente), esta clase permite ordenar interactivamente la tabla por cualquier columna: el primer clic del ratón ordena en una dirección, el segundo clic ordena en la dirección opuesta, y el tercero cancela la ordenación.

Aquí no describiremos la clase en detalle, pero puede estudiar su código fuente. Sólo es importante señalar que la interfaz se basa en objetos gráficos. De hecho, las celdas de la tabla están formadas por objetos del tipo OBJ_LABEL, y todas sus propiedades son ya familiares para el lector. Sin embargo, algunas de las técnicas utilizadas en el código fuente del scoreboard, en concreto, el trabajo con recursos gráficos y la medición del texto de visualización se presentará más adelante, en la séptima parte.

El constructor de la clase *tableau* toma varios parámetros:

- prefix - prefijo para los nombres de los objetos gráficos creados
- rows - número de filas
- cols - número de columnas
- height - altura de la línea en píxeles (-1 significa el doble del tamaño de la fuente)
- width - anchura de la celda en píxeles
- c - un ángulo del gráfico para anclar objetos
- g - el espacio en píxeles entre celdas
- f - tamaño de letra
- font - nombre de la fuente para las celdas regulares
- bold - el nombre de la fuente en negrita para los títulos
- bgc - color de fondo
- bgt - transparencia de fondo

```
class Tableau
{
public:
   Tableau(const string prefix, const int rows, const int cols,
      const int height = 16, const int width = 100,
      const ENUM_BASE_CORNER c = CORNER_RIGHT_LOWER, const int g = 8,
      const int f = 8, const string font = "Consolas", const string bold = "Arial Bla
      const int mask = TBL_FLAG_COL_0_HEADER,
      const color bgc = 0x808080, const uchar bgt = 0xC0)
      ...
};
```

El usuario puede configurar la mayoría de estos parámetros en las variables de entrada del Asesor Experto de *LotMarginExposureTable.mq5*.

```
input ENUM_BASE_CORNER Corner = CORNER_RIGHT_LOWER;
input int Gap = 16;
input int FontSize = 8;
input string DefaultFontName = "Consolas";
input string TitleFontName = "Arial Black";
input string MotoTypeFontsHint = "Consolas/Courier/Courier New/Lucida Console/Lucida
input color BackgroundColor = 0x808080;
input uchar BackgroundTransparency = 0xC0; // BackgroundTransparency (255 - opaque, 0
```

El número de columnas de la tabla está predeterminado, el número de líneas es igual al número de símbolos, más la línea superior con encabezados.

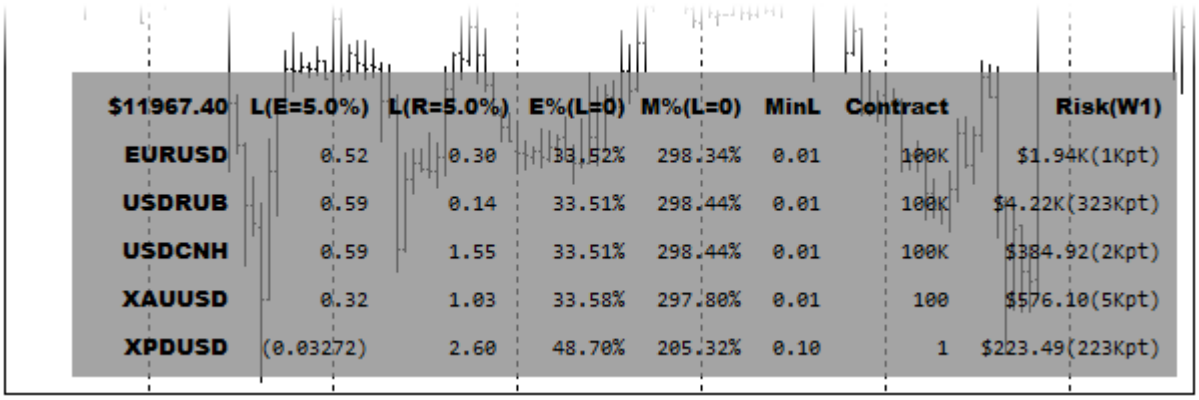
Es importante tener en cuenta que las fuentes para la tabla deben seleccionarse sin letra proporcional, por lo que en la variable *MotoTypeFontsHint* se proporciona una información sobre herramientas con un conjunto de fuentes monoespaciado estándar de Windows.

Los objetos gráficos creados se rellenan utilizando el método *fill* de la clase *Tableau*.

```
bool fill(const string &data[], const string &hint[]) const;
```

Nuestro Asesor Experto pasa el array *data* de cadenas que se obtienen del array *LME* a través de una serie de transformaciones a través de *StringFormat*, así como el array *hint* con información sobre herramientas para los títulos.

En la siguiente imagen se muestra una parte del gráfico con el Asesor Experto en ejecución con la configuración predeterminada, pero con una lista especificada de símbolos «EURUSD,USDRUB,USDCNH,XAUUSD,XPDUSD».



$11967.40	L(E=5.0%)	L(R=5.0%)	E%(L=0)	M%(L=0)	MinL	Contract	Risk(W1)
EURUSD	0.52	0.30	33.52%	298.34%	0.01	100K	$1.94K(1Kpt)
USDRUB	0.59	0.14	33.51%	298.44%	0.01	100K	$4.22K(323Kpt)
USDCNH	0.59	1.55	33.51%	298.44%	0.01	100K	$384.92(2Kpt)
XAUUSD	0.32	1.03	33.58%	297.80%	0.01	100	$576.10(5Kpt)
XPDUSD	(0.03272)	2.60	48.70%	205.32%	0.10	1	$223.49(223Kpt)

Niveles de carga de márgenes y depósito con un lote mínimo para cada símbolo

Los nombres de los símbolos aparecen en la columna de la izquierda. Como encabezamiento de la primera columna se muestra el importe de los fondos (en este caso, libres en la cuenta en el momento actual, ya que en el parámetro de entrada *Money* se deja a 0). Al pasar el ratón por encima del nombre de la columna, aparecerá una información sobre herramientas con una explicación.

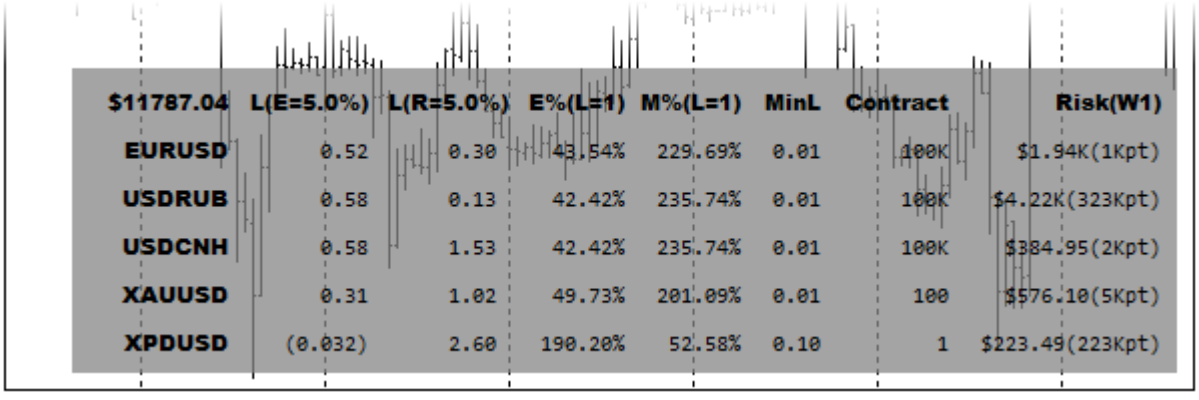
En las siguientes columnas:

- L(E) –lote calculado para cargar el nivel E del depósito del 5 % tras la transacción
- L(R) –lote calculado a riesgo R para el 5 % del depósito después de una operación sin éxito (rango en puntos e importe del riesgo - en la última columna)
- E % –carga de depósito después de la entrada con un lote mínimo
- M % –nivel de margen tras la entrada con el lote mínimo
- MinL –lote mínimo para cada símbolo
- Contrato –tamaño del contrato (1 lote) para cada símbolo
- Riesgo –beneficio/pérdida en dinero al negociar 1 lote y el mismo rango en puntos

En las columnas E % y M %, en este caso, se utilizan los lotes mínimos, ya que el parámetro de entrada *Lot* es 0 (por defecto).

Al cargar un 5 % del depósito, es posible operar con todos los símbolos seleccionados excepto «XPDUSD». Para este último, el volumen resultó ser de 0.03272, inferior al lote mínimo de 0.1, por lo que el resultado figura entre paréntesis. Si permitimos una carga del 20 % (introduzca 20 en el parámetro *Exposure*), obtendremos el lote mínimo para «XPDUSD» 0.1.

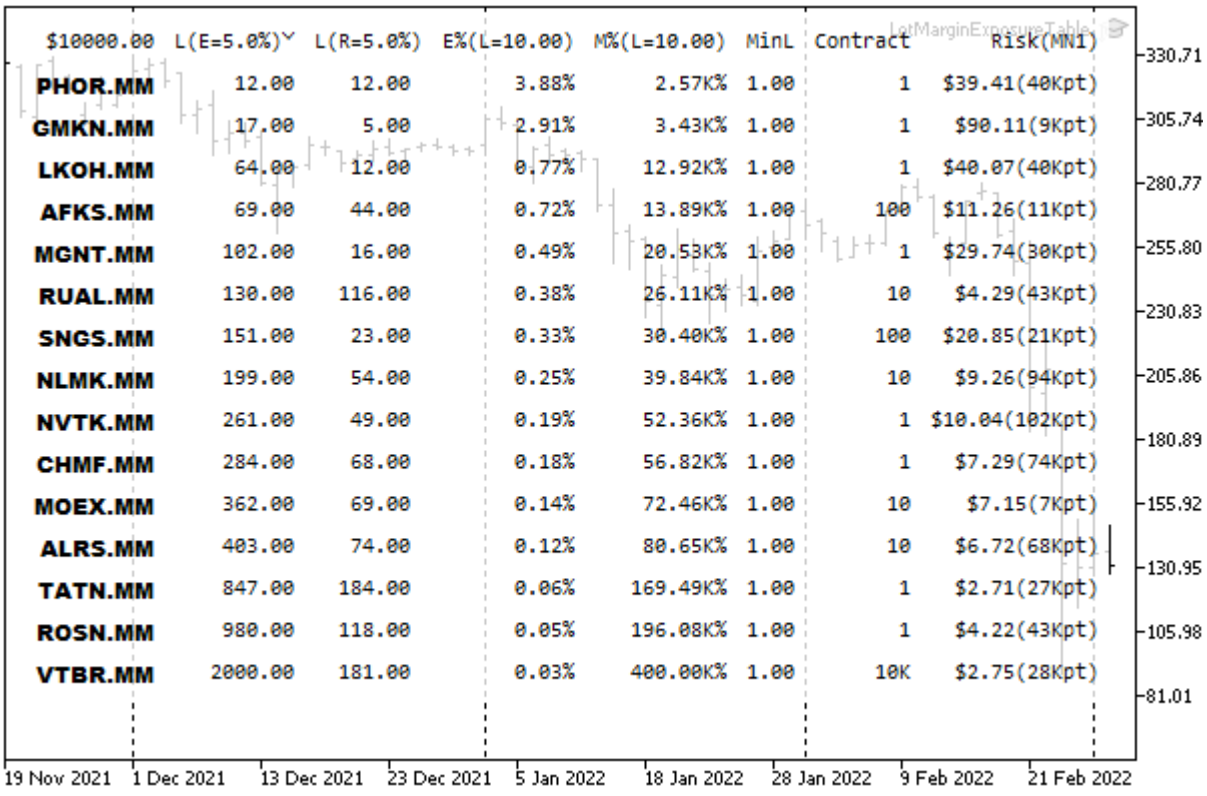
Si introducimos el valor 1 en el parámetro *Lot*, veremos actualizados los valores de las columnas E % y M % de la tabla (aumentará la carga y disminuirá el nivel de margen).



$11787.04	L(E=5.0%)	L(R=5.0%)	E%(L=1)	M%(L=1)	MinL	Contract	Risk(W1)
EURUSD	0.52	0.30	43.54%	229.69%	0.01	100K	$1.94K(1Kpt)
USDRUB	0.58	0.13	42.42%	235.74%	0.01	100K	$4.22K(323Kpt)
USDCNH	0.58	1.53	42.42%	235.74%	0.01	100K	$384.95(2Kpt)
XAUUSD	0.31	1.02	49.73%	201.09%	0.01	100	$576.10(5Kpt)
XPDUSD	(0.032)	2.60	190.20%	52.58%	0.10	1	$223.49(223Kpt)

Niveles de carga de márgenes y depósito con un solo lote para cada símbolo

En la última captura de pantalla que ilustra el trabajo del Asesor Experto se muestra un gran conjunto de blue chips de la bolsa rusa MOEX ordenados por volumen calculado para una carga de depósito del 5 % (2ª columna). Entre los ajustes no estándar, cabe señalar que *Lot=10*, y el periodo de cálculo de la horquilla de precios y el riesgo es igual a MN1. El fondo se hace blanco translúcido, el anclaje se hace a la esquina superior izquierda del gráfico.



Lotes, carga de depósito y nivel de margen para instrumentos MOEX

6.4.8 Estimación del beneficio de una operación de trading: OrderCalcProfit

Una de las funciones de la API de MQL5, *OrderCalcProfit*, permite preevaluar el resultado financiero de una operación de trading si se cumplen las condiciones previstas. Por ejemplo, utilizando esta función puede averiguar la cantidad de beneficios al alcanzar el nivel *Take Profit*, y la cantidad de pérdidas cuando se activa *Stop Loss*.

```
bool OrderCalcProfit(ENUM_ORDER_TYPE action, const string symbol, double volume,
 double openPrice, double closePrice, double &profit)
```

La función calcula el beneficio o la pérdida en la divisa de la cuenta para el entorno de mercado actual basándose en los parámetros pasados.

El tipo de orden se especifica en el parámetro *action*. Sólo se permiten órdenes de mercado ORDER_TYPE_BUY u ORDER_TYPE_SELL de la enumeración ENUM_ORDER_TYPE. El nombre del instrumento financiero y su volumen se pasan en los parámetros *symbol* y *volume*. Los precios de entrada y salida del mercado se fijan mediante los parámetros *openPrice* y *closePrice*, respectivamente. La variable *profit* se pasa por referencia como último parámetro, y en ella se escribirá el valor del beneficio.

La función devuelve un indicador de éxito (*true*) o de error (*false*).

La fórmula de cálculo del resultado financiero utilizada en *OrderCalcProfit* depende del tipo de símbolo.

Identificador	Formula
SYMBOL_CALC_MODE_FOREX	(ClosePrice - OpenPrice) * ContractSize * Lots
SYMBOL_CALC_MODE_FOREX_NO_LEVERAGE	(ClosePrice - OpenPrice) * ContractSize * Lots
SYMBOL_CALC_MODE_CFD	(ClosePrice - OpenPrice) * ContractSize * Lots
SYMBOL_CALC_MODE_CFDINDEX	(ClosePrice - OpenPrice) * ContractSize * Lots
SYMBOL_CALC_MODE_CFDLEVERAGE	(ClosePrice - OpenPrice) * ContractSize * Lots
SYMBOL_CALC_MODE_EXCH_STOCKS	(ClosePrice - OpenPrice) * ContractSize * Lots
SYMBOL_CALC_MODE_EXCH_STOCKS_MOEX	(ClosePrice - OpenPrice) * ContractSize * Lots
SYMBOL_CALC_MODE_FUTURES	(ClosePrice - OpenPrice) * Lots * TickPrice / TickSize
SYMBOL_CALC_MODE_EXCH_FUTURES	(ClosePrice - OpenPrice) * Lots * TickPrice / TickSize
SYMBOL_CALC_MODE_EXCH_FUTURES_FORTS	(ClosePrice - OpenPrice) * Lots * TickPrice / TickSize
SYMBOL_CALC_MODE_EXCH_BONDS	Lots * ContractSize * (ClosePrice * FaceValue + AccruedInterest)
SYMBOL_CALC_MODE_EXCH_BONDS_MOEX	Lots * ContractSize * (ClosePrice * FaceValue + AccruedInterest)
SYMBOL_CALC_MODE_SERV_COLLATERAL	Lots * ContractSize * MarketPrice * LiqudityRate

En las fórmulas se utiliza la siguiente notación:

- Lotes - volumen de la posición en lotes (acciones del contrato)
- ContractSize - tamaño del contrato (un lote, SYMBOL_TRADE_CONTRACT_SIZE)
- TickPrice - precio del tick (SYMBOL_TRADE_TICK_VALUE)
- TickSize - tamaño del tick (SYMBOL_TRADE_TICK_SIZE)
- MarketPrice - último precio de *oferta/compra (Bid/Ask)* conocido según el tipo de transacción
- OpenPrice - precio de apertura de la posición
- ClosePrice - precio de cierre de la posición
- FaceValue - valor nominal del bono (SYMBOL_TRADE_FACE_VALUE)
- LiqudityRate - ratio de liquidez (SYMBOL_TRADE_LIQUIDITY_RATE)
- AccruedInterest - ingresos por cupones acumulados (SYMBOL_TRADE_ACCRUED_INTEREST)

La función *OrderCalcProfit* sólo puede utilizarse en Asesores Expertos y scripts. Para calcular los beneficios/pérdidas potenciales en los indicadores es necesario aplicar un método alternativo, por ejemplo, cálculos independientes mediante fórmulas.

Para eludir la restricción de uso de las funciones *OrderCalcProfit* y *OrderCalcMargin* en indicadores, hemos desarrollado un conjunto de funciones que realizan cálculos utilizando las fórmulas de esta sección, así como la sección Requisitos de margen. Las funciones se encuentran en el archivo de encabezado *MarginProfitMeter.mqh*, dentro del espacio de nombres común *MPM* (de «Margin Profit Meter»).

En concreto, para calcular el resultado financiero, es importante disponer del valor de un punto de un instrumento concreto. En las fórmulas anteriores, participa indirectamente en la diferencia entre los precios de apertura y cierre *(ClosePrice - OpenPrice)*.

La función calcula el valor de un punto de precio *PointValue*.

```
namespace MPM
{
   double PointValue(const string symbol, const bool ask = false,
      const datetime moment = 0)
   {
      const double point = SymbolInfoDouble(symbol, SYMBOL_POINT);
      const double contract = SymbolInfoDouble(symbol, SYMBOL_TRADE_CONTRACT_SIZE);
      const ENUM_SYMBOL_CALC_MODE m =
         (ENUM_SYMBOL_CALC_MODE)SymbolInfoInteger(symbol, SYMBOL_TRADE_CALC_MODE);
      ...
```

Al principio de la función solicitamos todas las propiedades de los símbolos necesarias para el cálculo. A continuación, en función del tipo de símbolo, se obtiene el beneficio/pérdida en la divisa del beneficio de este instrumento. Tenga en cuenta que aquí no hay bonos cuyas fórmulas tengan en cuenta el precio nominal y los ingresos por cupones.

```
      double result = 0;
      switch(m)
      {
      case SYMBOL_CALC_MODE_FOREX_NO_LEVERAGE:
      case SYMBOL_CALC_MODE_FOREX:
      case SYMBOL_CALC_MODE_CFD:
      case SYMBOL_CALC_MODE_CFDINDEX:
      case SYMBOL_CALC_MODE_CFDLEVERAGE:
      case SYMBOL_CALC_MODE_EXCH_STOCKS:
      case SYMBOL_CALC_MODE_EXCH_STOCKS_MOEX:
         result = point * contract;
         break;

      case SYMBOL_CALC_MODE_FUTURES:
      case SYMBOL_CALC_MODE_EXCH_FUTURES:
      case SYMBOL_CALC_MODE_EXCH_FUTURES_FORTS:
         result = point * SymbolInfoDouble(symbol, SYMBOL_TRADE_TICK_VALUE)
            / SymbolInfoDouble(symbol, SYMBOL_TRADE_TICK_SIZE);
         break;
      default:
         PrintFormat("Unsupported symbol %s trade mode: %s", symbol, EnumToString(m))
      }
      ...
```

Por último, convertimos el importe a la divisa de la cuenta, si difiere.

```
      string account = AccountInfoString(ACCOUNT_CURRENCY);
      string current = SymbolInfoString(symbol, SYMBOL_CURRENCY_PROFIT);

      if(current != account)
      {
         if(!Convert(current, account, ask, result, moment)) return 0;
      }

      return result;
   }
   ...
};
```

La función de ayuda *Convert* se utiliza para convertir importes. A su vez, depende de la función *FindExchangeRate*, que busca entre todos los símbolos disponibles uno que contenga el tipo de cambio de la divisa *current* a la moneda *account*.

```
bool Convert(const string current, const string account,
   const bool ask, double &margin, const datetime moment = 0)
{
   string rate;
   int dir = FindExchangeRate(current, account, rate);
   if(dir == +1)
   {
      margin *= moment == 0 ?
         SymbolInfoDouble(rate, ask ? SYMBOL_BID : SYMBOL_ASK) :
         GetHistoricPrice(rate, moment, ask);
   }
   else if(dir == -1)
   {
      margin /= moment == 0 ?
         SymbolInfoDouble(rate, ask ? SYMBOL_ASK : SYMBOL_BID) :
         GetHistoricPrice(rate, moment, ask);
   }
   else
   {
      static bool once = false;
      if(!once)
      {
         Print("Can't convert ", current, " -> ", account);
         once = true;
      }
   }
   return true;
}
```

La función *FindExchangeRate* busca caracteres en *Observación de Mercado* y devuelve el nombre del primer símbolo de Forex que coincida, si hay varios, en el parámetro *result*. Si la cotización corresponde al orden directo de las divisas «*current/account*», la función devolverá +1, y si es al contrario, será «*account/current*», es decir, -1.

```
int FindExchangeRate(const string current, const string account, string &result)
{
   for(int i = 0; i < SymbolsTotal(true); i++)
   {
      const string symbol = SymbolName(i, true);
      const ENUM_SYMBOL_CALC_MODE m =
         (ENUM_SYMBOL_CALC_MODE)SymbolInfoInteger(symbol, SYMBOL_TRADE_CALC_MODE);
      if(m == SYMBOL_CALC_MODE_FOREX || m == SYMBOL_CALC_MODE_FOREX_NO_LEVERAGE)
      {
         string base = SymbolInfoString(symbol, SYMBOL_CURRENCY_BASE);
         string profit = SymbolInfoString(symbol, SYMBOL_CURRENCY_PROFIT);
         if(base == current && profit == account)
         {
            result = symbol;
            return +1;
         }
         else
         if(base == account && profit == current)
         {
            result = symbol;
            return -1;
         }
      }
   }
   return 0;
}
```

El código completo de las funciones se encuentra en el archivo adjunto *MarginProfitMeter.mqh*.

Vamos a comprobar el rendimiento de la función *OrderCalcProfit* y del grupo de funciones *MPM* con un script de prueba *ProfitMeter.mq5*: calcularemos la estimación de ganancias/pérdidas de las operaciones virtuales para todos los símbolos de Observación de Mercado, y lo haremos utilizando dos métodos: el integrado y el nuestro.

En los parámetros de entrada del script, puede seleccionar el tipo de operación *Action* (compra o venta), el tamaño del lote *Lot* y el tiempo de mantenimiento de la posición en barras *Duration*. El resultado financiero se calcula para las cotizaciones de las últimas barras *Duration* del marco temporal actual.

```
#property script_show_inputs

input ENUM_ORDER_TYPE Action = ORDER_TYPE_BUY; // Action (only Buy/Sell allowed)
input float Lot = 1;
input int Duration = 20; // Duration (bar number in past)
```

En el cuerpo del script, conectamos los archivos de encabezado y mostramos el encabezado con los parámetros.

```
#include <MQL5Book/MarginProfitMeter.mqh>
#include <MQL5Book/Periods.mqh>

void OnStart()
{
   // guarantee that the operation will only be a buy or a sell
   ENUM_ORDER_TYPE type = (ENUM_ORDER_TYPE)(Action % 2);
   const string text[] = {"buying", "selling"};
   PrintFormat("Profits/Losses for %s %s lots"
      " of %d symbols in Market Watch on last %d bars %s",
      text[type], (string)Lot, SymbolsTotal(true),
      Duration, PeriodToString(_Period));
   ...
```

A continuación, en un bucle a través de símbolos, realizamos los cálculos de dos maneras e imprimimos los resultados para compararlos.

```
   for(int i = 0; i < SymbolsTotal(true); i++)
   {
      const string symbol = SymbolName(i, true);
      const double enter = iClose(symbol, _Period, Duration);
      const double exit = iClose(symbol, _Period, 0);

      double profit1, profit2; // 2 adopted variables

      // standard method
      if(!OrderCalcProfit(type, symbol, Lot, enter, exit, profit1))
      {
         PrintFormat("OrderCalcProfit(%s) failed: %d", symbol, _LastError);
         continue;
      }

      // our own method
      const int points = (int)MathRound((exit - enter)
         / SymbolInfoDouble(symbol, SYMBOL_POINT));
      profit2 = Lot * points * MPM::PointValue(symbol);
      profit2 = NormalizeDouble(profit2,
         (int)AccountInfoInteger(ACCOUNT_CURRENCY_DIGITS));
      if(type == ORDER_TYPE_SELL) profit2 *= -1;

      // output to the log for comparison
      PrintFormat("%s: %f %f", symbol, profit1, profit2);
   }
}
```

Pruebe a ejecutar el script para diferentes cuentas y conjuntos de instrumentos.

```
Profits/Losses for buying 1.0 lots of 13 symbols in Market Watch on last 20 bars H1
EURUSD: 390.000000 390.000000
GBPUSD: 214.000000 214.000000
USDCHF: -254.270000 -254.270000
USDJPY: -57.930000 -57.930000
USDCNH: -172.570000 -172.570000
USDRUB: 493.360000 493.360000
AUDUSD: 84.000000 84.000000
NZDUSD: 13.000000 13.000000
USDCAD: -97.480000 -97.480000
USDSEK: -682.910000 -682.910000
XAUUSD: -1706.000000 -1706.000000
SP500m: 5300.000000 5300.000000
XPDUSD: -84.030000 -84.030000
```

Lo ideal es que los números de cada línea coincidan.

6.4.9 Estructura MqlTradeRequest

Las funciones de trading de la API de MQL5, en concreto *OrderCheck* y *OrderSend*, operan sobre varias estructuras integradas. Por lo tanto, tendremos que considerar estas estructuras antes de pasar a las funciones propiamente dichas.

Empecemos por la estructura *MqlTradeRequest*, que contiene todos los campos necesarios para ejecutar operaciones.

```
struct MqlTradeRequest
{
   ENUM_TRADE_REQUEST_ACTIONS action;       // Type of action to perform
   ulong                      magic;        // Unique Expert Advisor number
   ulong                      order;        // Order ticket
   string                     symbol;       // Name of the trading instrument
   double                     volume;       // Requested trade volume in lots
   double                     price;        // Price
   double                     stoplimit;    // StopLimit order level
   double                     sl;           // Stop Loss order level
   double                     tp;           // Take Profit order level
   ulong                      deviation;    // Maximum deviation from the given price
   ENUM_ORDER_TYPE            type;         // Order type
   ENUM_ORDER_TYPE_FILLING    type_filling; // Order type by execution
   ENUM_ORDER_TYPE_TIME       type_time;    // Order type by duration
   datetime                   expiration;   // Order expiration date
   string                     comment;      // Comment to the order
   ulong                      position;     // Position ticket
   ulong                      position_by;  // Opposite position ticket
};
```

No tenga miedo a que haya un gran número de campos: la estructura está diseñada para servir absolutamente a todos los tipos posibles de solicitudes comerciales; no obstante, sólo se suelen utilizar unos pocos campos para cada caso concreto.

Antes de rellenar los campos se recomienda anular la estructura mediante una inicialización explícita en su definición o llamando a la función *ZeroMemory*.

```
MqlTradeRequest request = {};
...
ZeroMemory(request);
```

De esta forma se evitarán posibles errores y efectos secundarios de pasar valores aleatorios a las funciones de la API en aquellos campos que no fueron asignados explícitamente.

En la siguiente tabla se ofrece una breve descripción de los campos, que veremos cómo rellenar cuando describamos las operaciones de trading.

Campo	Descripción
action	Tipo de operación de trading de ENUM_TRADE_REQUEST_ACTIONS
magic	ID de experto (opcional)
order	Ticket de orden pendiente cuya modificación se solicita
symbol	Nombre del instrumento de trading
volume	Volumen solicitado en lotes
price	El precio al que debe ejecutarse la orden
stoplimit	El precio al que se colocará una orden Limit cuando se activen las órdenes ORDER_TYPE_BUY_STOP_LIMIT y ORDER_TYPE_SELL_STOP_LIMIT.
sl	Precio al que se activará la orden *Stop Loss* cuando el precio se mueva en una dirección desfavorable.
tp	Precio al que se activará la orden *Take Profit* cuando el precio se mueva en una dirección favorable.
deviation	Desviación máxima aceptable del precio solicitado, en puntos
type	Tipo de orden de ENUM_ORDER_TYPE
type_filling	Tipo de llenado de orden de ENUM_ORDER_TYPE_FILLING
type_time	Tipo de vencimiento de la orden pendiente de ENUM_ORDER_TYPE_TIME
expiration	Fecha de vencimiento de órdenes pendientes
comment	Comentario a la orden
position	Ticket de posición
position_by	Ticket de posición opuesta para la operación TRADE_ACTION_CLOSE_BY

Para enviar órdenes de operaciones de trading, es necesario rellenar un conjunto diferente de campos, dependiendo de la naturaleza de la operación. Algunos campos son obligatorios y otros opcionales

(pueden omitirse al rellenarlos). A continuación, examinaremos más detenidamente los requisitos de campo en el contexto de acciones específicas.

El programa puede comprobar si una estructura *MqlTradeRequest* formada es correcta utilizando la función *OrderCheck* o enviarlo al servidor mediante la función *OrderSend*. Si tiene éxito, se realizará la operación solicitada.

El campo *action* es el único obligatorio para todas las actividades de trading.

Un número único en el campo *magic* suele indicarse sólo para solicitudes de compra/venta en el mercado o al crear una nueva orden pendiente. Esto conduce al posterior marcado de las transacciones y posiciones completadas con este número, lo que permite organizar el tratamiento analítico de las acciones de trading. Al modificar los niveles de precios de una posición o de órdenes pendientes, así como al eliminarlas, este campo no tiene ningún efecto.

Al realizar operaciones comerciales manualmente desde la interfaz de MetaTrader 5, el identificador mágico no se puede establecer, y por lo tanto es igual a cero. Esto proporciona una forma popular, aunque no del todo fiable, de distinguir entre el trading manual y el automatizado al analizar el historial. De hecho, los Asesores Expertos también pueden utilizar un identificador cero. Por lo tanto, para saber quién y cómo ha realizado determinadas acciones de trading, utilice las propiedades correspondientes de las órdenes (ORDER_REASON), transacciones (DEAL_REASON) y posiciones (POSITION_REASON).

Cada Asesor Experto puede establecer su propio ID único o incluso utilizar varios ID para diferentes propósitos (desglosados por estrategias de trading, señales, etc.). El número *magic* de la posición corresponde al número *magic* de la última operación implicada en la formación de la posición.

El nombre del símbolo en el campo *symbol* sólo es importante para abrir o aumentar posiciones, así como al colocar órdenes pendientes. En los casos de modificación y cierre de órdenes y posiciones, se ignorará, pero aquí hay una pequeña excepción. Dado que en las cuentas de compensación sólo puede existir una posición por cada símbolo, el campo *symbol* puede utilizarse para identificar una posición en una solicitud de modificación de sus niveles de precios de protección (*Stop Loss* y *Take Profit*).

El campo *volume* se utiliza de la misma manera: es necesario en órdenes de compra/venta inmediatas o al crear órdenes pendientes. Hay que tener en cuenta que el volumen real en la operación dependerá del modo de ejecución y puede diferir de lo solicitado.

El campo *price* también tiene algunas limitaciones: al enviar órdenes de mercado (TRADE_ACTION_DEAL en *action*) para instrumentos con modo de ejecución SYMBOL_TRADE_EXECUTION_MARKET o SYMBOL_TRADE_EXECUTION_EXCHANGE, este campo se ignora.

El campo *stoplimit* sólo tiene sentido cuando se establecen órdenes Stop-Limit, es decir, cuando el campo *type* contiene ORDER_TYPE_BUY_STOP_LIMIT u ORDER_TYPE_SELL_STOP_LIMIT. Este especifica el precio al que se colocará una orden Limit pendiente cuando el precio alcance el valor *price* (el servidor de MetaTrader 5 hace un seguimiento de este hecho, y por el momento una orden pendiente no se muestra en el sistema de trading).

Al colocar órdenes pendientes, sus reglas de vencimiento se establecen en un par de campos: *type_time* y *expiration*. Este último contiene un valor de tipo *datetime*, que sólo se tiene en cuenta si *type_time* es igual a ORDER_TIME_SPECIFIED u ORDER_TIME_SPECIFIED_DAY.

Por último, los dos últimos campos están relacionados con la identificación de posiciones en las consultas. Cada nueva posición creada a partir de órdenes (manual o programáticamente) obtiene un ticket asignado por el sistema, un número único. Por regla general, corresponde al ticket de la orden,

como resultado de la cual se abre la posición, pero puede cambiar sujeto a operaciones de servicio en el servidor, como por ejemplo, acumulación de swaps por reapertura de una posición.

Hablaremos de la obtención de las propiedades de posiciones, transacciones y órdenes en secciones separadas. Aquí, por ahora, es importante para nosotros que el campo *position* se rellene al cambiar y cerrar una posición para identificarla inequívocamente. En teoría, en las cuentas de compensación, basta con indicar el símbolo de posición en el campo *symbol*, pero para la unificación de los algoritmos, es mejor dejar el campo *position* en el trabajo.

El campo *position_by* se utiliza para cerrar posiciones opuestas (TRADE_ACTION_CLOSE_BY). Debe indicar una posición abierta para el mismo símbolo pero en sentido opuesto en relación con *position* (esto sólo es posible en cuentas de cobertura).

El campo *deviation* afecta a la ejecución de órdenes de mercado sólo en los modos Ejecución Instantánea y Solicitud de Ejecución.

En las secciones correspondientes se darán ejemplos de cómo rellenar la estructura para operaciones de trading de cada tipo.

6.4.10 Estructura MqlTradeCheckResult

Antes de enviar una solicitud de operación de trading al servidor se recomienda comprobar que la misma se ha completado sin errores formales. El control lo realiza la función *OrderCheck*, a la que pasamos la solicitud en la estructura *MqlTradeRequest* y la variable receptora del tipo de estructura *MqlTradeCheckResult*.

Además de la corrección de la solicitud, la estructura permite evaluar el estado de la cuenta tras la ejecución de una operación de trading; en concreto, el saldo, los fondos y el margen.

```
struct MqlTradeCheckResult
{
   uint   retcode;      // Response code
   double balance;      // Balance after the transaction
   double equity;       // Equity after the transaction
   double profit;       // Floating profit
   double margin;       // Margin requirements
   double margin_free;  // Free margin
   double margin_level; // Margin level
   string comment;      // Comment to the response code (description of the error)
};
```

En la siguiente tabla se describen los campos:

Campo	Descripción
retcode	Código de retorno asumido
balance	El valor del saldo, que se observará tras la ejecución de la operación de trading.
equity	El valor de los fondos propios, que se observará tras la ejecución de la operación de trading.
profit	El valor del beneficio flotante, que se observará tras la ejecución de la operación de trading.
margin	Margen total bloqueado tras una operación
margin_free	El volumen de fondos propios libres que quedarán tras la ejecución de la operación de trading
margin_level	El nivel de margen que se fijará tras la ejecución de una operación de trading.
comment	Comentario sobre el código de respuesta, descripción del error

En la estructura que se rellena llamando a *OrderCheck*, el campo *retcode* contendrá un código de resultado de entre los que admite la plataforma para procesar las solicitudes de operaciones reales y pone en un campo *retcode* similar de la estructura *MqlTradeResult* después de llamar a las funciones de trading *OrderSend* y *OrderSendAsync*.

Las constantes de código de retorno se presentan en Documentación MQL5. Para su salida más visual al registro cuando se depuran Asesores Expertos, la enumeración aplicada TRADE_RETCODE se definió en el archivo *TradeRetcode.mqh*. Todos sus elementos tienen identificadores que coinciden con las constantes integradas, pero sin el prefijo común «TRADE_RETCODE_». Por ejemplo:

```
enum TRADE_RETCODE
{
   OK_0           = 0,      // no standard constant
   REQUOTE        = 10004,  // TRADE_RETCODE_REQUOTE
   REJECT         = 10006,  // TRADE_RETCODE_REJECT
   CANCEL         = 10007,  // TRADE_RETCODE_CANCEL
   PLACED         = 10008,  // TRADE_RETCODE_PLACED
   DONE           = 10009,  // TRADE_RETCODE_DONE
   DONE_PARTIAL   = 10010,  // TRADE_RETCODE_DONE_PARTIAL
   ERROR          = 10011,  // TRADE_RETCODE_ERROR
   TIMEOUT        = 10012,  // TRADE_RETCODE_TIMEOUT
   INVALID        = 10013,  // TRADE_RETCODE_INVALID
   INVALID_VOLUME = 10014,  // TRADE_RETCODE_INVALID_VOLUME
   INVALID_PRICE  = 10015,  // TRADE_RETCODE_INVALID_PRICE
   INVALID_STOPS  = 10016,  // TRADE_RETCODE_INVALID_STOPS
   TRADE_DISABLED = 10017,  // TRADE_RETCODE_TRADE_DISABLED
   MARKET_CLOSED  = 10018,  // TRADE_RETCODE_MARKET_CLOSED
   ...
};
```

```
#define TRCSTR(X) EnumToString((TRADE_RETCODE)(X))
```

Así, el uso de *TRCSTR(r.retcode)*, donde *r* es una estructura, proporcionará una descripción mínima del código numérico.

Consideraremos un ejemplo de aplicación de una macro y de análisis de una estructura en la sección siguiente acerca de la función *OrderCheck*.

6.4.11 Solicitar validación: OrderCheck

Para realizar cualquier operación de trading, el programa MQL debe rellenar primero la estructura *MqlTradeRequest* con los datos necesarios. Antes de enviarla al servidor mediante funciones de trading, tiene sentido comprobar su corrección formal y evaluar las consecuencias de la solicitud; en concreto, la cantidad de margen que se requerirá y los fondos libres restantes. Esta comprobación la realiza la función *OrderCheck*.

bool OrderCheck(const MqlTradeRequest &request, MqlTradeCheckResult &result)

Si no hay fondos suficientes o si los parámetros se rellenan incorrectamente, la función devuelve *false*. Además, la función también reacciona con un rechazo cuando se desactiva el trading, tanto en el terminal en su conjunto como para un programa específico. Para el código de error, compruebe el campo *retcode* de la estructura *result*.

La comprobación satisfactoria de la estructura *request* y el entorno de trading finaliza con el estado *true*; sin embargo, esto no garantiza que la operación solicitada vaya a tener éxito con toda seguridad si se repite utilizando las funciones *OrderSend* o *OrderSendAsync*. Las condiciones de trading pueden cambiar entre llamadas o el bróker del servidor puede tener ajustes aplicados para un sistema de trading externo específico que no puedan satisfacerse en el algoritmo de verificación formal que realiza *OrderCheck*.

Para obtener una descripción del resultado financiero previsto, debe analizar los campos de la estructura *result*.

A diferencia de la función *OrderCalcMargin*, que calcula el margen estimado necesario para una única posición u orden propuesta, *OrderCheck* tiene en cuenta, aunque de modo simplificado, el estado general de la cuenta de trading. Así, rellena el campo *margin* de la estructura *MqlTradeCheckResult* y otros campos relacionados (*margin_free*, *margin_level*) con variables acumulativas que se formarán tras la ejecución de la orden. Por ejemplo, si una posición ya está abierta para cualquier instrumento en el momento de la llamada a *OrderCheck* y la solicitud que se está comprobando aumenta la posición, el campo *margin* reflejará el importe del depósito, incluidas las obligaciones de margen anteriores. Si la nueva orden contiene una operación en sentido opuesto, el margen no aumentará (en realidad, debería disminuir, porque una posición debería cerrarse completamente en una cuenta de compensación y el margen de cobertura debería aplicarse para posiciones opuestas en una cuenta de cobertura; sin embargo, la función no realiza cálculos tan precisos).

En primer lugar, *OrderCheck* es útil para los programadores que se encuentran en la fase inicial de familiarización con la API de trading, a fin de experimentar con las solicitudes sin enviarlas al servidor.

Vamos a probar el rendimiento de la función *OrderCheck* utilizando un simple Asesor Experto no de trading *CustomOrderCheck.mq5*. Lo hemos convertido en un Asesor Experto y no en un script para facilitar su uso: de esta manera permanecerá en el gráfico después de ser lanzado con la configuración actual, que se puede editar fácilmente cambiando los parámetros de entrada individuales. Con un

script, tendríamos que empezar de nuevo configurando los campos cada vez a partir de los valores por defecto.

Para ejecutar la comprobación, pongamos un temporizador en *OnInit*.

```
void OnInit()
{
   // initiate pending execution
   EventSetTimer(1);
}
```

En cuanto al manejador del temporizador, el algoritmo principal se implementará allí. Al principio cancelamos el temporizador ya que necesitamos que el código se ejecute una vez, y luego esperamos a que el usuario cambie los parámetros.

```
void OnTimer()
{
   // execute the code once and wait for new user settings
   EventKillTimer();
   ...
}
```

Los parámetros de entrada del Asesor Experto repiten completamente el conjunto de campos de la estructura de solicitud de operación.

```
input ENUM_TRADE_REQUEST_ACTIONS Action = TRADE_ACTION_DEAL;
input ulong Magic;
input ulong Order;
input string Symbol;     // Symbol (empty = current _Symbol)
input double Volume;     // Volume (0 = minimal lot)
input double Price;      // Price (0 = current Ask)
input double StopLimit;
input double SL;
input double TP;
input ulong Deviation;
input ENUM_ORDER_TYPE Type;
input ENUM_ORDER_TYPE_FILLING Filling;
input ENUM_ORDER_TYPE_TIME ExpirationType;
input datetime ExpirationTime;
input string Comment;
input ulong Position;
input ulong PositionBy;
```

Muchos de ellos no afectan a la comprobación ni a los resultados financieros, pero se dejan para que pueda estar seguro de ello.

Por defecto, el estado de las variables corresponde a la solicitud de abrir una posición con el lote mínimo del instrumento actual. En concreto, el parámetro *Type* sin inicialización explícita obtendrá el valor 0, que es igual al miembro ORDER_TYPE_BUY de la estructura ENUM_ORDER_TYPE. En el parámetro *Action*, especificamos una inicialización explícita porque 0 no corresponde a ningún elemento de la enumeración ENUM_TRADE_REQUEST_ACTIONS (el primer elemento de TRADE_ACTION_DEAL es 1).

```
void OnTimer()
{
   ...
   // initialize structures with zeros
   MqlTradeRequest request = {};
   MqlTradeCheckResult result = {};

   // default values
   const bool kindOfBuy = (Type & 1) == 0;
   const string symbol = StringLen(Symbol) == 0 ? _Symbol : Symbol;
   const double volume = Volume == 0 ?
      SymbolInfoDouble(symbol, SYMBOL_VOLUME_MIN) : Volume;
   const double price = Price == 0 ?
      SymbolInfoDouble(symbol, kindOfBuy ? SYMBOL_ASK : SYMBOL_BID) : Price;
   ...
```

Completemos la estructura. Los robots reales normalmente sólo necesitan asignar unos pocos campos, pero como esta prueba es genérica, debemos asegurarnos de que se pasen todos los parámetros que introduzca el usuario.

```
   request.action = Action;
   request.magic = Magic;
   request.order = Order;
   request.symbol = symbol;
   request.volume = volume;
   request.price = price;
   request.stoplimit = StopLimit;
   request.sl = SL;
   request.tp = TP;
   request.deviation = Deviation;
   request.type = Type;
   request.type_filling = Filling;
   request.type_time = ExpirationType;
   request.expiration = ExpirationTime;
   request.comment = Comment;
   request.position = Position;
   request.position_by = PositionBy;
   ...
```

Tenga en cuenta que aquí aún no normalizamos precios ni lotes, aunque ello es necesario en el programa real. Así, esta prueba permite introducir valores «desiguales» y asegurarse de que conducen a un error. En los siguientes ejemplos se activará la normalización.

A continuación, llamamos a *OrderCheck* y registramos las estructuras *request* y *result*. Sólo nos interesa el campo *retcode* de este último, por lo que se imprime adicionalmente con «descifrado» como texto, macro TRCSTR (*TradeRetcode.mqh*). También puede analizar un campo de cadena *comment*, pero su formato puede cambiar para que sea más adecuado para mostrarlo al usuario.

```
   ResetLastError();
   PRTF(OrderCheck(request, result));
   StructPrint(request, ARRAYPRINT_HEADER);
   Print(TRCSTR(result.retcode));
   StructPrint(result, ARRAYPRINT_HEADER, 2);
   ...
```

La salida de estructuras la proporciona una función auxiliar *StructPrint* basada en *ArrayPrint*. Por ello, seguiremos obteniendo una visualización «en bruto» de los datos. En concreto, los elementos de las enumeraciones se representan mediante números «tal cual». Más adelante desarrollaremos una función para que la salida de la estructura de *MqlTradeRequest* sea más transparente (fácil de usar) (véase *TradeUtils.mqh*).

Para facilitar el análisis de los resultados, al principio de la función *OnTimer* mostraremos el estado actual de la cuenta, y al final, para comparar, calcularemos el margen para una operación de trading determinada utilizando la función *OrderCalcMargin*.

```
void OnTimer()
{
   PRTF(AccountInfoDouble(ACCOUNT_EQUITY));
   PRTF(AccountInfoDouble(ACCOUNT_PROFIT));
   PRTF(AccountInfoDouble(ACCOUNT_MARGIN));
   PRTF(AccountInfoDouble(ACCOUNT_MARGIN_FREE));
   PRTF(AccountInfoDouble(ACCOUNT_MARGIN_LEVEL));
   ...
   // filling in the structure MqlTradeRequest
   // calling OrderCheck and printing results
   ...
   double margin = 0;
   ResetLastError();
   PRTF(OrderCalcMargin(Type, symbol, volume, price, margin));
   PRTF(margin);
}
```

A continuación se muestra un ejemplo de registros para XAUUSD con la configuración predeterminada:

```
AccountInfoDouble(ACCOUNT_EQUITY)=15565.22 / ok
AccountInfoDouble(ACCOUNT_PROFIT)=0.0 / ok
AccountInfoDouble(ACCOUNT_MARGIN)=0.0 / ok
AccountInfoDouble(ACCOUNT_MARGIN_FREE)=15565.22 / ok
AccountInfoDouble(ACCOUNT_MARGIN_LEVEL)=0.0 / ok
OrderCheck(request,result)=true / ok
[action] [magic] [order] [symbol] [volume] [price] [stoplimit] [sl] [tp] [deviation]
1 0 0 "XAUUSD" 0.01 1899.97 0.00 0.00 0.00 0 0 »
» [type_filling] [type_time] [expiration] [comment] [position] [position_by] [reserve
» 0 0 1970.01.01 00:00:00 "" 0 0 0
OK_0
[retcode] [balance] [equity] [profit] [margin] [margin_free] [margin_level] [comment]
0 15565.22 15565.22 0.00 19.00 15546.22 81922.21 "Done" 0
OrderCalcMargin(Type,symbol,volume,price,margin)=true / ok
margin=19.0 / ok
```

En el siguiente ejemplo se muestra una estimación del aumento previsto del margen en la cuenta, en la que ya existe una posición abierta que vamos a doblar.

```
AccountInfoDouble(ACCOUNT_EQUITY)=9999.540000000001 / ok
AccountInfoDouble(ACCOUNT_PROFIT)=-0.83 / ok
AccountInfoDouble(ACCOUNT_MARGIN)=79.22 / ok
AccountInfoDouble(ACCOUNT_MARGIN_FREE)=9920.32 / ok
AccountInfoDouble(ACCOUNT_MARGIN_LEVEL)=12622.49431961626 / ok
OrderCheck(request,result)=true / ok
[action] [magic] [order] [symbol] [volume] [price] [stoplimit] [sl] [tp] [deviation]
1 0 0 "PLZL.MM" 1.0 12642.0 0.0 0.0 0.0 0 0 »
» [type_filling] [type_time] [expiration] [comment] [position] [position_by] [reserve
» 0 0 1970.01.01 00:00:00 "" 0 0 0
OK_0
[retcode] [balance] [equity] [profit] [margin] [margin_free] [margin_level] [comment]
0 10000.87 9999.54 -0.83 158.26 9841.28 6318.43 "Done" 0
OrderCalcMargin(Type,symbol,volume,price,margin)=true / ok
margin=79.04000000000001 / ok
```

Pruebe a cambiar cualquier parámetro de la solicitud y compruebe si la solicitud tiene éxito. Las combinaciones incorrectas de parámetros provocarán códigos de error de la lista estándar, pero como hay muchas más opciones no válidas que reservadas (los errores más comunes), la función puede devolver a menudo el código genérico TRADE_RETCODE_INVALID (10013). A este respecto, se recomienda que emplee sus propias comprobaciones de estructura con un mayor grado de diagnóstico.

Al enviar solicitudes reales al servidor se utiliza el mismo código TRADE_RETCODE_INVALID en diversas circunstancias imprevistas; por ejemplo, al intentar reeditar una orden cuya operación de modificación ya se ha iniciado (pero aún no ha finalizado) en el sistema de trading externo.

6.4.12 Solicitar resultado del envío: estructura MqlTradeResult

En respuesta a una solicitud de operación ejecutada por las funciones *OrderSend* u *OrderSendAsync* que veremos en la siguiente sección, el servidor devuelve los resultados del procesamiento de la solicitud. Para ello se utiliza una estructura especial *MqlTradeResult* predefinida.

```
struct MqlTradeResult
{
   uint     retcode;          // Operation result code
   ulong    deal;             // Transaction ticket, if it is completed
   ulong    order;            // Order ticket, if it is placed
   double   volume;           // Trade volume confirmed by the broker
   double   price;            // Trade price confirmed by the broker
   double   bid;              // Current market bid price
   double   ask;              // Current market offer price
   string   comment;          // Broker's comment on the operation
   uint     request_id;       // Request identifier, set by the terminal when sending
   uint     retcode_external; // Response code of the external trading system
};
```

En la siguiente tabla se describen sus campos:

Campo	Descripción
retcode	Código de retorno del servidor de trading
deal	Ticket de transacción si se realiza (durante la operación de trading TRADE_ACTION_DEAL)
order	Ticket de orden si se realiza (durante la operación de trading TRADE_ACTION_PENDING)
volume	Volumen de trading confirmado por el bróker (depende de los modos de ejecución de la orden)
price	El precio de la transacción confirmado por el bróker (depende del campo *deviation* de la solicitud de operación, modo de ejecución y la operación de trading)
bid	Precio de oferta actual del mercado
ask	Precio de demanda actual del mercado
comment	Comentario del bróker sobre la operación (por defecto, se rellena con la desencriptación del código de retorno del servidor de trading)
request_id	ID de la solicitud que establece el terminal al enviarla al servidor de trading.
retcode_external	Código de error devuelto por el sistema de trading externo

Como veremos más adelante al realizar las operaciones de trading se pasa una variable de tipo *MqlTradeResult* como segundo parámetro por referencia en la función *OrderSend* u *OrderSendAsync*. Devuelve el resultado.

Al enviar una solicitud de operación al servidor, el terminal establece el identificador *request_id* en un valor único. Esto es necesario para el análisis de los eventos de trading posteriores, lo cual es necesario si se utiliza una función asíncrona *OrderSendAsync*. Este identificador permite asociar la solicitud enviada con el resultado de su procesamiento pasado al manejador de eventos *OnTradeTransaction*.

La presencia y los tipos de errores en el campo *retcode_external* dependen del bróker y del sistema de trading externo al que se envían las operaciones de trading.

Los resultados de las solicitudes se analizan de distintas maneras, en función de las operaciones de trading y de la forma en que se envían. Nos ocuparemos de ello en secciones posteriores sobre acciones específicas: compra y venta en el mercado, colocación y eliminación de órdenes pendientes, y modificación y cierre de posiciones.

6.4.13 Enviar una solicitud de trading: OrderSend y OrderSendAsync

Para realizar operaciones de trading, la API de MQL5 proporciona dos funciones: *OrderSend* y *OrderSendAsync*. Al igual que *OrderCheck*, realizan una comprobación formal de los parámetros de la solicitud pasados en forma de la estructura *MqlTradeRequest* y, a continuación, si tiene éxito, envía una solicitud al servidor.

La diferencia entre ambas funciones es la siguiente. *OrderSend* espera que la orden se ponga en cola para ser procesado en el servidor y recibe de éste datos significativos en los campos de la estructura *MqlTradeResult* que se pasa como segundo parámetro de la función. *OrderSendAsync* devuelve inmediatamente el control al código de llamada independientemente de cómo responda el servidor. Al mismo tiempo, de todos los campos de la estructura *MqlTradeResult* excepto *retcode*, la información importante se rellena sólo en *request_id*. Utilizando este identificador de solicitud, un programa MQL puede recibir más información sobre el progreso del procesamiento de esta solicitud en el evento *OnTradeTransaction*. Un enfoque alternativo consiste en analizar periódicamente las listas de órdenes, transacciones y posiciones. Esto también puede hacerse en bucle, estableciendo un tiempo de espera en caso de problemas de comunicación.

Es importante señalar que, a pesar del sufijo «Async» en el nombre de la segunda función, la primera función sin este sufijo tampoco es totalmente síncrona. El hecho es que el resultado del procesamiento de órdenes por el servidor, en concreto, la ejecución de una transacción (o, probablemente, varias transacciones basadas en una orden) y la apertura de una posición, generalmente se produce de forma asíncrona en un sistema de trading externo. Por lo tanto, la función *OrderSend* también requiere la recopilación y el análisis en diferido de las consecuencias de la ejecución de la solicitud, que los programas MQL deben, si es necesario, implementar ellos mismos. Más adelante veremos un ejemplo de envío verdaderamente síncrono de una solicitud y recepción de todos sus resultados (véase *MqlTradeSync.mqh*).

bool OrderSend(const MqlTradeRequest &request, MqlTradeResult &result)

La función devuelve *true* en caso de una comprobación básica correcta de la estructura *request* en el terminal y algunas comprobaciones adicionales en el servidor. Sin embargo, esto sólo indica la aceptación de la orden por parte del servidor y no garantiza una ejecución satisfactoria de la operación de trading.

El servidor de trading puede rellenar los valores del campo *deal* o *order* en la estructura *result* devuelta si estos datos son conocidos en el momento en que el servidor formatea una respuesta a la llamada *OrderSend*. Sin embargo, en el caso general, los eventos de ejecución de transacciones o colocación de órdenes Limit correspondientes a una orden pueden producirse después de que se envíe la respuesta al programa MQL en el terminal. Por lo tanto, para cualquier tipo de solicitud de trading, al recibir el resultado de la ejecución *OrderSend*, es necesario comprobar el código de retorno del servidor de trading *retcode* y el código de respuesta del sistema de trading externo *retcode_external* (si es necesario) que están disponibles en la estructura *result* devuelta. Basándose en ellos, debe decidir si esperar a las acciones pendientes en el servidor o emprender sus propias acciones.

Cada orden aceptada se almacena en el servidor de trading a la espera de ser procesada hasta que se produzca alguno de los siguientes eventos que afectan a su ciclo de vida:

- ejecución cuando aparece una solicitud de contador
- se activa cuando llega el precio de ejecución
- fecha de vencimiento
- cancelación por el usuario o el programa MQL
- eliminación por el bróker (por ejemplo, en caso de compensación o escasez de fondos, *Stop Out*)

El prototipo *OrderSendAsync* repite por completo el de *OrderSend*.

bool OrderSendAsync(const MqlTradeRequest &request, MqlTradeResult &result)

La función está pensada para trading de alta frecuencia, cuando, según las condiciones del algoritmo, es inaceptable perder tiempo esperando una respuesta del servidor. El uso de *OrderSendAsync* no

acelera el procesamiento de solicitudes por parte del servidor ni el envío de solicitudes al sistema de trading externo.

> ¡Atención! En el probador, la función *OrderSendAsync* funciona como *OrderSend*, lo cual dificulta la depuración del procesamiento pendiente de solicitudes asíncronas.

La función devuelve *true* al enviar correctamente la solicitud al servidor de MetaTrader 5. Sin embargo, esto no significa que la solicitud haya llegado al servidor y haya sido aceptada para su procesamiento. Al mismo tiempo, el código de respuesta de la estructura *result* receptora contiene el valor TRADE_RETCODE_PLACED (10008), es decir, «la orden se ha realizado».

Al procesar la solicitud recibida, el servidor enviará un mensaje de respuesta al terminal sobre un cambio en el estado actual de las posiciones, órdenes y transacciones, lo que lleva a la generación del evento *OnTrade* en un programa MQL. Allí, el programa puede analizar el nuevo entorno de trading y el historial de la cuenta. A continuación veremos ejemplos relevantes.

Además, los detalles de la ejecución de la solicitud del operador en el servidor pueden rastrearse utilizando el manejador *OnTradeTransaction*. Al mismo tiempo, debe tenerse en cuenta que, como resultado de la ejecución de una solicitud de trading, el manejador *OnTradeTransaction* será llamado varias veces. Por ejemplo, al enviar una solicitud de compra de mercado, ésta es aceptada para su procesamiento por el servidor, se crea la correspondiente orden de «compra» para la cuenta, se ejecuta la orden y se realiza la operación, como resultado de lo cual se elimina de la lista de órdenes abiertas y se añade al historial de órdenes. A continuación, la operación se añade al historial y se crea una nueva posición. Para cada uno de estos eventos, se llamará a la función *OnTradeTransaction*.

Empecemos con un ejemplo sencillo de Asesor Experto *CustomOrderSend.mq5*. Permite establecer todos los campos de la solicitud en los parámetros de entrada, lo que es similar a *CustomOrderCheck.mq5*, pero además difiere en que envía una solicitud al servidor en lugar de una simple comprobación en el terminal. Ejecute el Asesor Experto en su cuenta demo. Después de completar los experimentos, no olvide eliminar el Asesor Experto del gráfico o cerrar el gráfico para no enviar una solicitud de prueba cada vez que vuelva a iniciar el terminal.

El nuevo ejemplo presenta otras mejoras. En primer lugar, se añade el parámetro de entrada *Async*.

```
input bool Async = false;
```

Esta opción permite seleccionar la función que enviará la solicitud al servidor. Por defecto, el parámetro es igual a *false* y se utiliza la función *OrderSend*. Si se establece en *true*, se llamará a *OrderSendAsync*.

Además, con este ejemplo, empezaremos a describir y completar un conjunto especial de funciones en el archivo de encabezado *TradeUtils.mqh*, que resultará muy útil para simplificar la codificación de los robots. Todas las funciones se colocan en el espacio de nombres TU (de «Trade Utilities»), y en primer lugar, introducimos funciones para la conveniente salida registro de estructura *MqlTradeRequest* y *MqlTradeResult*.

```
namespace TU
{
   string StringOf(const MqlTradeRequest &r)
   {
      SymbolMetrics p(r.symbol);

      // main block: action, type, symbol
      string text = EnumToString(r.action);
      if(r.symbol != NULL) text += ", " + r.symbol;
      text += ", " + EnumToString(r.type);
      // volume block
      if(r.volume != 0) text += ", V=" + p.StringOf(r.volume, p.lotDigits);
      text += ", " + EnumToString(r.type_filling);
      // block of all prices
      if(r.price != 0) text += ", @ " + p.StringOf(r.price);
      if(r.stoplimit != 0) text += ", X=" + p.StringOf(r.stoplimit);
      if(r.sl != 0) text += ", SL=" + p.StringOf(r.sl);
      if(r.tp != 0) text += ", TP=" + p.StringOf(r.tp);
      if(r.deviation != 0) text += ", D=" + (string)r.deviation;
      // pending orders expiration block
      if(IsPendingType(r.type)) text += ", " + EnumToString(r.type_time);
      if(r.expiration != 0) text += ", " + TimeToString(r.expiration);
      // modification block
      if(r.order != 0) text += ", #=" + (string)r.order;
      if(r.position != 0) text += ", #P=" + (string)r.position;
      if(r.position_by != 0) text += ", #b=" + (string)r.position_by;
      // auxiliary data
      if(r.magic != 0) text += ", M=" + (string)r.magic;
      if(StringLen(r.comment)) text += ", " + r.comment;

      return text;
   }

   string StringOf(const MqlTradeResult &r)
   {
      string text = TRCSTR(r.retcode);
      if(r.deal != 0) text += ", D=" + (string)r.deal;
      if(r.order != 0) text += ", #=" + (string)r.order;
      if(r.volume != 0) text += ", V=" + (string)r.volume;
      if(r.price != 0) text += ", @ " + (string)r.price;
      if(r.bid != 0) text += ", Bid=" + (string)r.bid;
      if(r.ask != 0) text += ", Ask=" + (string)r.ask;
      if(StringLen(r.comment)) text += ", " + r.comment;
      if(r.request_id != 0) text += ", Req=" + (string)r.request_id;
      if(r.retcode_external != 0) text += ", Ext=" + (string)r.retcode_external;

      return text;
   }
   ...
};
```

El objetivo de las funciones es proporcionar todos los campos significativos (no vacíos) de forma concisa pero conveniente: se muestran en una línea con una designación única para cada uno.

Como puede ver, la función utiliza la clase *SymbolMetrics* para *MqlTradeRequest*. Facilita la normalización de múltiples precios o volúmenes para el mismo instrumento. No olvide que la normalización de precios y volúmenes es un requisito previo para preparar una solicitud de operación correcta.

```
class SymbolMetrics
{
public:
   const string symbol;
   const int digits;
   const int lotDigits;

   SymbolMetrics(const string s): symbol(s),
      digits((int)SymbolInfoInteger(s, SYMBOL_DIGITS)),
      lotDigits((int)MathLog10(1.0 / SymbolInfoDouble(s, SYMBOL_VOLUME_STEP)))
   { }

   double price(const double p)
   {
      return TU::NormalizePrice(p, symbol);
   }

   double volume(const double v)
   {
      return TU::NormalizeLot(v, symbol);
   }

   string StringOf(const double v, const int d = INT_MAX)
   {
      return DoubleToString(v, d == INT_MAX ? digits : d);
   }
};
```

La normalización directa de los valores se confía a las funciones auxiliares *NormalizePrice* y *NormalizeLot* (el esquema de esta última es idéntico al que vimos en el archivo *LotMarginExposure.mqh*).

```
double NormalizePrice(const double price, const string symbol = NULL)
{
   const double tick = SymbolInfoDouble(symbol, SYMBOL_TRADE_TICK_SIZE);
   return MathRound(price / tick) * tick;
}
```

Si conectamos *TradeUtils.mqh*, el ejemplo *CustomOrderSend.mq5* tiene la siguiente forma (los fragmentos de código '...' omitidos no se han modificado con respecto a *CustomOrderCheck.mq5*).

```
void OnTimer()
{
   ...
   MqlTradeRequest request = {};
   MqlTradeCheckResult result = {};

   TU::SymbolMetrics sm(symbol);

   // fill in the request structure
   request.action = Action;
   request.magic = Magic;
   request.order = Order;
   request.symbol = symbol;
   request.volume = sm.volume(volume);
   request.price = sm.price(price);
   request.stoplimit = sm.price(StopLimit);
   request.sl = sm.price(SL);
   request.tp = sm.price(TP);
   request.deviation = Deviation;
   request.type = Type;
   request.type_filling = Filling;
   request.type_time = ExpirationType;
   request.expiration = ExpirationTime;
   request.comment = Comment;
   request.position = Position;
   request.position_by = PositionBy;

   // send the request and display the result
   ResetLastError();
   if(Async)
   {
      PRTF(OrderSendAsync(request, result));
   }
   else
   {
      PRTF(OrderSend(request, result));
   }
   Print(TU::StringOf(request));
   Print(TU::StringOf(result));
}
```

Dado que ahora los precios y el volumen están normalizados, puede intentar introducir valores desiguales en los parámetros de entrada correspondientes. A menudo se obtienen en los programas durante los cálculos, y nuestro código los convierte de acuerdo con la especificación de símbolos.

Con la configuración por defecto, el Asesor Experto crea una solicitud para comprar el lote mínimo del instrumento actual por mercado y lo hace utilizando la función *OrderSend*.

```
OrderSend(request,result)=true / ok
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 1.12462
DONE, D=1250236209, #=1267684253, V=0.01, @ 1.12462, Bid=1.12456, Ask=1.12462, Reques
```

Por regla general, con la negociación permitida, esta operación debería completarse con éxito (estado HECHO, comentario «Solicitud ejecutada»). En la estructura *result* recibimos inmediatamente el número de transacción *D*.

Si abrimos la configuración del Asesor Experto y sustituimos el valor del parámetro *Async* por *true*, enviaremos una solicitud similar pero con la función *OrderSendAsync*.

```
OrderSendAsync(request,result)=true / ok
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 1.12449
PLACED, Order placed, Req=2
```

En este caso, el estado es PLACED (colocado), y no se conoce el número de operación en el momento en que se devuelve la función. Sólo conocemos el identificador único de la solicitud *Req=2*. Para obtener el número de transacción y de posición, debe interceptar el mensaje TRADE_TRANSACTION_REQUEST con el mismo ID de solicitud en el manejador *OnTradeTransaction*, donde la estructura rellenada se recibirá como parámetro *MqlTradeResult*.

Desde el punto de vista del usuario, ambas solicitudes deben ser igual de rápidas.

Será posible comparar el rendimiento de estas dos funciones directamente en el código de un programa MQL utilizando otro ejemplo de Asesor Experto (véase la sección sobre solicitudes síncronas y asíncronas), que consideraremos después de estudiar el modelo de eventos de trading.

> Cabe señalar que los eventos de trading se envían al manejador *OnTradeTransaction* (si está presente en el código), independientemente de la función que se utilice para enviar las solicitudes, *OrderSend* o *OrderSendAsync*. La situación es la siguiente: en caso de aplicar *OrderSend*, la totalidad o parte de la información sobre la ejecución de la orden está disponible inmediatamente en la estructura receptora *MqlTradeResult*. Sin embargo, en el caso general, el resultado se distribuye en el tiempo y el volumen; por ejemplo, cuando una orden se «rellena» en varias transacciones. A continuación, puede obtenerse información completa a partir de los eventos de trading o analizando el historial de transacciones y órdenes.

Si intenta enviar una solicitud deliberadamente incorrecta, por ejemplo, cambiar el tipo de orden a ORDER_TYPE_BUY_STOP pendiente, recibirá un mensaje de error, ya que para este tipo de órdenes debería utilizar la acción TRADE_ACTION_PENDING. Además, deben situarse a una distancia del precio actual (utilizamos el precio de mercado por defecto). Antes de esta prueba es importante no olvidar volver a cambiar el modo de consulta a síncrono (*Async=false*) para ver inmediatamente el error en la estructura *MqlTradeResult* tras finalizar la llamada a *OrderSend*. De lo contrario, *OrderSendAsync* devolvería *true*, pero el orden seguiría sin establecerse, y el programa sólo podría recibir información al respecto en *OnTradeTransaction*, que aún no tenemos.

```
OrderSend(request,result)=false / TRADE_SEND_FAILED(4756)
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_FILLING_FOK, @ 1.12452,
REQUOTE, Bid=1.12449, Ask=1.12452, Requote, Req=5
```

En este caso, el error informa de un precio de recotización no válido.

En las secciones siguientes se presentarán ejemplos del uso de funciones para realizar acciones de trading específicas.

6.4.14 Operaciones de compraventa

En esta sección comenzaremos finalmente a estudiar la aplicación de las funciones MQL5 para tareas específicas de trading. El objetivo de estas funciones es rellenar la estructura *MqlTradeRequest* de una manera especial y llamar a la función *OrderSend* u *OrderSendAsync*.

La primera acción que aprenderemos es comprar o vender un instrumento financiero al precio actual de mercado. El procedimiento para realizar esta acción incluye:

- Creación de una orden de mercado basada en una orden enviada
- Ejecución de una operación (una o varias) en virtud de una orden
- El resultado debería ser una posición abierta

Como vimos en la sección sobre tipos de operaciones de trading, la compra/venta instantánea corresponde al elemento TRADE_ACTION_DEAL de la enumeración ENUM_TRADE_REQUEST_ACTIONS. Por lo tanto, al rellenar la estructura *MqlTradeRequest*, escriba TRADE_ACTION_DEAL en el campo *action*.

La dirección de la operación se establece mediante el campo *type*, que debe contener uno de los tipos de órdenes: ORDER_TYPE_BUY u ORDER_TYPE_SELL.

Por supuesto, para comprar o vender, debe especificar el nombre del símbolo en el campo *symbol* y su volumen deseado en el campo *volume*.

El campo *type_filling* debe rellenarse con una de las políticas de relleno de la enumeración ENUM_ORDER_TYPE_FILLING, que se elige en función de la propiedad del carácter SYMBOL_FILLING_MODE con las políticas permitidas.

De manera opcional, el programa puede rellenar los campos con niveles de precios de protección (*sl* y *tp*), un comentario (*comment*) y un ID de Asesor Experto (*magic*).

El contenido de otros campos se establece de forma diferente en función del modo de ejecución del precio para el símbolo seleccionado. En algunos modos, algunos campos no surten ningún efecto. Por ejemplo, en los modos de Ejecución Solicitada y Ejecución Instantánea, el campo con *price* debe rellenarse con un precio adecuado (el último conocido *Ask* para comprar y *Bid* para vender), y el campo *deviation* puede contener la desviación máxima permitida del precio con respecto al precio fijado para la ejecución satisfactoria de una transacción. En Exchange Execution y Market Execution, estos campos se ignoran. Para simplificar el código fuente, puede rellenar el precio y el deslizamiento de manera uniforme en todos los modos, pero en las dos últimas opciones, el precio seguirá siendo seleccionado y sustituido por el servidor de trading según las reglas de los modos.

Otros campos de la estructura *MqlTradeRequest* no mencionados aquí no se utilizan para estas operaciones de trading.

En la siguiente tabla se resumen las reglas para cumplimentar los campos para los distintos modos de ejecución. Los campos obligatorios están marcados con un asterisco, mientras que los campos opcionales están marcados con un signo más.

Campo	Solicitud	Instantánea	Bolsa	Mercado
action	*	*	*	*
symbol	*	*	*	*
volume	*	*	*	*
type	*	*	*	*
type_filling	*	*	*	*
price	*	*		
sl	+	+	+	+
tp	+	+	+	+
deviation	+	+		
magic	+	+	+	+
comment	+	+	+	+

Dependiendo de la configuración del servidor, puede estar prohibido rellenar campos con niveles de protección *sl* y *tp* en el momento de abrir una posición. Este suele ser el caso de los modos de ejecución de bolsa o ejecución de mercado, pero la API de MQL5 no proporciona propiedades para aclarar esta circunstancia de antemano. En estos casos, *Stop Loss* y *Take Profit* deben establecerse mediante la modificación de una posición ya abierta. Por cierto, este método se puede recomendar para todos los modos de ejecución, ya que es el único que permite aplazar con precisión los niveles de protección del precio real de apertura de la posición. Por otro lado, crear y establecer una posición en dos movimientos puede llevar a una situación en la que la posición está abierta, pero la segunda solicitud para establecer niveles de protección ha fallado por una razón u otra.

Con independencia de la dirección de la operación (compra/venta), la orden *Stop Loss* siempre se coloca como una orden stop (ORDER_TYPE_BUY_STOP u ORDER_TYPE_SELL_STOP), y la orden *Take Profit* se coloca como una orden Limit (ORDER_TYPE_BUY_LIMIT u ORDER_TYPE_SELL_LIMIT). Además, las órdenes stop siempre están controladas por el servidor de MetaTrader 5 y sólo cuando el precio alcanza el nivel especificado, se envían al sistema de trading externo. En cambio, las órdenes Limit pueden enviarse directamente a un sistema de trading externo. En concreto, éste suele ser el caso de los instrumentos cotizados en bolsa.

Con el fin de simplificar la codificación de las operaciones de trading, no sólo de compra y venta, sino también todas las demás, comenzaremos en esta sección por desarrollar clases, o más bien estructuras que proporcionen un llenado automático y correcto de los campos para las solicitudes de trading, así como una espera verdaderamente sincrónica del resultado. Esto último es especialmente importante, dado que las funciones *OrderSend* y *OrderSendAsync* devuelven el control al código de

llamada antes de que la acción de trading se haya completado en su totalidad. En concreto, para la compra y venta en el mercado, el algoritmo suele necesitar saber no el número de ticket de la orden creada en el servidor, sino si la posición está abierta o no. En función de ello, puede, por ejemplo, modificar la posición fijando *Stop Loss* y *Take Profit* si se ha abierto o repetir los intentos de apertura si la orden ha sido rechazada.

Un poco más adelante descubriremos los eventos de trading *OnTrade* y *OnTradeTransaction*, que informan al programa sobre los cambios en el estado de la cuenta, incluido el estado de las órdenes, transacciones y posiciones. Sin embargo, dividir el algoritmo en dos fragmentos -generar órdenes por separado en función de determinadas señales o reglas, y analizar la situación por separado en manejadores de eventos- hace que el código sea menos comprensible y mantenible.

En teoría, el paradigma de programación asíncrona no es inferior al síncrono ni en velocidad ni en facilidad de codificación. Sin embargo, las formas de implementarlo pueden ser diferentes; por ejemplo, basándose en punteros directos a funciones de devolución de llamada (una técnica básica en Java, JavaScript y muchos otros lenguajes) o eventos (como en MQL5), lo que predetermina algunas características, que se tratarán en la sección *OnTradeTransaction*. El modo asíncrono permite acelerar el envío de solicitudes gracias al control diferido sobre su ejecución. Pero este control seguirá teniendo que hacerse tarde o temprano en el mismo hilo, por lo que el rendimiento medio de los circuitos es el mismo.

Todas las estructuras nuevas se colocarán en el archivo *MqlTradeSync.mqh*. Para no «reinventar la rueda», tomemos como punto de partida las estructuras MQL5 integradas y describamos nuestras estructuras como estructuras hijas. Por ejemplo, para obtener resultados de consulta, vamos a definir *MqlTradeResultSync*, que se deriva de *MqlTradeResult*. Aquí añadiremos campos y métodos útiles, en concreto el campo *position* para almacenar un ticket de posición abierta como resultado de una operación de compra o venta en el mercado.

```
struct MqlTradeResultSync: public MqlTradeResult
{
   ulong position;
   ...
};
```

La segunda mejora importante será un constructor que restablezca todos los campos (esto nos ahorra tener que especificar una inicialización explícita al describir variables de tipo estructura).

```
   MqlTradeResultSync()
   {
      ZeroMemory(this);
   }
```

A continuación, introduciremos un mecanismo de sincronización universal, es decir, la espera de los resultados de una solicitud (cada tipo de solicitud tendrá sus propias reglas para comprobar la disponibilidad).

Definamos el tipo de la función de devolución de llamada *condition*. Una función de este tipo debe tomar el parámetro de estructura *MqlTradeResultSync* y devolver *true* si tiene éxito: se recibe el resultado de la operación.

```
typedef bool (*condition)(MqlTradeResultSync &ref);
```

Este tipo de funciones están pensadas para ser pasadas al método *wait*, que implementa una comprobación cíclica de la disponibilidad del resultado durante un tiempo de espera predefinido en milisegundos.

```
bool wait(condition p, const ulong msc = 1000)
{
   const ulong start = GetTickCount64();
   bool success;
   while(!(success = p(this)) && GetTickCount64() - start < msc);
   return success;
}
```

Aclaremos de entrada que timeout es el tiempo máximo de espera: aunque se fije en un valor muy grande, el bucle terminará inmediatamente en cuanto se reciba el resultado, lo que puede ocurrir al instante. Por supuesto, un timeout significativo no debería durar más de unos segundos.

Veamos un ejemplo de un método que se utilizará para esperar de forma sincrónica a que aparezca una orden en el servidor (no importa con qué estado: el análisis del estado es tarea del código de llamada).

```
static bool orderExist(MqlTradeResultSync &ref)
{
   return OrderSelect(ref.order) || HistoryOrderSelect(ref.order);
}
```

Aquí se aplican dos funciones integradas de la API de MQL5, *OrderSelect* y *HistoryOrderSelect*: buscan y seleccionan lógicamente una orden por su ticket en el entorno de trading interno del terminal. En primer lugar, esto confirma la existencia de una orden (si una de las funciones devuelve *true*), y en segundo lugar, permite leer sus propiedades utilizando otras funciones, lo cual no es importante para nosotros todavía. Abordaremos todas estas características en secciones por separado. Las dos funciones se escriben conjuntamente porque una orden de mercado puede ejecutarse tan rápidamente que su fase activa (que cae en *OrderSelect*) pasará inmediatamente al historial (*HistoryOrderSelect*).

Tenga en cuenta que el método se declara estático. Esto se debe al hecho de que MQL5 no admite punteros a métodos de objetos. Si este fuera el caso, podríamos declarar el método como no estático mientras utilizamos el prototipo del puntero a las funciones callback de *condition* sin que el parámetro haga referencia a *MqlTradeResultSync* (ya que todos los campos están presentes dentro del objeto *this*).

El mecanismo de espera puede iniciarse del siguiente modo:

```
if(wait(orderExist))
{
   // there is an order
}
else
{
   // timeout
}
```

Por supuesto, este fragmento debe ejecutarse después de que recibamos un resultado del servidor con el estado TRADE_RETCODE_DONE o TRADE_RETCODE_DONE_PARTIAL, y se garantiza que el campo *order* de la estructura *MqlTradeResultSync* contiene un ticket de orden. Tenga en cuenta que, debido a

la naturaleza distribuida del sistema, es posible que una orden del servidor no se muestre inmediatamente en el entorno del terminal. Por eso necesita tiempo de espera.

Mientras la función *orderExist* devuelva *false* al método *wait*, el bucle de espera se ejecutará hasta que expire el tiempo de espera. En circunstancias normales, encontraremos casi instantáneamente una orden en el entorno terminal, y el bucle finalizará con una señal de éxito (*true*).

La función *positionExist* que comprueba la presencia de una posición abierta de forma similar pero un poco más complicada. Dado que la función *orderExist* anterior ha completado la comprobación de la orden, se confirma que su ticket contenido en el campo *ref.order* de la estructura funciona.

```
static bool positionExist(MqlTradeResultSync &ref)
{
   ulong posid, ticket;
   if(HistoryOrderGetInteger(ref.order, ORDER_POSITION_ID, posid))
   {
      // in most cases, the position ID is equal to the ticket,
      // but not always: the full code implements getting a ticket by ID,
      // for which there are no built-in MQL5 tools
      ticket = posid;

      if(HistorySelectByPosition(posid))
      {
         ref.position = ticket;
         ...
         return true;
      }
   }
   return false;
}
```

Utilizando las funciones *HistoryOrderGetInteger* y *HistorySelectByPosition* obtenemos el ID y el ticket de la posición basada en la orden.

Más adelante veremos el uso de *orderExist* y *positionExist* al verificar una solicitud de compra/venta, pero ahora pasemos a otra estructura: *MqlTradeRequestSync*. También se hereda de la integrada y contiene campos adicionales, en particular, una estructura con un resultado (a fin de no describirlo en el código de llamada) y un timeout para las solicitudes síncronas.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   MqlTradeResultSync result;
   ulong timeout;
   ...
```

Dado que los campos heredados de la nueva estructura son públicos, el programa MQL puede asignarles valores explícitamente, al igual que se hacía con la estructura estándar *MqlTradeRequest*. Los métodos que añadiremos para realizar operaciones de trading considerarán, comprobarán y, si es necesario, corregirán estos valores por los válidos.

En el constructor, reseteamos todos los campos y establecemos el símbolo al valor por defecto si se omite el parámetro.

```
MqlTradeRequestSync(const string s = NULL, const ulong t = 1000): timeout(t)
{
   ZeroMemory(this);
   symbol = s == NULL ? _Symbol : s;
}
```

En teoría, debido a que todos los campos de la estructura son públicos, técnicamente se pueden asignar de forma directa, pero esto no es recomendable para aquellos campos que requieren validación y para los que implementamos métodos setter: serán llamados antes de realizar operaciones de trading. El primero de estos métodos es *setSymbol*.

Rellena el campo *symbol* asegurándose de que el ticker transmitido existe e inicia la configuración posterior del modo de relleno de volumen.

```
bool setSymbol(const string s)
{
   if(s == NULL)
   {
      if(symbol == NULL)
      {
         Print("symbol is NULL, defaults to " + _Symbol);
         symbol = _Symbol;
         setFilling();
      }
      else
      {
         Print("new symbol is NULL, current used " + symbol);
      }
   }
   else
   {
      if(SymbolInfoDouble(s, SYMBOL_POINT) == 0)
      {
         Print("incorrect symbol " + s);
         return false;
      }
      if(symbol != s)
      {
         symbol = s;
         setFilling();
      }
   }
   return true;
}
```

Por lo tanto, al cambiar el símbolo con *setSymbol* se seleccionará automáticamente el modo de relleno correcto mediante una llamada anidada a *setFilling*.

El método *setFilling* proporciona la especificación automática del método de llenado de volumen basado en las propiedades de símbolo SYMBOL_FILLING_MODE y SYMBOL_TRADE_EXEMODE (véase la sección Condiciones de trading de símbolos y modos de ejecución de órdenes).

```
private:
   void setFilling()
   {
      const int filling = (int)SymbolInfoInteger(symbol, SYMBOL_FILLING_MODE);
      const bool market = SymbolInfoInteger(symbol, SYMBOL_TRADE_EXEMODE)
         == SYMBOL_TRADE_EXECUTION_MARKET;

      // the field may already be filled
      // and bit match means a valid mode
      if(((type_filling + 1) & filling) != 0
         || (type_filling == ORDER_FILLING_RETURN && !market)) return;

      if((filling & SYMBOL_FILLING_FOK) != 0)
      {
         type_filling = ORDER_FILLING_FOK;
      }
      else if((filling & SYMBOL_FILLING_IOC) != 0)
      {
         type_filling = ORDER_FILLING_IOC;
      }
      else
      {
         type_filling = ORDER_FILLING_RETURN;
      }
   }
```

Este método corrige implícitamente (sin errores ni mensajes) el campo *type_filling* si el Asesor Experto lo ha configurado incorrectamente. Si su algoritmo requiere un método de llenado específico garantizado, sin el cual el trading es imposible, realice las modificaciones oportunas para interrumpir el proceso.

> Para el conjunto de estructuras que se están desarrollando, se supone que, además del campo *type_filling*, sólo se pueden establecer directamente campos opcionales sin requisitos específicos para su contenido, tales como *magic* o *comment*.

En lo que sigue, muchos de los métodos se presentan de forma abreviada en aras de la simplicidad. Tienen partes para los tipos de operaciones que veremos más adelante, así como comprobación de errores ramificada.

Para las operaciones de compra y venta necesitamos los campos *price* y *volume*; estos dos valores deben normalizarse y comprobarse en cuanto al rango aceptable. Para ello se utiliza el método *setVolumePrices*.

```
bool setVolumePrices(const double v, const double p,
   const double stop, const double take)
{
   TU::SymbolMetrics sm(symbol);
   volume = sm.volume(v);

   if(p != 0) price = sm.price(p);
   else price = sm.price(TU::GetCurrentPrice(type, symbol));

   return setSLTP(stop, take);
}
```

Si el precio de transacción no está fijado (*p == 0*), el programa tomará automáticamente el precio actual del tipo correcto, dependiendo de la dirección, que se lee del campo *type*.

Aunque los niveles *Stop Loss* y *Take Profit* no son necesarios, también deben normalizarse si están presentes, por lo que se añaden a los parámetros de este método.

Ya conocemos la abreviatura TU: representa el espacio de nombres en el archivo *TradeUtilits.mqh* con muchas funciones útiles, incluidas las de normalización de precios y volúmenes.

El procesamiento de los campos *sl* y *tp* se realiza mediante el método independiente *setSLTP* porque ello es necesario no sólo en las operaciones de compra y venta, sino también al modificar una posición existente.

```
bool setSLTP(const double stop, const double take)
{
   TU::SymbolMetrics sm(symbol);
   TU::TradeDirection dir(type);

   if(stop != 0)
   {
      sl = sm.price(stop);
      if(!dir.worse(sl, price))
      {
         PrintFormat("wrong SL (%s) against price (%s)",
            TU::StringOf(sl), TU::StringOf(price));
         return false;
      }
   }
   else
   {
      sl = 0; // remove SL
   }

   if(take != 0)
   {
      tp = sm.price(take);
      if(!dir.better(tp, price))
      {
         PrintFormat("wrong TP (%s) against price (%s)",
            TU::StringOf(tp), TU::StringOf(price));
         return false;
      }
   }
   else
   {
      tp = 0; // remove TP
   }
   return true;
}
```

Además de normalizar y asignar valores a los campos *sl* y *tp*, este método comprueba la ubicación mutuamente correcta de los niveles con respecto a *price*. Para este fin, la clase *TradeDirection* se describe en el espacio TU.

Sus constructores permiten especificar la dirección analizada del trading: compra o venta, en cuyo contexto es fácil identificar un acuerdo mutuo rentable o no rentable de dos precios. Con esta clase, el análisis se realiza de forma unificada y las comprobaciones en el código se reducen 2 veces, ya que no es necesario procesar por separado las operaciones de compra y venta. En concreto, el método *worse* tiene dos parámetros de precio *p1*, *p2*, y devuelve *true* si el precio *p1* se coloca peor, es decir, no es rentable, en relación con el precio *p2*. Un método similar *better* representa la lógica inversa: devolverá *true* si el precio *p1* es mejor que el precio *p2*. Por ejemplo, para una venta, el mejor precio se sitúa más abajo porque *Take Profit* está por debajo del precio actual.

```
TU::TradeDirection dir(ORDER_TYPE_SELL);
Print(dir.better(100, 200)); // true
```

Ahora, si una orden se realiza incorrectamente, la función *setSLTP* registra una advertencia y aborta el proceso de verificación sin intentar corregir los valores, ya que la respuesta adecuada puede variar en los distintos programas. Por ejemplo, de los dos niveles pasados *stop* y *take* sólo uno puede ser incorrecto, y entonces probablemente tenga sentido utilizar el segundo (correcto).

Puede cambiar el comportamiento, por ejemplo, omitiendo la asignación de valores no válidos (los niveles de protección simplemente no se cambiarán) o añadiendo un campo con una bandera de error a la estructura (para tal estructura, un intento de enviar una solicitud debería suprimirse para no cargar el servidor con solicitudes obviamente imposibles). El envío de una solicitud no válida finalizará con el código de error *retcode* igual a TRADE_RETCODE_INVALID_STOPS.

El método *setSLTP* también realiza comprobaciones para asegurarse de que los niveles de protección no estén situados más cerca del precio actual que el número de puntos de la propiedad SYMBOL_TRADE_STOPS_LEVEL del símbolo (si esta propiedad está establecida, es decir, es mayor que 0), y no se solicita la modificación de la posición cuando está dentro de la zona de congelación SYMBOL_TRADE_FREEZE_LEVEL (si está establecida). Estos matices no se muestran aquí: pueden consultarse en el código fuente.

Ahora estamos listos para implementar un grupo de métodos de trading. Por ejemplo, para la compraventa con el conjunto más completo de campos, definimos los métodos *buy* y *sell*.

```
public:
   ulong buy(const string name, const double lot, const double p = 0,
      const double stop = 0, const double take = 0)
   {
      type = ORDER_TYPE_BUY;
      return _market(name, lot, p, stop, take);
   }
   ulong sell(const string name, const double lot, const double p = 0,
      const double stop = 0, const double take = 0)
   {
      type = ORDER_TYPE_SELL;
      return _market(name, lot, p, stop, take);
   }
```

Como ya se ha mencionado, para establecer campos opcionales como *deviation*, *comment* y *magic* se debe hacer una asignación directa antes de llamar a *buy/sell*. Esto es tanto más conveniente por cuanto que *deviation* y *magic* en la mayoría de los casos se establecen una vez, y se utilizan en consultas posteriores.

Los métodos devuelven un ticket de orden, pero a continuación mostraremos en acción el mecanismo de recepción «síncrona» de un ticket de posición, y éste será un ticket de una posición creada o modificada (si se realizó incremento de posición o cierre parcial).

Los métodos *buy* y *sell* sólo difieren en el valor del campo *type*, mientras que todo lo demás es igual. Por ello, la parte general se enmarca en un método independiente *_market*. Aquí es donde fijamos *action* en TRADE_ACTION_DEAL, y llamamos a *setSymbol* y *setVolumePrices*.

```
private:
   ulong _market(const string name, const double lot, const double p = 0,
      const double stop = 0, const double take = 0)
   {
      action = TRADE_ACTION_DEAL;
      if(!setSymbol(name)) return 0;
      if(!setVolumePrices(lot, p, stop, take)) return 0;
      ...
```

A continuación, podríamos simplemente llamar a *OrderSend*, pero dada la posibilidad de recotizaciones (actualizaciones de precios en el servidor durante el tiempo en que se envió la orden), vamos a envolver la llamada en un bucle. Debido a esto, el método será capaz de reintentar varias veces, pero no más que el número preestablecido de veces MAX_REQUOTES (la macro se elige para ser 10 en el código).

```
      int count = 0;
      do
      {
         ZeroMemory(result);
         if(OrderSend(this, result)) return result.order;
         // automatic price selection means automatic processing of requotes
         if(result.retcode == TRADE_RETCODE_REQUOTE)
         {
            Print("Requote N" + (string)++count);
            if(p == 0)
            {
               price = TU::GetCurrentPrice(type, symbol);
            }
         }
      }
      while(p == 0 && result.retcode == TRADE_RETCODE_REQUOTE
         && ++count < MAX_REQUOTES);
      return 0;
   }
```

Dado que el instrumento financiero se establece por defecto en el constructor de la estructura, podemos proporcionar un par de sobrecargas simplificadas de métodos *buy/sell* sin el parámetro *symbol*.

```
public:
   ulong buy(const double lot, const double p = 0,
      const double stop = 0, const double take = 0)
   {
      return buy(symbol, lot, p, stop, take);
   }

   ulong sell(const double lot, const double p = 0,
      const double stop = 0, const double take = 0)
   {
      return sell(symbol, lot, p, stop, take);
   }
```

Así, en una configuración mínima, bastará con que el programa llame a *request.buy(1.0)* para realizar una operación de compra de un lote.

Volvamos ahora al problema de obtener el resultado final de la solicitud, que en el caso de la operación TRADE_ACTION_DEAL significa el ticket de la posición. En la estructura *MqlTradeRequestSync*, este problema se resuelve con el método *completed*: para cada tipo de operación, debe solicitar la estructura anidada *MqlTradeResultSync* con el fin de esperar a que se llene de acuerdo con el tipo de operación.

```
bool completed()
{
   if(action == TRADE_ACTION_DEAL)
   {
      const bool success = result.opened(timeout);
      if(success) position = result.position;
      return success;
   }
   ...
   return false;
}
```

La apertura de la posición se controla con el método *opened*. Dentro encontraremos un par de llamadas al método *wait* descrito anteriormente: la primera es para *orderExist*, y la segunda es para *positionExist*.

```
bool opened(const ulong msc = 1000)
{
   if(retcode != TRADE_RETCODE_DONE
      && retcode != TRADE_RETCODE_DONE_PARTIAL)
   {
      return false;
   }

   if(!wait(orderExist, msc))
   {
      Print("Waiting for order: #" + (string)order);
   }

   if(deal != 0)
   {
      if(HistoryDealGetInteger(deal, DEAL_POSITION_ID, position))
      {
         return true;
      }
      Print("Waiting for position for deal D=" + (string)deal);
   }

   if(!wait(positionExist, msc))
   {
      Print("Timeout");
      return false;
   }
   position = result.position;

   return true;
}
```

Por supuesto, tiene sentido esperar a que aparezca una orden y una posición sólo si el estado de *retcode* indica éxito. Otros estados se refieren a errores o anulación de la operación, o a códigos intermedios específicos (TRADE_RETCODE_PLACED, TRADE_RETCODE_TIMEOUT) que no van acompañados de información útil en otros campos. En ambos casos, esto impide el procesamiento posterior dentro de este marco «síncrono».

Es importante tener en cuenta que estamos utilizando *OrderSync* y por lo tanto nos basamos en la presencia obligatoria del ticket de orden en la estructura recibida del servidor.

En algunos casos, el sistema no sólo envía un ticket de orden, sino también un ticket de transacción al mismo tiempo. Partiendo de la transacción podrá encontrar la posición más rápidamente. Pero aunque haya información sobre la operación, el entorno de trading del terminal puede no tener temporalmente información sobre la nueva posición. Por eso debe esperar a recibirla con *wait(positionExist)*.

Resumamos para el resultado intermedio. Las estructuras creadas le permiten escribir el siguiente código para comprar 1 lote del símbolo actual:

```
MqlTradeRequestSync request;
if(request.buy(1.0) && request.completed())
{
   Print("OK Position: P=", request.result.position);
}
```

Entramos en el bloque del operador condicional sólo con una posición abierta garantizada, y conocemos su ticket. Si utilizáramos únicamente métodos *buy/sell*, recibirían un ticket de orden a su salida y tendrían que comprobar por sí mismos la ejecución. En caso de error, no entraremos en el bloque *if*, y el código del servidor estará contenido en *request.result.retcode*.

Cuando implementemos métodos para otras operaciones en las secciones siguientes, podrán ejecutarse en un modo de «bloqueo» similar; por ejemplo, para modificar los niveles de stop:

```
if(request.adjust(SL, TP) && request.completed())
{
   Print("OK Adjust")
}
```

Por supuesto, no es necesario llamar a *completed* si no se desea comprobar el resultado de la operación en modo de bloqueo. En lugar de ello, puede ceñirse al paradigma asíncrono y analizar el entorno en manejadores de eventos de trading. Pero incluso en este caso, la estructura *MqlTradeRequestAsync* puede ser útil para comprobar y normalizar los parámetros de funcionamiento.

Vamos a escribir un Asesor Experto de prueba *MarketOrderSend.mq5* para poner todo esto junto. Los parámetros de entrada proporcionarán valores para los campos principales y algunos opcionales de la solicitud de operación.

```
enum ENUM_ORDER_TYPE_MARKET
{
   MARKET_BUY = ORDER_TYPE_BUY,  // ORDER_TYPE_BUY
   MARKET_SELL = ORDER_TYPE_SELL // ORDER_TYPE_SELL
};

input string Symbol;          // Symbol (empty = current _Symbol)
input double Volume;          // Volume (0 = minimal lot)
input double Price;           // Price (0 = current Ask)
input ENUM_ORDER_TYPE_MARKET Type;
input string Comment;
input ulong Magic;
input ulong Deviation;
```

La enumeración ENUM_ORDER_TYPE_MARKET es un subconjunto de la estándar ENUM_ORDER_TYPE y se introduce para limitar los tipos de operaciones disponibles a sólo dos: compra y venta en el mercado.

La acción se ejecutará una vez en un temporizador, del mismo modo que en los ejemplos anteriores.

```
void OnInit()
{
   // scheduling a delayed start
   EventSetTimer(1);
}
```

En el manejador del temporizador, desactivamos el temporizador para que la solicitud se ejecute sólo una vez. Para el próximo lanzamiento necesitará cambiar los parámetros del Asesor Experto.

```
void OnTimer()
{
   EventKillTimer();
   ...
```

Vamos a describir una variable de tipo *MqlTradeRequestSync* y a preparar los valores para los campos principales.

```
   const bool wantToBuy = Type == MARKET_BUY;
   const string symbol = StringLen(Symbol) == 0 ? _Symbol : Symbol;
   const double volume = Volume == 0 ?
      SymbolInfoDouble(symbol, SYMBOL_VOLUME_MIN) : Volume;

   MqlTradeRequestSync request(symbol);
   ...
```

Los campos opcionales se rellenarán directamente.

```
   request.magic = Magic;
   request.deviation = Deviation;
   request.comment = Comment;
   ...
```

Entre los campos opcionales, puede seleccionar el modo de relleno (*type_filling*). Por defecto, *MqlTradeRequestSync* escribe automáticamente en este campo el primero de los modos permitidos ENUM_ORDER_TYPE_FILLING. Recordemos que la estructura dispone de un método especial *setFilling* para ello.

A continuación, llamamos al método *buy* o *sell* con parámetros, y si devuelve un ticket de orden, esperamos a que aparezca una posición abierta.

```
   ResetLastError();
   const ulong order = (wantToBuy ?
      request.buy(volume, Price) :
      request.sell(volume, Price));
   if(order != 0)
   {
      Print("OK Order: #=", order);
      if(request.completed()) // waiting for an open position
      {
         Print("OK Position: P=", request.result.position);
      }
   }
   Print(TU::StringOf(request));
   Print(TU::StringOf(request.result));
}
```

Al final de la función, las estructuras de consulta y resultado se registran como referencia.

Si ejecutamos el Asesor Experto con los parámetros por defecto (comprando el símbolo actual con el lote mínimo), podemos obtener el siguiente resultado para «XTIUSD».

```
OK Order: #=218966930
Waiting for position for deal D=215494463
OK Position: P=218966930
TRADE_ACTION_DEAL, XTIUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 109.340, P=21
DONE, D=215494463, #=218966930, V=0.01, @ 109.35, Request executed, Req=8
```

Preste atención a la advertencia sobre la ausencia temporal de una posición: siempre aparecerá debido al procesamiento distribuido de las solicitudes (las propias advertencias pueden desactivarse eliminando la macro SHOW_WARNINGS en el código del Asesor Experto, pero la situación se mantendrá). Sin embargo, gracias a las nuevas estructuras desarrolladas, el código aplicado no se desvía por estas complejidades internas y se escribe en forma de secuencia de pasos sencillos, donde cada uno de los siguientes «confía» en el éxito de los anteriores.

En una cuenta de compensación podemos conseguir un efecto interesante de inversión de posiciones mediante la venta posterior con un lote mínimo duplicado (0.02 en este caso).

```
OK Order: #=218966932
Waiting for position for deal D=215494468
Position ticket <> id: 218966932, 218966930
OK Position: P=218966932
TRADE_ACTION_DEAL, XTIUSD, ORDER_TYPE_SELL, V=0.02, ORDER_FILLING_FOK, @ 109.390, P=2
DONE, D=215494468, #=218966932, V=0.02, @ 109.39, Request executed, Req=9
```

Es importante señalar que, tras la inversión, el ticket de posición deja de ser igual al identificador de posición: el identificador permanece desde la primera orden, y el ticket permanece desde la segunda. Hemos omitido deliberadamente la tarea de encontrar el ticket de posición por su identificador para simplificar la presentación. En la mayoría de los casos, el ticket y el ID son iguales, pero para un control preciso, utilice la función *TU::PositionSelectById*. Los interesados pueden estudiar el código fuente adjunto.

Los identificadores son constantes en tanto exista la posición (hasta que se cierra a cero en términos de volumen) y son útiles para analizar el historial de la cuenta. Los tickets describen las posiciones mientras están abiertas (no existe el concepto de ticket de posición en el historial) y se utilizan en algunos tipos de solicitudes; en concreto, para modificar los niveles de protección o cerrar con una posición opuesta. Sin embargo, existen matices a este respecto. Hablaremos más sobre las propiedades de posición en una sección por separado.

Al realizar una operación de compra o de venta, nuestros métodos *buy/sell* le permiten fijar inmediatamente los niveles *Stop Loss* y/o *Take Profit* . Para ello, basta con pasarlos como parámetros adicionales obtenidos a partir de variables de entrada o calculados mediante algunas fórmulas. Por ejemplo:

```
input double SL;
input double TP;
...
void OnTimer()
{
   ...
   const ulong order = (wantToBuy ?
      request.buy(symbol, volume, Price, SL, TP) :
      request.sell(symbol, volume, Price, SL, TP));
   ...
```

Todos los métodos de las nuevas estructuras proporcionan una normalización automática de los parámetros pasados, por lo que no es necesario utilizar *NormalizeDouble* u otra cosa.

Ya se ha señalado anteriormente que algunos ajustes del servidor pueden prohibir el establecimiento de niveles de protección en el momento de apertura de la posición. En este caso, deberá configurar los campos *sl* y *tp* mediante una solicitud independiente. Exactamente la misma solicitud se utiliza también en aquellos casos en los que se requiere modificar niveles ya establecidos; en concreto, para implementar trailing stop o trailing profit.

En la siguiente sección completaremos el ejemplo actual con una configuración retardada de *sl* y *tp* con la segunda solicitud después de la apertura exitosa de una posición.

6.4.15 Modificar los niveles de Stop Loss y/o Take Profit de una posición

Un programa MQL puede cambiar los niveles de precios de protección *Stop Loss* y *Take Profit* para una posición abierta. El elemento TRADE_ACTION_SLTP de la enumeración ENUM_TRADE_REQUEST_ACTIONS está pensado para este fin, es decir, cuando se rellena la estructura *MqlTradeRequest* debemos escribir TRADE_ACTION_SLTP en el campo *action*.

Este es el único campo obligatorio. La necesidad de rellenar otros campos viene determinada por el modo de funcionamiento de la cuenta ENUM_ACCOUNT_MARGIN_MODE. En las cuentas de cobertura, debe rellenar el campo *symbol*, pero puede omitir el ticket de posición. En las cuentas de cobertura, por el contrario, es obligatorio indicar el ticket de posición *position*, pero puede omitir el símbolo. Esto se debe a las particularidades de la identificación de posiciones en cuentas de distintos tipos. Durante la compensación, sólo puede existir una posición para cada símbolo.

Para unificar el código, se recomienda rellenar ambos campos si se dispone de información.

Los niveles de precios de protección se fijan en los campos *sl* y *tp*. Es posible calcular sólo uno de los campos. Para eliminar los niveles de protección, asígneles valores cero.

En la siguiente tabla se resumen los requisitos para rellenar los campos en función de los modos de recuento. Los campos obligatorios están marcados con un asterisco, los campos opcionales están marcados con un signo más.

Campo	Compensación	Cobertura
action	*	*
symbol	*	+
posición	+	*
sl	+	+
tp	+	+

Para realizar la operación de modificación de los niveles de protección, introducimos varias sobrecargas del método *adjust* en la estructura *MqlTradeRequestSync*.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   ...
   bool adjust(const ulong pos, const double stop = 0, const double take = 0);
   bool adjust(const string name, const double stop = 0, const double take = 0);
   bool adjust(const double stop = 0, const double take = 0);
   ...
};
```

Como vimos anteriormente, dependiendo del entorno, la modificación puede hacerse sólo por ticket o sólo por símbolo de posición. Estas opciones se tienen en cuenta en los dos primeros prototipos.

Además, dado que la estructura puede haberse utilizado ya para solicitudes anteriores, es posible que haya rellenado los campos *position* y *symbols*. A continuación, puede llamar al método con el último prototipo.

Todavía no mostramos la implementación de estos tres métodos, porque está claro que debe tener un cuerpo común con el envío de una solicitud. Esta parte se enmarca como un método de ayuda privado *_adjust* con un conjunto completo de opciones. Aquí se da su código con algunas abreviaturas que no afectan a la lógica de trabajo.

```
private:
   bool _adjust(const ulong pos, const string name,
      const double stop = 0, const double take = 0)
   {
      action = TRADE_ACTION_SLTP;
      position = pos;
      type = (ENUM_ORDER_TYPE)PositionGetInteger(POSITION_TYPE);
      if(!setSymbol(name)) return false;
      if(!setSLTP(stop, take)) return false;
      ZeroMemory(result);
      return OrderSend(this, result);
   }
```

Rellenamos todos los campos de la estructura según las reglas anteriores, llamando a los métodos anteriormente descritos *setSymbol* y *setSLTP*, y a continuación enviamos una solicitud al servidor. El resultado es un estado de éxito (*true*) o error (*false*).

Cada uno de los métodos *adjust* sobrecargados prepara por separado los parámetros de origen para la solicitud. Así es como se hace en presencia de un ticket de posición:

```
public:
   bool adjust(const ulong pos, const double stop = 0, const double take = 0)
   {
      if(!PositionSelectByTicket(pos))
      {
         Print("No position: P=" + (string)pos);
         return false;
      }
      return _adjust(pos, PositionGetString(POSITION_SYMBOL), stop, take);
   }
```

Aquí, utilizando la función integrada *PositionSelectByTicket*, comprobamos la presencia de una posición y su selección en el entorno de trading del terminal, lo cual es necesario para la posterior lectura de sus propiedades, en este caso, el símbolo (*PositionGetString(POSITION_SYMBOL)*). Entonces la variante universal se denomina *adjust*.

Al modificar una posición por nombre de símbolo (que sólo está disponible en una cuenta de compensación), puede utilizar otra opción *adjust*.

```
   bool adjust(const string name, const double stop = 0, const double take = 0)
   {
      if(!PositionSelect(name))
      {
         Print("No position: " + s);
         return false;
      }

      return _adjust(PositionGetInteger(POSITION_TICKET), name, stop, take);
   }
```

En este caso, la selección de la posición se realiza mediante la función integrada *PositionSelect*, y el número de ticket se obtiene a partir de sus propiedades (*PositionGetInteger(POSITION_TICKET)*).

Todas estas características se abordarán en detalle en sus respectivas secciones sobre trabajar con posiciones y propiedades de posiciones.

La versión del método *adjust* con el conjunto de parámetros más minimalista, es decir, con sólo los niveles *stop* y *take*, es la siguiente:

```
bool adjust(const double stop = 0, const double take = 0)
{
   if(position != 0)
   {
      if(!PositionSelectByTicket(position))
      {
         Print("No position with ticket P=" + (string)position);
         return false;
      }
      const string s = PositionGetString(POSITION_SYMBOL);
      if(symbol != NULL && symbol != s)
      {
         Print("Position symbol is adjusted from " + symbol + " to " + s);
      }
      symbol = s;
   }
   else if(AccountInfoInteger(ACCOUNT_MARGIN_MODE)
      != ACCOUNT_MARGIN_MODE_RETAIL_HEDGING
      && StringLen(symbol) > 0)
   {
      if(!PositionSelect(symbol))
      {
         Print("Can't select position for " + symbol);
         return false;
      }
      position = PositionGetInteger(POSITION_TICKET);
   }
   else
   {
      Print("Neither position ticket nor symbol was provided");
      return false;
   }
   return _adjust(position, symbol, stop, take);
}
```

Este código garantiza que los campos *position* y *symbols* se rellenan correctamente en varios modos o que sale antes de tiempo con un mensaje de error en el registro. Al final, se llama a la versión privada de *_adjust*, que envía la solicitud a través de *OrderSend*.

De forma similar a los métodos de *buy/sell*, el conjunto de métodos de *adjust* funciona de forma «asíncrona»: a su finalización, sólo se conoce el estado de envío de la solicitud, pero no hay confirmación de la modificación de los niveles. Como sabemos, para la bolsa, el nivel *Take Profit* puede transmitirse como una orden Limit. Por lo tanto, en la estructura *MqlTradeResultSync*, debemos proporcionar una espera «sincrónica» hasta que los cambios surtan efecto.

El mecanismo general de espera formado como el método *MqlTradeResultSync::wait* ya está listo y se ha utilizado para esperar la apertura de una posición. El método *wait* recibe como primer parámetro un

puntero a otro método con un prototipo predefinido *condition* para sondear en un bucle hasta que se cumpla la condición requerida o se produzca un timeout. En este caso, este método compatible con *condition* debe realizar una comprobación aplicada de los niveles de parada de la posición.

Vamos a añadir un nuevo método llamado *adjusted*.

```
struct MqlTradeResultSync: public MqlTradeResult
{
   ...
   bool adjusted(const ulong msc = 1000)
   {
      if(retcode != TRADE_RETCODE_DONE || retcode != TRADE_RETCODE_PLACED)
      {
         return false;
      }

      if(!wait(checkSLTP, msc))
      {
         Print("SL/TP modification timeout: P=" + (string)position);
         return false;
      }

      return true;
   }
```

En primer lugar, por supuesto, comprobamos el estado en el campo *retcode*. Si hay un estado estándar, seguimos comprobando los niveles en sí, pasando a *wait* un método auxiliar *checkSLTP*.

```
struct MqlTradeResultSync: public MqlTradeResult
{
   ...
   static bool checkSLTP(MqlTradeResultSync &ref)
   {
      if(PositionSelectByTicket(ref.position))
      {
         return TU::Equal(PositionGetDouble(POSITION_SL), /*.?.*/)
            && TU::Equal(PositionGetDouble(POSITION_TP), /*.?.*/);
      }
      else
      {
         Print("PositionSelectByTicket failed: P=" + (string)ref.position);
      }
      return false;
   }
```

Este código garantiza que la posición se selecciona mediante ticket en el entorno de trading del terminal utilizando *PositionSelectByTicket* y lee las propiedades de posición POSITION_SL y POSITION_TP, que deben compararse con lo que había en la solicitud. El problema es que aquí no tenemos acceso al objeto de solicitud y debemos pasar aquí de alguna manera un par de valores para los lugares marcados con '.?.'.

Básicamente, dado que estamos diseñando la estructura *MqlTradeResultSync*, podemos añadirle los campos *sl* y *tp* y rellenarlos con los valores de *MqlTradeRequestSync* antes de enviar la solicitud (el

núcleo no «sabe» de nuestros campos añadidos y los dejará intactos durante la llamada a*OrderSend*). Pero para simplificar, utilizaremos lo que ya está disponible. Los campos *bid* y *ask* de la estructura *MqlTradeResultSync* sólo se utilizan para informar de los precios de recotización (estado TRADE_RETCODE_REQUOTE), que no está relacionado con la solicitud TRADE_ACTION_SLTP, por lo que podemos almacenar en ellos *sl* y *tp* de la *MqlTradeRequestSync* completada.

Es lógico hacerlo en el método *completed* de la estructura *MqlTradeRequestSync* que inicia una espera de bloqueo para los resultados de la operación de trading con un tiempo de espera predefinido. Hasta ahora, su código sólo tenía una rama para la acción TRADE_ACTION_DEAL. Para continuar, vamos a añadir una rama para TRADE_ACTION_SLTP.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   ...
   bool completed()
   {
      if(action == TRADE_ACTION_DEAL)
      {
         const bool success = result.opened(timeout);
         if(success) position = result.position;
         return success;
      }
      else if(action == TRADE_ACTION_SLTP)
      {
         // pass the original request data for comparison with the position propertie
         // by default they are not in the result structure
         result.position = position;
         result.bid = sl; // bid field is free in this result type, use under StopLos
         result.ask = tp; // ask field is free in this type of result, we use it unde
         return result.adjusted(timeout);
      }
      return false;
   }
```

Como puede ver, después de establecer el ticket de posición y los niveles de precio desde la solicitud, llamamos al método *adjusted* comentado anteriormente que comprueba *wait(checkSLTP)*. Ahora podemos volver al método de ayuda *checkSLTP* en la estructura *MqlTradeResultSync* y llevarlo a su forma final.

```
struct MqlTradeResultSync: public MqlTradeResult
{
   ...
   static bool checkSLTP(MqlTradeResultSync &ref)
   {
      if(PositionSelectByTicket(ref.position))
      {
         return TU::Equal(PositionGetDouble(POSITION_SL), ref.bid) // sl from request
            && TU::Equal(PositionGetDouble(POSITION_TP), ref.ask); // tp from request
      }
      else
      {
         Print("PositionSelectByTicket failed: P=" + (string)ref.position);
      }
      return false;
   }
```

Esto completa la ampliación de la funcionalidad de las estructuras *MqlTradeRequestSync* y *MqlTradeResultSync* para la operación de modificación de *Stop Loss* y *Take Profit*.

Con esto en mente, vamos a continuar con el ejemplo del Asesor Experto *MarketOrderSend.mq5* que comenzamos en la sección anterior. Añadámosle un parámetro de entrada *Distance2SLTP*, que permite especificar la distancia en puntos a los niveles *Stop Loss* y *Take Profit*.

```
input int Distance2SLTP = 0; // Distance to SL/TP in points (0 = no)
```

Si es cero, no se establecerá ningún nivel de protección.

En el código de trabajo, tras recibir la confirmación de apertura de una posición, calculamos los valores de los niveles en las variables SL y TP y realizamos una modificación sincrónica: *request.adjust(SL, TP) && request.completed()*.

```
   ...
   const ulong order = (wantToBuy ?
      request.buy(symbol, volume, Price) :
      request.sell(symbol, volume, Price));
   if(order != 0)
   {
      Print("OK Order: #=", order);
      if(request.completed()) // waiting for position opening
      {
         Print("OK Position: P=", request.result.position);
         if(Distance2SLTP != 0)
         {
            // position "selected" in the trading environment of the terminal inside
            // so it is not required to do this explicitly on the ticket
            // PositionSelectByTicket(request.result.position);

            // with the selected position, you can find out its properties, but we ne
            // to step back from it by a given number of points
            const double price = PositionGetDouble(POSITION_PRICE_OPEN);
            const double point = SymbolInfoDouble(symbol, SYMBOL_POINT);
            // we count the levels using the auxiliary class TradeDirection
            TU::TradeDirection dir((ENUM_ORDER_TYPE)Type);
            // SL is always "worse" and TP is always "better" of the price: the code
            const double SL = dir.negative(price, Distance2SLTP * point);
            const double TP = dir.positive(price, Distance2SLTP * point);
            if(request.adjust(SL, TP) && request.completed())
            {
               Print("OK Adjust");
            }
         }
      }
   }
   Print(TU::StringOf(request));
   Print(TU::StringOf(request.result));
}
```

En la primera llamada de *completed* tras una operación de compra o venta exitosa, el ticket de posición se guarda en el campo *position* de la estructura de solicitud. Por lo tanto, para modificar los stops, bastan sólo los niveles de precios, y el símbolo y el ticket de la posición ya están presentes en *request*.

Vamos a intentar ejecutar una operación de compra utilizando el Asesor Experto con la configuración por defecto pero con *Distance2SLTP* fijado en 500 puntos.

```
OK Order: #=1273913958
Waiting for position for deal D=1256506526
OK Position: P=1273913958
OK Adjust
TRADE_ACTION_SLTP, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 1.10889, »
»  SL=1.10389, TP=1.11389, P=1273913958
DONE, Bid=1.10389, Ask=1.11389, Request executed, Req=26
```

Las dos últimas líneas corresponden a la salida de depuración al registro del contenido de las estructuras *request* y *request.result*, iniciada al final de la función. En estas líneas, es interesante que

los campos almacenen una simbiosis de valores a partir de dos consultas: primero se abrió una posición y luego se modificó. En particular, los campos con volumen (0.01) y precio (1.10889) de la solicitud permanecieron después de TRADE_ACTION_DEAL, pero no impidieron la ejecución de TRADE_ACTION_SLTP. En teoría, es fácil deshacerse de esto reseteando la estructura entre dos solicitudes; sin embargo, hemos preferido dejarlos como están, porque entre los campos rellenados también los hay útiles: el campo *position* recibió el ticket que necesitamos para solicitar la modificación. Si restableciéramos la estructura tendríamos que introducir una variable para el almacenamiento intermedio del ticket.

En casos generales, por supuesto, es deseable adherirse a una estricta política de inicialización de datos, pero saber cómo usarlos en escenarios específicos (como dos o más solicitudes relacionadas de un tipo predefinido) le permite optimizar su código.

Además, no debe sorprendernos que en la estructura con el resultado, veamos los niveles solicitados *sl* y *tp* en los campos para los precios *Bid* y *Ask*: fueron escritos allí por el método *MqlTradeRequestSync::completed* con el propósito de compararlos con los cambios de posición reales. Al ejecutar la solicitud, el núcleo del sistema sólo rellena *retcode* (DONE), *comment* («Solicitud ejecutada») y *request_id* (26) en la estructura *result*.

A continuación veremos otro ejemplo de modificación de nivel que implementa trailing stop.

6.4.16 Trailing stop

Una de las tareas más comunes en las que se utiliza la capacidad de cambiar los niveles de precios de protección es desplazar secuencialmente *Stop Loss* a un precio mejor a medida que continúa la tendencia favorable. Se trata de trailing stop. Lo implementamos utilizando las nuevas estructuras *MqlTradeRequestSync* y *MqlTradeResultSync* de las secciones anteriores.

Para poder conectar el mecanismo a cualquier Asesor Experto, vamos a declararlo como la clase *Trailing Stop* (véase el archivo *TrailingStop.mqh*). Almacenaremos el número de la posición controlada, su símbolo y el tamaño del punto de precio, así como la distancia requerida del nivel de stop loss desde el precio actual, y el paso de cambios de nivel en las variables personales de la clase.

```
#include <MQL5Book/MqlTradeSync.mqh>

class TrailingStop
{
   const ulong ticket;  // ticket of controlled position
   const string symbol; // position symbol
   const double point;  // symbol price pip size
   const uint distance; // distance to the stop in points
   const uint step;     // movement step (sensitivity) in points
   ...
```

La distancia sólo es necesaria para el algoritmo de seguimiento de posición estándar proporcionado por la clase base. Las clases derivadas podrán mover el nivel de protección según otros principios, como medias móviles, canales, el indicador SAR, etc. Después de familiarizarnos con la clase base, daremos un ejemplo de una clase derivada con una media móvil.

Vamos a crear la variable *level* para el nivel de precio stop actual. En la variable *ok* mantendremos el estado actual de la posición: *true* si la posición sigue existiendo y *false* si se ha producido un error y la posición se ha cerrado.

```
protected:
   double level;
   bool ok;
   virtual double detectLevel()
   {
      return DBL_MAX;
   }
```

Un método virtual *detectLevel* está pensado para ser sobrescrito en clases descendientes, donde el precio stop debe calcularse de acuerdo con un algoritmo arbitrario. En esta implementación se devuelve un valor especial DBL_MAX, que indica el trabajo según el algoritmo estándar (véase más abajo).

En el constructor, rellene todos los campos con los valores de los parámetros correspondientes. La función *PositionSelectByTicket* comprueba la existencia de una posición con un ticket dado y la asigna en el entorno del programa de manera que la llamada posterior de *PositionGetString* devuelve su propiedad string con el nombre del símbolo.

```
public:
   TrailingStop(const ulong t, const uint d, const uint s = 1) :
      ticket(t), distance(d), step(s),
      symbol(PositionSelectByTicket(t) ? PositionGetString(POSITION_SYMBOL) : NULL),
      point(SymbolInfoDouble(symbol, SYMBOL_POINT))
   {
      if(symbol == NULL)
      {
         Print("Position not found: " + (string)t);
         ok = false;
      }
      else
      {
         ok = true;
      }
   }

   bool isOK() const
   {
      return ok;
   }
```

Consideremos ahora el método público principal de la clase *trail*. El programa MQL tendrá que llamarlo en cada tick o por temporizador para realizar un seguimiento de la posición. El método devuelve *true* mientras exista la posición.

```
virtual bool trail()
{
   if(!PositionSelectByTicket(ticket))
   {
      ok = false;
      return false; // position closed
   }

   // find out prices for calculations: current quote and stop level
   const double current = PositionGetDouble(POSITION_PRICE_CURRENT);
   const double sl = PositionGetDouble(POSITION_SL);
   ...
```

Aquí y más abajo utilizamos las funciones de lectura de las propiedades de posición. Se abordarán en detalle en una sección por separado. En concreto, debemos averiguar la dirección de la negociación - compra y venta- para saber en qué dirección debe fijarse el nivel de stop.

```
   // POSITION_TYPE_BUY  = 0 (false)
   // POSITION_TYPE_SELL = 1 (true)
   const bool sell = (bool)PositionGetInteger(POSITION_TYPE);
   TU::TradeDirection dir(sell);
   ...
```

Para los cálculos y comprobaciones, utilizaremos la clase de ayuda *TU::TradeDirection* y su objeto *dir*. Por ejemplo, su método *negative* permite calcular el precio situado a una distancia determinada del precio actual en una dirección perdedora, con independencia del tipo de operación. Esto simplifica el código porque, de lo contrario, habría que hacer cálculos «espejo» para las compras y las ventas.

```
   level = detectLevel();
   // we can't trail without a level: removing the stop level must be done by the
   if(level == 0) return true;
   // if there is a default value, make a standard offset from the current price
   if(level == DBL_MAX) level = dir.negative(current, point * distance);
   level = TU::NormalizePrice(level, symbol);

   if(!dir.better(current, level))
   {
      return true; // you can't set a stop level on the profitable side<
   }
   ...
```

El método *better* de la clase *TU::TradeDirection* comprueba que el nivel de stop recibido se encuentra a la derecha del precio. Sin este método, tendríamos que volver a escribir la comprobación dos veces (para las compras y las ventas).

Podemos obtener un valor de nivel de stop incorrecto ya que el método *detectLevel* puede sobrescribirse en clases derivadas. Con el cálculo estándar, este problema desaparece porque el nivel lo calcula el objeto *dir*.

Por último, una vez calculado el nivel, es necesario aplicarlo a la posición. Si la posición aún no tiene un stop loss, cualquier nivel válido servirá. Si ya se ha fijado el stop loss, entonces el nuevo valor debe ser mejor que el anterior y diferir en más que el paso especificado.

```
      if(sl == 0)
      {
         PrintFormat("Initial SL: %f", level);
         move(level);
      }
      else
      {
         if(dir.better(level, sl) && fabs(level - sl) >= point * step)
         {
            PrintFormat("SL: %f -> %f", sl, level);
            move(level);
         }
      }

      return true; // success
   }
```

El envío de una solicitud de modificación de posición se implementa en el método *move*, que utiliza el conocido método *adjust* de la estructura *MqlTradeRequestSync* (véase la sección Modificación de los niveles de Stop Loss y/o Take Profit).

```
   bool move(const double sl)
   {
      MqlTradeRequestSync request;
      request.position = ticket;
      if(request.adjust(sl, 0) && request.completed())
      {
         Print("OK Trailing: ", TU::StringOf(sl));
         return true;
      }
      return false;
   }
};
```

Ahora todo está listo para añadir trailing al Asesor Experto *TrailingStop.mq5* de prueba. En los parámetros de entrada, puede especificar la dirección de trading, la distancia al nivel de stop en puntos y el paso en puntos. El parámetro *TrailingDistance* es igual a 0 por defecto, lo que significa cálculo automático del rango diario de cotizaciones y uso de la mitad del mismo como distancia.

```
#include <MQL5Book/MqlTradeSync.mqh>
#include <MQL5Book/TrailingStop.mqh>

enum ENUM_ORDER_TYPE_MARKET
{
   MARKET_BUY = ORDER_TYPE_BUY,    // ORDER_TYPE_BUY
   MARKET_SELL = ORDER_TYPE_SELL   // ORDER_TYPE_SELL
};

input int TrailingDistance = 0;    // Distance to Stop Loss in points (0 = autodetect)
input int TrailingStep = 10;       // Trailing Step in points
input ENUM_ORDER_TYPE_MARKET Type;
input string Comment;
input ulong Deviation;
input ulong Magic = 1234567890;
```

Al iniciarse, el Asesor Experto buscará si existe una posición en el símbolo actual con el número *Magic* especificado y la creará si no existe.

El trailing se realizará mediante un objeto de la clase *TrailingStop* envuelto en un puntero inteligente *AutoPtr*. Gracias a esto último, no necesitamos eliminar manualmente el objeto antiguo cuando se necesita un nuevo objeto de seguimiento que lo sustituya para la nueva posición que se está creando. Cuando se asigna un nuevo objeto a un puntero inteligente, el objeto antiguo se elimina automáticamente. Recordemos que la desreferenciación de un puntero inteligente, es decir, el acceso al objeto de trabajo almacenado en su interior, se realiza mediante el operador sobrecargado [].

```
#include <MQL5Book/AutoPtr.mqh>

AutoPtr<TrailingStop> tr;
```

En el manejador *OnTick* comprobamos si hay un objeto. Si lo hay, comprueba si existe una posición (el atributo se devuelve desde el método *trail*). Inmediatamente después de iniciarse el programa, el objeto no está ahí, y el puntero es NULL. En este caso, debe crear una nueva posición o encontrar una ya abierta y crear un objeto *Trailing Stop* para ella; de ello se encarga la función *Setup*. En llamadas posteriores de *OnTick*, el objeto se inicia y continúa el seguimiento, evitando que el programa entre en el bloque *if* mientras la posición esté «viva».

```
void OnTick()
{
   if(tr[] == NULL || !tr[].trail())
   {
      // if there is no trailing yet, create or find a suitable position
      Setup();
   }
}
```

Y aquí está la función *Setup* propiamente dicha:

```
void Setup()
{
   int distance = 0;
   const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);

   if(trailing distance == 0) // auto-detect the daily range of prices
   {
      distance = (int)((iHigh(_Symbol, PERIOD_D1, 1) - iLow(_Symbol, PERIOD_D1, 1))
         / point / 2);
      Print("Autodetected daily distance (points): ", distance);
   }
   else
   {
      distance = TrailingDistance;
   }

   // process only the position of the current symbol and our Magic
   if(GetMyPosition(_Symbol, Magic))
   {
      const ulong ticket = PositionGetInteger(POSITION_TICKET);
      Print("The next position found: ", ticket);
      tr = new TrailingStop(ticket, distance, TrailingStep);
   }
   else // there is no our position
   {
      Print("No positions found, lets open it...");
      const ulong ticket = OpenPosition();
      if(ticket)
      {
         tr = new TrailingStop(ticket, distance, TrailingStep);
      }
   }

   if(tr[] != NULL)
   {
      // Execute trailing for the first time immediately after creating or finding a
      tr[].trail();
   }
}
```

La búsqueda de una posición abierta adecuada se implementa en la función *GetMyPosition*, y la apertura de una nueva posición se realiza mediante la función *OpenPosition*. Ambas se presentan a continuación. En cualquier caso, obtenemos un ticket de posición y creamos un objeto de trailing para él.

```
bool GetMyPosition(const string s, const ulong m)
{
   for(int i = 0; i < PositionsTotal(); ++i)
   {
      if(PositionGetSymbol(i) == s && PositionGetInteger(POSITION_MAGIC) == m)
      {
         return true;
      }
   }
   return false;
}
```

El propósito y el significado general del algoritmo deben quedar claros a partir de los nombres de las funciones integradas. En el bucle a través de todas las posiciones abiertas (*PositionsTotal*), seleccionamos secuencialmente cada una de ellas utilizando *PositionGetSymbol* y obtenemos su símbolo. Si el símbolo coincide con el solicitado, leemos y comparamos la propiedad de posición POSITION_MAGIC con la «magic» pasada. Todas las funciones para trabajar con posiciones se abordarán en una sección aparte.

La función devolverá *true* en cuanto se encuentre la primera posición coincidente. Al mismo tiempo, la posición permanecerá seleccionada en el entorno de trading del terminal, lo que hace posible que el resto del código lea sus otras propiedades si es necesario.

Ya conocemos el algoritmo para abrir una posición.

```
ulong OpenPosition()
{
   MqlTradeRequestSync request;

   // default values
   const bool wantToBuy = Type == MARKET_BUY;
   const double volume = SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN);
   // optional fields are filled directly in the structure
   request.magic = Magic;
   request.deviation = Deviation;
   request.comment = Comment;
   ResetLastError();
   // execute the selected trade operation and wait for its confirmation
   if((bool)(wantToBuy ? request.buy(volume) : request.sell(volume))
      && request.completed())
   {
      Print("OK Order/Deal/Position");
   }

   return request.position; // non-zero value - sign of success
}
```

Para mayor claridad, veamos cómo funciona este programa en el probador, en modo visual.

Después de la compilación, abramos el panel del probador de estrategias en el terminal, en la pestaña *Review*, y elijamos la primera opción: *Single test*.

En la pestaña *Settings*, seleccione lo siguiente:

- en la lista desplegable *Expert Advisor*, MQL5Book\p6\TralingStop
- *Symbol*: EURUSD
- *Timeframe*: H1
- *Interval*: último año, mes o personalizado
- *Forward*: No
- *Delays*: deshabilitado
- *Modeling*: basado en ticks reales o generados
- *Optimization*: deshabilitado
- *Visual mode*: habilitado

Una vez que pulse *Start* verá algo parecido a esto en una ventana de comprobación independiente:

Trailing stop estándar en el probador

El registro mostrará entradas como las siguientes:

```
2022.01.10 00:02:00 Autodetected daily distance (points): 373
2022.01.10 00:02:00 No positions found, let's open it...
2022.01.10 00:02:00 instant buy 0.01 EURUSD at 1.13612 (1.13550 / 1.13612 / 1.13550)
2022.01.10 00:02:00 deal #2 buy 0.01 EURUSD at 1.13612 done (based on order #2)
2022.01.10 00:02:00 deal performed [#2 buy 0.01 EURUSD at 1.13612]
2022.01.10 00:02:00 order performed buy 0.01 at 1.13612 [#2 buy 0.01 EURUSD at 1.1361
2022.01.10 00:02:00 Waiting for position for deal D=2
2022.01.10 00:02:00 OK Order/Deal/Position
2022.01.10 00:02:00 Initial SL: 1.131770
2022.01.10 00:02:00 position modified [#2 buy 0.01 EURUSD 1.13612 sl: 1.13177]
2022.01.10 00:02:00 OK Trailing: 1.13177
2022.01.10 00:06:13 SL: 1.131770 -> 1.131880
2022.01.10 00:06:13 position modified [#2 buy 0.01 EURUSD 1.13612 sl: 1.13188]
2022.01.10 00:06:13 OK Trailing: 1.13188
2022.01.10 00:09:17 SL: 1.131880 -> 1.131990
2022.01.10 00:09:17 position modified [#2 buy 0.01 EURUSD 1.13612 sl: 1.13199]
2022.01.10 00:09:17 OK Trailing: 1.13199
2022.01.10 00:09:26 SL: 1.131990 -> 1.132110
2022.01.10 00:09:26 position modified [#2 buy 0.01 EURUSD 1.13612 sl: 1.13211]
2022.01.10 00:09:26 OK Trailing: 1.13211
2022.01.10 00:09:35 SL: 1.132110 -> 1.132240
2022.01.10 00:09:35 position modified [#2 buy 0.01 EURUSD 1.13612 sl: 1.13224]
2022.01.10 00:09:35 OK Trailing: 1.13224
2022.01.10 10:06:38 stop loss triggered #2 buy 0.01 EURUSD 1.13612 sl: 1.13224 [#3 se
2022.01.10 10:06:38 deal #3 sell 0.01 EURUSD at 1.13221 done (based on order #3)
2022.01.10 10:06:38 deal performed [#3 sell 0.01 EURUSD at 1.13221]
2022.01.10 10:06:38 order performed sell 0.01 at 1.13221 [#3 sell 0.01 EURUSD at 1.13
2022.01.10 10:06:38 Autodetected daily distance (points): 373
2022.01.10 10:06:38 No positions found, let's open it...
```

Observe cómo el algoritmo desplaza el nivel de SL hacia arriba con un movimiento favorable del precio, hasta el momento en que la posición se cierra por stop loss. Inmediatamente después de liquidar una posición, el programa abre otra nueva.

Para comprobar la posibilidad de utilizar mecanismos de seguimiento no estándar, aplicamos un ejemplo de algoritmo sobre una media móvil. Para ello, volvamos al archivo *TrailingStop.mqh* y describamos la clase derivada *TrailingStopByMA*.

```
class TrailingStopByMA: public TrailingStop
{
   int handle;

public:
   TrailingStopByMA(const ulong t, const int period,
      const int offset = 1,
      const ENUM_MA_METHOD method = MODE_SMA,
      const ENUM_APPLIED_PRICE type = PRICE_CLOSE): TrailingStop(t, 0, 1)
   {
      handle = iMA(_Symbol, PERIOD_CURRENT, period, offset, method, type);
   }

   virtual double detectLevel() override
   {
      double array[1];
      ResetLastError();
      if(CopyBuffer(handle, 0, 0, 1, array) != 1)
      {
         Print("CopyBuffer error: ", _LastError);
         return 0;
      }
      return array[0];
   }
};
```

Crea la instancia del indicador *iMA* en el constructor: el periodo, el método de promediación y el tipo de precio se pasan como parámetros.

En el método overridden *detectLevel*, leemos el valor del búfer del indicador, y por defecto, esto se hace con un offset de 1 barra, es decir, la barra está cerrada, y sus lecturas no cambian cuando llegan los ticks. Aquellos que lo deseen pueden tomar el valor de la barra cero, pero tales señales son inestables para todos los tipos de precio, excepto para PRICE_OPEN.

Para utilizar una nueva clase en el mismo Asesor Experto *TrailingStop.mq5* de prueba, vamos a añadir otro parámetro de entrada *MATrailingPeriod* con un período móvil (vamos a dejar otros parámetros del indicador sin cambios).

```
input int MATrailingPeriod = 0;   // Period for Trailing by MA (0 = disabled)
```

El valor 0 en este parámetro desactiva la media móvil de trailing. Si está activada, se ignoran los ajustes de distancia del parámetro *TrailingDistance*.

Dependiendo de este parámetro, crearemos un objeto de trailing estándar *TrailingStop* o el derivado de *iMA* -*TrailingStopByMA*.

```
...
tr = MATrailingPeriod > 0 ?
   new TrailingStopByMA(ticket, MATrailingPeriod) :
   new TrailingStop(ticket, distance, TrailingStep);
...
```

Veamos cómo se comporta el programa actualizado en el probador. En la configuración del Asesor Experto, establezca un período distinto de cero para MA, por ejemplo, 10.

Trailing stop en la media móvil en el probador

Tenga en cuenta que en los momentos en los que la media se acerca al precio, se produce un efecto de activación frecuente del stop-loss y de cierre de la posición. Cuando la media está por encima de las cotizaciones, no se fija ningún nivel de protección, porque no es correcto para comprar. Esto es consecuencia del hecho de que nuestro Asesor Experto no tiene ninguna estrategia y siempre abre posiciones del mismo tipo, independientemente de la situación en el mercado. En el caso de las ventas, de vez en cuando se produce la misma situación paradójica cuando la media se sitúa por debajo del precio, lo que significa que el mercado está creciendo, y el robot se pone «obstinadamente» en posición corta.

En las estrategias de trabajo, por regla general, la dirección de la posición se elige teniendo en cuenta el movimiento del mercado, y la media móvil se sitúa a la derecha del precio actual, donde se permite colocar un stop loss.

6.4.17 Cierre de una posición: total y parcial

Técnicamente, el cierre de una posición puede considerarse una operación de trading opuesta a la utilizada para abrirla. Por ejemplo, para salir de una compra, es necesario realizar una operación de venta (ORDER_TYPE_SELL en el campo *type*) y para salir de la de venta es necesario comprar (ORDER_TYPE_BUY en el campo *type*).

El tipo de operación de trading en el campo *action* de la estructura *MqlTradeTransaction* sigue siendo el mismo: TRADE_ACTION_DEAL.

En una cuenta de cobertura, la posición que se desea cerrar debe especificarse mediante un ticket en el campo *position*. Para las cuentas de compensación, sólo puede especificar el nombre del símbolo en el campo *symbol*, ya que en ellas sólo es posible una posición de símbolo. Sin embargo, también puede cerrar posiciones aquí mediante un ticket.

Para unificar el código, tiene sentido rellenar los campos *position* y *symbol* independientemente del tipo de cuenta.

Asegúrese también de ajustar el volumen en el campo *volume*. Si es igual al volumen de posición, se cerrará completamente. Sin embargo, si se especifica un valor inferior, es posible cerrar sólo una parte de la posición.

En la siguiente tabla, todos los campos obligatorios de la estructura están marcados con un asterisco y los campos opcionales están marcados con un signo más.

Campo	Compensación	Cobertura
action	*	*
symbol	*	+
posición	+	*
type	*	*
type_filling	*	*
volume	*	*
price	*'	*'
deviation	±	±
magic	+	+
comment	+	+

El campo *price* marcado está con un asterisco con un tick porque se requiere sólo para los símbolos con los modos de ejecución *Request* y *Instant*), mientras que para la ejecución *Exchange* y *Market*, el precio en la estructura no se tiene en cuenta.

Por una razón similar, el campo *deviation* está marcado con «±». Sólo tiene efecto para los modos *Instant* y *Request*.

Para simplificar la implementación programática del cierre de una posición, volvamos a nuestra estructura extendida *MqlTradeRequestSync* en el archivo *MqlTradeSync.mqh*. El método para cerrar una posición por ticket tiene el siguiente código:

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   double partial; // volume after partial closing
   ...
   bool close(const ulong ticket, const double lot = 0)
   {
      if(!PositionSelectByTicket(ticket)) return false;

      position = ticket;
      symbol = PositionGetString(POSITION_SYMBOL);
      type = (ENUM_ORDER_TYPE)(PositionGetInteger(POSITION_TYPE) ^ 1);
      price = 0;
      ...
```

Aquí comprobamos primero la existencia de una posición llamando a la función *PositionSelectByTicket*. Además, esta llamada hace que la posición sea seleccionada en el entorno de trading del terminal, lo que permite leer sus propiedades mediante las funciones posteriores. En concreto, averiguamos el símbolo de una posición a partir de la propiedad POSITION_SYMBOL e «invertimos» su tipo de POSITION_TYPE al opuesto para obtener el tipo de orden requerido.

Los tipos de posición de a enumeración ENUM_POSITION_TYPE son POSITION_TYPE_BUY (valor 0) y POSITION_TYPE_SELL (valor 1). En la enumeración de tipos de órdenes ENUM_ORDER_TYPE, las operaciones de mercado ocupan exactamente los mismos valores: ORDER_TYPE_BUY y ORDER_TYPE_SELL. Por eso podemos llevar la primera enumeración a la segunda, y para obtener el sentido opuesto de la operación, basta con conmutar el bit cero mediante la operación OR exclusiva ('^'): obtenemos 1 a partir de 0, y 0 a partir de 1.

Poner a cero el campo *price* significa seleccionar automáticamente el precio actual correcto (*Ask* o *Bid*) antes de enviar la solicitud: esto se hace un poco más tarde, dentro del método de ayuda *setVolumePrices*, que se llama más adelante en el algoritmo, desde el método *market*.

La llamada al método *_market* se produce un par de líneas más abajo. El método *_market* genera una orden de mercado para todo el volumen o una parte, teniendo en cuenta todos los campos completos de la estructura.

```
      const double total = lot == 0 ? PositionGetDouble(POSITION_VOLUME) : lot;
      partial = PositionGetDouble(POSITION_VOLUME) - total;
      return _market(symbol, total);
   }
```

Este fragmento está ligeramente simplificado en comparación con el código fuente actual. El código completo contiene el manejo de una situación rara pero posible cuando el volumen de la posición excede el volumen máximo permitido en una orden por símbolo (propiedad SYMBOL_VOLUME_MAX). En este caso, la posición debe cerrarse por partes, mediante varias órdenes.

También hay que tener en cuenta que como la posición se puede cerrar parcialmente, hemos tenido que añadir un campo a la estructura *partial*, donde se coloca el saldo previsto del volumen después de la operación. Por supuesto, para un cierre completo, esto será 0. Esta información será necesaria para seguir verificando la realización de la operación.

Para las cuentas de compensación, existe una versión del método *close* que identifica la posición por el nombre del símbolo: selecciona una posición por símbolo, obtiene su ticket y, a continuación, remite a la versión anterior de *close*.

```
bool close(const string name, const double lot = 0)
{
   if(!PositionSelect(name)) return false;
   return close(PositionGetInteger(POSITION_TICKET), lot);
}
```

En la estructura *MqlTradeRequestSync*, tenemos el método *completed* que proporciona una espera sincrónica para la finalización de la operación, si es necesario. Ahora necesitamos complementarlo para cerrar posiciones, en la rama donde *action* es igual a TRADE_ACTION_DEAL. Distinguiremos entre abrir una posición y cerrarla por un valor cero en el campo *position*: no tiene ticket cuando se abre una posición, y tiene uno cuando se cierra.

```
bool completed()
{
   if(action == TRADE_ACTION_DEAL)
   {
      if(position == 0)
      {
         const bool success = result.opened(timeout);
         if(success) position = result.position;
         return success;
      }
      else
      {
         result.position = position;
         result.partial = partial;
         return result.closed(timeout);
      }
   }
```

Para comprobar el cierre real de una posición, hemos añadido el método *closed* en la estructura *MqlTradeResultSync*. Antes de llamarlo, escribimos el ticket de la posición en el campo *result.position* para que la estructura de resultados pueda realizar un seguimiento del momento en que el ticket correspondiente desaparece del entorno de trading del terminal, o cuando el volumen es igual a *result.partial* en caso de cierre parcial.

He aquí el método *closed*. Se basa en un principio bien conocido: comprobar primero el éxito del código de retorno del servidor y luego esperar con el método *wait* a que se cumpla alguna condición.

```
struct MqlTradeResultSync: public MqlTradeResult
{
   ...
   bool closed(const ulong msc = 1000)
   {
      if(retcode != TRADE_RETCODE_DONE)
      {
         return false;
      }
      if(!wait(positionRemoved, msc))
      {
         Print("Position removal timeout: P=" + (string)position);
      }

      return true;
   }
```

En este caso, para comprobar la condición para que la posición desaparezca, hemos tenido que implementar una nueva función *positionRemoved*.

```
   static bool positionRemoved(MqlTradeResultSync &ref)
   {
      if(ref.partial)
      {
         return PositionSelectByTicket(ref.position)
            && TU::Equal(PositionGetDouble(POSITION_VOLUME), ref.partial);
      }
      return !PositionSelectByTicket(ref.position);
   }
```

Probaremos el funcionamiento del cierre de posiciones utilizando el Asesor Experto *TradeClose.mq5*, que implementa una sencilla estrategia de trading: entrar en el mercado si hay dos barras consecutivas en la misma dirección, y en cuanto la siguiente barra cierre en sentido opuesto a la tendencia anterior, salir del mercado. Las señales repetitivas durante tendencias continuas serán ignoradas, es decir, habrá un máximo de una posición (lote mínimo) o ninguna en el mercado.

El Asesor Experto no tendrá ningún parámetro ajustable, a excepción de (*Deviation*) y un número único (*Magic*). Los parámetros implícitos son el marco temporal y el símbolo de trabajo del gráfico.

Para rastrear la presencia de una posición ya abierta, utilizamos la función *GetMyPosition* del ejemplo anterior *TradeTrailing.mq5*: busca entre las posiciones por símbolo y número de Asesor Experto y devuelve un *true* lógico si se encuentra una posición adecuada.

También tomamos la función casi sin cambios *OpenPosition*: abre una posición según el tipo de orden de mercado pasada en el parámetro único. Aquí, este parámetro provendrá del algoritmo de detección de tendencias, y anteriormente (en *TrailingStop.mq5*) el tipo de orden era establecido por el usuario a través de una variable de entrada.

Una nueva función que implementa el cierre de una posición es *ClosePosition*. Dado que el archivo de encabezado *MqlTradeSync.mqh* se hizo cargo de toda la rutina, sólo tenemos que llamar al método *request.close(ticket)* para el ticket de posición enviado y esperar a que se complete el borrado mediante *request.completed()*.

En teoría, esto último puede evitarse si el Asesor Experto analiza la situación en cada tick. En este caso, un posible problema con la eliminación de la posición se revelará rápidamente en el siguiente tick, y el Asesor Experto puede intentar eliminarla de nuevo. Sin embargo, este Asesor Experto tiene una lógica de trading basada en barras, y por lo tanto no tiene sentido analizar cada tick. A continuación, implementamos un mecanismo especial para trabajar barra por barra y, en este sentido, controlamos de forma sincrónica la eliminación, ya que, de lo contrario, la posición se quedaría «colgada» durante toda una barra.

```
ulong LastErrorCode = 0;

ulong ClosePosition(const ulong ticket)
{
   MqlTradeRequestSync request; // empty structure

   // optional fields are filled directly in the structure
   request.magic = Magic;
   request.deviation = Deviation;

   ResetLastError();
   // perform close and wait for confirmation
   if(request.close(ticket) && request.completed())
   {
      Print("OK Close Order/Deal/Position");
   }
   else // print diagnostics in case of problems
   {
      Print(TU::StringOf(request));
      Print(TU::StringOf(request.result));
      LastErrorCode = request.result.retcode;
      return 0; // error, code to parse in LastErrorCode
   }

   return request.position; // non-zero value - success
}
```

Podríamos obligar a las funciones de *ClosePosition* a devolver 0 en caso de borrado correcto de la posición, y un código de error en caso contrario. Este enfoque aparentemente eficaz haría que el comportamiento de las dos funciones *OpenPosition* y *ClosePosition* fuera diferente: en el código de llamada, sería necesario anidar las llamadas de estas funciones en expresiones lógicas de significado opuesto, lo que introduciría confusión. Además, necesitaríamos la variable global *LastErrorCode* en cualquier caso, a fin de añadir información sobre el error dentro de la función *OpenPosition*. Además, la comprobación *if(condition)* se interpreta más orgánicamente como un éxito que *if(!condition)*.

La función que genera señales de trading según la estrategia anterior se denomina *GetTradeDirection*.

```
ENUM_ORDER_TYPE GetTradeDirection()
{
   if(iClose(_Symbol, _Period, 1) > iClose(_Symbol, _Period, 2)
      && iClose(_Symbol, _Period, 2) > iClose(_Symbol, _Period, 3))
   {
      return ORDER_TYPE_BUY; // open a long position
   }

   if(iClose(_Symbol, _Period, 1) < iClose(_Symbol, _Period, 2)
      && iClose(_Symbol, _Period, 2) < iClose(_Symbol, _Period, 3))
   {
      return ORDER_TYPE_SELL; // open a short position
   }

   return (ENUM_ORDER_TYPE)-1; // close
}
```

La función devuelve un valor del tipo ENUM_ORDER_TYPE con dos elementos estándar (ORDER_TYPE_BUY y ORDER_TYPE_SELL) que activan compras y ventas, respectivamente. El valor especial -1 (no en la enumeración) se utilizará como señal de cierre.

Para activar el Asesor Experto basado en el algoritmo de trading utilizamos el manejador *OnTick*. Como recordamos, otras opciones son adecuadas para otras estrategias; por ejemplo, un temporizador para operar en las noticias o eventos de Profundidad de Mercado para trading de volumen.

En primer lugar, analicemos la función de forma simplificada, sin manejar posibles errores. Al principio, hay un bloque que garantiza que el algoritmo posterior sólo se active cuando se abra una nueva barra.

```
void OnTick()
{
   static datetime lastBar = 0;
   if(iTime(_Symbol, _Period, 0) == lastBar) return;
   lastBar = iTime(_Symbol, _Period, 0);
   ...
```

A continuación obtenemos la señal actual de la función *GetTradeDirection*.

```
   const ENUM_ORDER_TYPE type = GetTradeDirection();
```

Si hay una posición, comprobamos si se ha recibido una señal para cerrarla y llamamos a *ClosePosition* si es necesario. Si todavía no hay ninguna posición y hay una señal para entrar en el mercado, llamamos a *OpenPosition*.

```
      if(GetMyPosition(_Symbol, Magic))
      {
         if(type != ORDER_TYPE_BUY && type != ORDER_TYPE_SELL)
         {
            ClosePosition(PositionGetInteger(POSITION_TICKET));
         }
      }
      else if(type == ORDER_TYPE_BUY || type == ORDER_TYPE_SELL)
      {
         OpenPosition(type);
      }
   }
```

Para analizar errores, necesitará encerrar las llamadas a *OpenPosition* y *ClosePosition* en sentencias condicionales y tomar alguna acción para restaurar el estado de funcionamiento del programa. En el caso más sencillo, basta con repetir la solicitud en el siguiente tick, pero es deseable hacerlo un número limitado de veces. Por lo tanto, crearemos variables estáticas con un contador y un límite de error.

```
   void OnTick()
   {
      static int errors = 0;
      static const int maxtrials = 10; // no more than 10 attempts per bar

      // expect a new bar to appear if there were no errors
      static datetime lastBar = 0;
      if(iTime(_Symbol, _Period, 0) == lastBar && errors == 0) return;
      lastBar = iTime(_Symbol, _Period, 0);
      ...
```

El mecanismo barra a barra se desactiva temporalmente si aparecen errores, ya que conviene superarlos lo antes posible.

Los errores se cuentan en las sentencias condicionales en torno a *ClosePosition* y *OpenPosition*.

```
      const ENUM_ORDER_TYPE type = GetTradeDirection();

      if(GetMyPosition(_Symbol, Magic))
      {
         if(type != ORDER_TYPE_BUY && type != ORDER_TYPE_SELL)
         {
            if(!ClosePosition(PositionGetInteger(POSITION_TICKET)))
            {
               ++errors;
            }
            else
            {
               errors = 0;
            }
         }
      }
      else if(type == ORDER_TYPE_BUY || type == ORDER_TYPE_SELL)
      {
         if(!OpenPosition(type))
         {
            ++errors;
         }
         else
         {
            errors = 0;
         }
      }
    // too many errors per bar
      if(errors >= maxtrials) errors = 0;
    // error serious enough to pause
      if(IS_TANGIBLE(LastErrorCode)) errors = 0;
   }
```

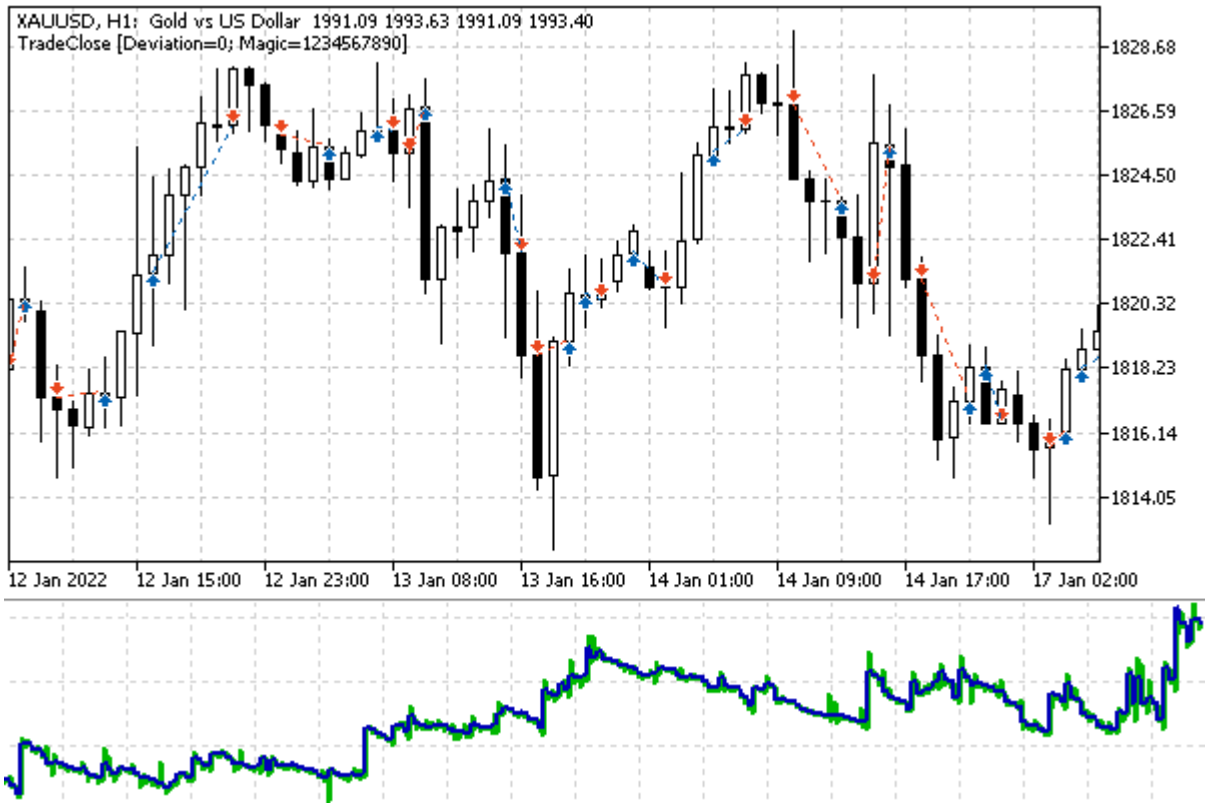
Si se establece la variable *errors* en 0, se activa de nuevo el mecanismo barra a barra y se detienen los intentos de repetir la solicitud hasta la siguiente barra.

La macro IS_TANGIBLE se define en *TradeRetcode.mqh* como:

```
#define IS_TANGIBLE(T) ((T) >= TRADE_RETCODE_ERROR)
```

Los errores con códigos más pequeños son operativos, es decir, normales en cierto sentido. Los códigos grandes requieren análisis y diferentes acciones, dependiendo de la causa del problema: parámetros de solicitud incorrectos, prohibiciones permanentes o temporales en el entorno de trading, falta de fondos, etc. Presentaremos un clasificador de errores mejorado en la sección Modificación de orden pendiente.

Vamos a ejecutar el Asesor Experto en el probador en XAUUSD, H1 desde principios de 2022, simulando ticks reales. En el siguiente collage se muestra un fragmento de un gráfico con transacciones, así como la curva de balance.



Resultados de las pruebas TradeClose en XAUUSD, H1

Basándonos en el informe y el registro, podemos ver que la combinación de nuestra sencilla lógica de trading y las dos operaciones de apertura y cierre de posiciones funciona correctamente.

Además de simplemente cerrar una posición, la plataforma admite la posibilidad de un mutuo cierre de dos posiciones opuestas en cuentas de cobertura.

6.4.18 Cierre de posiciones opuestas: total y parcial (cobertura)

En las cuentas de cobertura se permite abrir varias posiciones al mismo tiempo y, en la mayoría de los casos, estas posiciones pueden ser en sentido opuesto. En algunas jurisdicciones, las cuentas de cobertura están restringidas: sólo se pueden tener posiciones en una dirección al mismo tiempo. En este caso, recibirá el código de error TRADE_RETCODE_HEDGE_PROHIBITED cuando intente ejecutar una operación de trading opuesta. Además, esta restricción a menudo se correlaciona con el ajuste de la propiedad de cuenta ACCOUNT_FIFO_CLOSE a *true*.

Cuando se abren dos posiciones opuestas al mismo tiempo, la plataforma admite el mecanismo de su cierre mutuo simultáneo mediante la operación TRADE_ACTION_CLOSE_BY. Para realizar esta acción, debe rellenar dos campos más en la estructura *MqlTradeTransaction* además del campo *action*: *position* y *position_by* deben contener los tickets de las posiciones que se van a cerrar.

La disponibilidad de esta característica depende de la propiedad SYMBOL_ORDER_MODE del instrumento financiero: SYMBOL_ORDER_CLOSEBY (64) debe estar presente en la máscara de bits de las banderas permitidas.

Esta operación no sólo simplifica el cierre (una operación en lugar de dos), sino que también ahorra un diferencial.

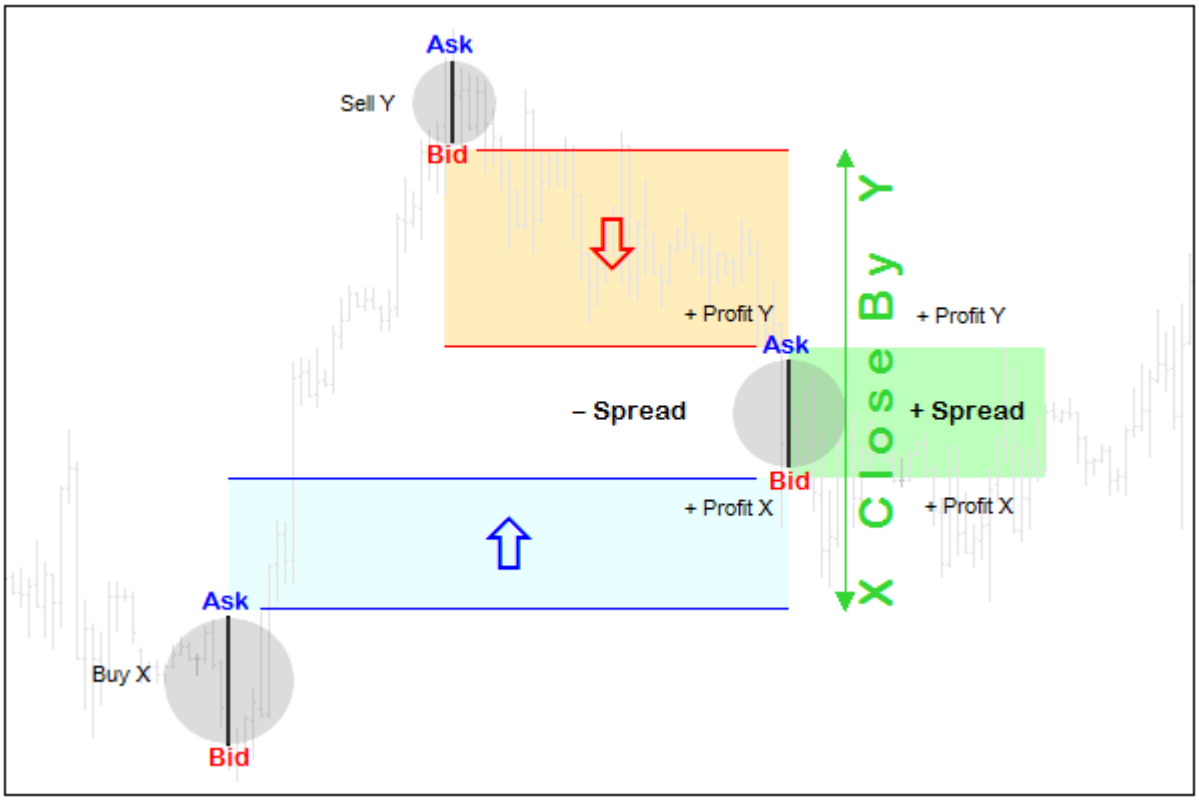
Como sabe, cualquier posición nueva comienza a operar con una pérdida igual al diferencial. Por ejemplo, cuando se compra un instrumento financiero, la transacción se concluye al precio *Ask*, pero para una transacción de salida, es decir, una venta, el precio real es *Bid*. Para una posición corta, la situación es inversa: inmediatamente después de entrar al precio *Bid*, empezamos a realizar un seguimiento del precio *Ask* para una posible salida.

Si cierra posiciones al mismo tiempo de forma regular, sus precios de salida estarán a una distancia del diferencial actual entre sí. Sin embargo, si utiliza la operación TRADE_ACTION_CLOSE_BY, ambas posiciones se cerrarán sin tener en cuenta los precios actuales. El precio al que se compensan las posiciones es igual al precio de apertura de la posición *position_by* (en la estructura de solicitudes). Se especifica en la orden ORDER_TYPE_CLOSE_BY generada por la solicitud TRADE_ACTION_CLOSE_BY.

Lamentablemente, en los informes en el contexto de transacciones y posiciones, los precios de cierre y apertura de posiciones/transacciones opuestas se muestran en pares de valores idénticos, en sentido espejo, lo que da la impresión de una doble ganancia o pérdida. De hecho, el resultado financiero de la operación (la diferencia entre los precios ajustados por el lote) sólo se registra para la operación de salida de la primera posición (el campo *position* de la estructura de la solicitud). El resultado de la segunda operación de salida es siempre 0, independientemente de la diferencia de precios.

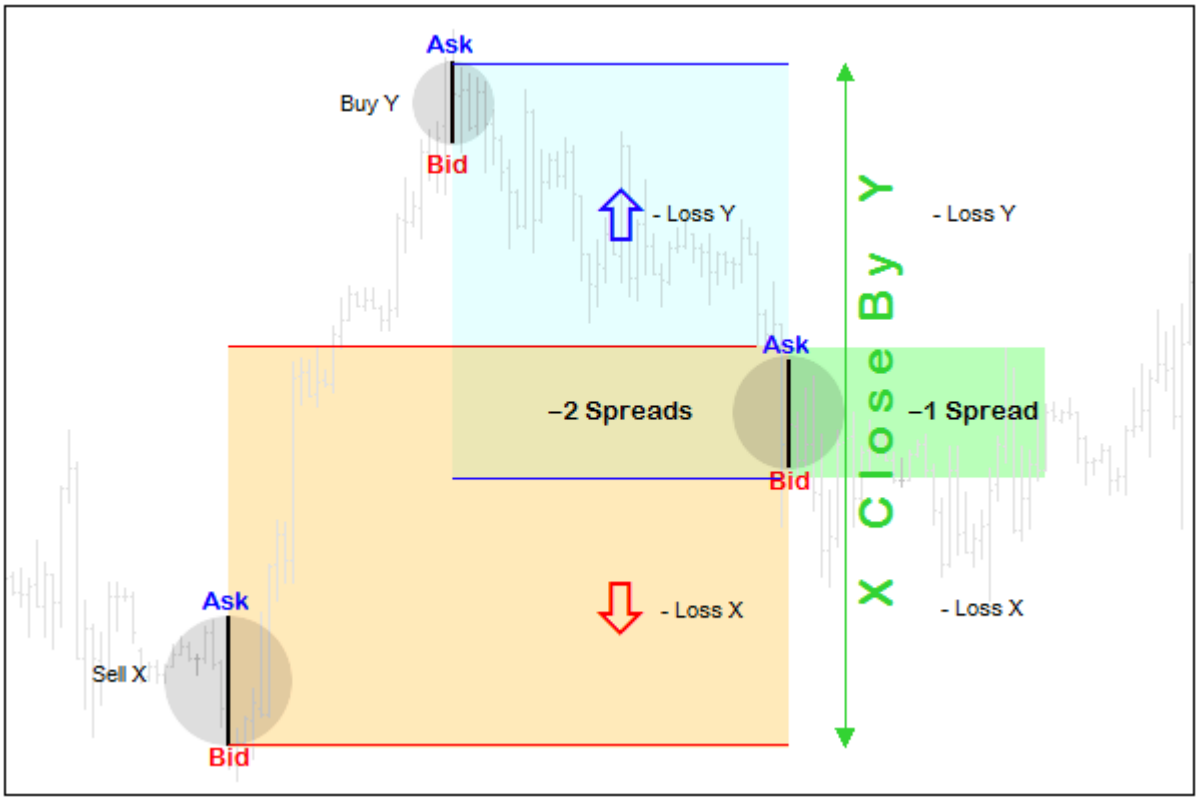
Otra consecuencia de esta asimetría es que al cambiar los lugares de los tickets en los campos *position* y *position_by*, las estadísticas de pérdidas y ganancias en el contexto de operaciones largas y cortas cambian en el informe de trading; por ejemplo, las operaciones largas rentables pueden aumentar exactamente tanto como disminuye el número de operaciones cortas rentables. Pero esto, en teoría, no debería afectar al resultado global, si suponemos que el retraso en la ejecución de la orden no depende del orden de transferencia de los tickets.

En el siguiente diagrama se muestra una explicación gráfica del proceso (los diferenciales están intencionadamente exagerados).



Contabilidad de diferenciales al cerrar posiciones rentables

He aquí un caso de un par de posiciones rentables. Si las posiciones tenían direcciones opuestas y eran deficitarias, al cerrarlas por separado, el diferencial se tendría en cuenta dos veces (en cada una). El contracierre permite reducir la pérdida en un diferencial.



Contabilización del diferencial al cerrar posiciones no rentables

Las posiciones invertidas no tienen por qué ser del mismo tamaño. La operación de cierre opuesto funcionará en el mínimo de los dos volúmenes.

En el archivo *MqlTradeSync.mqh*, la operación del cierre opuesto se implementa utilizando el método *closeby* con dos parámetros para los tickets de posición.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   ...
   bool closeby(const ulong ticket1, const ulong ticket2)
   {
      if(!PositionSelectByTicket(ticket1)) return false;
      double volume1 = PositionGetDouble(POSITION_VOLUME);
      if(!PositionSelectByTicket(ticket2)) return false;
      double volume2 = PositionGetDouble(POSITION_VOLUME);

      action = TRADE_ACTION_CLOSE_BY;
      position = ticket1;
      position_by = ticket2;

      ZeroMemory(result);
      if(volume1 != volume2)
      {
         // remember which position should disappear
         if(volume1 < volume2)
            result.position = ticket1;
         else
            result.position = ticket2;
      }
      return OrderSend(this, result);
   }
```

Para controlar el resultado del cierre, almacenamos el ticket de una posición menor en la variable *result.position*. Todo en el método *completed* y en la estructura *MqlTradeResultSync* está listo para el seguimiento síncrono del cierre de posición: el mismo algoritmo funcionó para el cierre normal de una posición.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   ...
   bool completed()
   {
      ...
      else if(action == TRADE_ACTION_CLOSE_BY)
      {
         return result.closed(timeout);
      }
      return false;
   }
```

Las posiciones opuestas suelen utilizarse como sustituto de una orden de stop o un intento de obtener beneficios en una corrección a corto plazo mientras se permanece en el mercado y se sigue la tendencia principal. La opción de utilizar una orden de pseudo-stop le permite posponer la decisión de cerrar realmente las posiciones durante algún tiempo, continuando el análisis de los movimientos del mercado a la espera de que el precio se invierta en la dirección correcta. Sin embargo, hay que tener en cuenta que las posiciones «bloqueadas» requieren mayores depósitos y están sujetas a swaps. Por eso es difícil imaginar una estrategia de trading basada en posiciones opuestas en su forma pura, que pueda servir de ejemplo para esta sección.

Vamos a desarrollar la idea de la estrategia basada en barras precio-acción esbozada en el ejemplo anterior. El nuevo Asesor Experto es *TradeCloseBy.mq5*.

Utilizaremos la señal anterior para entrar en el mercado al detectar dos velas consecutivas que cerraron en la misma dirección. Una función responsable de su formación es de nuevo *GetTradeDirection*. No obstante, vamos a permitir reingresos si la tendencia continúa. El número máximo total de posiciones permitidas se fijará en la variable de entrada *PositionLimit*; el valor por defecto es 5.

La función *GetMyPositions* sufrirá algunos cambios: tendrá dos parámetros, que serán referencias a arrays que aceptan tickets de posición: compra y venta por separado.

```
#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1, ArraySize(A) * 2) - 1] = V)

int GetMyPositions(const string s, const ulong m,
   ulong &ticketsLong[], ulong &ticketsShort[])
{
   for(int i = 0; i < PositionsTotal(); ++i)
   {
      if(PositionGetSymbol(i) == s && PositionGetInteger(POSITION_MAGIC) == m)
      {
         if((ENUM_POSITION_TYPE)PositionGetInteger(POSITION_TYPE) == POSITION_TYPE_BU
            PUSH(ticketsLong, PositionGetInteger(POSITION_TICKET));
         else
            PUSH(ticketsShort, PositionGetInteger(POSITION_TICKET));
      }
   }

   const int min = fmin(ArraySize(ticketsLong), ArraySize(ticketsShort));
   if(min == 0) return -fmax(ArraySize(ticketsLong), ArraySize(ticketsShort));
   return min;
}
```

La función devuelve el tamaño del array más pequeño de los dos. Cuando es mayor que cero, tenemos la oportunidad de cerrar posiciones opuestas.

Si el array mínimo es de tamaño cero, la función devolverá el tamaño de otro array, pero con un signo menos, para que el código que llama sepa que todas las posiciones están en la misma dirección.

Si no hay posiciones en ninguna dirección, la función devolverá 0.

Las posiciones de apertura seguirán bajo el control de la función *OpenPosition*; aquí no hay cambios.

El cierre sólo se realizará en el modo de dos posiciones opuestas en la nueva función *CloseByPosition*. En otras palabras: este Asesor Experto no es capaz de cerrar posiciones de una en una, de la forma habitual. Por supuesto, en un robot real es improbable que se produzca un principio así, pero como ejemplo de un cierre que se aproxima, encaja muy bien. Si necesitamos cerrar una sola posición, basta con abrir una posición opuesta para ella (en este momento la ganancia o pérdida flotante es fija) y llamar a *CloseByPosition* para dos.

```
bool CloseByPosition(const ulong ticket1, const ulong ticket2)
{
   MqlTradeRequestSync request;
   request.magic = Magic;

   ResetLastError();
   // send a request and wait for it to complete
   if(request.closeby(ticket1, ticket2))
   {
      Print("Positions collapse initiated");
      if(request.completed())
      {
         Print("OK CloseBy Order/Deal/Position");
         return true; // success
      }
   }

   Print(TU::StringOf(request));
   Print(TU::StringOf(request.result));

   return false; // error
}
```

El código utiliza el método *request.closeby* descrito anteriormente. Se rellenan los campos *position* y *position_by* y se llama a *OrderSend*.

La lógica de trading se describe en el manejador *OnTick* que analiza la configuración de precios sólo en el momento de la formación de una nueva barra y recibe una señal de la función *GetTradeDirection*.

```
void OnTick()
{
   static bool error = false;
   // waiting for the formation of a new bar, if there is no error
   static datetime lastBar = 0;
   if(iTime(_Symbol, _Period, 0) == lastBar && !error) return;
   lastBar = iTime(_Symbol, _Period, 0);

   const ENUM_ORDER_TYPE type = GetTradeDirection();
   ...
```

A continuación, rellenamos los arrays *ticketsLong* y *ticketsShort* con los tickets de posición del símbolo de trabajo y con el número *Magic* dado. Si la función *GetMyPositions* devuelve un valor mayor que cero, da el número de pares formados de posiciones opuestas. Pueden cerrarse en bucle mediante la función *CloseByPosition*. La combinación de pares en este caso se elige aleatoriamente (por orden de posiciones en el entorno del terminal); sin embargo, en la práctica, puede ser importante seleccionar los pares por volumen o de una forma tal que se cierren primero los más rentables.

```
      ulong ticketsLong[], ticketsShort[];
      const int n = GetMyPositions(_Symbol, Magic, ticketsLong, ticketsShort);
      if(n > 0)
      {
         for(int i = 0; i < n; ++i)
         {
            error = !CloseByPosition(ticketsShort[i], ticketsLong[i]) && error;
         }
      }
      ...
```

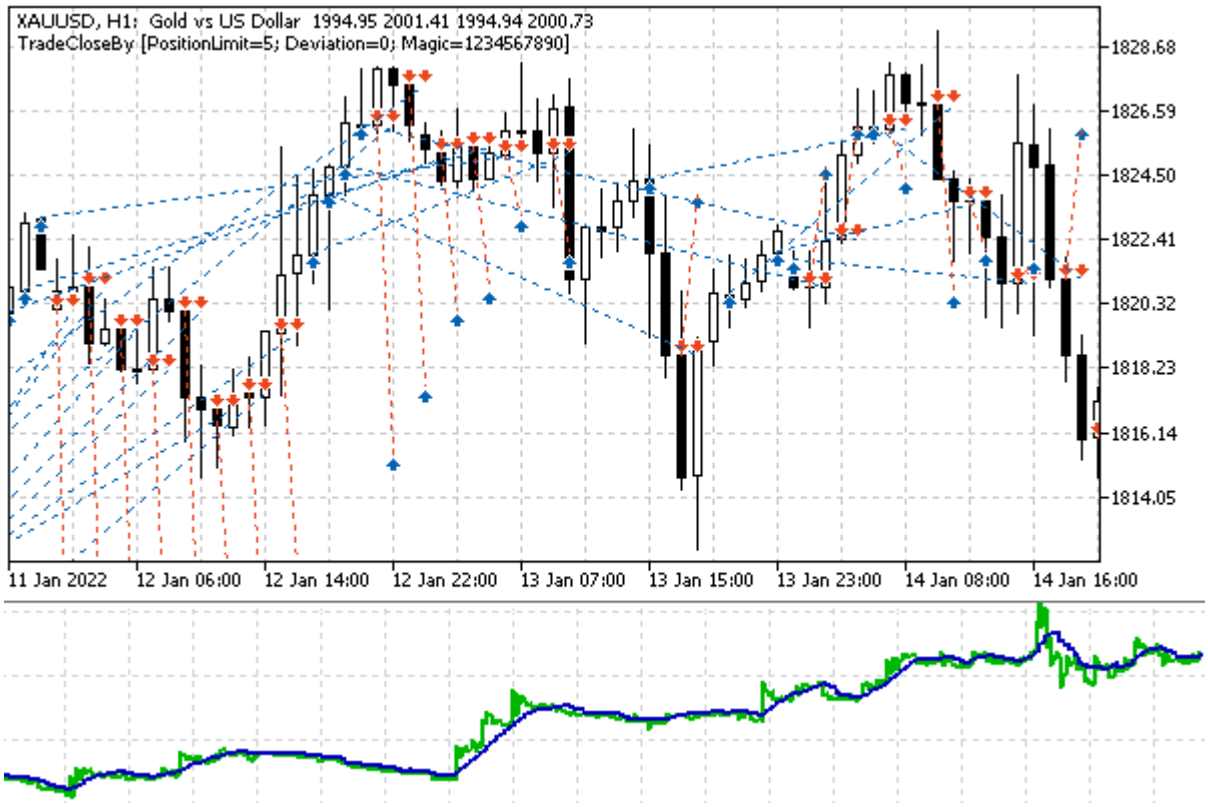
Para cualquier otro valor de *n*, debe comprobar si existe una señal (posiblemente repetida) para entrar en el mercado y ejecutarla llamando a *OpenPosition*.

```
      else if(type == ORDER_TYPE_BUY || type == ORDER_TYPE_SELL)
      {
         error = !OpenPosition(type);
      }
      ...
```

Por último, si todavía hay posiciones abiertas, pero están en la misma dirección, comprobamos si su número ha alcanzado el límite, en cuyo caso formamos una posición opuesta para «colapsar» dos de ellas en la siguiente barra (cerrando así una de cualquier posición de las antiguas).

```
      else if(n < 0)
      {
         if(-n >= (int)PositionLimit)
         {
            if(ArraySize(ticketsLong) > 0)
            {
               error = !OpenPosition(ORDER_TYPE_SELL);
            }
            else // (ArraySize(ticketsShort) > 0)
            {
               error = !OpenPosition(ORDER_TYPE_BUY);
            }
         }
      }
   }
```

Vamos a ejecutar el Asesor Experto en el probador en XAUUSD, H1 desde principios de 2022, con la configuración predeterminada. A continuación se muestra el gráfico con las posiciones en el proceso del programa, así como la curva de balance.

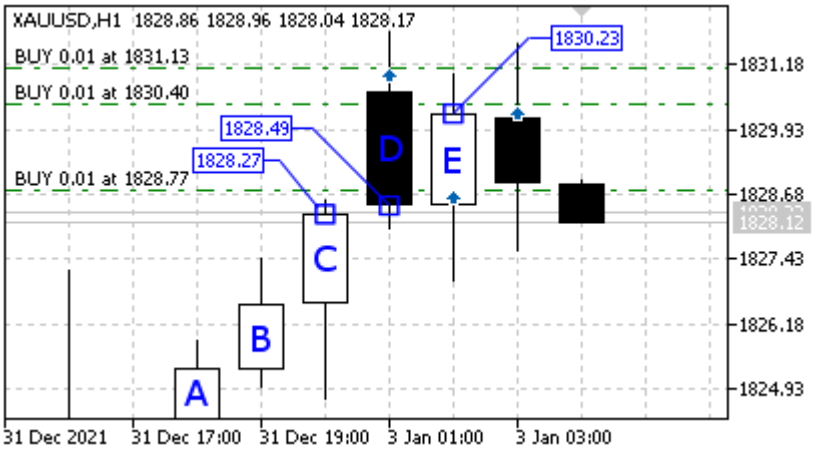


Resultados de la prueba TradeCloseBy en XAUUSD, H1

Es fácil encontrar en el registro los momentos en los que finaliza una tendencia (compra con tickets del nº 2 al nº 4), y comienzan a generarse transacciones en sentido opuesto (venta nº 5), tras lo cual se activa un cierre de contador.

```
2022.01.03 01:05:00 instant buy 0.01 XAUUSD at 1831.13 (1830.63 / 1831.13 / 1830.63)
2022.01.03 01:05:00 deal #2 buy 0.01 XAUUSD at 1831.13 done (based on order #2)
2022.01.03 01:05:00 deal performed [#2 buy 0.01 XAUUSD at 1831.13]
2022.01.03 01:05:00 order performed buy 0.01 at 1831.13 [#2 buy 0.01 XAUUSD at 1831.1
2022.01.03 01:05:00 Waiting for position for deal D=2
2022.01.03 01:05:00 OK New Order/Deal/Position
2022.01.03 02:00:00 instant buy 0.01 XAUUSD at 1828.77 (1828.47 / 1828.77 / 1828.47)
2022.01.03 02:00:00 deal #3 buy 0.01 XAUUSD at 1828.77 done (based on order #3)
2022.01.03 02:00:00 deal performed [#3 buy 0.01 XAUUSD at 1828.77]
2022.01.03 02:00:00 order performed buy 0.01 at 1828.77 [#3 buy 0.01 XAUUSD at 1828.7
2022.01.03 02:00:00 Waiting for position for deal D=3
2022.01.03 02:00:00 OK New Order/Deal/Position
2022.01.03 03:00:00 instant buy 0.01 XAUUSD at 1830.40 (1830.16 / 1830.40 / 1830.16)
2022.01.03 03:00:00 deal #4 buy 0.01 XAUUSD at 1830.40 done (based on order #4)
2022.01.03 03:00:00 deal performed [#4 buy 0.01 XAUUSD at 1830.40]
2022.01.03 03:00:00 order performed buy 0.01 at 1830.40 [#4 buy 0.01 XAUUSD at 1830.4
2022.01.03 03:00:00 Waiting for position for deal D=4
2022.01.03 03:00:00 OK New Order/Deal/Position
2022.01.03 05:00:00 instant sell 0.01 XAUUSD at 1826.22 (1826.22 / 1826.45 / 1826.22)
2022.01.03 05:00:00 deal #5 sell 0.01 XAUUSD at 1826.22 done (based on order #5)
2022.01.03 05:00:00 deal performed [#5 sell 0.01 XAUUSD at 1826.22]
2022.01.03 05:00:00 order performed sell 0.01 at 1826.22 [#5 sell 0.01 XAUUSD at 1826
2022.01.03 05:00:00 Waiting for position for deal D=5
2022.01.03 05:00:00 OK New Order/Deal/Position
2022.01.03 06:00:00 close position #5 sell 0.01 XAUUSD by position #2 buy 0.01 XAUUSD
2022.01.03 06:00:00 deal #6 buy 0.01 XAUUSD at 1831.13 done (based on order #6)
2022.01.03 06:00:00 deal #7 sell 0.01 XAUUSD at 1826.22 done (based on order #6)
2022.01.03 06:00:00 Positions collapse initiated
2022.01.03 06:00:00 OK CloseBy Order/Deal/Position
```

La transacción nº 3 es un artefacto interesante. Un lector atento observará que abrió por debajo de la anterior, violando aparentemente nuestra estrategia. De hecho, aquí no hay ningún error, y esto es consecuencia de que las condiciones de las señales están escritas de la forma más sencilla posible: sólo en función de los precios de cierre de las barras. Por lo tanto, una vela de reversión bajista (D), que abrió con un precio más alto y cerró por encima del final de la vela alcista anterior (C), generó una señal de compra. Esta situación se ilustra en la siguiente captura de pantalla:



Transacciones con tendencia al alza a precios de cierre

Todas las velas de la secuencia A, B, C, D y E cierran por encima de la anterior y animan a seguir comprando. Para excluir tales artefactos, habría que analizar además la dirección de las propias barras.

Lo último a lo que hay que prestar atención en este ejemplo es la función *OnInit*. Dado que el Asesor Experto utiliza la operación TRADE_ACTION_CLOSE_BY, aquí se comprueba la configuración de la cuenta y del símbolo de trabajo correspondientes.

```
int OnInit()
{
   ...
   if(AccountInfoInteger(ACCOUNT_MARGIN_MODE) != ACCOUNT_MARGIN_MODE_RETAIL_HEDGING)
   {
      Alert("An account with hedging is required for this EA!");
      return INIT_FAILED;
   }

   if((SymbolInfoInteger(_Symbol, SYMBOL_ORDER_MODE) & SYMBOL_ORDER_CLOSEBY) == 0)
   {
      Alert("'Close By' mode is not supported for ", _Symbol);
      return INIT_FAILED;
   }

   return INIT_SUCCEEDED;
}
```

Si una de las propiedades no admite cierre cruzado, el Asesor Experto no podrá seguir funcionando. Al crear robots de trabajo, estas comprobaciones, por regla general, se llevan a cabo dentro del algoritmo de trading y cambian el programa a modos alternativos; en concreto, a un único cierre de posiciones y al mantenimiento de una posición agregada en caso de compensación.

6.4.19 Colocar una orden pendiente

En Tipos de órdenes hemos considerado, en teoría, todas las opciones de colocación de órdenes pendientes que admite la plataforma. Desde un punto de vista práctico, las órdenes se crean utilizando las funciones *OrderSend/OrderSendAsync*, para las que la estructura de solicitud *MqlTradeRequest* se rellena previamente de acuerdo con reglas especiales. En concreto, el campo *action* debe contener el valor TRADE_ACTION_PENDING de la enumeración ENUM_TRADE_REQUEST_ACTIONS. Teniendo esto en cuenta, los siguientes campos son obligatorios:

- action
- symbol
- volume
- price
- type (el valor por defecto 0 corresponde a ORDER_TYPE_BUY)
- type_filling (por defecto 0 corresponde a ORDER_FILLING_FOK)
- type_time (el valor por defecto 0 corresponde a ORDER_TIME_GTC)
- expiration (por defecto 0, no se utiliza para ORDER_TIME_GTC)

Si los valores predeterminados cero son adecuados para la tarea, pueden omitirse algunos de los cuatro últimos campos.

El campo *stoplimit* es obligatorio sólo para las órdenes de los tipos ORDER_TYPE_BUY_STOP_LIMIT y ORDER_TYPE_SELL_STOP_LIMIT.

Los siguientes campos son opcionales:

- sl
- tp
- magic
- comment

Los valores cero en *sl* y *tp* indican la ausencia de niveles protectores.

Añadamos los métodos para comprobar valores y rellenar campos en nuestras estructuras en el archivo *MqlTradeSync.mqh*. El principio de formación de todos los tipos de órdenes es el mismo, así que analicemos un par de casos especiales de colocación de órdenes de compra y venta limitadas. El resto de tipos sólo se diferenciarán en el valor del tipo de campo. Los métodos públicos con un conjunto completo de campos obligatorios, así como los niveles de protección, se denominan según los tipos: *buyLimit* y *sellLimit*.

```
ulong buyLimit(const string name, const double lot, const double p,
   const double stop = 0, const double take = 0,
   ENUM_ORDER_TYPE_TIME duration = ORDER_TIME_GTC, datetime until = 0)
{
   type = ORDER_TYPE_BUY_LIMIT;
   return _pending(name, lot, p, stop, take, duration, until);
}

ulong sellLimit(const string name, const double lot, const double p,
   const double stop = 0, const double take = 0,
   ENUM_ORDER_TYPE_TIME duration = ORDER_TIME_GTC, datetime until = 0)
{
   type = ORDER_TYPE_SELL_LIMIT;
   return _pending(name, lot, p, stop, take, duration, until);
}
```

Dado que la estructura contiene el campo *symbol* que se inicializa opcionalmente en el constructor, existen métodos similares sin el parámetro *name*: llaman a los métodos anteriores pasando *symbol* como primer parámetro. Así, para crear una orden con el mínimo esfuerzo, escriba lo siguiente:

```
MqlTradeRequestSync request; // by default uses the current chart symbol
request.buyLimit(volume, price);
```

La parte general del código para comprobar los valores pasados, normalizarlos, guardarlos en campos de estructura y crear una orden pendiente se ha trasladado al método de ayuda *_pending*. Devuelve el ticket de orden en caso de éxito, o 0 en caso de fallo.

```
ulong _pending(const string name, const double lot, const double p,
   const double stop = 0, const double take = 0,
   ENUM_ORDER_TYPE_TIME duration = ORDER_TIME_GTC, datetime until = 0,
   const double origin = 0)
{
   action = TRADE_ACTION_PENDING;
   if(!setSymbol(name)) return 0;
   if(!setVolumePrices(lot, p, stop, take, origin)) return 0;
   if(!setExpiration(duration, until)) return 0;
   if((SymbolInfoInteger(name, SYMBOL_ORDER_MODE) & (1 << (type / 2))) == 0)
   {
      Print(StringFormat("pending orders %s not allowed for %s",
         EnumToString(type), name));
      return 0;
   }
   ZeroMemory(result);
   if(OrderSend(this, result)) return result.order;
   return 0;
}
```

Ya sabemos cómo rellenar el campo *action* y cómo llamar a los métodos *setSymbol* y *setVolumePrices* desde operaciones de trading anteriores.

El operador multilínea *if* garantiza que la operación que se está preparando está presente entre las operaciones de símbolos permitidas especificadas en la propiedad SYMBOL_ORDER_MODE. La división de tipo entero *type* que divide por la mitad y desplaza el valor resultante en 1, establece el bit correcto en la máscara de tipos de orden permitidos. Esto se debe a la combinación de constantes en la enumeración ENUM_ORDER_TYPE y la propiedad SYMBOL_ORDER_MODE. Por ejemplo, ORDER_TYPE_BUY_STOP y ORDER_TYPE_SELL_STOP tienen los valores 4 y 5, que divididos por 2 dan ambos 2 (sin decimales). La operación *1 << 2* tiene un resultado 4 igual a SYMBOL_ORDER_STOP.

Una característica especial de las órdenes pendientes es el tratamiento de la fecha de vencimiento. El método *setExpiration* se ocupa de ello. En este método hay que asegurarse de que el modo de vencimiento especificado ENUM_ORDER_TYPE_TIME de *duration* se permite para el símbolo y la fecha y hora en *until* se rellenan correctamente.

```
bool setExpiration(ENUM_ORDER_TYPE_TIME duration = ORDER_TIME_GTC, datetime until
{
   const int modes = (int)SymbolInfoInteger(symbol, SYMBOL_EXPIRATION_MODE);
   if(((1 << duration) & modes) != 0)
   {
      type_time = duration;
      if((duration == ORDER_TIME_SPECIFIED || duration == ORDER_TIME_SPECIFIED_DAY
         && until == 0)
      {
         Print(StringFormat("datetime is 0, "
            "but it's required for order expiration mode %s",
            EnumToString(duration)));
         return false;
      }
      if(until > 0 && until <= TimeTradeServer())
      {
         Print(StringFormat("expiration datetime %s is in past, server time is %s"
            TimeToString(until), TimeToString(TimeTradeServer())));
         return false;
      }
      expiration = until;
   }
   else
   {
      Print(StringFormat("order expiration mode %s is not allowed for %s",
         EnumToString(duration), symbol));
      return false;
   }
   return true;
}
```

La máscara de bits de los modos permitidos está disponible en la propiedad SYMBOL_EXPIRATION_MODE. La combinación de bits en la máscara y las constantes ENUM_ORDER_TYPE_TIME es tal que basta con evaluar la expresión *1 << duration* y superponerla a la máscara: un valor distinto de cero indica la presencia del modo.

Para los modos ORDER_TIME_SPECIFIED y ORDER_TIME_SPECIFIED_DAY, el campo *expiration* con el valor específico *datetime* no puede estar vacío. Además, la fecha y hora especificadas no pueden estar en el pasado.

Dado que el método *_pending* presentado anteriormente envía una solicitud al servidor utilizando *OrderSend* al final, nuestro programa debe asegurarse de que la orden con el ticket recibido fue realmente creada (esto es especialmente importante para las órdenes Limit que pueden ser emitidas a un sistema de trading externo). Por lo tanto, en el método *completed*, que se utiliza para el control de «bloqueo» del resultado, añadiremos una rama para la operación TRADE_ACTION_PENDING.

```
bool completed()
{
   // old processing code
   // TRADE_ACTION_DEAL
   // TRADE_ACTION_SLTP
   // TRADE_ACTION_CLOSE_BY
   ...
   else if(action == TRADE_ACTION_PENDING)
   {
      return result.placed(timeout);
   }
   ...
   return false;
}
```

En la estructura *MqlTradeResultSync*, añadimos el método *placed*.

```
bool placed(const ulong msc = 1000)
{
   if(retcode != TRADE_RETCODE_DONE
      && retcode != TRADE_RETCODE_DONE_PARTIAL)
   {
      return false;
   }

   if(!wait(orderExist, msc))
   {
      Print("Waiting for order: #" + (string)order);
      return false;
   }
   return true;
}
```

Su tarea principal es esperar a que aparezca la orden utilizando la espera en la función *orderExist*: ya se ha utilizado en la primera fase de verificación de apertura de posición.

Para probar la nueva funcionalidad, vamos a implementar el Asesor Experto *PendingOrderSend.mq5*. Ello permite seleccionar el tipo de orden pendiente y todos sus atributos mediante variables de entrada, tras lo cual se ejecuta una solicitud de confirmación.

```
enum ENUM_ORDER_TYPE_PENDING
{                                                    // UI interface strings
   PENDING_BUY_STOP = ORDER_TYPE_BUY_STOP,           // ORDER_TYPE_BUY_STOP
   PENDING_SELL_STOP = ORDER_TYPE_SELL_STOP,         // ORDER_TYPE_SELL_STOP
   PENDING_BUY_LIMIT = ORDER_TYPE_BUY_LIMIT,         // ORDER_TYPE_BUY_LIMIT
   PENDING_SELL_LIMIT = ORDER_TYPE_SELL_LIMIT,       // ORDER_TYPE_SELL_LIMIT
   PENDING_BUY_STOP_LIMIT = ORDER_TYPE_BUY_STOP_LIMIT,    // ORDER_TYPE_BUY_STOP_LIMIT
   PENDING_SELL_STOP_LIMIT = ORDER_TYPE_SELL_STOP_LIMIT, // ORDER_TYPE_SELL_STOP_LIMI
};

input string Symbol;             // Symbol (empty = current _Symbol)
input double Volume;             // Volume (0 = minimal lot)
input ENUM_ORDER_TYPE_PENDING Type = PENDING_BUY_STOP;
input int Distance2SLTP = 0;     // Distance to SL/TP in points (0 = no)
input ENUM_ORDER_TYPE_TIME Expiration = ORDER_TIME_GTC;
input datetime Until = 0;
input ulong Magic = 1234567890;
input string Comment;
```

El Asesor Experto creará una nueva orden cada vez que se inicie o se cambien los parámetros. La eliminación de orden automática no se ofrece todavía. Hablaremos de este tipo de operación más adelante. A este respecto, no olvide borrar las órdenes manualmente.

La colocación de una orden única se realiza, como en algunos ejemplos anteriores, basándose en un temporizador (por lo tanto, primero debe asegurarse de que el mercado esté abierto).

```
void OnTimer()
{
   // execute once and wait for the user to change the settings
   EventKillTimer();

   const string symbol = StringLen(Symbol) == 0 ? _Symbol : Symbol;
   if(PlaceOrder((ENUM_ORDER_TYPE)Type, symbol, Volume,
      Distance2SLTP, Expiration, Until, Magic, Comment))
   {
      Alert("Pending order placed - remove it manually, please");
   }
}
```

La función *PlaceOrder* acepta todos los ajustes como parámetros, envía una solicitud y devuelve un indicador de éxito (ticket distinto de cero). Las órdenes de todos los tipos admitidos se ofrecen con distancias preestablecidas desde el precio actual que se calculan como parte del rango diario de cotizaciones.

```
ulong PlaceOrder(const ENUM_ORDER_TYPE type,
   const string symbol, const double lot,
   const int sltp, ENUM_ORDER_TYPE_TIME expiration, datetime until,
   const ulong magic = 0, const string comment = NULL)
{
   static double coefficients[] = // indexed by order type
   {
      0  ,   // ORDER_TYPE_BUY - not used
      0  ,   // ORDER_TYPE_SELL - not used
     -0.5,   // ORDER_TYPE_BUY_LIMIT - slightly below the price
     +0.5,   // ORDER_TYPE_SELL_LIMIT - slightly above the price
     +1.0,   // ORDER_TYPE_BUY_STOP - far above the price
     -1.0,   // ORDER_TYPE_SELL_STOP - far below the price
     +0.7,   // ORDER_TYPE_BUY_STOP_LIMIT - average above the price
     -0.7,   // ORDER_TYPE_SELL_STOP_LIMIT - average below the price
      0  ,   // ORDER_TYPE_CLOSE_BY - not used
   };
   ...
```

Por ejemplo, el coeficiente de -0.5 para ORDER_TYPE_BUY_LIMIT significa que la orden se colocará por debajo del precio actual en la mitad del rango diario (rebote dentro del rango), y el coeficiente de +1.0 para ORDER_TYPE_BUY_STOP significa que la orden se situará en el límite superior del rango (ruptura).

El rango diario propiamente dicho se calcula del siguiente modo:

```
   const double range = iHigh(symbol, PERIOD_D1, 1) - iLow(symbol, PERIOD_D1, 1);
   Print("Autodetected daily range: ", (float)range);
   ...
```

A continuación se indican los valores de punto y volumen que serán necesarios.

```
   const double volume = lot == 0 ? SymbolInfoDouble(symbol, SYMBOL_VOLUME_MIN) : lot
   const double point = SymbolInfoDouble(symbol, SYMBOL_POINT);
```

El nivel de precios para colocar una orden se calcula en la variable *price* a partir de los coeficientes dados del rango total.

```
   const double price = TU::GetCurrentPrice(type, symbol) + range * coefficients[type
```

El campo *stoplimit* sólo debe rellenarse para las órdenes *_STOP_LIMIT. Sus valores se almacenan en la variable *origin*.

```
   const bool stopLimit =
      type == ORDER_TYPE_BUY_STOP_LIMIT ||
      type == ORDER_TYPE_SELL_STOP_LIMIT;
   const double origin = stopLimit ? TU::GetCurrentPrice(type, symbol) : 0;
```

Cuando se activan estos dos tipos de órdenes se coloca una nueva orden pendiente al precio actual. De hecho, en este escenario, el precio se mueve desde el valor actual hasta el nivel *price*, donde se activa la orden, y por lo tanto el precio «actual anterior» se convierte en el nivel de rebote correcto indicado por una orden Limit. A continuación ilustraremos esta situación.

Los niveles de protección se determinan utilizando el objeto *TU::TradeDirection*. Para las órdenes Stop-Limit, calculamos a partir de *origin*.

```
TU::TradeDirection dir(type);
const double stop = sltp == 0 ? 0 :
   dir.negative(stopLimit ? origin : price, sltp * point);
const double take = sltp == 0 ? 0 :
   dir.positive(stopLimit ? origin : price, sltp * point);
```

A continuación, se describe la estructura y se rellenan los campos opcionales.

```
MqlTradeRequestSync request(symbol);

request.magic = magic;
request.comment = comment;
// request.type_filling = SYMBOL_FILLING_FOK;
```

Aquí puede seleccionar el modo de relleno. Por defecto, *MqlTradeRequestSync* selecciona automáticamente el primero de los modos permitidos, ENUM_ORDER_TYPE_FILLING.

Dependiendo del tipo de orden elegido por el usuario, llamamos a uno u otro método de trading.

```
ResetLastError();
// fill in and check the required fields, send the request
ulong order = 0;
switch(type)
{
case ORDER_TYPE_BUY_STOP:
   order = request.buyStop(volume, price, stop, take, expiration, until);
   break;
case ORDER_TYPE_SELL_STOP:
   order = request.sellStop(volume, price, stop, take, expiration, until);
   break;
case ORDER_TYPE_BUY_LIMIT:
   order = request.buyLimit(volume, price, stop, take, expiration, until);
   break;
case ORDER_TYPE_SELL_LIMIT:
   order = request.sellLimit(volume, price, stop, take, expiration, until);
   break;
case ORDER_TYPE_BUY_STOP_LIMIT:
   order = request.buyStopLimit(volume, price, origin, stop, take, expiration, unt
   break;
case ORDER_TYPE_SELL_STOP_LIMIT:
   order = request.sellStopLimit(volume, price, origin, stop, take, expiration, un
   break;
}
...
```

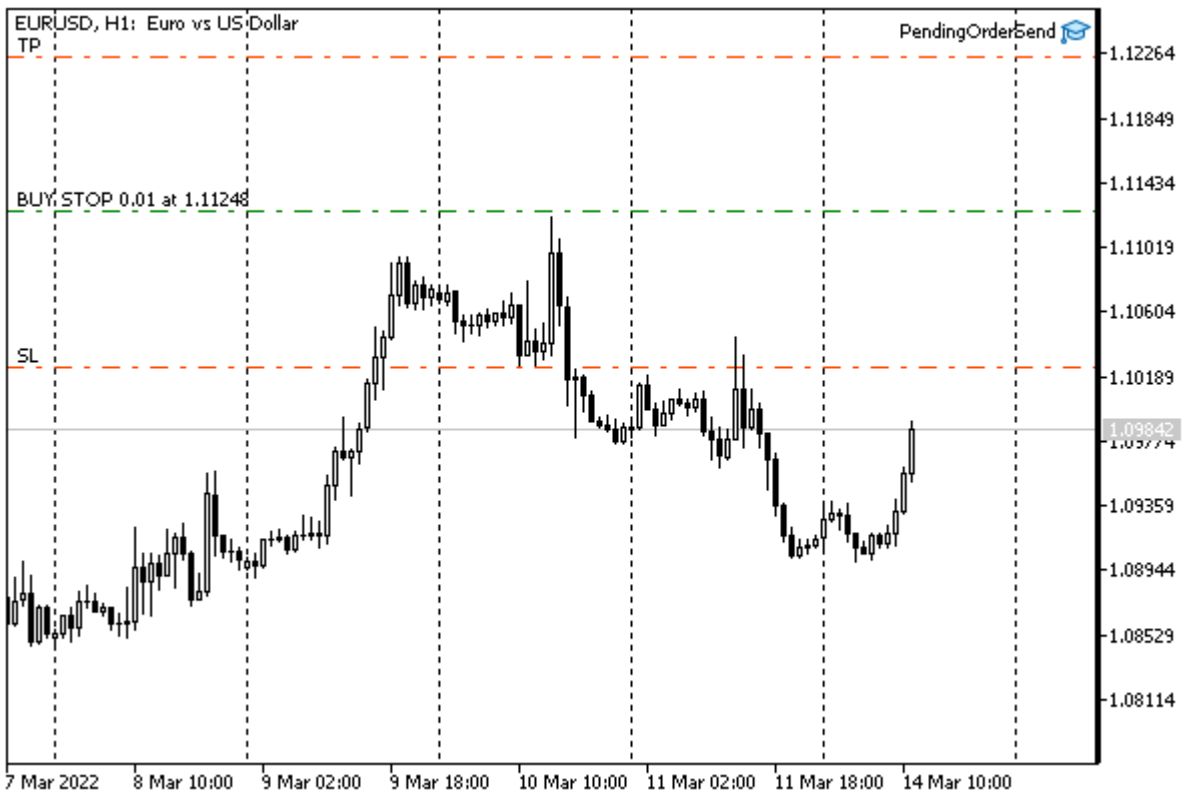
Si el ticket se ha recibido, esperamos a que aparezca en el entorno de trading del terminal.

```
      if(order != 0)
      {
         Print("OK order sent: #=", order);
         if(request.completed()) // expect result (order confirmation)
         {
            Print("OK order placed");
         }
      }
      Print(TU::StringOf(request));
      Print(TU::StringOf(request.result));
      return order;
   }
```

Vamos a ejecutar el Asesor Experto en el gráfico EURUSD con la configuración predeterminada y, además, seleccionamos la distancia a los niveles de protección de 1000 puntos. Veremos las siguientes entradas en el registro (suponiendo que la configuración por defecto coincide con los permisos para EURUSD de su cuenta).

```
Autodetected daily range: 0.01413
OK order sent: #=1282106395
OK order placed
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_FILLING_FOK, »
  » @ 1.11248, SL=1.10248, TP=1.12248, ORDER_TIME_GTC, M=1234567890
DONE, #=1282106395, V=0.01, Request executed, Req=91
Alert: Pending order placed - remove it manually, please
```

Este es el aspecto que tiene en el gráfico:



Orden ORDER_TYPE_BUY_STOP pendiente

Eliminemos la orden manualmente y cambiemos el tipo de orden a ORDER_TYPE_BUY_STOP_LIMIT. El resultado es una imagen más compleja:

Orden ORDER_TYPE_BUY_STOP_LIMIT pendiente

El precio en el que se encuentra el par superior de líneas discontinuas es el precio de activación de la orden, como resultado de lo cual se colocará una orden ORDER_TYPE_BUY_LIMIT en el nivel de precios actual, con los valores *Stop Loss* y *Take Profit* marcados con líneas rojas. El nivel *Take Profit* de la futura orden ORDER_TYPE_BUY_LIMIT coincide prácticamente con el nivel de activación de la orden preliminar recién creada ORDER_TYPE_BUY_STOP_LIMIT.

Como ejemplo adicional para el autoaprendizaje, se incluye con el libro un Asesor Experto *AllPendingsOrderSend.mq5*; el Asesor Experto establece 6 órdenes pendientes a la vez: una de cada tipo.

Órdenes pendientes de todo tipo

Como resultado de ejecutarlo con la configuración predeterminada, puede obtener entradas de registro como las siguientes:

```
Autodetected daily range: 0.01413
OK order placed: #=1282032135
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, »
  » @ 1.08824, ORDER_TIME_GTC, M=1234567890
DONE, #=1282032135, V=0.01, Request executed, Req=73
OK order placed: #=1282032136
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, »
  » @ 1.10238, ORDER_TIME_GTC, M=1234567890
DONE, #=1282032136, V=0.01, Request executed, Req=74
OK order placed: #=1282032138
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_FILLING_FOK, »
  » @ 1.10944, ORDER_TIME_GTC, M=1234567890
DONE, #=1282032138, V=0.01, Request executed, Req=75
OK order placed: #=1282032141
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP, V=0.01, ORDER_FILLING_FOK, »
  » @ 1.08118, ORDER_TIME_GTC, M=1234567890
DONE, #=1282032141, V=0.01, Request executed, Req=76
OK order placed: #=1282032142
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, »
  » @ 1.10520, X=1.09531, ORDER_TIME_GTC, M=1234567890
DONE, #=1282032142, V=0.01, Request executed, Req=77
OK order placed: #=1282032144
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK,
  » @ 1.08542, X=1.09531, ORDER_TIME_GTC, M=1234567890
DONE, #=1282032144, V=0.01, Request executed, Req=78
Alert: 6 pending orders placed - remove them manually, please
```

6.4.20 Modificar una orden pendiente

MetaTrader 5 permite modificar ciertas propiedades de una orden pendiente, incluyendo el precio de activación, los niveles de protección y la fecha de vencimiento. Las propiedades principales, como el tipo de orden o el volumen, no pueden modificarse. En tales casos, deberá eliminar la orden y sustituirla por otra. El único caso en el que el tipo de orden puede ser modificado por el propio servidor es la activación de una orden stop limitada, que se convierte en la orden Limit correspondiente.

La modificación programática de las órdenes se realiza mediante la operación TRADE_ACTION_MODIFY: es esta constante la que hay que escribir en el campo *action* de la estructura *MqlTradeRequest* antes de enviarlo al servidor mediante la función *OrderSend* o *OrderSendAsync*. El ticket de la orden modificada se indica en el campo *order*. Teniendo en cuenta *action* y *order*, la lista completa de campos obligatorios para esta operación incluye:

- action
- order
- price
- type_time (el valor por defecto 0 corresponde a ORDER_TIME_GTC)
- expiration (por defecto 0, no importante para ORDER_TIME_GTC)
- type_filling (por defecto 0 corresponde a ORDER_FILLING_FOK)
- stoplimit (sólo para órdenes de los tipos ORDER_TYPE_BUY_STOP_LIMIT y ORDER_TYPE_SELL_STOP_LIMIT)

Campos opcionales:

- sl
- tp

Si ya se han establecido niveles de protección para la orden, deben especificarse para que puedan guardarse. Los valores cero indican la supresión de *Stop Loss* y/o *Take Profit*.

En la estructura *MqlTradeRequestSync* (*MqlTradeSync.mqh*), la implementación de la modificación de la orden se sitúa en el método *modify*.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   ...
   bool modify(const ulong ticket,
      const double p, const double stop = 0, const double take = 0,
      ENUM_ORDER_TYPE_TIME duration = ORDER_TIME_GTC, datetime until = 0,
      const double origin = 0)
   {
      if(!OrderSelect(ticket)) return false;

      action = TRADE_ACTION_MODIFY;
      order = ticket;

      // the following fields are needed for checks inside subfunctions
      type = (ENUM_ORDER_TYPE)OrderGetInteger(ORDER_TYPE);
      symbol = OrderGetString(ORDER_SYMBOL);
      volume = OrderGetDouble(ORDER_VOLUME_CURRENT);

      if(!setVolumePrices(volume, p, stop, take, origin)) return false;
      if(!setExpiration(duration, until)) return false;
      ZeroMemory(result);
      return OrderSend(this, result);
   }
```

La ejecución real de la solicitud se realiza de nuevo en el método *completed*, en la rama dedicada del operador *if*.

```
bool completed()
{
   ...
   else if(action == TRADE_ACTION_MODIFY)
   {
      result.order = order;
      result.bid = sl;
      result.ask = tp;
      result.price = price;
      result.volume = stoplimit;
      return result.modified(timeout);
   }
   ...
}
```

Para que la estructura *MqlTradeResultSync* conozca los nuevos valores de las propiedades de la orden editada y pueda compararlos con el resultado, los escribimos en campos libres (no son rellenados por el servidor en este tipo de solicitud). Además, en el método *modified*, la estructura resultante está a la espera de que se aplique la modificación.

```
struct MqlTradeResultSync: public MqlTradeResult
{
   ...
   bool modified(const ulong msc = 1000)
   {
      if(retcode != TRADE_RETCODE_DONE && retcode != TRADE_RETCODE_PLACED)
      {
         return false;
      }

      if(!wait(orderModified, msc))
      {
         Print("Order not found in environment: #" + (string)order);
         return false;
      }
      return true;
   }

   static bool orderModified(MqlTradeResultSync &ref)
   {
      if(!(OrderSelect(ref.order) || HistoryOrderSelect(ref.order)))
      {
         Print("OrderSelect failed: #=" + (string)ref.order);
         return false;
      }
      return TU::Equal(ref.bid, OrderGetDouble(ORDER_SL))
         && TU::Equal(ref.ask, OrderGetDouble(ORDER_TP))
         && TU::Equal(ref.price, OrderGetDouble(ORDER_PRICE_OPEN))
         && TU::Equal(ref.volume, OrderGetDouble(ORDER_PRICE_STOPLIMIT));
   }
```

Aquí vemos cómo se leen las propiedades de orden utilizando la función *OrderGetDouble* y se compara con los valores especificados. Todo esto sucede según el procedimiento ya conocido, en un bucle dentro de la función wait, dentro de un tiempo de espera determinado de *msc* (1000 milisegundos por defecto).

Como ejemplo, vamos a utilizar el Asesor Experto *PendingOrderModify.mq5*, mientras heredamos algunos fragmentos de código de *PendingOrderSend.mq5*. En concreto, un conjunto de parámetros de entrada y la función *PlaceOrder* para crear una nueva orden. Se utiliza en el primer lanzamiento si no hay ninguna orden para la combinación dada del símbolo y número *Magic*, asegurando así que el Asesor Experto tiene algo que modificar.

Se necesitaba una nueva función para encontrar una orden adecuada: *GetMyOrder*. Es muy similar a la función *GetMyPosition*, que se utilizó en el ejemplo con seguimiento de posición (*TrailingStop.mq5*) para encontrar una posición adecuada. La finalidad de las funciones integradas en la API de MQL5 que se utilizan en *GetMyOrder* debería quedar clara a partir de sus nombres, y la descripción técnica se presentará en secciones independientes.

```
ulong GetMyOrder(const string name, const ulong magic)
{
   for(int i = 0; i < OrdersTotal(); ++i)
   {
      ulong t = OrderGetTicket(i);
      if(OrderGetInteger(ORDER_MAGIC) == magic
         && OrderGetString(ORDER_SYMBOL) == name)
      {
         return t;
      }
   }

   return 0;
}
```

Ahora falta el parámetro de entrada *Distance2SLTP*. En su lugar, el nuevo Asesor Experto calculará automáticamente el rango diario de precios y colocará niveles de protección a una distancia de la mitad de este rango. Al comienzo de cada día, se recalcularán el rango y los nuevos niveles de los campos *sl* y *tp*. Las solicitudes de modificación de órdenes se generarán en función de los nuevos valores.

Las órdenes pendientes que se activen y se conviertan en posiciones se cerrarán al alcanzar *Stop Loss* o *Take Profit*. El terminal puede informar al programa MQL sobre la activación de órdenes pendientes y el cierre de posiciones si usted describe en él manejadores de evento de trading. Esto permitiría, por ejemplo, evitar la creación de una nueva orden si existe una posición abierta. No obstante, también puede utilizarse la estrategia actual. Así pues, nos ocuparemos de los eventos más adelante.

La lógica principal del Asesor Experto se implementa en el manejador *OnTick*.

```
void OnTick()
{
   static datetime lastDay = 0;
   static const uint DAYLONG = 60 * 60 * 24; // number of seconds in a day
   //discard the "fractional" part, i.e. time
   if(TimeTradeServer() / DAYLONG * DAYLONG == lastDay) return;
   ...
```

Dos líneas al principio de la función garantizan que el algoritmo se ejecute una vez al principio de cada día. Para ello, calculamos la fecha actual sin hora y la comparamos con el valor de la variable *lastDay* que contiene la última fecha correcta. Por supuesto, el estado de éxito o error queda claro al final de la función, así que volveremos a ello más adelante.

A continuación, se calcula el rango de precios del día anterior.

```
   const string symbol = StringLen(Symbol) == 0 ? _Symbol : Symbol;
   const double range = iHigh(symbol, PERIOD_D1, 1) - iLow(symbol, PERIOD_D1, 1);
   Print("Autodetected daily range: ", (float)range);
   ...
```

Dependiendo de si existe o no una orden en la función *GetMyOrder*, crearemos una nueva orden a través de *PlaceOrder* o editaremos la existente utilizando *ModifyOrder*.

```
   uint retcode = 0;
   ulong ticket = GetMyOrder(symbol, Magic);
   if(!ticket)
   {
      retcode = PlaceOrder((ENUM_ORDER_TYPE)Type, symbol, Volume,
         range, Expiration, Until, Magic);
   }
   else
   {
      retcode = ModifyOrder(ticket, range, Expiration, Until);
   }
   ...
```

Ambas funciones, *PlaceOrder* y *ModifyOrder*, trabajan sobre la base de los parámetros de entrada del Asesor Experto y el rango de precios encontrado. Devuelven el estado de la solicitud, que habrá que analizar de alguna manera para decidir qué acción tomar:

- Actualizar la variable *lastDay* si la solicitud tiene éxito (la orden se ha actualizado y el Asesor Experto duerme hasta el comienzo del día siguiente).
- Dejar el día anterior en *lastDay* durante algún tiempo para volver a intentarlo en los próximos ticks si hay problemas temporales (por ejemplo, la sesión de trading aún no ha comenzado).
- Detener el Asesor Experto si se detectan problemas graves (por ejemplo, el tipo de orden seleccionada o la dirección de la operación no están permitidos en el símbolo).

```
   ...
   if(/* some kind of retcode analysis */)
   {
      lastDay = TimeTradeServer() / DAYLONG * DAYLONG;
   }
}
```

En la sección Cierre de una posición: total y parcial hemos utilizado un análisis simplificado con la macro IS_TANGIBLE, que daba una respuesta en las categorías «sí» y «no» para indicar si había error o no. Evidentemente, este planteamiento debe mejorarse, y volveremos sobre este tema próximamente. Por ahora, nos centraremos en la funcionalidad principal del Asesor Experto.

El código fuente de la función *PlaceOrder* prácticamente no ha cambiado con respecto al ejemplo anterior. *ModifyOrder* se muestra a continuación.

Recordemos que determinamos la localización de las órdenes en función del rango diario, al que se aplicó la tabla de coeficientes. El principio no ha cambiado; sin embargo, dado que ahora tenemos dos funciones que trabajan con órdenes, *PlaceOrder* y *ModifyOrder*, la tabla *Coefficients* se sitúa en un contexto global. No lo repetiremos aquí y pasaremos directamente a la función *ModifyOrder*.

```
uint ModifyOrder(const ulong ticket, const double range,
   ENUM_ORDER_TYPE_TIME expiration, datetime until)
{
   // default values
   const string symbol = OrderGetString(ORDER_SYMBOL);
   const double point = SymbolInfoDouble(symbol, SYMBOL_POINT);
   ...
```

Los niveles de precios se calculan en función del tipo de orden y del rango pasado.

```
   const ENUM_ORDER_TYPE type = (ENUM_ORDER_TYPE)OrderGetInteger(ORDER_TYPE);
   const double price = TU::GetCurrentPrice(type, symbol) + range * Coefficients[type

   // origin is filled only for orders *_STOP_LIMIT
   const bool stopLimit =
      type == ORDER_TYPE_BUY_STOP_LIMIT ||
      type == ORDER_TYPE_SELL_STOP_LIMIT;
   const double origin = stopLimit ? TU::GetCurrentPrice(type, symbol) : 0;

   TU::TradeDirection dir(type);
   const int sltp = (int)(range / 2 / point);
   const double stop = sltp == 0 ? 0 :
      dir.negative(stopLimit ? origin : price, sltp * point);
   const double take = sltp == 0 ? 0 :
      dir.positive(stopLimit ? origin : price, sltp * point);
   ...
```

Después de calcular todos los valores, creamos un objeto de la estructura *MqlTradeRequestSync* y ejecutamos la solicitud.

```
   MqlTradeRequestSync request(symbol);

   ResetLastError();
   // pass the data for the fields, send the order and wait for the result
   if(request.modify(ticket, price, stop, take, expiration, until, origin)
      && request.completed())
   {
      Print("OK order modified: #=", ticket);
   }

   Print(TU::StringOf(request));
   Print(TU::StringOf(request.result));
   return request.result.retcode;
}
```

Para analizar *retcode* que tenemos que ejecutar en el bloque de llamada dentro de *OnTick*, se desarrolló un nuevo mecanismo que complementaba el archivo *TradeRetcode.mqh*. Todos los códigos de devolución del servidor se dividen en varios grupos de «gravedad», descritos por los elementos de la enumeración TRADE_RETCODE_SEVERITY.

```
enum TRADE_RETCODE_SEVERITY
{
   SEVERITY_UNDEFINED,    // something non-standard - just output to the log
   SEVERITY_NORMAL,       // normal operation
   SEVERITY_RETRY,        // try updating environment/prices again (probably several t
   SEVERITY_TRY_LATER,    // we should wait and try again
   SEVERITY_REJECT,       // request denied, probably(!) you can try again
                          //
   SEVERITY_INVALID,      // need to fix the request
   SEVERITY_LIMITS,       // need to check the limits and fix the request
   SEVERITY_PERMISSIONS, // it is required to notify the user and change the program/
   SEVERITY_ERROR,        // stop, output information to the log and to the user
};
```

De forma simplista, la primera mitad corresponde a errores recuperables: suele bastar con esperar un poco y reintentar la solicitud. La segunda mitad requiere que cambie el contenido de la solicitud, compruebe la configuración de la cuenta o del símbolo, los permisos para el programa y, en el peor de los casos, que deje de operar. Aquellos que lo deseen pueden dibujar una línea separadora condicional, no después de SEVERITY_REJECT, como se resalta visualmente ahora, sino antes.

La división de todos los códigos en grupos se realiza mediante la función *TradeCodeSeverity* (indicada con abreviaturas).

```
TRADE_RETCODE_SEVERITY TradeCodeSeverity(const uint retcode)
{
   static const TRADE_RETCODE_SEVERITY severities[] =
   {
      ...
      SEVERITY_RETRY,       // REQUOTE (10004)
      SEVERITY_UNDEFINED,
      SEVERITY_REJECT,      // REJECT (10006)
      SEVERITY_NORMAL,      // CANCEL (10007)
      SEVERITY_NORMAL,      // PLACED (10008)
      SEVERITY_NORMAL,      // DONE (10009)
      SEVERITY_NORMAL,      // DONE_PARTIAL (10010)
      SEVERITY_ERROR,       // ERROR (10011)
      SEVERITY_RETRY,       // TIMEOUT (10012)
      SEVERITY_INVALID,     // INVALID (10013)
      SEVERITY_INVALID,     // INVALID_VOLUME (10014)
      SEVERITY_INVALID,     // INVALID_PRICE (10015)
      SEVERITY_INVALID,     // INVALID_STOPS (10016)
      SEVERITY_PERMISSIONS, // TRADE_DISABLED (10017)
      SEVERITY_TRY_LATER,   // MARKET_CLOSED (10018)
      SEVERITY_LIMITS,      // NO_MONEY (10019)
      ...
   };

   if(retcode == 0) return SEVERITY_NORMAL;
   if(retcode < 10000 || retcode > HEDGE_PROHIBITED) return SEVERITY_UNDEFINED;
   return severities[retcode - 10000];
}
```

Gracias a esta funcionalidad, el manejador *OnTick* puede complementarse con un tratamiento de errores «inteligente». Una variable estática *RetryFrequency* almacena la frecuencia con la que el programa intentará repetir la solicitud en caso de errores no críticos. La última vez que se realizó un intento de este tipo se almacenó en la variable *RetryRecordTime*.

```
void OnTick()
{
   ...
   const static int DEFAULT_RETRY_TIMEOUT = 1; // seconds
   static int RetryFrequency = DEFAULT_RETRY_TIMEOUT;
   static datetime RetryRecordTime = 0;
   if(TimeTradeServer() - RetryRecordTime < RetryFrequency) return;
   ...
```

Una vez que la función *PlaceOrder* o *ModifyOrder* devuelve el valor de *retcode*, nos enteramos de su gravedad y, en función de la misma, elegimos una de las tres alternativas: detener el Asesor Experto, esperar un tiempo u operar de forma regular (marcando la modificación exitosa de la orden por el día actual en *lastDay*).

```
   const TRADE_RETCODE_SEVERITY severity = TradeCodeSeverity(retcode);
   if(severity >= SEVERITY_INVALID)
   {
      Alert("Can't place/modify pending order, EA is stopped");
      RetryFrequency = INT_MAX;
   }
   else if(severity >= SEVERITY_RETRY)
   {
      RetryFrequency += (int)sqrt(RetryFrequency + 1);
      RetryRecordTime = TimeTradeServer();
      PrintFormat("Problems detected, waiting for better conditions "
         "(timeout enlarged to %d seconds)",
         RetryFrequency);
   }
   else
   {
      if(RetryFrequency > DEFAULT_RETRY_TIMEOUT)
      {
         RetryFrequency = DEFAULT_RETRY_TIMEOUT;
         PrintFormat("Timeout restored to %d second", RetryFrequency);
      }
      lastDay = TimeTradeServer() / DAYLONG * DAYLONG;
   }
```

En caso de problemas repetidos clasificados como solucionables, el tiempo de espera de *RetryFrequency* aumenta gradualmente con cada error subsiguiente, pero se restablece a 1 segundo cuando la solicitud se procesa correctamente.

Cabe señalar que los métodos de la estructura aplicada *MqlTradeRequestSync* comprueban la corrección de un gran número de combinaciones de parámetros y, si se encuentran problemas, interrumpen el proceso antes de la llamada a *SendRequest*. Este comportamiento está activado por defecto, pero puede desactivarse definiendo una macro RETURN(X) vacía antes de la directiva *#include* con *MqlTradeSync.mqh*.

```
#define RETURN(X)
#include <MQL5Book/MqlTradeSync.mqh>
```

Con esta definición de macro, las comprobaciones no sólo imprimirán advertencias en el registro sino que seguirán ejecutando métodos hasta la llamada a *SendRequest*.

En cualquier caso, tras llamar a uno u otro método de la estructura *MqlTradeResultSync*, el código de error se añadirá a *retcode*. Esto lo hará el servidor o los algoritmos de comprobación de la estructura *MqlTradeRequestSync* (aquí utilizamos el hecho de que la instancia *MqlTradeResultSync* está incluida dentro de *MqlTradeRequestSync*). No se proporciona aquí la descripción de la devolución de códigos de error y el uso de la macro RETURN en los métodos *MqlTradeRequestSync* en aras de la brevedad. Los interesados pueden ver el código fuente completo en el archivo *MqlTradeSync.mqh*.

Vamos a ejecutar el Asesor Experto *PendingOrderModify.mq5* en el probador, con el modo visual activado, utilizando los datos de XAUUSD, H1 (todos los ticks o el modo de ticks reales). Con la configuración por defecto, el Asesor Experto colocará órdenes del tipo ORDER_TYPE_BUY_STOP con un lote mínimo. Comprobemos en el registro y en el historial de operaciones que el programa coloca órdenes pendientes y las modifica al principio de cada día.

```
2022.01.03 01:05:00 Autodetected daily range: 14.37
2022.01.03 01:05:00 buy stop 0.01 XAUUSD at 1845.73 sl: 1838.55 tp: 1852.91 (1830.63
2022.01.03 01:05:00 OK order placed: #=2
2022.01.03 01:05:00 TRADE_ACTION_PENDING, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_
» @ 1845.73, SL=1838.55, TP=1852.91, ORDER_TIME_GTC, M=1234567890
2022.01.03 01:05:00 DONE, #=2, V=0.01, Bid=1830.63, Ask=1831.36, Request executed
2022.01.04 01:05:00 Autodetected daily range: 33.5
2022.01.04 01:05:00 order modified [#2 buy stop 0.01 XAUUSD at 1836.56]
2022.01.04 01:05:00 OK order modified: #=2
2022.01.04 01:05:00 TRADE_ACTION_MODIFY, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_F
» @ 1836.56, SL=1819.81, TP=1853.31, ORDER_TIME_GTC, #=2
2022.01.04 01:05:00 DONE, #=2, @ 1836.56, Bid=1819.81, Ask=1853.31, Request executed,
2022.01.05 01:05:00 Autodetected daily range: 18.23
2022.01.05 01:05:00 order modified [#2 buy stop 0.01 XAUUSD at 1832.56]
2022.01.05 01:05:00 OK order modified: #=2
2022.01.05 01:05:00 TRADE_ACTION_MODIFY, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_F
» @ 1832.56, SL=1823.45, TP=1841.67, ORDER_TIME_GTC, #=2
2022.01.05 01:05:00 DONE, #=2, @ 1832.56, Bid=1823.45, Ask=1841.67, Request executed,
...
2022.01.11 01:05:00 Autodetected daily range: 11.96
2022.01.11 01:05:00 order modified [#2 buy stop 0.01 XAUUSD at 1812.91]
2022.01.11 01:05:00 OK order modified: #=2
2022.01.11 01:05:00 TRADE_ACTION_MODIFY, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_F
» @ 1812.91, SL=1806.93, TP=1818.89, ORDER_TIME_GTC, #=2
2022.01.11 01:05:00 DONE, #=2, @ 1812.91, Bid=1806.93, Ask=1818.89, Request executed,
2022.01.11 18:10:58 order [#2 buy stop 0.01 XAUUSD at 1812.91] triggered
2022.01.11 18:10:58 deal #2 buy 0.01 XAUUSD at 1812.91 done (based on order #2)
2022.01.11 18:10:58 deal performed [#2 buy 0.01 XAUUSD at 1812.91]
2022.01.11 18:10:58 order performed buy 0.01 at 1812.91 [#2 buy stop 0.01 XAUUSD at 1
2022.01.11 20:28:59 take profit triggered #2 buy 0.01 XAUUSD 1812.91 sl: 1806.93 tp:
» [#3 sell 0.01 XAUUSD at 1818.89]
2022.01.11 20:28:59 deal #3 sell 0.01 XAUUSD at 1818.91 done (based on order #3)
2022.01.11 20:28:59 deal performed [#3 sell 0.01 XAUUSD at 1818.91]
2022.01.11 20:28:59 order performed sell 0.01 at 1818.91 [#3 sell 0.01 XAUUSD at 1818
2022.01.12 01:05:00 Autodetected daily range: 23.28
2022.01.12 01:05:00 buy stop 0.01 XAUUSD at 1843.77 sl: 1832.14 tp: 1855.40 (1820.14
2022.01.12 01:05:00 OK order placed: #=4
2022.01.12 01:05:00 TRADE_ACTION_PENDING, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_
» @ 1843.77, SL=1832.14, TP=1855.40, ORDER_TIME_GTC, M=1234567890
2022.01.12 01:05:00 DONE, #=4, V=0.01, Bid=1820.14, Ask=1820.49, Request executed, Re
```

La orden puede activarse en cualquier momento, tras lo cual la posición se cierra al cabo de un tiempo mediante stop loss o take profit (como en el código anterior).

En algunos casos, puede darse la situación de que la posición siga existiendo al comienzo del día siguiente, y entonces se creará una nueva orden además de ésta, como en la captura de pantalla siguiente:

El Asesor Experto con una estrategia de trading basada en órdenes pendientes en el probador

Tenga en cuenta que, debido al hecho de que solicitamos cotizaciones del marco temporal PERIOD_D1 para calcular el rango diario, el probador visual abre el gráfico correspondiente, además del gráfico de trabajo actual. Este servicio funciona no sólo para marcos temporales distintos del de trabajo, sino también para otros símbolos. Esto será útil, en particular, a la hora de desarrollar Asesores expertos multidivisa.

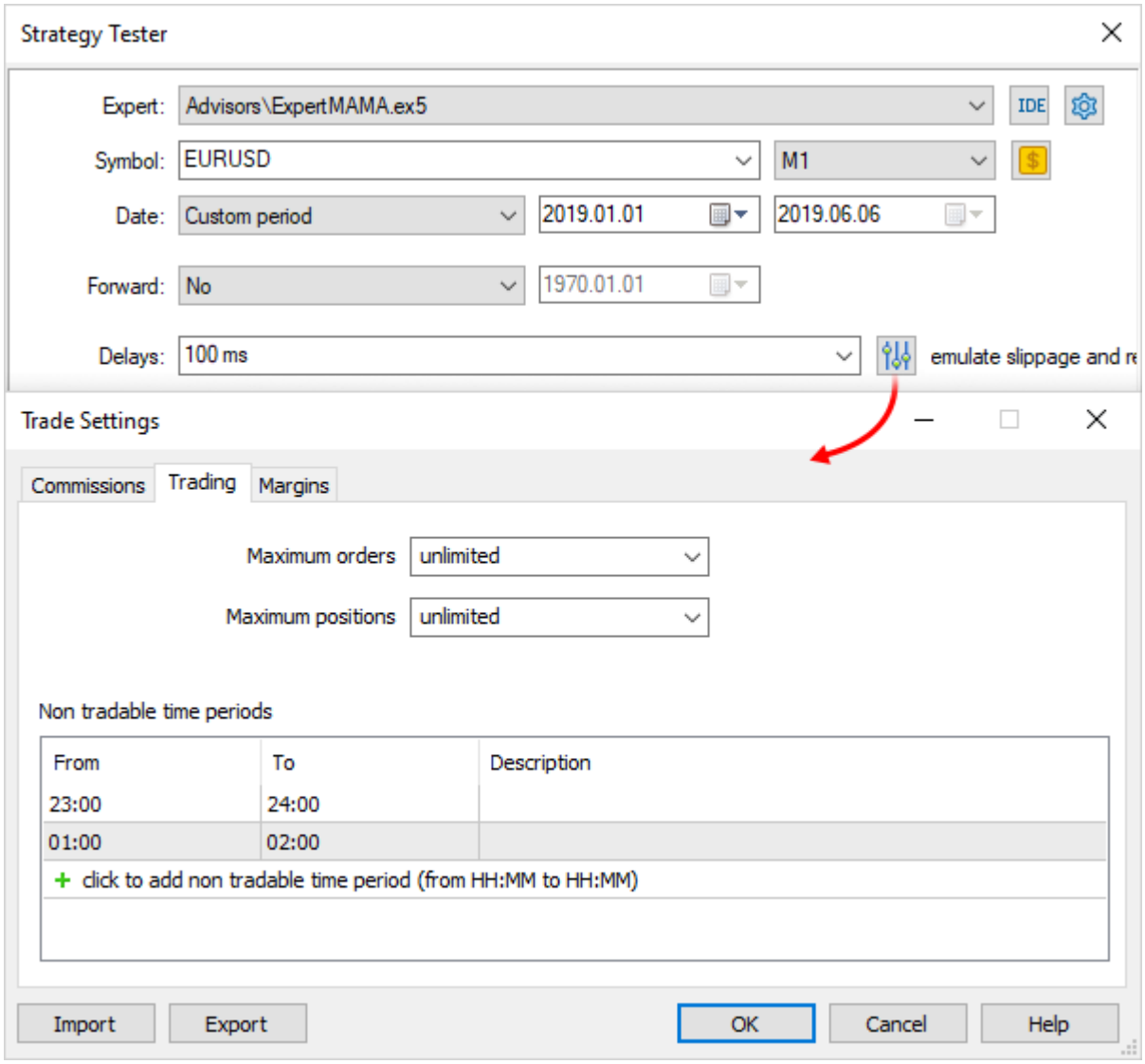
Para comprobar cómo funciona el tratamiento de errores, pruebe a desactivar el trading para el Asesor Experto. El registro contendrá lo siguiente:

```
Autodetected daily range: 34.48
TRADE_ACTION_PENDING, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_FILLING_FOK, »
  » @ 1975.73, SL=1958.49, TP=1992.97, ORDER_TIME_GTC, M=1234567890
CLIENT_DISABLES_AT, AutoTrading disabled by client
Alert: Can't place/modify pending order, EA is stopped
```

Este error es crítico, y el Asesor Experto deja de funcionar.

Para demostrar uno de los errores más sencillos, podríamos utilizar el manejador *OnTimer* en lugar de *OnTick*. Entonces, lanzar el mismo Asesor Experto en símbolos en los que las sesiones de trading duran sólo una parte del día generaría periódicamente una secuencia de errores no críticos sobre un mercado cerrado («Market closed»). En este caso, el Asesor Experto seguiría intentando comenzar a operar, aumentando constantemente el tiempo de espera.

Esto, en particular, es fácil de comprobar en el probador, que permite establecer sesiones de trading arbitrarias para cualquier símbolo. En la pestaña *Settings*, a la derecha de la lista desplegable *Delays*, hay un botón que abre el cuadro de diálogo *Trade setup*. Allí deberá incluir la opción *Use your settings* y, en la pestaña *Trade*, añadir al menos un registro a la tabla *Non-trading periods*.



Establecer periodos no de trading en el probador

Tenga en cuenta que lo que se configura aquí son los periodos no de trading, no las sesiones de trading; es decir, este ajuste actúa exactamente al revés en comparación con la especificación del símbolo.

Muchos errores potenciales relacionados con las restricciones de trading pueden eliminarse mediante un análisis preliminar del entorno utilizando una clase como *Permissions* presentada en la sección Restricciones y permisos para las transacciones de la cuenta.

6.4.21 Borrar una orden pendiente

La eliminación de una orden pendiente se realiza a nivel de programa mediante la operación TRADE_ACTION_REMOVE: esta constante debe asignarse al campo *action* de la estructura *MqlTradeRequest* antes de llamar a una de las versiones de la función *OrderSend*. El único campo obligatorio además de *action* es *order* para especificar el ticket de la orden que se va a borrar.

El método *remove* de la estructura de la aplicación *MqlTradeRequestSync* del archivo *MqlTradeSync.mqh* es bastante básico.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   ...
   bool remove(const ulong ticket)
   {
      if(!OrderSelect(ticket)) return false;
      action = TRADE_ACTION_REMOVE;
      order = ticket;
      ZeroMemory(result);
      return OrderSend(this, result);
   }
```

La comprobación del hecho de borrar una orden se realiza tradicionalmente en el método *completed*.

```
   bool completed()
   {
      ...
      else if(action == TRADE_ACTION_REMOVE)
      {
         result.order = order;
         return result.removed(timeout);
      }
      ...
   }
```

La espera de la retirada efectiva de la orden se realiza en el método *removed* de la estructura *MqlTradeResultSync*.

```
struct MqlTradeResultSync: public MqlTradeResult
{
   ...
   bool removed(const ulong msc = 1000)
   {
      if(retcode != TRADE_RETCODE_DONE)
      {
         return false;
      }

      if(!wait(orderRemoved, msc))
      {
         Print("Order removal timeout: #=" + (string)order);
         return false;
      }

      return true;
   }

   static bool orderRemoved(MqlTradeResultSync &ref)
   {
      return !OrderSelect(ref.order) && HistoryOrderSelect(ref.order);
   }
```

Ejemplo del Asesor Experto (*PendingOrderDelete.mq5*) que muestra la eliminación de una orden que construiremos casi por entero basada en *PendingOrderSend.mq5*. Esto se debe al hecho de que es más fácil garantizar la existencia de una orden antes de borrarla. Así, inmediatamente después del lanzamiento, el Asesor Experto creará una nueva orden con los parámetros especificados. A continuación, la orden se eliminará en el manejador *OnDeinit*. Si cambia los parámetros de entrada del Asesor Experto, el símbolo o el marco temporal del gráfico, también se eliminará la orden antigua y se creará una nueva.

Se ha añadido la variable global *OwnOrder* para almacenar el ticket de la orden. Se rellena como resultado de la llamada a *PlaceOrder* (la función en sí no se modifica).

```
ulong OwnOrder = 0;

void OnTimer()
{
   // execute the code once for the current parameters
   EventKillTimer();

   const string symbol = StringLen(Symbol) == 0 ? _Symbol : Symbol;
   OwnOrder = PlaceOrder((ENUM_ORDER_TYPE)Type, symbol, Volume,
      Distance2SLTP, Expiration, Until, Magic, Comment);
}
```

He aquí una simple función de borrado *RemoveOrder*, que crea el objeto *request* y llama secuencialmente a los métodos *remove* y *completed* para él.

```
void OnDeinit(const int)
{
   if(OwnOrder != 0)
   {
      RemoveOrder(OwnOrder);
   }
}

void RemoveOrder(const ulong ticket)
{
   MqlTradeRequestSync request;
   if(request.remove(ticket) && request.completed())
   {
      Print("OK order removed");
   }
   Print(TU::StringOf(request));
   Print(TU::StringOf(request.result));
}
```

En el siguiente registro se muestran las entradas que aparecieron como resultado de colocar el Asesor Experto en el gráfico EURUSD, después de lo cual el símbolo fue cambiado a XAUUSD, y luego el Asesor Experto fue eliminado.

```
(EURUSD,H1)   Autodetected daily range: 0.0094
(EURUSD,H1)   OK order placed: #=1284920879
(EURUSD,H1)   TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_FILLIN
» @ 1.11011, ORDER_TIME_GTC, M=1234567890
(EURUSD,H1)   DONE, #=1284920879, V=0.01, Request executed, Req=1
(EURUSD,H1)   OK order removed
(EURUSD,H1)   TRADE_ACTION_REMOVE, EURUSD, ORDER_TYPE_BUY, ORDER_FILLING_FOK, #=12849
(EURUSD,H1)   DONE, #=1284920879, Request executed, Req=2
(XAUUSD,H1)   Autodetected daily range: 47.45
(XAUUSD,H1)   OK order placed: #=1284921672
(XAUUSD,H1)   TRADE_ACTION_PENDING, XAUUSD, ORDER_TYPE_BUY_STOP, V=0.01, ORDER_FILLIN
» @ 1956.68, ORDER_TIME_GTC, M=1234567890
(XAUUSD,H1)   DONE, #=1284921672, V=0.01, Request executed, Req=3
(XAUUSD,H1)   OK order removed
(XAUUSD,H1)   TRADE_ACTION_REMOVE, XAUUSD, ORDER_TYPE_BUY, ORDER_FILLING_FOK, #=12849
(XAUUSD,H1)   DONE, #=1284921672, Request executed, Req=4
```

Vamos a ver otro ejemplo de supresión de órdenes para aplicar la estrategia OCO («One Cancel Other») en la sección de eventos *OnTrade*.

6.4.22 Obtener una lista de órdenes activas

Los programas Asesor Experto a menudo necesitan enumerar las órdenes activas existentes y analizar sus propiedades. En concreto, en la sección sobre modificaciones de órdenes pendientes, en el ejemplo *PendingOrderModify.mq5*, hemos creado una función especial *GetMyOrder* para encontrar las órdenes pertenecientes al Asesor Experto para modificar esta orden. Allí, el análisis se llevó a cabo por nombre de símbolo e ID de Asesor Experto (*Magic*). En teoría, debería haberse aplicado el mismo enfoque en el ejemplo de eliminación de una orden pendiente *PendingOrderDelete.mq5* de la sección anterior.

En este último caso, por simplicidad, creamos una orden y almacenamos su ticket en una variable global. Pero esto no puede hacerse en el caso general porque el Asesor Experto y todo el terminal pueden detenerse o reiniciarse en cualquier momento. Por lo tanto, el Asesor Experto debe contener un algoritmo para restablecer el estado interno, incluido el análisis de todo el entorno de trading, junto con las órdenes, las transacciones, las posiciones, el saldo de la cuenta, etc.

En esta sección estudiaremos las funciones MQL5 para obtener una lista de órdenes activas y seleccionar cualquiera de ellas en el entorno de trading, lo que permite leer todas sus propiedades.

int OrdersTotal()

La función *OrdersTotal* devuelve el número de órdenes activas actualmente. Entre ellas se incluyen las órdenes pendientes, así como las órdenes de mercado que aún no se han ejecutado. Por regla general, una orden de mercado se ejecuta rápidamente, por lo que no suele ser posible captarla en la fase activa, pero esto puede ocurrir si no hay suficiente liquidez en el mercado. En cuanto se ejecuta la orden (se completa una transacción), se transfiere desde la categoría de activas al historial. Hablaremos del trabajo con el historial de órdenes en una sección aparte.

> Tenga en cuenta que sólo las órdenes pueden estar activas y ser históricas. Esto distingue significativamente las órdenes de las transacciones que siempre se crean en el historial y de las posiciones que sólo existen en línea. Para restablecer el historial de posiciones, debe analizar el historia de transacciones.

ulong OrderGetTicket(uint index)

La función *OrderGetTicket* devuelve el ticket de la orden por su número en la lista de órdenes del entorno de trading del terminal. El parámetro *index* debe estar comprendido entre 0 y el valor *OrdersTotal()-1*, ambos inclusive. La forma en que se organizan las órdenes no está regulada.

La función *OrderGetTicket* selecciona una orden, es decir, copia datos sobre él en alguna caché interna para que el programa MQL pueda leer todas sus propiedades utilizando las llamadas posteriores de la función *OrderGetDouble*, *OrderGetInteger* o *OrderGetString*, que se abordarán en una sección aparte.

La presencia de una caché de este tipo indica que los datos recibidos de ella pueden quedar obsoletos: la orden puede haber dejado de existir o puede haber sido modificada (por ejemplo, puede tener un estado, un precio de apertura, unos niveles *Stop Loss* o *Take Profit* y un vencimiento diferentes). Por lo tanto, para garantizar la recepción de datos relevantes sobre la orden, se recomienda llamar a la función *OrderGetTicket* inmediatamente antes de solicitar los datos. He aquí cómo se hace en el ejemplo de *PendingOrderModify.mq5*.

```
ulong GetMyOrder(const string name, const ulong magic)
{
   for(int i = 0; i < OrdersTotal(); ++i)
   {
      ulong t = OrderGetTicket(i);
      if(OrderGetInteger(ORDER_MAGIC) == magic
      && OrderGetString(ORDER_SYMBOL) == name)
      {
         return t;
      }
   }
   return 0;
}
```

Cada programa MQL mantiene su propia caché (contexto del entorno de trading), que incluye la orden seleccionada. En las siguientes secciones descubriremos que además de las órdenes, un programa MQL puede seleccionar posiciones y fragmentos del historial con transacciones y órdenes en el contexto activo.

La función *OrderSelect* realiza una selección similar de una orden con copia de sus datos a la caché interna.

bool OrderSelect(ulong ticket)

La función comprueba la presencia de una orden y prepara la posibilidad de seguir leyendo sus propiedades. En este caso, la orden no se especifica mediante un número de serie, sino mediante un ticket que el programa MQL debe recibir antes de una forma u otra; en concreto, como resultado de la ejecución de *OrderSend/OrderSendAsync*.

La función devuelve *true* en caso de éxito. Si se recibe *false*, ello normalmente significa que no hay ninguna orden con el ticket especificado. La razón más común es cuando el estado de la orden ha cambiado de activo a historial; por ejemplo, como resultado de su ejecución o cancelación (más adelante aprenderemos a determinar el estado exacto). Las órdenes pueden seleccionarse en el historial mediante las funciones relevantes.

Anteriormente utilizamos la función *OrderSelect* en la estructura *MqlTradeResultSync* para hacer un seguimiento de la creación y eliminación de órdenes pendientes.

6.4.23 Propiedades de una orden (activas y del historial)

En las secciones relacionadas con las operaciones de trading, en concreto para hacer compra/venta, cerrar una posición y colocar una orden pendiente, hemos visto que las solicitudes se envían al servidor en función de la cumplimentación de campos específicos de la estructura *MqlTradeRequest*, la mayoría de los cuales definen directamente las propiedades de las órdenes resultantes. La API de MQL5 permite conocer estas y algunas otras propiedades establecidas por el propio sistema de trading, como ticket, hora de registro y estado.

Es importante señalar que la lista de propiedades de la orden es común tanto para las órdenes activas como para las históricas, aunque, por supuesto, los valores de muchas propiedades serán diferentes para ellas.

Las propiedades de orden se agrupan en MQL5 según el principio que ya nos es familiar basado en el tipo de valores: entero (compatible con *long/ulong*), real (*double*) y cadenas. Cada grupo de propiedades tiene su propia enumeración.

Las propiedades de los enteros se resumen en ENUM_ORDER_PROPERTY_INTEGER y se presentan en la tabla siguiente:

Identificador	Descripción	Tipo
ORDER_TYPE	Tipo de orden	ENUM_ORDER_TYPE
ORDER_TYPE_FILLING	Tipo de ejecución por volumen	ENUM_ORDER_TYPE_FILLING
ORDER_TYPE_TIME	Duración de la orden (pendiente)	ENUM_ORDER_TYPE_TIME
ORDER_TIME_EXPIRATION	Hora de vencimiento de la orden (pendiente)	datetime
ORDER_MAGIC	Identificador arbitrario establecido por el Asesor Experto que colocó la orden.	ulong
ORDER_TICKET	Ticket de la orden; número único asignado por el servidor a cada orden.	ulong
ORDER_STATE	Estado de la orden	ENUM_ORDER_STATE (véase más abajo)
ORDER_REASON	Motivo u origen de la orden	ENUM_ORDER_REASON (véase más abajo)
ORDER_TIME_SETUP	Tiempo de colocación de la orden	datetime
ORDER_TIME_DONE	Hora de ejecución o retirada de la orden	datetime
ORDER_TIME_SETUP_MSC	Tiempo de colocación de la orden para su ejecución en milisegundos	ulong
ORDER_TIME_DONE_MSC	Tiempo de ejecución/retirada de la orden en milisegundos	ulong
ORDER_POSITION_ID	ID de la posición que la orden generó o modificó al ejecutarse	ulong
ORDER_POSITION_BY_ID	Identificador de posición opuesta para órdenes de tipo ORDER_TYPE_CLOSE_BY	ulong

Cada orden ejecutada genera una transacción que abre una posición nueva o modifica una existente. El ID de esta posición se asigna a la orden ejecutada en la propiedad ORDER_POSITION_ID.

La enumeración ENUM_ORDER_STATE contiene elementos que describen los estados de las órdenes. Vea a continuación un esquema simplificado (diagrama de estados) de las órdenes:

Identificador	Descripción
ORDER_STATE_STARTED	Se ha comprobado que la orden es correcta, pero el servidor aún no la ha aceptado.
ORDER_STATE_PLACED	La orden ha sido aceptada por el servidor.
ORDER_STATE_CANCELED	La orden ha sido cancelada por el cliente (usuario o programa MQL).
ORDER_STATE_PARTIAL	La orden se ha ejecutado parcialmente.
ORDER_STATE_FILLED	La orden se ha realizado en su totalidad.
ORDER_STATE_REJECTED	La orden ha sido rechazada por el servidor.
ORDER_STATE_EXPIRED	La orden se ha cancelado al vencimiento.
ORDER_STATE_REQUEST_ADD	La orden se está registrando (se está colocando en el sistema de trading).
ORDER_STATE_REQUEST_MODIFY	Se modifica la orden (se cambian sus parámetros).
ORDER_STATE_REQUEST_CANCEL	La orden se está borrando (eliminando del sistema de trading).

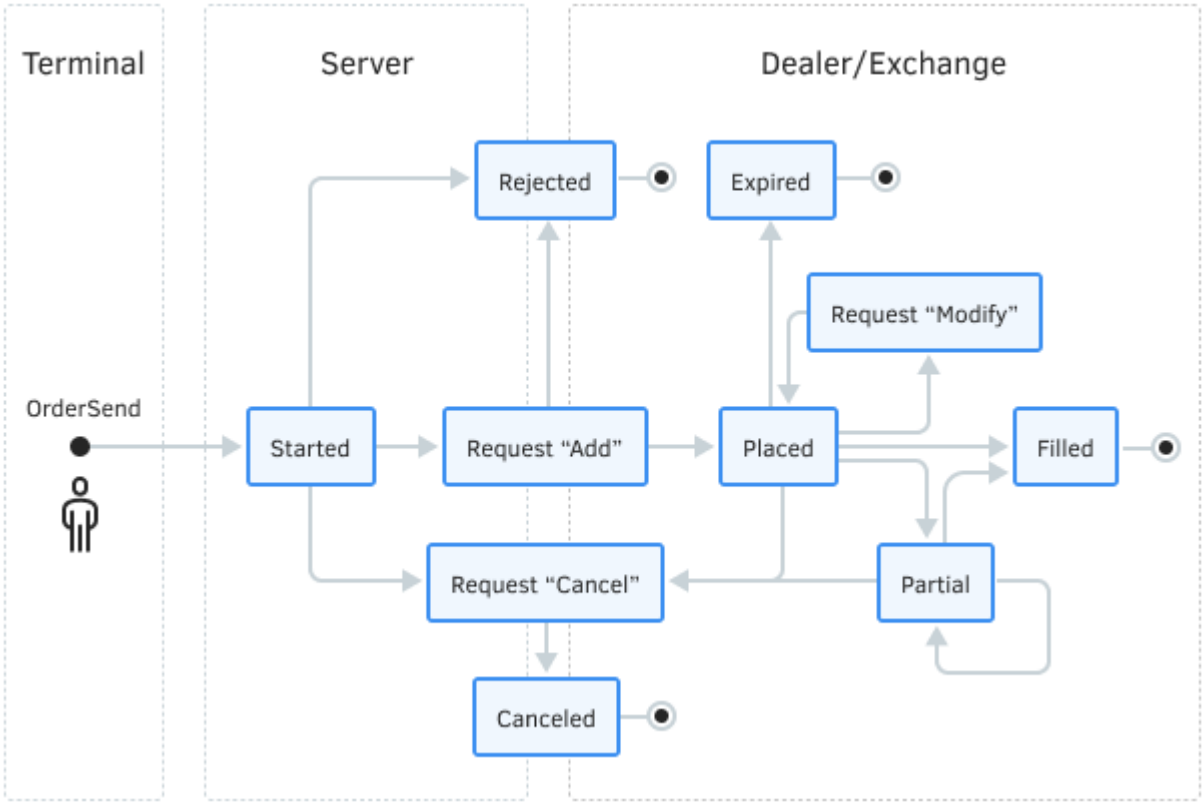


Diagrama del estado de la orden

El cambio de estado sólo es posible para las órdenes activas. Para las órdenes históricas (ejecutadas o canceladas), el estado es fijo.

Puede cancelar una orden que ya se ha cumplido parcialmente, y entonces su estado en el historial será ORDER_STATE_CANCELED.

ORDER_STATE_PARTIAL sólo se produce para órdenes activas. Las órdenes ejecutadas (históricas) siempre tienen el estado ORDER_STATE_FILLED.

La enumeración ENUM_ORDER_REASON especifica las posibles opciones de origen de la orden.

Identificador	Descripción
ORDER_REASON_CLIENT	Orden cursada manualmente desde el terminal de sobremesa
ORDER_REASON_EXPERT	Orden cursada desde el terminal de sobremesa por un Asesor Experto o un script
ORDER_REASON_MOBILE	Orden cursada desde la aplicación móvil
ORDER_REASON_WEB	Orden cursada desde el terminal web (navegador)
ORDER_REASON_SL	Orden cursada por el servidor como resultado de la activación de Stop Loss
ORDER_REASON_TP	Orden cursada por el servidor como resultado de la activación de Take Profit
ORDER_REASON_SO	Orden cursada por el servidor como resultado del evento Stop Out

Las propiedades reales se recopilan en la enumeración ENUM_ORDER_PROPERTY_DOUBLE.

Identificador	Descripción
ORDER_VOLUME_INITIAL	Volumen inicial al cursar una orden
ORDER_VOLUME_CURRENT	Volumen actual (inicial o restante tras la ejecución parcial)
ORDER_PRICE_OPEN	El precio indicado en la orden
ORDER_PRICE_CURRENT	El precio actual del símbolo de una orden que aún no se ha ejecutado o el precio de ejecución.
ORDER_SL	Nivel de Stop Loss
ORDER_TP	Nivel de Take Profit
ORDER_PRICE_STOPLIMIT	Precio de colocación de una orden Limit cuando se activa una orden StopLimit.

La propiedad ORDER_PRICE_CURRENT contiene el precio *Ask* actual para órdenes pendientes de compra activas o el precio *Bid* para órdenes pendientes de venta activas. «Actual» se refiere al precio conocido en el entorno de trading en el momento en que se selecciona la orden mediante *OrderSelect* o

OrderGetTicket. Para las órdenes ejecutadas en el historial, esta propiedad contiene el precio de ejecución, que puede diferir del especificado en la orden debido al deslizamiento.

Las propiedades ORDER_VOLUME_INITIAL y ORDER_VOLUME_CURRENT no son iguales entre sí sólo si el estado de la orden es ORDER_STATE_PARTIAL.

Si la orden se ejecutó por partes, entonces su propiedad ORDER_VOLUME_INITIAL en el historial será igual al tamaño de la última parte ejecutada, y todas las demás «ejecuciones» relacionadas con el volumen completo original se ejecutarán como órdenes (y transacciones) independientes.

Las propiedades de cadena se describen en la enumeración ENUM_ORDER_PROPERTY_STRING.

Identificador	Descripción
ORDER_SYMBOL	El símbolo sobre el que se coloca la orden
ORDER_COMMENT	Comentario
ORDER_EXTERNAL_ID	ID de la orden en el sistema de trading externo (en la bolsa)

Para leer todas las propiedades anteriores existen dos conjuntos de funciones diferentes: para órdenes activas y para órdenes históricas. En primer lugar, consideraremos las funciones para órdenes activas, y volveremos a las históricas después de familiarizarnos con los principios de selección del periodo necesario en el historial.

6.4.24 Funciones para leer las propiedades de órdenes activas

Los conjuntos de funciones que pueden utilizarse para obtener los valores de todas las propiedades de las órdenes difieren para las órdenes activas y las históricas. En esta sección se describen las funciones para leer las propiedades de las órdenes activas. Para conocer las funciones que permiten acceder a las propiedades de las órdenes en el historial, consulte la sección correspondiente.

Las propiedades de los enteros pueden leerse utilizando la función *OrderGetInteger*, que tiene dos formas: la primera devuelve directamente el valor de la propiedad; la segunda devuelve un signo lógico de éxito (*true*) o error (*false*), y el segundo parámetro pasado por referencia se rellena con el valor de la propiedad.

```
long OrderGetInteger(ENUM_ORDER_PROPERTY_INTEGER property)
bool OrderGetInteger(ENUM_ORDER_PROPERTY_INTEGER property, long &value)
```

Ambas funciones permiten obtener la propiedad de orden solicitada de un tipo compatible con enteros (*datetime*, *long/ulong* o listado). Aunque el prototipo menciona *long*, desde un punto de vista técnico, el valor se almacena como una celda de 8 bytes, que puede convertirse a tipos compatibles sin ninguna conversión de la representación interna, en concreto, a *ulong*, que se utiliza para todos los tickets.

Un par de funciones similares están pensadas para propiedades de tipo real *double*.

```
double OrderGetDouble(ENUM_ORDER_PROPERTY_DOUBLE property)
bool OrderGetDouble(ENUM_ORDER_PROPERTY_DOUBLE property, double &value)
```

Por último, las propiedades de cadena están disponibles a través de un par de funciones *OrderGetString*.

string OrderGetString(ENUM_ORDER_PROPERTY_STRING property)
bool OrderGetString(ENUM_ORDER_PROPERTY_STRING property, string &value)

Como primer parámetro, todas las funciones toman el identificador de la propiedad que nos interesa. Debe ser un elemento de una de las enumeraciones (ENUM_ORDER_PROPERTY_INTEGER, ENUM_ORDER_PROPERTY_DOUBLE o ENUM_ORDER_PROPERTY_STRING) que se describen en la sección anterior.

Tenga en cuenta que antes de llamar a cualquiera de las funciones anteriores, debe seleccionar primero una orden utilizando *OrderSelect* o *OrderGetTicket*.

Para leer todas las propiedades de una orden específica, desarrollaremos la clase *OrderMonitor* (*OrderMonitor.mqh*) que funciona según el mismo principio que los monitores de símbolo (*SymbolMonitor.mqh*) y cuenta de trading (*AccountMonitor.mqh*) considerados anteriormente.

Estas y otras clases de monitores que se abordan en el libro ofrecen una forma unificada de analizar propiedades mediante versiones sobrecargadas de métodos *get* virtuales.

Mirando un poco más adelante, digamos que las transacciones y las posiciones tienen la misma agrupación de propiedades según los tres tipos principales de valores, y también necesitamos implementar monitores para ellos. A este respecto, tiene sentido separar el algoritmo general en una clase abstracta base *MonitorInterface* (*TradeBaseMonitor.mqh*). Esta es una clase de plantilla con tres parámetros destinados a especificar los tipos de enumeraciones concretas, para los grupos de propiedades entero (I), real (D) y cadena (S).

```
#include <MQL5Book/EnumToArray.mqh>

template<typename I,typename D,typename S>
class MonitorInterface
{
protected:
   bool ready;
public:
   MonitorInterface(): ready(false) { }

   bool isReady() const
   {
      return ready;
   }
   ...
```

Debido al hecho de que encontrar una orden (transacción o posición) en el entorno de trading puede fallar por diversas razones, la clase tiene una variable reservada *ready* en la que las clases derivadas tendrán que escribir una señal de inicialización de éxito, es decir, la elección de un objeto para leer sus propiedades.

Varios métodos puramente virtuales declaran el acceso a propiedades de los tipos correspondientes.

```
virtual long get(const I property) const = 0;
virtual double get(const D property) const = 0;
virtual string get(const S property) const = 0;
virtual long get(const int property, const long) const = 0;
virtual double get(const int property, const double) const = 0;
virtual string get(const int property, const string) const = 0;
...
```

En los tres primeros métodos, el tipo de propiedad se especifica mediante uno de los parámetros de la plantilla. En otros tres métodos, el tipo se especifica mediante el segundo parámetro del propio método: esto es necesario porque los últimos métodos no toman las constantes de una enumeración concreta, sino simplemente un número entero como primer parámetro. Por un lado, esto es conveniente para la numeración continua de identificadores (las constantes de enumeración de los tres tipos no se cruzan). Por otra parte, necesitamos otra fuente para determinar el tipo de valor, ya que el tipo devuelto por la función/método no participa en el proceso de elección de la sobrecarga pertinente.

Este enfoque le permite obtener propiedades basadas en varias entradas disponibles en el código de llamada. A continuación, crearemos clases basadas en *OrderMonitor* (así como en las futuras *DealMonitor* y *PositionMonitor*) para seleccionar objetos según un conjunto de condiciones arbitrarias, y allí todos estos métodos estarán en demanda.

Muy a menudo, los programas necesitan obtener una representación de cadena de cualquier propiedad; por ejemplo, para el registro. En los nuevos monitores, esto se implementa mediante los métodos *stringify*. Obviamente, obtienen los valores de las propiedades solicitadas a través de las llamadas a los métodos *get* mencionados anteriormente.

```
virtual string stringify(const long v, const I property) const = 0;

virtual string stringify(const I property) const
{
   return stringify(get(property), property);
}

virtual string stringify(const D property, const string format = NULL) const
{
   if(format == NULL) return (string)get(property);
   return StringFormat(format, get(property));
}

virtual string stringify(const S property) const
{
   return get(property);
}
...
```

El único método que no ha recibido implementación es la primera versión de *stringify* para el tipo *long*. Esto se debe al hecho de que el grupo de propiedades de enteros, como vimos en la sección anterior, en realidad contienen diferentes tipos de aplicaciones, incluyendo fecha y hora, enumeraciones y enteros. Por lo tanto, sólo las clases derivadas pueden proporcionar su conversión a cadenas comprensibles. Esta situación es común a todas las entidades de trading, no sólo a las órdenes, sino también a las transacciones y posiciones, cuyas propiedades estudiaremos más adelante.

Cuando una propiedad de enteros contiene un elemento de enumeración (por ejemplo, ENUM_ORDER_TYPE, ORDER_TYPE_FILLING, etc.), debe utilizar la función *EnumToString* para convertirla en una cadena. Esta tarea se realiza por medio de un método auxiliar *enumstr*. Pronto veremos su uso generalizado en clases específicas de monitores, empezando por *OrderMonitor* tras un par de párrafos.

```
template<typename E>
static string enumstr(const long v)
{
   return EnumToString((E)v);
}
```

Para registrar todas las propiedades de un tipo determinado, hemos creado el método *list2log* que utiliza *stringify* en un bucle.

```
template<typename E>
void list2log() const
{
   E e = (E)0; // suppress warning 'possible use of uninitialized variable'
   int array[];
   const int n = EnumToArray(e, array, 0, USHORT_MAX);
   Print(typename(E), " Count=", n);
   for(int i = 0; i < n; ++i)
   {
      e = (E)array[i];
      PrintFormat("% 3d %s=%s", i, EnumToString(e), stringify(e));
   }
}
```

Por último, para facilitar el registro de las propiedades de los tres grupos, existe un método *print* que llama a *list2log* tres veces para cada grupo de propiedades.

```
virtual void print() const
{
   if(!ready) return;

   Print(typename(this));
   list2log<I>();
   list2log<D>();
   list2log<S>();
}
```

Teniendo a nuestra disposición una clase de plantilla base *MonitorInterface*, describimos *OrderMonitorInterface*, donde especificamos ciertos tipos de enumeración para las órdenes de la sección anterior y proporcionamos una implementación de *stringify* para las propiedades de enteros de las órdenes.

```
class OrderMonitorInterface:
   public MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,
   ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PROPERTY_STRING>
{
public:
   // description of properties according to subtypes
   virtual string stringify(const long v,
      const ENUM_ORDER_PROPERTY_INTEGER property) const override
   {
      switch(property)
      {
         case ORDER_TYPE:
            return enumstr<ENUM_ORDER_TYPE>(v);
         case ORDER_STATE:
            return enumstr<ENUM_ORDER_STATE>(v);
         case ORDER_TYPE_FILLING:
            return enumstr<ENUM_ORDER_TYPE_FILLING>(v);
         case ORDER_TYPE_TIME:
            return enumstr<ENUM_ORDER_TYPE_TIME>(v);
         case ORDER_REASON:
            return enumstr<ENUM_ORDER_REASON>(v);

         case ORDER_TIME_SETUP:
         case ORDER_TIME_EXPIRATION:
         case ORDER_TIME_DONE:
            return TimeToString(v, TIME_DATE | TIME_SECONDS);

         case ORDER_TIME_SETUP_MSC:
         case ORDER_TIME_DONE_MSC:
            return STR_TIME_MSC(v);
      }

      return (string)v;
   }
};
```

La macro STR_TIME_MSC para mostrar el tiempo en milisegundos se define como sigue:

```
#define STR_TIME_MSC(T) (TimeToString((T) / 1000, TIME_DATE | TIME_SECONDS) \
    + StringFormat("'%03d", (T) % 1000))
```

Ahora estamos listos para describir la clase final para leer las propiedades de cualquier orden: *OrderMonitor* derivada de *OrderMonitorInterface*. El ticket de la orden se pasa al constructor, y se selecciona en el entorno de trading mediante *OrderSelect*.

```
class OrderMonitor: public OrderMonitorInterface
{
public:
   const ulong ticket;
   OrderMonitor(const long t): ticket(t)
   {
      if(!OrderSelect(ticket))
      {
         PrintFormat("Error: OrderSelect(%lld) failed: %s",
            ticket, E2S(_LastError));
      }
      else
      {
         ready = true;
      }
   }
   ...
```

La parte principal de trabajo del monitor consiste en redefiniciones de funciones virtuales para la lectura de propiedades. Aquí vemos las llamadas a las funciones *OrderGetInteger*, *OrderGetDouble* y *OrderGetString*.

```
   virtual long get(const ENUM_ORDER_PROPERTY_INTEGER property) const override
   {
      return OrderGetInteger(property);
   }

   virtual double get(const ENUM_ORDER_PROPERTY_DOUBLE property) const override
   {
      return OrderGetDouble(property);
   }

   virtual string get(const ENUM_ORDER_PROPERTY_STRING property) const override
   {
      return OrderGetString(property);
   }

   virtual long get(const int property, const long) const override
   {
      return OrderGetInteger((ENUM_ORDER_PROPERTY_INTEGER)property);
   }

   virtual double get(const int property, const double) const override
   {
      return OrderGetDouble((ENUM_ORDER_PROPERTY_DOUBLE)property);
   }

   virtual string get(const int property, const string)  const override
   {
      return OrderGetString((ENUM_ORDER_PROPERTY_STRING)property);
   }
};
```

Este fragmento de código se presenta de forma abreviada: se han eliminado de él los operadores para trabajar con órdenes en el historial. Veremos el código completo de *OrderMonitor* más adelante, cuando exploremos este aspecto en las secciones siguientes.

Es importante señalar que el objeto monitor no almacena copias de sus propiedades. Por lo tanto, el acceso a los métodos *get* debe realizarse inmediatamente después de la creación del objeto y, en consecuencia, la llamada *OrderSelect*. Para leer las propiedades en un período posterior, tendrá que volver a asignar la orden en la caché interna del programa MQL, por ejemplo, llamando al método *refresh*.

```
   void refresh()
   {
      ready = OrderSelect(ticket);
   }
```

Probemos el funcionamiento de *OrderMonitor* añadiéndolo al Asesor Experto *MarketOrderSend.mq5*. Una nueva versión llamada *MarketOrderSendMonitor.mq5* conecta el archivo *OrderMonitor.mqh* mediante la directiva *#include*, y en el cuerpo de la función *OnTimer* (en el bloque de confirmación exitosa de apertura de una posición en una orden) crea un objeto monitor y llama a su método *print*.

```
#include <MQL5Book/OrderMonitor.mqh>
...
void OnTimer()
{
   ...
   const ulong order = (wantToBuy ?
      request.buy(volume, Price) :
      request.sell(volume, Price));
   if(order != 0)
   {
      Print("OK Order: #=", order);
      if(request.completed())
      {
         Print("OK Position: P=", request.result.position);

         OrderMonitor m(order);
         m.print();
         ...
      }
   }
}
```

En el registro, deberíamos ver nuevas líneas que contienen todas las propiedades de la orden.

```
OK Order: #=1287846602
Waiting for position for deal D=1270417032
OK Position: P=1287846602
MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER, »
   » ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PROPERTY_STRING>
ENUM_ORDER_PROPERTY_INTEGER Count=14
  0 ORDER_TIME_SETUP=2022.03.21 13:28:59
  1 ORDER_TIME_EXPIRATION=1970.01.01 00:00:00
  2 ORDER_TIME_DONE=2022.03.21 13:28:59
  3 ORDER_TYPE=ORDER_TYPE_BUY
  4 ORDER_TYPE_FILLING=ORDER_FILLING_FOK
  5 ORDER_TYPE_TIME=ORDER_TIME_GTC
  6 ORDER_STATE=ORDER_STATE_FILLED
  7 ORDER_MAGIC=1234567890
  8 ORDER_POSITION_ID=1287846602
  9 ORDER_TIME_SETUP_MSC=2022.03.21 13:28:59'572
 10 ORDER_TIME_DONE_MSC=2022.03.21 13:28:59'572
 11 ORDER_POSITION_BY_ID=0
 12 ORDER_TICKET=1287846602
 13 ORDER_REASON=ORDER_REASON_EXPERT
ENUM_ORDER_PROPERTY_DOUBLE Count=7
  0 ORDER_VOLUME_INITIAL=0.01
  1 ORDER_VOLUME_CURRENT=0.0
  2 ORDER_PRICE_OPEN=1.10275
  3 ORDER_PRICE_CURRENT=1.10275
  4 ORDER_PRICE_STOPLIMIT=0.0
  5 ORDER_SL=0.0
  6 ORDER_TP=0.0
ENUM_ORDER_PROPERTY_STRING Count=3
  0 ORDER_SYMBOL=EURUSD
  1 ORDER_COMMENT=
  2 ORDER_EXTERNAL_ID=
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, »
   » @ 1.10275, P=1287846602, M=1234567890
DONE, D=1270417032, #=1287846602, V=0.01, @ 1.10275, Bid=1.10275, Ask=1.10275, »
   » Request executed, Req=3
```

La cuarta línea inicia la salida del método *print*, que incluye el nombre completo del objeto monitor *MonitorInterface* junto con los tipos de parámetros (en este caso, triple ENUM_ORDER_PROPERTY) y, a continuación, todas las propiedades de una orden concreta.

No obstante, la impresión de propiedades no es la acción más interesante que puede ofrecer un monitor. La tarea de seleccionar ordenes por condiciones (valores de propiedades arbitrarias) es mucho más demandada entre los Asesores Expertos. Utilizando el monitor como herramienta auxiliar, crearemos un mecanismo de filtrado de órdenes similar al que hemos hecho para los símbolos: *SymbolFilter.mqh* pertinente.

6.4.25 Seleccionar órdenes por propiedades

En una de las secciones sobre propiedades de los símbolos hemos introducido la clase *SymbolFilter* para seleccionar instrumentos financieros con características específicas. Ahora aplicaremos el mismo enfoque a las órdenes.

Dado que no sólo tenemos que analizar órdenes, sino también transacciones y posiciones de forma similar, separaremos la parte general del algoritmo de filtrado en la clase base *TradeFilter* (*TradeFilter.mqh*). Repite casi exactamente el código fuente de *SymbolFilter*. Por lo tanto, no volveremos a explicarlo aquí.

Quienes lo deseen pueden realizar una comparación contextual de los archivos *SymbolFilter.mqh* y *TradeFilter.mqh* para comprobar su similitud y localizar pequeñas modificaciones.

La principal diferencia es que la clase *TradeFilter* es una plantilla, ya que tiene que manejar las propiedades de diferentes objetos: órdenes, transacciones y posiciones.

```
enum IS // supported comparison conditions in filters
{
   EQUAL,
   GREATER,
   NOT_EQUAL,
   LESS
};

enum ENUM_ANY // dummy enum to cast all enums to it
{
};

template<typename T,typename I,typename D,typename S>
class TradeFilter
{
protected:
   MapArray<ENUM_ANY,long> longs;
   MapArray<ENUM_ANY,double> doubles;
   MapArray<ENUM_ANY,string> strings;
   MapArray<ENUM_ANY,IS> conditions;
   ...

   template<typename V>
   static bool equal(const V v1, const V v2);

   template<typename V>
   static bool greater(const V v1, const V v2);

   template<typename V>
   bool match(const T &m, const MapArray<ENUM_ANY,V> &data) const;

public:
   // methods for adding conditions to the filter
   TradeFilter *let(const I property, const long value, const IS cmp = EQUAL);
   TradeFilter *let(const D property, const double value, const IS cmp = EQUAL);
   TradeFilter *let(const S property, const string value, const IS cmp = EQUAL);
   // methods for getting into arrays of records matching the filter
   template<typename E,typename V>
   bool select(const E property, ulong &tickets[], V &data[],
      const bool sort = false) const;
   template<typename E,typename V>
   bool select(const E &property[], ulong &tickets[], V &data[][],
      const bool sort = false) const
   bool select(ulong &tickets[]) const;
   ...
}
```

Los parámetros de plantilla I, D y S son enumeraciones para grupos de propiedades de tres tipos principales (entero, real y cadena): para las órdenes, se describieron en secciones anteriores, por lo que, para mayor claridad, puede imaginar que I=ENUM_ORDER_PROPERTY_INTEGER, D=ENUM_ORDER_PROPERTY_DOUBLE, S=ENUM_ORDER_PROPERTY_STRING.

El tipo T ha sido diseñado para especificar una clase de monitor. Por el momento sólo tenemos listo un monitor, *OrderMonitor*. Más adelante implementaremos *DealMonitor* y *PositionMonitor*.

Anteriormente, en la clase *SymbolFilter*, no utilizamos parámetros de plantilla porque para los símbolos, todos los tipos de enumeraciones de propiedades son invariablemente conocidos, y existe una única clase *SymbolMonitor*.

Recordemos la estructura de la clase filtro. Un grupo de métodos *let* permite registrar una combinación de pares «propiedad=valor» en el filtro, que luego se utilizará para seleccionar objetos en los métodos *select*. La propiedad ID se especifica en el parámetro *property*, y el valor está en el parámetro *value*.

También existen varios métodos *select*. Permiten al código de llamada rellenar un array con los tickets seleccionados, así como, si es necesario, arrays adicionales con los valores de las propiedades de los objetos solicitados. Los identificadores específicos de las propiedades solicitadas se establecen en el primer parámetro del método *select*; puede ser una propiedad o varias. En función de esto, el array receptor debe ser unidimensional o bidimensional.

La combinación de propiedad y valor puede comprobarse no sólo para la igualdad (EQUAL), sino también para las operaciones mayor/menor (GREATER/LESS). Para las propiedades de cadena, es aceptable especificar un patrón de búsqueda con el carácter «*» que indique cualquier secuencia de caracteres (por ejemplo, «*[tp]*» para la propiedad ORDER_COMMENT coincidirá con todos los comentarios en los que aparezca «[tp]» en cualquier lugar, aunque esto es sólo una demostración de la posibilidad -mientras que para buscar órdenes resultantes de *Take Profit* activado debe analizar ORDER_REASON).

Como el algoritmo requiere la implementación de un bucle a través de todos los objetos y los objetos pueden ser de diferentes tipos (hasta ahora son órdenes, pero luego aparecerá la compatibilidad para transacciones y posiciones), necesitamos describir dos métodos abstractos en la clase *TradeFilter*, *total* y *get*:

```
virtual int total() const = 0;
virtual ulong get(const int i) const = 0;
```

El primero devuelve el número de objetos y el segundo devuelve el ticket de orden por su número. Esto debería recordarle el par de funciones *OrdersTotal* y *OrderGetTicket*. De hecho, se utilizan en implementaciones específicas de métodos para filtrar órdenes.

A continuación se muestra la clase *OrderFilter* (*OrderFilter.mqh*) en su totalidad.

```
#include <MQL5Book/OrderMonitor.mqh>
#include <MQL5Book/TradeFilter.mqh>

class OrderFilter: public TradeFilter<OrderMonitor,
   ENUM_ORDER_PROPERTY_INTEGER,
   ENUM_ORDER_PROPERTY_DOUBLE,
   ENUM_ORDER_PROPERTY_STRING>
{
protected:
   virtual int total() const override
   {
      return OrdersTotal();
   }
   virtual ulong get(const int i) const override
   {
      return OrderGetTicket(i);
   }
};
```

Esta simplicidad es especialmente importante dado que se crearán filtros similares sin esfuerzo para operaciones y posiciones.

Con la ayuda de la nueva clase, podemos comprobar mucho más fácilmente la presencia de órdenes pertenecientes a nuestro Asesor Experto, es decir, reemplazar cualquier versión autoescrita de la función *GetMyOrder* utilizada en el ejemplo *PendingOrderModify.mq5*.

```
OrderFilter filter;
ulong tickets[];

// set a condition for orders for the current symbol and our "magic" number
filter.let(ORDER_SYMBOL, _Symbol).let(ORDER_MAGIC, Magic);
// select suitable tickets in an array
if(filter.select(tickets))
{
   ArrayPrint(tickets);
}
```

Con «cualquier versión» nos referimos a que, gracias a la clase de filtro, podemos crear condiciones arbitrarias para seleccionar órdenes y modificarlas «sobre la marcha» (por ejemplo, por indicación del usuario, no del programador).

Como ejemplo de cómo utilizar el filtro, utilicemos un Asesor Experto que crea una cuadrícula de órdenes pendientes para operar en un rebote desde niveles dentro de un determinado rango de precios, es decir, diseñado para un mercado fluctuante. A partir de esta sección y durante las siguientes, modificaremos el Asesor Experto en el contexto del material que se está estudiando.

La primera versión del Asesor Experto *PendingOrderGrid1.mq5* construye una cuadrícula de un tamaño determinado a partir de órdenes Limit y Stop Limit. Los parámetros serán el número de niveles de precios y el paso en puntos entre ellos. El esquema de funcionamiento se ilustra en el siguiente gráfico:

Cuadrícula de órdenes pendientes en 4 niveles con un paso de 200 puntos

A una hora inicial determinada, que puede venir determinada por el horario intradiario y corresponder, por ejemplo, a la «zona plana de la noche», el precio actual se redondea al tamaño del paso de la cuadrícula, y a partir de este nivel se establece un número determinado de niveles hacia arriba y hacia abajo.

En cada nivel superior colocamos una orden límite de venta y una orden límite de compra *stoplimit* con el precio de la orden Limit futura un nivel por debajo. En cada nivel inferior, colocamos una orden límite de compra y una orden de venta *stoplimit* con el precio de la orden Limit futura un nivel por encima.

Cuando el precio toca uno de los niveles, la orden Limit que se encuentra allí se convierte en compra o venta (posición). Al mismo tiempo, una orden Stop Limit del mismo nivel es convertida automáticamente por el sistema en una orden Limit de sentido opuesto en el nivel siguiente.

Por ejemplo, si el precio traspasa el nivel mientras se mueve al alza, obtendremos una posición corta, y se creará una orden Limit de compra a la distancia de paso por debajo del mismo.

El Asesor Experto controlará que en cada nivel haya una orden Stop Limit emparejada con una orden Limit. Por lo tanto, tras detectar una nueva orden límite de compra, el programa le añadirá una orden Stop Limit de venta al mismo nivel, y el precio objetivo de la futura orden Limit será el nivel situado junto al superior, es decir, aquel en el que se abre la posición.

Digamos que el precio baja y activa una orden Limit al nivel inferior: obtendremos una posición larga. Al mismo tiempo, la orden Stop Limit se convierte en una orden Limit para vender al nivel inmediatamente superior. Ahora el Asesor Experto detectará de nuevo una orden Limit «básica» y creará una orden Stop Limit para comprar como par a ella, al mismo nivel que el precio de la futura orden Limit un nivel por debajo.

Si hay posiciones opuestas, las cerraremos. También se establecerá un periodo intradiario en el que el sistema de trading estará habilitado, y durante el resto del tiempo se eliminarán todas las órdenes y

posiciones. Esto, en particular, es útil para la «zona plana de la noche», cuando las fluctuaciones de rentabilidad del mercado son especialmente pronunciadas.

Por supuesto, ésta es sólo una de las muchas posibles implementaciones de la estrategia de cuadrícula que carece de muchas de las personalizaciones de las cuadrículas, pero no complicaremos demasiado el ejemplo.

El Asesor Experto analizará la situación en cada barra (presumiblemente marco temporal H1 o menos). En teoría, la lógica de funcionamiento de este Asesor Experto debe mejorarse respondiendo rápidamente a eventos de trading pero aún no las hemos explorado. Por lo tanto, en lugar de realizar un seguimiento constante y un restablecimiento «manual» instantáneo de las órdenes Limit en los niveles de cuadrícula vacantes, confiamos este trabajo al servidor mediante el uso de órdenes Stop Limit. Sin embargo, aquí hay un matiz.

El hecho es que las órdenes Limit y Stop Limit en cada nivel son de tipos opuestos (*buy/sell*) y por lo tanto son activadas por diferentes tipos de precios.

Resulta que si el mercado subiera al siguiente nivel de la mitad superior de la cuadrícula, el precio *Ask* podría tocar el nivel y activar una orden de compra Stop Limit, pero el precio *Bid* no alcanzaría el nivel, y la orden límite de venta se mantendría tal cual (no se convertiría en una posición). En la mitad inferior de la cuadrícula, cuando el mercado se mueve a la baja, la situación se repite. Cualquier nivel es tocado en primer lugar por el precio *Bid*, y activa una orden Stop Limit de venta, y sólo con un nuevo descenso el nivel es alcanzado también por el precio *Ask*. Si no hay movimiento, la orden Limit de compra permanecerá como está.

Este problema se vuelve crítico a medida que aumenta el diferencial. Por lo tanto, el Asesor Experto requerirá un control adicional sobre las órdenes Limit «extra». En otras palabras: el Asesor Experto no generará una orden Stop Limit que falte en el nivel si ya existe una orden Limit a su supuesto precio objetivo (nivel adyacente).

El código fuente se incluye en el archivo *PendingOrderGrid1.mq5*. En los parámetros de entrada, puede establecer el *Volume* de cada operación (por defecto, si se deja igual a 0, se toma el lote mínimo del símbolo del gráfico), el número de niveles de la cuadrícula *GridSize* (debe ser par), y el paso *GridStep* entre niveles en puntos. Las horas de inicio y fin del segmento intradiario sobre el que se permite trabajar a la estrategia se especifican en los parámetros *StartTime* y *StopTime*: en ambos, sólo importa la hora.

```
#include <MQL5Book/MqlTradeSync.mqh>
#include <MQL5Book/OrderFilter.mqh>
#include <MQL5Book/MapArray.mqh>

input double Volume;                                    // Volume (0 = minimal lot
input uint GridSize = 6;                                // GridSize (even number o
input uint GridStep = 200;                              // GridStep (points)
input ENUM_ORDER_TYPE_TIME Expiration = ORDER_TIME_GTC;
input ENUM_ORDER_TYPE_FILLING Filling = ORDER_FILLING_FOK;
input datetime StartTime = D'1970.01.01 00:00:00';      // StartTime (hh:mm:ss)
input datetime StopTime = D'1970.01.01 09:00:00';       // StopTime (hh:mm:ss)
input ulong Magic = 1234567890;
```

El segmento de tiempo de trabajo puede estar dentro de un día (*StartTime* < *StopTime*) o cruzar el límite del día (*StartTime* > *StopTime*); por ejemplo, de 22:00 a 09:00. Si las dos horas son iguales, se entiende que las operaciones se realizan las veinticuatro horas del día.

Antes de proceder a la aplicación de la idea de trading, vamos a simplificar la tarea de configurar las consultas y la salida de información de diagnóstico en el registro. Para ello, describimos nuestra propia estructura *MqlTradeRequestSyncLog*, la derivada de *MqlTradeRequestSync*.

```
const ulong DAYLONG = 60 * 60 * 24; // length of the day in seconds

struct MqlTradeRequestSyncLog: public MqlTradeRequestSync
{
   MqlTradeRequestSyncLog()
   {
      magic = Magic;
      type_filling = Filling;
      type_time = Expiration;
      if(Expiration == ORDER_TIME_SPECIFIED)
      {
         expiration = (datetime)(TimeCurrent() / DAYLONG * DAYLONG
            + StopTime % DAYLONG);
         if(StartTime > StopTime)
         {
            expiration = (datetime)(expiration + DAYLONG);
         }
      }
   }
   ~MqlTradeRequestSyncLog()
   {
      Print(TU::StringOf(this));
      Print(TU::StringOf(this.result));
   }
};
```

En el constructor, rellenamos todos los campos con valores invariables. En el destructor, registramos campos de consulta y resultado significativos. Obviamente, el destructor de objetos automáticos siempre será llamado en el momento de salida del bloque de código donde se formó y envió la orden, es decir, se imprimirán los datos enviados y recibidos.

En *OnInit* vamos a realizar algunas comprobaciones de la corrección de las variables de entrada; en concreto, para un tamaño de cuadrícula uniforme.

```
int OnInit()
{
   if(GridSize < 2 || !!(GridSize % 2))
   {
      Alert("GridSize should be 2, 4, 6+ (even number)");
      return INIT_FAILED;
   }
   return INIT_SUCCEEDED;
}
```

El principal punto de entrada del algoritmo es el manejador *OnTick*. En él, por brevedad, omitiremos el mismo mecanismo de gestión de errores basado en TRADE_RETCODE_SEVERITY que en el ejemplo *PendingOrderModify.mq5*.

Para trabajar barra a barra, la función dispone de una variable estática *lastBar*, en la que almacenamos la hora de la última barra procesada con éxito. Se omiten todos los ticks posteriores de la misma barra.

```
void OnTick()
{
   static datetime lastBar = 0;
   if(iTime(_Symbol, _Period, 0) == lastBar) return;
   uint retcode = 0;

   ... // main algorithm (see further)

   const TRADE_RETCODE_SEVERITY severity = TradeCodeSeverity(retcode);
   if(severity < SEVERITY_RETRY)
   {
      lastBar = iTime(_Symbol, _Period, 0);
   }
}
```

En lugar de una elipsis, seguirá el algoritmo principal, dividido en varias funciones auxiliares con fines de sistematización. En primer lugar, determinemos si el periodo de trabajo del día está fijado y, en caso afirmativo, si la estrategia está activada en ese momento. Este atributo se almacena en la variable *tradeScheduled*.

```
   ...
   bool tradeScheduled = true;

   if(StartTime != StopTime)
   {
      const ulong now = TimeCurrent() % DAYLONG;

      if(StartTime < StopTime)
      {
         tradeScheduled = now >= StartTime && now < StopTime;
      }
      else
      {
         tradeScheduled = now >= StartTime || now < StopTime;
      }
   }
   ...
```

Con el trading activado, compruebe en primer lugar si ya existe una red de órdenes mediante la función *CheckGrid*. Si no hay red, la función devolverá la constante GRID_EMPTY y deberemos crear la red llamando a *Setup Grid*. Si la red ya se ha construido, tiene sentido comprobar si hay posiciones opuestas que cerrar: de ello se encarga la función *CompactPositions*.

```
   if(tradeScheduled)
   {
      retcode = CheckGrid();

      if(retcode == GRID_EMPTY)
      {
         retcode = SetupGrid();
      }
      else
      {
         retcode = CompactPositions();
      }
   }
   ...
```

Tan pronto como finalice el periodo de trading es necesario eliminar las órdenes y cerrar todas las posiciones (si las hubiera). Esto se hace, respectivamente, mediante las funciones *RemoveOrders* y *CompactPositions function*, pero con una bandera booleana (*true*): este único argumento opcional ordena aplicar un cierre simple para las posiciones restantes después del cierre opuesto.

```
   else
   {
      retcode = CompactPositions(true);
      if(!retcode) retcode = RemoveOrders();
   }
```

Todas las funciones devuelven un código de servidor, cuyo éxito o fracaso se analiza con *TradeCodeSeverity*. Los códigos de aplicación especial GRID_EMPTY y GRID_OK también se consideran estándar según TRADE_RETCODE_SEVERITY.

```
#define GRID_OK    +1
#define GRID_EMPTY  0
```

Veamos ahora las funciones una por una.

La función *CheckGrid* utiliza la clase *OrderFilter* presentada al principio de esta sección. El filtro solicita todas las órdenes pendientes para el símbolo actual y con «nuestro» número de identificación, y los tickets de las órdenes encontradas se almacenan en el array.

```
uint CheckGrid()
{
   OrderFilter filter;
   ulong tickets[];

   filter.let(ORDER_SYMBOL, _Symbol).let(ORDER_MAGIC, Magic)
      .let(ORDER_TYPE, ORDER_TYPE_SELL, IS::GREATER)
      .select(tickets);
   const int n = ArraySize(tickets);
   if(!n) return GRID_EMPTY;
   ...
```

La integridad de la cuadrícula se analiza utilizando la ya conocida clase MapArray, que almacena pares «clave=valor». En este caso, la clave es el nivel (precio convertido en puntos), y el valor es la máscara

de bits (superposición) de tipos de orden en el nivel dado. Además, las órdenes Limit y Stop Limit se contabilizan en las variables *limits* y *stops*, respectivamente.

```
// price levels => masks of types of orders existing there
MapArray<ulong,uint> levels;

const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);
int limits = 0;
int stops = 0;

for(int i = 0; i < n; ++i)
{
   if(OrderSelect(tickets[i]))
   {
      const ulong level = (ulong)MathRound(OrderGetDouble(ORDER_PRICE_OPEN) / poin
      const ulong type = OrderGetInteger(ORDER_TYPE);
      if(type == ORDER_TYPE_BUY_LIMIT || type == ORDER_TYPE_SELL_LIMIT)
      {
         ++limits;
         levels.put(level, levels[level] | (1 << type));
      }
      else if(type == ORDER_TYPE_BUY_STOP_LIMIT
         || type == ORDER_TYPE_SELL_STOP_LIMIT)
      {
         ++stops;
         levels.put(level, levels[level] | (1 << type));
      }
   }
}
...
```

Si el número de órdenes de cada tipo coincide y es igual al tamaño de cuadrícula especificado, entonces todo está en orden.

```
if(limits == stops)
{
   if(limits == GridSize) return GRID_OK; // complete grid

   Alert("Error: Order number does not match requested");
   return TRADE_RETCODE_ERROR;
}
...
```

La situación en la que el número de órdenes Limit es mayor que las Stop Limit es normal: significa que, debido al movimiento del precio, una o más órdenes Stop Limit se han convertido en Limit. A continuación, el programa debe añadir órdenes Stop Limit en los niveles en los que no haya suficientes. La función *RepairGridLevel* permite colocar una orden independiente de un tipo específico para un nivel concreto.

```
      if(limits > stops)
      {
         const uint stopmask =
            (1 << ORDER_TYPE_BUY_STOP_LIMIT) | (1 << ORDER_TYPE_SELL_STOP_LIMIT);
         for(int i = 0; i < levels.getSize(); ++i)
         {
            if((levels[i] & stopmask) == 0) // there is no stop-limit order at this leve
            {
               // the direction of the limit is required to set the reverse stop limit
               const bool buyLimit = (levels[i] & (1 << ORDER_TYPE_BUY_LIMIT));
               // checks for "extra" orders due to the spread are omitted here (see the
               ...
               // create a stop-limit order in the desired direction
               const uint retcode = RepairGridLevel(levels.getKey(i), point, buyLimit);
               if(TradeCodeSeverity(retcode) > SEVERITY_NORMAL)
               {
                  return retcode;
               }
            }
         }
         return GRID_OK;
      }
      ...
```

La situación cuando el número de órdenes Stop Limit es mayor que las Limit se trata como un error (probablemente el servidor omitió el precio por alguna razón).

```
      Alert("Error: Orphaned Stop-Limit orders found");
      return TRADE_RETCODE_ERROR;
   }
```

La función *RepairGridLevel* realiza las siguientes acciones:

```
uint RepairGridLevel(const ulong level, const double point, const bool buyLimit)
{
   const double price = level * point;
   const double volume = Volume == 0 ?
      SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN) : Volume;

   MqlTradeRequestSyncLog request;

   request.comment = "repair";

   // if there is an unpaired buy-limit, set the sell-stop-limit to it
   // if there is an unpaired sell-limit, set buy-stop-limit to it
   const ulong order = (buyLimit ?
      request.sellStopLimit(volume, price, price + GridStep * point) :
      request.buyStopLimit(volume, price, price - GridStep * point));
   const bool result = (order != 0) && request.completed();
   if(!result) Alert("RepairGridLevel failed");
   return request.result.retcode;
}
```

Tenga en cuenta que no necesitamos rellenar realmente la estructura (excepto un comentario que puede hacerse más informativo si es necesario) ya que algunos de los campos son rellenados automáticamente por el constructor, y pasamos el volumen y el precio directamente al método *sellStopLimit* o *buyStopLimit*.

Un enfoque similar se utiliza en la función *SetupGrid*, que crea una nueva red completa de órdenes. Al principio de la función preparamos las variables para los cálculos y describimos el array de estructuras *MqlTradeRequestSyncLog*.

```
uint SetupGrid()
{
   const double current = SymbolInfoDouble(_Symbol, SYMBOL_BID);
   const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);
   const double volume = Volume == 0 ?
      SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN) : Volume;
   // central price of the range rounded to the nearest step,
   // from it up and down we identify the levels
   const double base = ((ulong)MathRound(current / point / GridStep) * GridStep)
      * point;
   const string comment = "G[" + DoubleToString(base,
      (int)SymbolInfoInteger(_Symbol, SYMBOL_DIGITS)) + "]";
   const static string message = "SetupGrid failed: ";
   MqlTradeRequestSyncLog request[][2]; // limit and stop-limit - one pair
   ArrayResize(request, GridSize);      // 2 pending orders per level
```

A continuación, generamos órdenes para la mitad inferior y superior de la cuadrícula, divergiendo desde el centro hacia los lados.

```
   for(int i = 0; i < (int)GridSize / 2; ++i)
   {
      const int k = i + 1;

      // bottom half of the grid
      request[i][0].comment = comment;
      request[i][1].comment = comment;

      if(!(request[i][0].buyLimit(volume, base - k * GridStep * point)))
      {
         Alert(message + (string)i + "/BL");
         return request[i][0].result.retcode;
      }
      if(!(request[i][1].sellStopLimit(volume, base - k * GridStep * point,
         base - (k - 1) * GridStep * point)))
      {
         Alert(message + (string)i + "/SSL");
         return request[i][1].result.retcode;
      }

      // top half of the grid
      const int m = i + (int)GridSize / 2;

      request[m][0].comment = comment;
      request[m][1].comment = comment;

      if(!(request[m][0].sellLimit(volume, base + k * GridStep * point)))
      {
         Alert(message + (string)m + "/SL");
         return request[m][0].result.retcode;
      }
      if(!(request[m][1].buyStopLimit(volume, base + k * GridStep * point,
         base + (k - 1) * GridStep * point)))
      {
         Alert(message + (string)m + "/BSL");
         return request[m][1].result.retcode;
      }
   }
```

Luego comprobamos si está listo.

```
      for(int i = 0; i < (int)GridSize; ++i)
      {
         for(int j = 0; j < 2; ++j)
         {
            if(!request[i][j].completed())
            {
               Alert(message + (string)i + "/" + (string)j + " post-check");
               return request[i][j].result.retcode;
            }
         }
      }
      return GRID_OK;
   }
```

Aunque la comprobación (llamada de *completed*) se espacia con el envío de órdenes, nuestra estructura sigue utilizando internamente la forma síncrona *OrderSend*. De hecho, para acelerar el envío de un lote de órdenes (como en nuestro Asesor Experto de cuadrícula), es mejor utilizar la versión asíncrona *OrderSendAsync*. Pero entonces el estado de ejecución de la orden debe iniciarse desde el manejador de eventos *OnTradeTransaction*, que estudiaremos más adelante.

Un error en el envío de cualquier orden provoca una salida anticipada del bucle y la devolución del código desde el servidor. Este Asesor Experto de simulación simplemente detendrá su trabajo en caso de error. Para un robot real es deseable proporcionar un análisis intelectual del significado del error y, si es necesario, eliminar todas las órdenes y cerrar posiciones.

Las posiciones generadas por órdenes pendientes se cierran mediante la función *CompactPositions*.

```
uint CompactPositions(const bool cleanup = false)
```

El parámetro *cleanup* igual a *false* por defecto significa una «limpieza» regular de las posiciones dentro del periodo de trading, es decir, el cierre de las posiciones opuestas (si las hay). El valor *cleanup=true* se utiliza para forzar el cierre de todas las posiciones al final del periodo de trading.

La función rellena los arrays *ticketsLong* y *ticketsShort* con tickets de posiciones largas y cortas utilizando una función de ayuda *GetMyPositions*. Ya hemos utilizado esta última en el ejemplo *TradeCloseBy.mq5* de la sección Cierre de posiciones opuestas: total y parcial, La función *CloseByPosition* de ese ejemplo ha sufrido cambios mínimos en el nuevo Asesor Experto: devuelve un código del servidor en lugar de un indicador lógico de éxito o error.

```
uint CompactPositions(const bool cleanup = false)
{
   uint retcode = 0;
   ulong ticketsLong[], ticketsShort[];
   const int n = GetMyPositions(_Symbol, Magic, ticketsLong, ticketsShort);
   if(n > 0)
   {
      Print("CompactPositions, pairs: ", n);
      for(int i = 0; i < n; ++i)
      {
         retcode = CloseByPosition(ticketsShort[i], ticketsLong[i]);
         if(retcode) return retcode;
      }
   }
   ...
```

La segunda parte de *CompactPositions* sólo funciona cuando *cleanup=true*, lo cual está lejos de ser perfecto y se reescribirá en breve.

```
   if(cleanup)
   {
      if(ArraySize(ticketsLong) > ArraySize(ticketsShort))
      {
         retcode = CloseAllPositions(ticketsLong, ArraySize(ticketsShort));
      }
      else if(ArraySize(ticketsLong) < ArraySize(ticketsShort))
      {
         retcode = CloseAllPositions(ticketsShort, ArraySize(ticketsLong));
      }
   }

   return retcode;
}
```

Para todas las posiciones restantes encontradas se realiza el cierre habitual llamando a *CloseAllPositions*.

```
uint CloseAllPositions(const ulong &tickets[], const int start = 0)
{
   const int n = ArraySize(tickets);
   Print("CloseAllPositions ", n);
   for(int i = start; i < n; ++i)
   {
      MqlTradeRequestSyncLog request;
      request.comment = "close down " + (string)(i + 1 - start)
         + " of " + (string)(n - start);
      if(!(request.close(tickets[i]) && request.completed()))
      {
         Print("Error: position is not closed ", tickets[i]);
         return request.result.retcode;
      }
   }
   return 0; // success
}
```

Ahora sólo tenemos que considerar la función *RemoveOrders*. También utiliza el filtro de orden para obtener una lista de las mismas, y luego llama al método *remove* en un bucle.

```
uint RemoveOrders()
{
   OrderFilter filter;
   ulong tickets[];
   filter.let(ORDER_SYMBOL, _Symbol).let(ORDER_MAGIC, Magic)
      .select(tickets);
   const int n = ArraySize(tickets);
   for(int i = 0; i < n; ++i)
   {
      MqlTradeRequestSyncLog request;
      request.comment = "removal " + (string)(i + 1) + " of " + (string)n;
      if(!(request.remove(tickets[i]) && request.completed()))
      {
         Print("Error: order is not removed ", tickets[i]);
         return request.result.retcode;
      }
   }
   return 0;
}
```

Vamos a comprobar cómo funciona el Asesor Experto en el probador con la configuración por defecto (período de trading de 00:00 a 09:00). A continuación se muestra una captura de pantalla para el lanzamiento en EURUSD, H1:

Estrategia de cuadrícula PendingOrderGrid1.mq5 en el probador

En el registro, además de las entradas periódicas sobre la creación por lotes de varias órdenes (al principio del día) y su eliminación por la mañana, veremos regularmente el restablecimiento de la red (añadiendo órdenes en lugar de las activadas) y el cierre de posiciones.

```
buy stop limit 0.01 EURUSD at 1.14200 (1.14000) (1.13923 / 1.13923)
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, »
   » @ 1.14200, X=1.14000, ORDER_TIME_GTC, M=1234567890, repair
DONE, #=159, V=0.01, Bid=1.13923, Ask=1.13923, Request executed, Req=287
CompactPositions, pairs: 1
close position #152 sell 0.01 EURUSD by position #153 buy 0.01 EURUSD (1.13923 / 1.13
deal #18 buy 0.01 EURUSD at 1.13996 done (based on order #160)
deal #19 sell 0.01 EURUSD at 1.14202 done (based on order #160)
Positions collapse initiated
OK CloseBy Order/Deal/Position
TRADE_ACTION_CLOSE_BY, EURUSD, ORDER_TYPE_BUY, ORDER_FILLING_FOK, P=152, b=153, »
   » M=1234567890, compacting
DONE, D=18, #=160, Request executed, Req=288
```

Ahora es el momento de analizar las funciones de MQL5 para trabajar con posiciones y mejorar su selección y análisis en nuestro Asesor Experto. Ello se abordará en las siguientes secciones.

6.4.26 Obtener la lista de posiciones

En muchos ejemplos de Asesores Expertos hemos utilizado ya las funciones de la API de MQL5 diseñadas para analizar las posiciones de trading abiertas. En esta sección se presenta su descripción formal.

Es importante tener en cuenta que las funciones de este grupo no permiten crear, modificar o eliminar posiciones. Como hemos visto antes, todas estas acciones se realizan indirectamente mediante el envío de órdenes. Si se ejecutan con éxito, se realizan transacciones, como resultado de las cuales se forman posiciones.

Otra característica es que las funciones sólo son aplicables a las posiciones en línea. Para restablecer el historial de posiciones, es necesario analizar el historial de operaciones.

La función *PositionsTotal* permite averiguar el número total de posiciones abiertas en la cuenta (para todos los instrumentos financieros).

```
int PositionsTotal()
```

Con la contabilidad de compensación de posiciones (ACCOUNT_MARGIN_MODE_RETAIL_NETTING y ACCOUNT_MARGIN_MODE_EXCHANGE) sólo puede haber una posición para cada símbolo en cualquier momento. Esta posición puede ser el resultado de una o varias transacciones.

Con la representación independiente de las posiciones (ACCOUNT_MARGIN_MODE_RETAIL_HEDGING) se pueden abrir varias posiciones simultáneamente para cada símbolo, incluidas las multidireccionales. Cada operación de entrada en el mercado crea una posición independiente, por lo que la ejecución parcial paso a paso de una orden puede generar varias posiciones.

La función *PositionGetSymbol* devuelve el símbolo de una posición por su número.

```
string PositionGetSymbol(int index)
```

El índice debe estar comprendido entre 0 y N-1, donde N es el valor recibido por la llamada previa de *PositionsTotal*. El orden de las posiciones no está regulado.

Si no se encuentra la posición, se devolverá una cadena vacía y el código de error estará disponible en *_LastError*.

Se han ofrecido ejemplos de uso de estas dos funciones en varios Asesores Expertos de prueba (*TrailingStop.mq5*, *TradeCloseBy.mq5*, etc.) en funciones con nombres *GetMyPosition/GetMyPositions*.

Una posición abierta se caracteriza por un ticket único que es el número que la distingue de otras posiciones, pero puede cambiar durante su vida en algunos casos, como una inversión de posición en el modo de compensación por una operación, o como resultado de operaciones de servicio en el servidor (reapertura para cobro de swap, compensación).

Para obtener un ticket de posición por su número, utilizamos la función *PositionGetTicket*.

```
ulong PositionGetTicket(int index)
```

Además, la función resalta una posición en el entorno de trading del terminal, lo que permite leer sus propiedades mediante un grupo de funciones *PositionGet*especiales. En otras palabras: por analogía con las órdenes, el terminal mantiene una caché interna para cada programa MQL a fin de almacenar las propiedades de una posición. Para resaltar una posición, además de *PositionGetTicket*, existen dos funciones: *PositionSelect* y *PositionSelectByTicket*, que analizaremos a continuación.

En caso de error, la función *PositionGetTicket* devolverá 0.

> El ticket no debe confundirse con el identificador que se asigna a cada posición y que nunca cambia. Son los identificadores que se utilizan para vincular posiciones con órdenes y operaciones. Hablaremos de ello un poco más adelante.
>
> Los tickets son necesarios para satisfacer las solicitudes que implican posiciones: los tickets se

especifican en los campos *position* y *position_by* de la estructura *MqlTradeRequest*. Además, al guardar el ticket en una variable, el programa puede seleccionar posteriormente una posición concreta mediante la función *PositionSelectByTicket* (véase más adelante) y trabajar con ella sin recurrir a la enumeración repetida de posiciones en el bucle.

Cuando se invierte una posición en una cuenta de compensación, POSITION_TICKET se cambia por el ticket de la orden que inició esta operación. Sin embargo, una posición de este tipo puede rastrearse utilizando un ID. La inversión de posiciones no se admite en el modo de cobertura.

bool PositionSelect(const string symbol)

La función selecciona una posición abierta por el nombre del instrumento financiero.

Con la representación independiente de las posiciones (ACCOUNT_MARGIN_MODE_RETAIL_HEDGING) puede haber varias posiciones abiertas para cada símbolo al mismo tiempo. En este caso, *PositionSelect* seleccionará la posición con el ticket más pequeño.

El resultado devuelto indica una ejecución correcta (*true*) o incorrecta (*false*) de la función.

El hecho de que las propiedades de la posición seleccionada se almacenen en caché significa que la posición en sí puede dejar de existir, o puede cambiar si el programa lee sus propiedades al cabo de algún tiempo. Se recomienda llamar a la función *PositionSelect* justo antes de acceder a los datos.

bool PositionSelectByTicket(ulong ticket)

La función selecciona una posición abierta para seguir trabajando en el ticket especificado.

Veremos ejemplos de uso de funciones más adelante, cuando estudiemos las propiedades y funciones *PositionGet* relacionadas.

Al construir algoritmos que utilicen las funciones *PositionsTotal*, *OrdersTotal* y similares, deben tenerse en cuenta los principios asíncronos del funcionamiento de los terminales. Ya hemos tocado este tema al escribir las clases *MqlTradeSync.mqh* e implementar la espera de los resultados de ejecución de las solicitudes de operaciones. Sin embargo, esta espera no siempre es posible en el lado del cliente. En concreto, si colocamos una orden pendiente, su transformación en una orden de mercado y su posterior ejecución tendrán lugar en el servidor. En este momento, la orden puede dejar de figurar entre las activas (*OrdersTotal* devolverá 0), pero la posición aún no se muestra (*PositionsTotal* también es igual a 0). Por lo tanto, un programa MQL que tiene una condición para colocar una orden en ausencia de una posición puede iniciar erróneamente una nueva orden, como resultado de lo cual la posición al final se duplicará.

Para resolver este problema, un programa MQL debe analizar el entorno de trading más profundamente que simplemente comprobar el número de órdenes y posiciones a la vez. Por ejemplo, puede mantener una instantánea del último estado correcto del entorno de trading y no permitir que desaparezca ninguna entidad sin algún tipo de confirmación. Sólo entonces podrá formarse una nueva conversión. Así, una orden sólo puede borrarse junto con un cambio de posición (creación, cierre) o trasladarse al historial con un estado de cancelación. Una de las posibles soluciones se propone en forma de clase *TradeGuard* en el archivo *TradeGuard.mqh*. El libro también incluye el script de demostración *TradeGuardExample.mq5* que puede estudiar de forma adicional.

6.4.27 Propiedades de posiciones

Todas las propiedades de posición se dividen en tres grupos según el tipo de valores: enteros y compatibles con ellos, números reales y cadenas. Se utilizan para leer funciones *PositionGet* similares a

las funciones *OrderGet*. Describiremos las funciones en sí en la siguiente sección, y aquí daremos los identificadores de todas las propiedades que están disponibles para especificar en el primer parámetro de estas funciones.

Las propiedades de enteros se proporcionan en la enumeración ENUM_POSITION_PROPERTY_INTEGER.

Identificador	Descripción	Tipo
POSITION_TICKET	Ticket de posición	ulong
POSITION_TIME	Hora de apertura de la posición	datetime
POSITION_TIME_MSC	Hora de apertura de la posición en milisegundos	ulong
POSITION_TIME_UPDATE	Hora de cambio de posición (volumen)	datetime
POSITION_TIME_UPDATE_MSC	Hora de cambio de posición (volumen) en milisegundos	ulong
POSITION_TYPE	Tipo de posición	ENUM_POSITION_TYPE
POSITION_MAGIC	Número mágico de la posición (basado en ORDER_MAGIC)	ulong
POSITION_IDENTIFIER	Identificador de la posición; un número único que se asigna a cada posición recién abierta y que no cambia durante toda su vida.	ulong
POSITION_REASON	Motivo de la apertura de una posición	ENUM_POSITION_REASON

Por regla general, POSITION_IDENTIFIER corresponde al ticket de la orden que abrió la posición. El identificador de la posición se indica en cada orden (ORDER_POSITION_ID) y transacción (DEAL_POSITION_ID) que la abrió, modificó o cerró. Por lo tanto, es conveniente utilizarlo para buscar órdenes y transacciones relacionadas con una posición.

Si la orden se ejecuta parcialmente, pueden existir simultáneamente la posición y la orden pendiente activa para el volumen restante con entradas coincidentes. Además, dicha posición puede cerrarse a tiempo, y en la siguiente cumplimentación del resto de la orden pendiente, volverá a aparecer una posición con el mismo ticket.

En el modo de compensación, la inversión de una posición con una operación se considera un cambio de posición, no una nueva, por lo que POSITION_IDENTIFIER se conserva. Una nueva posición en un símbolo sólo es posible tras cerrar la anterior en volumen cero.

La propiedad POSITION_TIME_UPDATE sólo responde a cambios de volumen (por ejemplo, como resultado de un cierre parcial o un aumento de posición), pero no a otros parámetros como niveles *Stop Loss/Take Profit* o cargos de swap.

Sólo existen dos tipos de posiciones (ENUM_POSITION_TYPE):

Identificador	Descripción
POSITION_TYPE_BUY	Comprar
POSITION_TYPE_SELL	Vender

Las opciones para el origen de una posición, es decir, cómo se abrió la posición, se proporcionan en la enumeración ENUM_POSITION_REASON.

Identificador	Descripción
POSITION_REASON_CLIENT	Activación de una orden cursada desde el terminal de sobremesa.
POSITION_REASON_MOBILE	Activación de una orden cursada desde una aplicación móvil.
POSITION_REASON_WEB	Activación de una orden cursada desde la plataforma web (navegador).
POSITION_REASON_EXPERT	Activación de una orden cursada por un Asesor Experto o un script.

Las propiedades reales se recopilan en ENUM_POSITION_PROPERTY_DOUBLE.

Identificador	Descripción
POSITION_VOLUME	Volumen de la posición
POSITION_PRICE_OPEN	Precio de la posición
POSITION_SL	Precio Stop Loss
POSITION_TP	Precio Take profit
POSITION_PRICE_CURRENT	Precio actual del símbolo
POSITION_SWAP	Swap acumulado
POSITION_PROFIT	Beneficio actual

El tipo de precio actual corresponde a la operación de cierre de posición. Por ejemplo, una posición larga debe cerrarse vendiendo, por lo que el precio *Bid* de la misma se rastrea en POSITION_PRICE_CURRENT.

Por último, se admiten las siguientes propiedades de cadena (ENUM_POSITION_PROPERTY_STRING) para las posiciones.

Identificador	Descripción
POSITION_SYMBOL	El símbolo en el que se abre la posición.
POSITION_COMMENT	Comentario de la posición
POSITION_EXTERNAL_ID	ID de posición en el sistema externo (en la bolsa)

Después de revisar la lista de propiedades de posición, estamos listos para ver las funciones para leer estas propiedades.

6.4.28 Funciones de lectura de propiedades de posición

Un programa MQL puede obtener propiedades de posición utilizando varias funciones *PositionGet* dependiendo del tipo de propiedades. En todas las funciones, la propiedad específica que se solicita se define en el primer parámetro, que toma el ID de una de las enumeraciones ENUM_POSITION_PROPERTY comentadas en la sección anterior.

Para cada tipo de propiedad, existe una forma corta y una forma larga de la función: la primera devuelve directamente el valor de la propiedad, y la segunda lo escribe en el segundo parámetro, pasado por referencia.

Las propiedades de enteros y las propiedades de tipos compatibles (*datetime*, enumeraciones) pueden obtenerse mediante la función *PositionGetInteger*.

```
long PositionGetInteger(ENUM_POSITION_PROPERTY_INTEGER property)
bool PositionGetInteger(ENUM_POSITION_PROPERTY_INTEGER property, long &value)
```

Si falla, la función devuelve 0 o *false*.

La función *PositionGetDouble* se utiliza para obtener propiedades reales.

```
double PositionGetDouble(ENUM_POSITION_PROPERTY_DOUBLE property)
bool PositionGetDouble(ENUM_POSITION_PROPERTY_DOUBLE property, double &value)
```

Por último, la función *PositionGetString* devuelve las propiedades de las cadenas.

```
string PositionGetString(ENUM_POSITION_PROPERTY_STRING property)
bool PositionGetString(ENUM_POSITION_PROPERTY_STRING property, string &value)
```

En caso de fallo, la primera forma de la función devuelve una cadena vacía.

Para leer las propiedades de posición, tenemos ya lista una interfaz abstracta *MonitorInterface* (*TradeBaseMonitor.mqh*) que utilizamos para escribir un monitor de órdenes. Ahora será fácil implantar un monitor similar para las posiciones. El resultado se adjunta en el archivo *PositionMonitor.mqh*.

La clase *PositionMonitorInterface* se hereda de *MonitorInterface* con asignación a los tipos de plantilla I, D y S de las enumeraciones consideradas ENUM_POSITION_PROPERTY, y anula un par de métodos de *stringify* teniendo en cuenta las particularidades de las propiedades de posición.

```
class PositionMonitorInterface:
   public MonitorInterface<ENUM_POSITION_PROPERTY_INTEGER,
   ENUM_POSITION_PROPERTY_DOUBLE,ENUM_POSITION_PROPERTY_STRING>
{
public:
   virtual string stringify(const long v,
      const ENUM_POSITION_PROPERTY_INTEGER property) const override
   {
      switch(property)
      {
         case POSITION_TYPE:
            return enumstr<ENUM_POSITION_TYPE>(v);
         case POSITION_REASON:
            return enumstr<ENUM_POSITION_REASON>(v);

         case POSITION_TIME:
         case POSITION_TIME_UPDATE:
            return TimeToString(v, TIME_DATE | TIME_SECONDS);

         case POSITION_TIME_MSC:
         case POSITION_TIME_UPDATE_MSC:
            return STR_TIME_MSC(v);
      }

      return (string)v;
   }

   virtual string stringify(const ENUM_POSITION_PROPERTY_DOUBLE property,
      const string format = NULL) const override
   {
      if(format == NULL &&
         (property == POSITION_PRICE_OPEN || property == POSITION_PRICE_CURRENT
         || property == POSITION_SL || property == POSITION_TP))
      {
         const int digits = (int)SymbolInfoInteger(PositionGetString(POSITION_SYMBOL)
            SYMBOL_DIGITS);
         return DoubleToString(PositionGetDouble(property), digits);
      }
      return MonitorInterface<ENUM_POSITION_PROPERTY_INTEGER,
         ENUM_POSITION_PROPERTY_DOUBLE,ENUM_POSITION_PROPERTY_STRING>
         ::stringify(property, format);
   }
```

La clase específica del monitor, preparada para ver posiciones, es la siguiente en la cadena de herencia y se basa en las funciones de *PositionGet*. La selección de una posición por ticket se realiza en el constructor.

```
class PositionMonitor: public PositionMonitorInterface
{
public:
   const ulong ticket;
   PositionMonitor(const ulong t): ticket(t)
   {
      if(!PositionSelectByTicket(ticket))
      {
         PrintFormat("Error: PositionSelectByTicket(%lld) failed: %s",
            ticket, E2S(_LastError));
      }
      else
      {
         ready = true;
      }
   }

   virtual long get(const ENUM_POSITION_PROPERTY_INTEGER property) const override
   {
      return PositionGetInteger(property);
   }

   virtual double get(const ENUM_POSITION_PROPERTY_DOUBLE property) const override
   {
      return PositionGetDouble(property);
   }

   virtual string get(const ENUM_POSITION_PROPERTY_STRING property) const override
   {
      return PositionGetString(property);
   }
   ...
};
```

Un sencillo script le permitirá registrar todas las características de la primera posición (si hay al menos una disponible).

```
void OnStart()
{
   PositionMonitor pm(PositionGetTicket(0));
   pm.print();
}
```

En el registro, deberíamos obtener algo como esto:

```
MonitorInterface<ENUM_POSITION_PROPERTY_INTEGER, »
   » ENUM_POSITION_PROPERTY_DOUBLE,ENUM_POSITION_PROPERTY_STRING>
ENUM_POSITION_PROPERTY_INTEGER Count=9
  0 POSITION_TIME=2022.03.24 23:09:45
  1 POSITION_TYPE=POSITION_TYPE_BUY
  2 POSITION_MAGIC=0
  3 POSITION_IDENTIFIER=1291755067
  4 POSITION_TIME_MSC=2022.03.24 23:09:45'261
  5 POSITION_TIME_UPDATE=2022.03.24 23:09:45
  6 POSITION_TIME_UPDATE_MSC=2022.03.24 23:09:45'261
  7 POSITION_TICKET=1291755067
  8 POSITION_REASON=POSITION_REASON_EXPERT
ENUM_POSITION_PROPERTY_DOUBLE Count=8
  0 POSITION_VOLUME=0.01
  1 POSITION_PRICE_OPEN=1.09977
  2 POSITION_PRICE_CURRENT=1.09965
  3 POSITION_SL=0.00000
  4 POSITION_TP=1.10500
  5 POSITION_COMMISSION=0.0
  6 POSITION_SWAP=0.0
  7 POSITION_PROFIT=-0.12
ENUM_POSITION_PROPERTY_STRING Count=3
  0 POSITION_SYMBOL=EURUSD
  1 POSITION_COMMENT=
  2 POSITION_EXTERNAL_ID=
```

Si no hay posiciones abiertas en ese momento, veremos un mensaje de error.

```
Error: PositionSelectByTicket(0) failed: TRADE_POSITION_NOT_FOUND
```

No obstante, el monitor es útil no sólo y no tanto por la salida de las propiedades en el registro. Basándonos en *PositionMonitor*, creamos una clase para seleccionar posiciones por condiciones, similar a lo que hicimos para las órdenes (*OrderFilter*). El objetivo final es mejorar nuestro Asesor Experto de cuadrícula.

Gracias a la programación orientada a objetos (POO), la creación de una nueva clase de filtro no requiere prácticamente ningún esfuerzo. A continuación se muestra el código fuente completo (archivo *PositionFilter.mqh*).

```
class PositionFilter: public TradeFilter<PositionMonitor,
   ENUM_POSITION_PROPERTY_INTEGER,
   ENUM_POSITION_PROPERTY_DOUBLE,
   ENUM_POSITION_PROPERTY_STRING>
{
protected:
   virtual int total() const override
   {
      return PositionsTotal();
   }
   virtual ulong get(const int i) const override
   {
      return PositionGetTicket(i);
   }
};
```

Ahora podemos escribir un script de este tipo para recibir beneficios específicos en posiciones con el número mágico dado, por ejemplo.

```
input ulong Magic;

void OnStart()
{
   PositionFilter filter;

   ENUM_POSITION_PROPERTY_DOUBLE properties[] =
      {POSITION_PROFIT, POSITION_VOLUME};

   double profits[][2];
   ulong tickets[];
   string symbols[];

   filter.let(POSITION_MAGIC, Magic).select(properties, tickets, profits);
   filter.select(POSITION_SYMBOL, tickets, symbols);

   for(int i = 0; i < ArraySize(symbols); ++i)
   {
      PrintFormat("%s[%lld]=%f",
         symbols[i], tickets[i], profits[i][0] / profits[i][1]);
   }
}
```

En este caso hemos tenido que llamar al método *select* dos veces, porque los tipos de propiedades que nos interesan son diferentes: beneficio real y lote, pero el nombre de cadena del instrumento. En una de las secciones al principio del capítulo, cuando estábamos desarrollando la clase de filtro para símbolos, describimos el concepto de tuplas. En MQL5 podemos implementarlo como plantillas de estructura con campos de tipos arbitrarios. Estas tuplas serían muy útiles para finalizar la jerarquía de clases de filtros, ya que entonces sería posible describir el método *select* que rellena un array de tuplas con campos de cualquier tipo.

Las tuplas se describen en el archivo *Tuples.mqh*. Todas sus estructuras tienen un nombre *TupleN<T1,...>*, donde N es un número de 2 a 8, y corresponde al número de parámetros de la plantilla (tipos Ti). Por ejemplo, *Tuple2*:

```
template<typename T1,typename T2>
struct Tuple2
{
   T1 _1;
   T2 _2;

   static int size() { return 2; };

   // M - order, position, deal monitor class, any MonitorInterface<>
   template<typename M>
   void assign(const int &properties[], M &m)
   {
      if(ArraySize(properties) != size()) return;
      _1 = m.get(properties[0], _1);
      _2 = m.get(properties[1], _2);
   }
};
```

En la clase *TradeFilter* (*TradeFilter.mqh*) vamos a añadir una versión de la función *select* con tuplas.

```
template<typename T,typename I,typename D,typename S>
class TradeFilter
{
   ...
 template<typename U> // type U must be Tuple<>, e.g. Tuple3<T1,T2,T3>
   bool select(const int &property[], U &data[], const bool sort = false) const
   {
      const int q = ArraySize(property);
      static const U u;                 // PRB: U::size() does not compile
      if(q != u.size()) return false;   // required condition

      const int n = total();
      // cycle through orders/positions/deals
      for(int i = 0; i < n; ++i)
      {
         const ulong t = get(i);
         // access to properties via monitor T
         T m(t);
         // check all filter conditions for different types of properties
         if(match(m, longs)
         && match(m, doubles)
         && match(m, strings))
         {
            // for a suitable object, store the properties in an array of tuples
            const int k = EXPAND(data);
            data[k].assign(property, m);
         }
      }

      if(sort)
      {
         sortTuple(data, u._1);
      }

      return true;
   }
```

Un array de tuplas puede ordenarse opcionalmente por el primer campo _1, por lo que puede estudiar adicionalmente el método de ayuda *sortTuple*.

Con las tuplas puede consultar un objeto de filtro para propiedades de tres tipos diferentes en una sola llamada a *select*.

A continuación se muestran posiciones con algún número de *Magic*, ordenadas por beneficio; para cada una se obtiene adicionalmente un símbolo y un ticket.

```
input ulong Magic;

  void OnStart()
  {
     int props[] = {POSITION_PROFIT, POSITION_SYMBOL, POSITION_TICKET};
     Tuple3<double,string,ulong> tuples[];
     PositionFilter filter;
     filter.let(POSITION_MAGIC, Magic).select(props, tuples, true);
     ArrayPrint(tuples);
  }
```

Por supuesto, los tipos de parámetros de la descripción del array de tuplas (en este caso, *Tuple3<double,string,ulong>*) deben coincidir con los tipos de enumeración de propiedades solicitados (POSITION_PROFIT, POSITION_SYMBOL, POSITION_TICKET).

Ahora podemos simplificar ligeramente el Asesor Experto de cuadrícula (lo que significa no sólo un código más corto, sino también más comprensible). La nueva versión se llama *PendingOrderGrid2.mq5*. Los cambios afectarán a todas las funciones relacionadas con la gestión de posiciones.

La función *GetMyPositions* rellena el array de tuplas *types4tickets* pasadas por referencia. Se supone que en cada tupla *Tuple2* se almacena el tipo y el ticket de la posición. En este caso concreto, podríamos arreglárnoslas simplemente con un array bidimensional *ulong* en lugar de tuplas porque ambas propiedades son del mismo tipo base. Sin embargo, utilizamos tuplas para demostrar cómo trabajar con ellas en el código de llamada.

```
#include <MQL5Book/Tuples.mqh>
#include <MQL5Book/PositionFilter.mqh>

int GetMyPositions(const string s, const ulong m,
   Tuple2<ulong,ulong> &types4tickets[])
{
   int props[] = {POSITION_TYPE, POSITION_TICKET};
   PositionFilter filter;
   filter.let(POSITION_SYMBOL, s).let(POSITION_MAGIC, m)
      .select(props, types4tickets, true);
   return ArraySize(types4tickets);
}
```

Observe que el último y tercer parámetro del método *select* es igual a *true*, que ordena el array por el primer campo, es decir, el tipo de posiciones. Así, tendremos compras al principio y ventas al final. Esto será necesario para el cierre del contador.

La reencarnación del método *CompactPositions* es la siguiente:

```
uint CompactPositions(const bool cleanup = false)
{
   uint retcode = 0;
   Tuple2<ulong,ulong> types4tickets[];
   int i = 0, j = 0;
   int n = GetMyPositions(_Symbol, Magic, types4tickets);
   if(n > 0)
   {
      Print("CompactPositions: ", n);
      for(i = 0, j = n - 1; i < j; ++i, --j)
      {
         if(types4tickets[i]._1 != types4tickets[j]._1) // as long as the types are d
         {
            retcode = CloseByPosition(types4tickets[i]._2, types4tickets[j]._2);
            if(retcode) return retcode; // error
         }
         else
         {
            break;
         }
      }
   }

   if(cleanup && j < n)
   {
      retcode = CloseAllPositions(types4tickets, i, j + 1);
   }

   return retcode;
}
```

La función *CloseAllPositions* es prácticamente la misma:

```
uint CloseAllPositions(const Tuple2<ulong,ulong> &types4tickets[],
   const int start = 0, const int end = 0)
{
   const int n = end == 0 ? ArraySize(types4tickets) : end;
   Print("CloseAllPositions ", n - start);
   for(int i = start; i < n; ++i)
   {
      MqlTradeRequestSyncLog request;
      request.comment = "close down " + (string)(i + 1 - start)
         + " of " + (string)(n - start);
      const ulong ticket = types4tickets[i]._2;
      if(!(request.close(ticket) && request.completed()))
      {
         Print("Error: position is not closed ", ticket);
         return request.result.retcode; // error
      }
   }
   return 0; // success
}
```

Puede comparar el trabajo de los Asesores Expertos *PendingOrderGrid1.mq5* y *PendingOrderGrid2.mq5* en el probador.

Los informes serán ligeramente diferentes, porque si hay varias posiciones, se cierran en combinaciones opuestas, debido a lo cual el cierre de otras posiciones no emparejadas se produce con respecto a sus diferenciales individuales.

6.4.29 Propiedades de transacciones

Una transacción refleja el hecho de que se ha realizado una operación de trading sobre la base de una orden. Una orden puede generar varias transacciones debido a la ejecución por partes o al cierre opuesto de posiciones.

Las transacciones se caracterizan por propiedades de tres tipos básicos: entero (y compatibles con ellos), real y cadena. Cada propiedad se describe mediante su propia constante en una de las enumeraciones: ENUM_DEAL_PROPERTY_INTEGER, ENUM_DEAL_PROPERTY_DOUBLE, ENUM_DEAL_PROPERTY_STRING.

Para leer las propiedades de las transacciones, utilice las funciones *HistoryDealGet* . Todos ellos asumen que la sección necesaria del historial fue solicitada previamente utilizando funciones especiales para la selección de órdenes y transacciones del historial

Las propiedades de enteros se describen en la enumeración ENUM_DEAL_PROPERTY_INTEGER.

Identificador	Descripción	Tipo
DEAL_TICKET	Ticket de transacción; un número único que se asigna a cada transacción.	ulong
DEAL_ORDER	El ticket de la orden en base a la cual se ejecutó la operación.	ulong
DEAL_TIME	Hora de la transacción	datetime
DEAL_TIME_MSC	Tiempo de la transacción en milisegundos	ulong
DEAL_TYPE	Tipo de transacción	ENUM_DEAL_TYPE (véase más abajo)
DEAL_ENTRY	Dirección de la transacción: entrada en el mercado, salida del mercado o inversión	ENUM_DEAL_ENTRY (véase más abajo)
DEAL_MAGIC	Número mágico de la transacción (basado en ORDER_MAGIC)	ulong
DEAL_REASON	Fuente o motivo de la transacción	ENUM_DEAL_REASON (véase más abajo)
DEAL_POSITION_ID	Identificador de la posición abierta, modificada o cerrada por la transacción	ulong

Los posibles tipos de transacción se representan mediante la enumeración ENUM_DEAL_TYPE.

Identificador	Descripción
DEAL_TYPE_BUY	Comprar
DEAL_TYPE_SELL	Vender
DEAL_TYPE_BALANCE	Saldo devengado
DEAL_TYPE_CREDIT	Devengo de créditos
DEAL_TYPE_CHARGE	Cargos adicionales
DEAL_TYPE_CORRECTION	Corrección
DEAL_TYPE_BONUS	Bonificaciones
DEAL_TYPE_COMMISSION	Comisión adicional
DEAL_TYPE_COMMISSION_DAILY	Comisión cobrada al final del día de trading
DEAL_TYPE_COMMISSION_MONTHLY	Comisión cobrada a final de mes

Identificador	Descripción
DEAL_TYPE_COMMISSION_AGENT_DAILY	Comisión de agente cobrada al final del día de trading
DEAL_TYPE_COMMISSION_AGENT_MONTHLY	Comisión del agente cobrada a final de mes
DEAL_TYPE_INTEREST	Devengo de intereses de los fondos libres
DEAL_TYPE_BUY_CANCELED	Transacción de compra cancelada
DEAL_TYPE_SELL_CANCELED	Transacción de venta cancelada
DEAL_DIVIDEND	Devengo de dividendos
DEAL_DIVIDEND_FRANKED	Devengo de un dividendo con franquicia (exento de impuestos)
DEAL_TAX	Devengo de impuestos

Las opciones DEAL_TYPE_BUY_CANCELED y DEAL_TYPE_SELL_CANCELED reflejan la situación cuando se cancela una transacción anterior. En este caso, el tipo de la transacción ejecutada previamente (DEAL_TYPE_BUY o DEAL_TYPE_SELL) se cambia a DEAL_TYPE_BUY_CANCELED o DEAL_TYPE_SELL_CANCELED, y su beneficio/pérdida se pone a cero. Los beneficios/pérdidas percibidos anteriormente se abonan/cargan en la cuenta como una operación de saldo independiente.

Las transacciones difieren en la forma de cambiar de posición. Puede tratarse de la simple apertura de una posición (entrada en el mercado), el aumento del volumen de una posición abierta previamente, el cierre de una posición con una transacción en sentido opuesto o la inversión de la posición cuando la transacción opuesta cubre el volumen de una posición abierta previamente. Esta última operación sólo se admite en las cuentas de compensación.

Todas estas situaciones se describen mediante los elementos de la enumeración ENUM_DEAL_ENTRY.

Identificador	Descripción
DEAL_ENTRY_IN	Entrada en el mercado
DEAL_ENTRY_OUT	Salida del mercado
DEAL_ENTRY_INOUT	Inversión
DEAL_ENTRY_OUT_BY	Cierre por una posición opuesta

Los motivos de la transacción se resumen en la enumeración ENUM_DEAL_REASON.

Identificador	Descripción
DEAL_REASON_CLIENT	Activación de una orden cursada desde el terminal de sobremesa.
DEAL_REASON_MOBILE	Activación de una orden cursada desde una aplicación móvil.
DEAL_REASON_WEB	Activación de una orden cursada desde la plataforma web.
DEAL_REASON_EXPERT	Activación de una orden cursada por un Asesor Experto o un script.
DEAL_REASON_SL	Orden Stop Loss activada
DEAL_REASON_TP	Activación de la orden Take Profit
DEAL_REASON_SO	Evento Stop Out
DEAL_REASON_ROLLOVER	Transferencia de posición a un nuevo día
DEAL_REASON_VMARGIN	Añadir/deducir margen de variación
DEAL_REASON_SPLIT	Dividir (precio más bajo) el instrumento en el que había una posición

Las propiedades de tipo real se representan mediante la enumeración ENUM_DEAL_PROPERTY_DOUBLE.

Identificador	Descripción
DEAL_VOLUME	Volumen de la transacción
DEAL_PRICE	Precio de la transacción
DEAL_COMMISSION	Comisión de la transacción
DEAL_SWAP	Permuta acumulada al cierre
DEAL_PROFIT	Resultado financiero de la transacción
DEAL_FEE	Comisión por la transacción que se cobra inmediatamente después de la misma.
DEAL_SL	Nivel de Stop Loss
DEAL_TP	Nivel de Take Profit

Las dos últimas propiedades se rellenan de la siguiente manera: para una transacción de entrada o de inversión, el valor *Stop Loss/Take Profit* se toma de la orden por la que se abrió o amplió la posición. Para la transacción de salida, el valor *Stop Loss/Take Profit* se toma de la posición en el momento de su cierre.

Las propiedades de transacción de cadenas están disponibles a través de las constantes de enumeración ENUM_DEAL_PROPERTY_STRING.

Identificador	Descripción
DEAL_SYMBOL	El nombre del símbolo por el que se ha realizado la transacción
DEAL_COMMENT	Comentarios de la transacción
DEAL_EXTERNAL_ID	Identificador de la transacción en el sistema de trading externo (en la bolsa)

Probaremos cómo leer las propiedades en la sección sobre las funciones *HistoryDealGet* a través de las clases *DealMonitor* y *DealFilter*.

6.4.30 Seleccionar órdenes y transacciones del historial

MetaTrader 5 le permite crear una instantánea del historial para un período de tiempo específico para un Asesor Experto o un script. La instantánea es una lista de órdenes y transacciones a la que se puede acceder posteriormente a través de las funciones correspondientes. Además, se puede solicitar el historial en relación con órdenes, transacciones o posiciones concretas.

La selección explícita del periodo deseado (por fechas) se realiza mediante la función *HistorySelect*. A continuación se puede buscar el tamaño de la lista de transacciones y de la lista de órdenes mediante las funciones *HistoryDealsTotal* y *HistoryOrdersTotal*, respectivamente. Los elementos de la lista de órdenes pueden comprobarse mediante la función *HistoryOrderGetTicket*; para los elementos de la lista de transacciones, utilice *HistoryDealGetTicket*.

> Es necesario distinguir entre las órdenes activas (de trabajo) y las órdenes del historial, es decir, las ejecutadas, canceladas o rechazadas. Para analizar las órdenes activas, utilice las funciones comentadas en las secciones relacionadas con obtener una lista de órdenes activas y leer sus propiedades

bool HistorySelect(datetime from, datetime to)

La función solicita el historial de transacciones y órdenes para el periodo especificado de tiempo del servidor (*from* y *to* inclusive, *to* >= *from*) y devuelve *true* en caso de éxito.

Incluso si no hay órdenes y transacciones en el periodo solicitado, la función devolverá *true* en ausencia de errores. Un error puede ser, por ejemplo, la falta de memoria para elaborar una lista de órdenes o transacciones.

> Tenga en cuenta que las órdenes tienen dos tiempos: establecimiento (ORDER_TIME_SETUP) y ejecución (ORDER_TIME_DONE). La función *HistorySelect* selecciona las órdenes por tiempo de ejecución.

Para extraer todo el historial de la cuenta, puede utilizar la sintaxis *HistorySelect(0, LONG_MAX)*.

Otra forma de acceder a una parte del historial es por ID de posición.

bool HistorySelectByPosition(ulong positionID)

La función solicita el historial de transacciones y órdenes con el ID de posición especificado en las propiedades ORDER_POSITION_ID, DEAL_POSITION_ID.

> ¡Atención! La función no selecciona órdenes por el ID de la posición opuesta para operaciones Close By. En otras palabras: la propiedad ORDER_POSITION_BY_ID se ignora, a pesar de que los datos de la orden intervinieron en la formación de la posición.
>
> Por ejemplo, un Asesor Experto podría completar una compra (orden nº 1) y una venta (orden nº 2) en una cuenta con cobertura. Así se formarán las posiciones nº 1 y nº 2. El cierre opuesto de posiciones requiere la orden ORDER_TYPE_CLOSE_BY (nº 3). Como resultado, la llamada *HistorySelectByPosition(#1)* seleccionará las órdenes nº 1 y nº 3, lo que es de esperar. Sin embargo, la llamada de *HistorySelectByPosition(#2)* seleccionará sólo la orden nº 2 (a pesar de que la orden nº 3 tiene nº 2 en la propiedad ORDER_POSITION_BY_ID, y estrictamente hablando, la orden nº 3 participó en el cierre de la posición nº 2).

Al ejecutar con éxito cualquiera de las dos funciones, *HistorySelect* o *HistorySelectByPosition*, el terminal genera una lista interna de órdenes y transacciones para el programa MQL. También puede cambiar el contexto histórico con las funciones *HistoryOrderSelect* y *HistoryDealSelect*, para lo cual necesita conocer de antemano el ticket del objeto correspondiente (por ejemplo, guardarlo del resultado de la solicitud).

Es importante tener en cuenta que *HistoryOrderSelect* sólo afecta a la lista de órdenes, y *HistoryDealSelect* sólo se utiliza para la lista de transacciones.

Todas las funciones de selección de contexto devuelven un valor *bool* en caso de éxito (*true*) o error (*false*). El código de error puede leerse en la variable integrada *_LastError*.

bool HistoryOrderSelect(ulong ticket)

La función *HistoryOrderSelect* selecciona una orden en el historial por su ticket. A continuación, la orden se utiliza para otras operaciones con la transacción (lectura de propiedades).

Durante la aplicación de la función *HistoryOrderSelect*, si la búsqueda de una orden por ticket ha tenido éxito, la nueva lista de órdenes seleccionadas en el historial estará formada por la única orden que se acaba de encontrar. En otras palabras: se restablece la lista anterior de órdenes seleccionadas (si la hubiera). Sin embargo, la función no restablece el historial de transacciones previamente seleccionado, es decir, no selecciona la(s) transacción(es) asociada(s) a la orden.

bool HistoryDealSelect(ulong ticket)

La función *HistoryDealSelect* selecciona una transacción en el historial para acceder a ella a través de las funciones correspondientes. La función no restablece el historial de órdenes, es decir, no selecciona la orden asociada a la transacción seleccionada.

Después de seleccionar un determinado contexto en el historial llamando a una de las funciones anteriores, el programa MQL puede llamar a las funciones para iterar sobre las órdenes y transacciones que entran en este contexto y leer sus propiedades.

int HistoryOrdersTotal()

La función *HistoryOrdersTotal* devuelve el número de órdenes en el historial (en la selección).

ulong HistoryOrderGetTicket(int index)

La función *HistoryOrderGetTicket* permite obtener un ticket de orden por su número de serie en el contexto del historial seleccionado. El índice debe estar comprendido entre 0 y N-1, donde N se obtiene de la función *HistoryOrdersTotal*.

Conociendo el ticket de la orden es fácil obtener todas las propiedades necesarias del mismo utilizando las funciones *HistoryOrderGet*. Las propiedades de las órdenes históricas son exactamente las mismas que las de órdenes existentes.

Existe un par de funciones similares para trabajar con transacciones.

int HistoryDealsTotal()

La función *HistoryDealsTotal* devuelve el número de transacciones en el historial (en la selección).

ulong HistoryDealGetTicket(int index)

La función *HistoryDealGetTicket* permite obtener un ticket de transacción por su número de serie en el contexto del historial seleccionado. Esto es necesario para el tratamiento posterior de la transacción mediante funciones *HistoryDealGet*. La lista de las propiedades de transacción accesibles a través de estas funciones se describió en la sección anterior.

Consideraremos un ejemplo de utilización de funciones después de estudiar las funciones *HistoryOrderGet* y *HistoryDealGet*.

6.4.31 Funciones para leer propiedades de órdenes del historial

Las funciones para la lectura de las propiedades de las órdenes históricas se dividen en 3 grupos, según el tipo básico de los valores de las propiedades, de acuerdo con la división de los identificadores de las propiedades disponibles en tres enumeraciones: ENUM_ORDER_PROPERTY_INTEGER, ENUM_ORDER_PROPERTY_DOUBLE y ENUM_ORDER_PROPERTY_STRING que abordamos anteriormente en una sección aparte al explorar las órdenes activas.

Antes de llamar a estas funciones, es necesario seleccionar de alguna manera el conjunto adecuado de tickets del historial.

Si intenta leer las propiedades de una orden o de una transacción que tenga tickets fuera del contexto de historial seleccionado, el entorno puede generar un error WRONG_INTERNAL_PARAMETER (4002), que puede analizarse a través de *_LastError*.

Para cada tipo de propiedad base, existen dos formas de función: una devuelve directamente el valor de la propiedad solicitada, la segunda lo escribe en un parámetro pasado por referencia y devuelve un indicador de éxito (*true*) o errores (*false*).

Para los tipos de enteros y compatibles (*datetime*, enums) de propiedades existe una función dedicada *HistoryOrderGetInteger*.

```
long HistoryOrderGetInteger(ulong ticket, ENUM_ORDER_PROPERTY_INTEGER property)
bool HistoryOrderGetInteger(ulong ticket, ENUM_ORDER_PROPERTY_INTEGER property,
 long &value)
```

La función le permite encontrar la orden *property* del historial seleccionado por su número de ticket.

Para las propiedades reales, se asigna la función *HistoryOrderGetDouble*.

```
double HistoryOrderGetDouble(ulong ticket, ENUM_ORDER_PROPERTY_DOUBLE property)
bool HistoryOrderGetDouble(ulong ticket, ENUM_ORDER_PROPERTY_DOUBLE property,
 double &value)
```

Por último, las propiedades de cadena pueden leerse con *HistoryOrderGetString*.

string HistoryOrderGetString(ulong ticket, ENUM_ORDER_PROPERTY_STRING property)
bool HistoryOrderGetString(ulong ticket, ENUM_ORDER_PROPERTY_STRING property,
 string &value)

Ahora podemos complementar la clase *OrderMonitor* (*OrderMonitor.mqh*) para trabajar con órdenes históricas. En primer lugar, vamos a añadir una variable booleana a la clase *history*, que rellenaremos en el constructor en función del segmento en el que se seleccionó la orden con el ticket pasado: entre los activos (*OrderSelect*) o en el historial (*HistoryOrderSelect*).

```
class OrderMonitor: public OrderMonitorInterface
{
   bool history;

public:
   const ulong ticket;
   OrderMonitor(const long t): ticket(t), history(!OrderSelect(t))
   {
      if(history && !HistoryOrderSelect(ticket))
      {
         PrintFormat("Error: OrderSelect(%lld) failed: %s", ticket, E2S(_LastError));
      }
      else
      {
         ResetLastError();
         ready = true;
      }
   }
   ...
```

Necesitamos llamar a la función *ResetLastError* en una rama exitosa de *if* para restablecer el posible error que podría establecer la función *OrderSelect* (si la orden está en el historial).

De hecho, esta versión del constructor contiene un grave error lógico, y volveremos a él dentro de unos párrafos.

Para leer propiedades en métodos get, llamamos ahora a diferentes funciones integradas, dependiendo del valor de la variable *history*.

```
virtuallongget(constENUM_ORDER_PROPERTY_INTEGERproperty)constoverride
{
returnhistory?HistoryOrderGetInteger(ticket,property) :OrderGetInteger(property);
}

virtualdoubleget(constENUM_ORDER_PROPERTY_DOUBLEproperty)constoverride
{
returnhistory?HistoryOrderGetDouble(ticket,property) :OrderGetDouble(property);
}

virtualstringget(constENUM_ORDER_PROPERTY_STRINGproperty)constoverride
{
returnhistory?HistoryOrderGetString(ticket,property) :OrderGetString(property);
}
...
```

El objetivo principal de la clase *OrderMonitor* es suministrar datos a otras clases analíticas. Los objetos *OrderMonitor* se utilizan para filtrar órdenes activas en la clase *OrderFilter*, y necesitamos una clase similar para seleccionar órdenes por condiciones arbitrarias en el historial: *HistoryOrderFilter*.

Escribamos esta clase en el mismo archivo *OrderFilter.mqh*, que utiliza dos nuevas funciones para trabajar con el historial: *HistoryOrdersTotal* y *HistoryOrderGetTicket*.

```
class HistoryOrderFilter: public TradeFilter<OrderMonitor,
   ENUM_ORDER_PROPERTY_INTEGER,
   ENUM_ORDER_PROPERTY_DOUBLE,
   ENUM_ORDER_PROPERTY_STRING>
{
protected:
   virtual int total() const override
   {
      return HistoryOrdersTotal();
   }
   virtual ulong get(const int i) const override
   {
      return HistoryOrderGetTicket(i);
   }
};
```

Este sencillo código hereda de la clase de plantilla *TradeFilter*, donde la clase se pasa como primer parámetro de la plantilla *OrderMonitor* para leer las propiedades de los objetos correspondientes (vimos un análogo para posiciones, y pronto crearemos uno para transacciones).

Aquí radica el problema con el constructor *OrderMonitor*. Como descubrimos en la sección Seleccionar órdenes y transacciones del historial, para analizar la cuenta, primero debemos establecer el contexto con una de las funciones, como *HistorySelect*. Así que aquí en el código fuente *HistoryOrderFilter* se asume que el programa MQL ya ha seleccionado el fragmento de historial requerido. Sin embargo, la nueva versión intermedia del constructor *OrderMonitor* utiliza la llamada *HistoryOrderSelect* para comprobar la existencia de un ticket en el historial. Mientras tanto, esta función restablece el contexto anterior de órdenes históricas y selecciona una única orden.

Por tanto, necesitamos un método de ayuda *historyOrderSelectWeak* para validar el ticket de una manera «suave», sin romper el contexto existente. Para ello, basta con comprobar si la propiedad ORDER_TICKET es igual al ticket pasado *t*: *(HistoryOrderGetInteger(t, ORDER_TICKET) == t)*. Si dicho

ticket ya ha sido seleccionado (disponible), la comprobación tendrá éxito y el monitor no necesitará manipular el historial.

```
class OrderMonitor: public OrderMonitorInterface
{
   bool historyOrderSelectWeak(const ulong t) const
   {
      return (((HistoryOrderGetInteger(t, ORDER_TICKET) == t) ||
         (HistorySelect(0, LONG_MAX) && (HistoryOrderGetInteger(t, ORDER_TICKET) == t
   }
   bool history;

public:
   const ulong ticket;
   OrderMonitor(const long t): ticket(t), history(!OrderSelect(t))
   {
      if(history && !historyOrderSelectWeak(ticket))
      {
         PrintFormat("Error: OrderSelect(%lld) failed: %s", ticket, E2S(_LastError));
      }
      else
      {
         ResetLastError();
         ready = true;
      }
   }
```

Un ejemplo de aplicación del filtrado de órdenes en el historial se estudiará en la siguiente sección, después de que preparemos una funcionalidad similar para las transacciones.

6.4.32 Funciones para leer propiedades de transacciones del historial

Para leer las propiedades de las transacciones existen grupos de funciones organizados por tipo de propiedad: entero, real y cadena. Antes de llamar a las funciones es necesario seleccionar el período del historial deseado y garantizar así la disponibilidad de transacciones con tickets que se pasan en el primer parámetro (*ticket*) de todas las funciones.

Existen dos formas para cada tipo de propiedad: devolver un valor directamente y escribir en una variable por referencia. La segunda devuelve *true* para indicar el éxito. La primera simplemente devolverá 0 en caso de error. El código de error se encuentra en la variable *_LastError*.

Los tipos de propiedad de enteros y compatibles (*datetime*, enumeraciones) pueden obtenerse utilizando la función *HistoryDealGetInteger*.

```
long HistoryDealGetInteger(ulong ticket, ENUM_DEAL_PROPERTY_INTEGER property)
bool HistoryDealGetInteger(ulong ticket, ENUM_DEAL_PROPERTY_INTEGER property,
 long &value)
```

Las propiedades reales se leen mediante la función *HistoryDealGetDouble*.

double HistoryDealGetDouble(ulong ticket, ENUM_DEAL_PROPERTY_DOUBLE property)
bool HistoryDealGetDouble(ulong ticket, ENUM_DEAL_PROPERTY_DOUBLE property,
 double &value)

Para las propiedades de cadena existe la función *HistoryDealGetString*.

string HistoryDealGetString(ulong ticket, ENUM_DEAL_PROPERTY_STRING property)
bool HistoryDealGetString(ulong ticket, ENUM_DEAL_PROPERTY_STRING property,
 string &value)

La clase *DealMonitor* (*DealMonitor.mqh*), organizada exactamente igual que *OrderMonitor* y *PositionMonitor*, proporcionará una lectura unificada de las propiedades de las transacciones. La clase base es *DealMonitorInterface*, heredada de la plantilla *MonitorInterface* (la describimos en la sección Funciones para leer las propiedades de órdenes activas). Es en este nivel donde se especifican los tipos concretos de enumeraciones ENUM_DEAL_PROPERTY como parámetros de plantilla y la implementación específica del método *stringify*.

```
#include <MQL5Book/TradeBaseMonitor.mqh>

class DealMonitorInterface:
   public MonitorInterface<ENUM_DEAL_PROPERTY_INTEGER,
   ENUM_DEAL_PROPERTY_DOUBLE,ENUM_DEAL_PROPERTY_STRING>
{
public:
   // property descriptions taking into account integer subtypes
   virtual string stringify(const long v,
      const ENUM_DEAL_PROPERTY_INTEGER property) const override
   {
      switch(property)
      {
         case DEAL_TYPE:
            return enumstr<ENUM_DEAL_TYPE>(v);
         case DEAL_ENTRY:
            return enumstr<ENUM_DEAL_ENTRY>(v);
         case DEAL_REASON:
            return enumstr<ENUM_DEAL_REASON>(v);

         case DEAL_TIME:
            return TimeToString(v, TIME_DATE | TIME_SECONDS);

         case DEAL_TIME_MSC:
            return STR_TIME_MSC(v);
      }

      return (string)v;
   }
};
```

La clase *DealMonitor* que se muestra a continuación es algo similar a una clase modificada recientemente para trabajar con el historial *OrderMonitor*. Además de la aplicación de las funciones *HistoryDeal* en lugar de las funciones *HistoryOrder*, cabe señalar que para las transacciones no es necesario comprobar el ticket en el entorno en línea porque las mismas sólo existen en el historial.

```
class DealMonitor: public DealMonitorInterface
{
   bool historyDealSelectWeak(const ulong t) const
   {
      return ((HistoryDealGetInteger(t, DEAL_TICKET) == t) ||
         (HistorySelect(0, LONG_MAX) && (HistoryDealGetInteger(t, DEAL_TICKET) == t))
   }
public:
   const ulong ticket;
   DealMonitor(const long t): ticket(t)
   {
      if(!historyDealSelectWeak(ticket))
      {
         PrintFormat("Error: HistoryDealSelect(%lld) failed", ticket);
      }
      else
      {
         ready = true;
      }
   }

   virtual long get(const ENUM_DEAL_PROPERTY_INTEGER property) const override
   {
      return HistoryDealGetInteger(ticket, property);
   }

   virtual double get(const ENUM_DEAL_PROPERTY_DOUBLE property) const override
   {
      return HistoryDealGetDouble(ticket, property);
   }

   virtual string get(const ENUM_DEAL_PROPERTY_STRING property) const override
   {
      return HistoryDealGetString(ticket, property);
   }
   ...
};
```

Basándose en *DealMonitor* y *TradeFilter* es fácil crear un filtro de transacciones (*DealFilter.mqh*). Recordemos que *TradeFilter*, como clase base para muchas entidades, se describió en la sección Seleccionar órdenes por propiedades.

```
#include <MQL5Book/DealMonitor.mqh>
#include <MQL5Book/TradeFilter.mqh>

class DealFilter: public TradeFilter<DealMonitor,
   ENUM_DEAL_PROPERTY_INTEGER,
   ENUM_DEAL_PROPERTY_DOUBLE,
   ENUM_DEAL_PROPERTY_STRING>
{
protected:
   virtual int total() const override
   {
      return HistoryDealsTotal();
   }
   virtual ulong get(const int i) const override
   {
      return HistoryDealGetTicket(i);
   }
};
```

Como ejemplo generalizado de trabajo con historiales, considere el script de recuperación de historiales de posición *TradeHistoryPrint.mq5*.

TradeHistoryPrint

El script creará un historial para el símbolo del gráfico actual.

Primero necesitamos filtros para transacciones y órdenes.

```
#include <MQL5Book/OrderFilter.mqh>
#include <MQL5Book/DealFilter.mqh>
```

De las transacciones extraeremos los identificadores de posición y, a partir de ellos, solicitaremos detalles sobre las órdenes.

El historial puede visualizarse en su totalidad o para una posición específica, para lo cual proporcionaremos una selección de modo y un campo de entrada para el identificador en las variables de entrada.

```
enum SELECTOR_TYPE
{
   TOTAL,    // Whole history
   POSITION, // Position ID
};

input SELECTOR_TYPE Type = TOTAL;
input ulong PositionID = 0; // Position ID
```

Hay que tener en cuenta que el muestreo de un historial de cuenta largo puede suponer una sobrecarga, por lo que es deseable prever el almacenamiento en caché de los resultados obtenidos del procesamiento del historial en los Asesores Expertos de trabajo, junto con la marca de tiempo del último procesamiento. Con cada análisis posterior del historial, puede iniciar el proceso no desde el principio, sino desde un momento recordado.

Para mostrar información sobre registros del historial con alineación de columnas de una forma visualmente atractiva, tiene sentido representarla como un array de estructuras. Sin embargo,

nuestros filtros ya permiten consultar datos almacenados en estructuras especiales: las tuplas. Por lo tanto, aplicaremos un truco: describiremos las estructuras de nuestra aplicación, observando las reglas de las tuplas:

- El primer campo debe tener el nombre _1; se utiliza opcionalmente en el algoritmo de ordenación.
- La función *size* que devuelve el número de campos debe describirse en la estructura.
- La estructura debe tener un método de plantilla *assign* para rellenar los campos a partir de las propiedades del objeto monitor pasado derivado de *MonitorInterface*.

En las tuplas estándar, el método *assign* se describe de la siguiente manera:

```
template<typename M>
void assign(const int &properties[], M &m);
```

Como primer parámetro, recibe un array con los ID de las propiedades correspondientes a los campos que nos interesan. De hecho, este es el array que pasa el código llamante al método *select* del filtro (*TradeFilter::select*), y que luego por referencia llega a *assign*. Pero como ahora crearemos, no unas tuplas estándar, sino nuestras propias estructuras que «conocen» la naturaleza aplicada de sus campos, podemos dejar el array con identificadores de propiedades dentro de la propia estructura y no «conducirlo» al filtro y de vuelta al método *assign* de la misma estructura.

En concreto, para solicitar transacciones, describimos la estructura *DealTuple* con 8 campos. Sus identificadores se especificarán en el array estático *fields*.

```
struct DealTuple
{
   datetime _1;   // deal time
   ulong deal;    // deal ticket
   ulong order;   // order ticket
   string type;   // ENUM_DEAL_TYPE as string
   string in_out; // ENUM_DEAL_ENTRY as string
   double volume;
   double price;
   double profit;

   static int size() { return 8; }; // number of properties
   static const int fields[]; // identifiers of the requested deal properties
   ...
};

static const int DealTuple::fields[] =
{
   DEAL_TIME, DEAL_TICKET, DEAL_ORDER, DEAL_TYPE,
   DEAL_ENTRY, DEAL_VOLUME, DEAL_PRICE, DEAL_PROFIT
};
```

Este enfoque reúne identificadores y campos para almacenar los valores correspondientes en un único lugar, lo que facilita la comprensión y el mantenimiento del código fuente.

Rellenar los campos con valores de propiedades requerirá una versión ligeramente modificada (simplificada) del método *assign* que toma los ID del array *fields* y no del parámetro de entrada.

```
struct DealTuple
{
   ...
   template<typename M> // M is derived from MonitorInterface<>
   void assign(M &m)
   {
      static const int DEAL_TYPE_ = StringLen("DEAL_TYPE_");
      static const int DEAL_ENTRY_ = StringLen("DEAL_ENTRY_");
      static const ulong L = 0; // default type declaration (dummy)

      _1 = (datetime)m.get(fields[0], L);
      deal = m.get(fields[1], deal);
      order = m.get(fields[2], order);
      const ENUM_DEAL_TYPE t = (ENUM_DEAL_TYPE)m.get(fields[3], L);
      type = StringSubstr(EnumToString(t), DEAL_TYPE_);
      const ENUM_DEAL_ENTRY e = (ENUM_DEAL_ENTRY)m.get(fields[4], L);
      in_out = StringSubstr(EnumToString(e), DEAL_ENTRY_);
      volume = m.get(fields[5], volume);
      price = m.get(fields[6], price);
      profit = m.get(fields[7], profit);
   }
};
```

Al mismo tiempo, convertimos los elementos numéricos de las enumeraciones ENUM_DEAL_TYPE y ENUM_DEAL_ENTRY en cadenas fáciles de usar. Por supuesto, esto sólo es necesario para el registro. Para el análisis programático, los tipos deben dejarse como están.

Dado que hemos inventado una nueva versión del método *assign* en sus tuplas, es necesario añadir una nueva versión del método *select* para ello en la clase *TradeFilter*. La innovación será sin duda útil para otros programas, por lo que la introduciremos directamente en *TradeFilter*, no en una nueva clase derivada.

```
template<typename T,typename I,typename D,typename S>
class TradeFilter
{
   ...
   template<typename U> // U must have first field _1 and method assign(T)
   bool select(U &data[], const bool sort = false) const
   {
      const int n = total();
      // loop through the elements
      for(int i = 0; i < n; ++i)
      {
         const ulong t = get(i);
         // read properties through the monitor object
         T m(t);
         // check all filtering conditions
         if(match(m, longs)
         && match(m, doubles)
         && match(m, strings))
         {
            // for a suitable object, add its properties to an array
            const int k = EXPAND(data);
            data[k].assign(m);
         }
      }

      if(sort)
      {
         static const U u;
         sortTuple(data, u._1);
      }

      return true;
   }
```

Recuerde que todos los métodos de plantilla no son implementados por el compilador hasta que se llaman en código con un tipo específico. Por lo tanto, la presencia de dichos patrones en *TradeFilter* no obliga a incluir ningún archivo de encabezado de tuplas ni a describir estructuras similares si no se utilizan.

Así, antes, para seleccionar transacciones usando una tupla estándar, tendríamos que escribir algo como lo siguiente:

```
#include <MQL5Book/Tuples.mqh>
...
DealFilter filter;
int properties[] =
{
   DEAL_TIME, DEAL_TICKET, DEAL_ORDER, DEAL_TYPE,
   DEAL_ENTRY, DEAL_VOLUME, DEAL_PRICE, DEAL_PROFIT
};
Tuple8<ulong,ulong,ulong,ulong,ulong,double,double,double> tuples[];
filter.let(DEAL_SYMBOL, _Symbol).select(properties, tuples);
```

Pero ahora, con una estructura personalizada, todo es mucho más sencillo:

```
DealFilter filter;
DealTuple tuples[];
filter.let(DEAL_SYMBOL, _Symbol).select(tuples);
```

De forma similar a la estructura *DealTuple*, vamos a describir la estructura de 10 campos para las órdenes *OrderTuple*.

```
struct OrderTuple
{
   ulong _1;       // ticket (also used as 'ulong' prototype)
   datetime setup;
   datetime done;
   string type;
   double volume;
   double open;
   double current;
   double sl;
   double tp;
   string comment;

   static int size() { return 10; }; // number of properties
   static const int fields[]; // identifiers of requested order properties

   template<typename M> // M is derived from MonitorInterface<>
   void assign(M &m)
   {
      static const int ORDER_TYPE_ = StringLen("ORDER_TYPE_");

      _1 = m.get(fields[0], _1);
      setup = (datetime)m.get(fields[1], _1);
      done = (datetime)m.get(fields[2], _1);
      const ENUM_ORDER_TYPE t = (ENUM_ORDER_TYPE)m.get(fields[3], _1);
      type = StringSubstr(EnumToString(t), ORDER_TYPE_);
      volume = m.get(fields[4], volume);
      open = m.get(fields[5], open);
      current = m.get(fields[6], current);
      sl = m.get(fields[7], sl);
      tp = m.get(fields[8], tp);
      comment = m.get(fields[9], comment);
   }
};

static const int OrderTuple::fields[] =
{
   ORDER_TICKET, ORDER_TIME_SETUP, ORDER_TIME_DONE, ORDER_TYPE, ORDER_VOLUME_INITIAL,
   ORDER_PRICE_OPEN, ORDER_PRICE_CURRENT, ORDER_SL, ORDER_TP, ORDER_COMMENT
};
```

Ahora todo está listo para implementar la función principal del script: *OnStart*. Al principio, describiremos los objetos de los filtros para transacciones y órdenes.

```
void OnStart()
{
   DealFilter filter;
   HistoryOrderFilter subfilter;
   ...
```

En función de las variables de entrada, elegimos todo el historial o una posición concreta.

```
if(PositionID == 0 || Type == TOTAL)
{
   HistorySelect(0, LONG_MAX);
}
else if(Type == POSITION)
{
   HistorySelectByPosition(PositionID);
}
...
```

A continuación, recopilaremos todos los identificadores de posición en un array, o dejaremos uno especificado por el usuario.

```
ulong positions[];
if(PositionID == 0)
{
   ulong tickets[];
   filter.let(DEAL_SYMBOL, _Symbol)
      .select(DEAL_POSITION_ID, tickets, positions, true); // true - sorting
   ArrayUnique(positions);
}
else
{
   PUSH(positions, PositionID);
}

const int n = ArraySize(positions);
Print("Positions total: ", n);
if(n == 0) return;
...
```

La función de ayuda *ArrayUnique* deja los elementos no repetidos en el array. Ello requiere que el array de origen esté ordenado para que funcione.

Además, en un bucle a través de las posiciones, solicitamos transacciones y órdenes relacionadas con cada una de ellas. Las transacciones se ordenan por el primer campo de la estructura *DealTuple*, es decir, por tiempo. Quizá lo más interesante sea el cálculo de los beneficios/pérdidas de una posición. Para ello, sumamos los valores del campo *profit* de todas las transacciones.

```
      for(int i = 0; i < n; ++i)
      {
         DealTuple deals[];
         filter.let(DEAL_POSITION_ID, positions[i]).select(deals, true);
         const int m = ArraySize(deals);
         if(m == 0)
         {
            Print("Wrong position ID: ", positions[i]);
            break; // invalid id set by user
         }
         double profit = 0; // TODO: need to take into account commissions, swaps and fe
         for(int j = 0; j < m; ++j) profit += deals[j].profit;
         PrintFormat("Position: % 8d %16lld Profit:%f", i + 1, positions[i], (profit));
         ArrayPrint(deals);

         Print("Order details:");
         OrderTuple orders[];
         subfilter.let(ORDER_POSITION_ID, positions[i], IS::OR_EQUAL)
            .let(ORDER_POSITION_BY_ID, positions[i], IS::OR_EQUAL)
            .select(orders);
         ArrayPrint(orders);
      }
   }
```

Este código no analiza las comisiones (DEAL_COMMISSION), los swaps (DEAL_SWAP) ni los honorarios (DEAL_FEE) en las propiedades de las transacciones. En los Asesores Expertos reales, esto probablemente debería hacerse (dependiendo de los requisitos de la estrategia). Veremos otro ejemplo de análisis del historial de trading en la sección sobre probar Asesores Expertos multidivisa y allí tendremos en cuenta este momento.

Puede comparar los resultados del script con la tabla de la pestaña Historial del terminal: su columna Beneficio muestra el beneficio neto de cada posición (los swaps, comisiones y honorarios están en columnas adyacentes, pero deben incluirse).

Es importante tener en cuenta que una orden del tipo ORDER_TYPE_CLOSE_BY se mostrará en ambas posiciones sólo si se selecciona todo el historial en los ajustes. Si se seleccionó una posición específica, el sistema incluirá dicha orden sólo en una de ellas (la que se especificó en la solicitud de operación en primer lugar, en el campo *position*) pero no en la segunda (la que se especificó en *position_by*).

A continuación se muestra un ejemplo del resultado del script para un símbolo con un pequeño historial:

```
Positions total: 3
Position: 1 1253500309 Profit:238.150000
[_1] [deal] [order] [type] [in_out] [volume] [price] [profit]
[0] 2022.02.04 17:34:57 1236049891 1253500309 "BUY" "IN" 1.00000 76.23900 0.00000
[1] 2022.02.14 16:28:41 1242295527 1259788704 "SELL" "OUT" 1.00000 76.42100 238.15000
Order details:
[_1] [setup] [done] [type] [volume] [open] [current] »
» [sl] [tp] [comment]
[0] 1253500309 2022.02.04 17:34:57 2022.02.04 17:34:57 "BUY" 1.00000 76.23900 76.2390
» 0.00 0.00 ""
[1] 1259788704 2022.02.14 16:28:41 2022.02.14 16:28:41 "SELL" 1.00000 76.42100 76.421
» 0.00 0.00 ""
Position: 2 1253526613 Profit:878.030000
[_1] [deal] [order] [type] [in_out] [volume] [price] [profit]
[0] 2022.02.07 10:00:00 1236611994 1253526613 "BUY" "IN" 1.00000 75.75000 0.00000
[1] 2022.02.14 16:28:40 1242295517 1259788693 "SELL" "OUT" 1.00000 76.42100 878.03000
Order details:
[_1] [setup] [done] [type] [volume] [open] [current] »
» [sl] [tp] [comment]
[0] 1253526613 2022.02.04 17:55:18 2022.02.07 10:00:00 "BUY_LIMIT" 1.00000 75.75000 7
» 0.00 0.00 ""
[1] 1259788693 2022.02.14 16:28:40 2022.02.14 16:28:40 "SELL" 1.00000 76.42100 76.421
» 0.00 0.00 ""
Position: 3 1256280710 Profit:4449.040000
[_1] [deal] [order] [type] [in_out] [volume] [price] [profit]
[0] 2022.02.09 13:17:52 1238797056 1256280710 "BUY" "IN" 2.00000 74.72100 0.00000
[1] 2022.02.14 16:28:39 1242295509 1259788685 "SELL" "OUT" 2.00000 76.42100 4449.0400
Order details:
[_1] [setup] [done] [type] [volume] [open] [current] »
» [sl] [tp] [comment]
[0] 1256280710 2022.02.09 13:17:52 2022.02.09 13:17:52 "BUY" 2.00000 74.72100 74.7210
» 0.00 0.00 ""
[1] 1259788685 2022.02.14 16:28:39 2022.02.14 16:28:39 "SELL" 2.00000 76.42100 76.421
» 0.00 0.00 ""
```

En el siguiente fragmento se muestra el caso del aumento de una posición (dos operaciones «IN») y su anulación (una operación «INOUT» de mayor volumen) en una cuenta de compensación.

```
Position: 5 219087383 Profit:0.170000
[_1] [deal] [order] [type] [in_out] [volume] [price] [profit]
[0] 2022.03.29 08:03:33 215612450 219087383 "BUY" "IN" 0.01000 1.10011 0.00000
[1] 2022.03.29 08:04:05 215612451 219087393 "BUY" "IN" 0.01000 1.10009 0.00000
[2] 2022.03.29 08:04:29 215612457 219087400 "SELL" "INOUT" 0.03000 1.10018 0.16000
[3] 2022.03.29 08:04:34 215612460 219087403 "BUY" "OUT" 0.01000 1.10017 0.01000
Order details:
[_1] [setup] [done] [type] [volume] [open] [current] »
» [sl] [tp] [comment]
[0] 219087383 2022.03.29 08:03:33 2022.03.29 08:03:33 "BUY" 0.01000 0.0000 1.10011 »
» 0.00 0.00 ""
[1] 219087393 2022.03.29 08:04:05 2022.03.29 08:04:05 "BUY" 0.01000 0.0000 1.10009 »
» 0.00 0.00 ""
[2] 219087400 2022.03.29 08:04:29 2022.03.29 08:04:29 "SELL" 0.03000 0.0000 1.10018 »
» 0.00 0.00 ""
[3] 219087403 2022.03.29 08:04:34 2022.03.29 08:04:34 "BUY" 0.01000 0.0000 1.10017 »
» 0.00 0.00 ""
```

Consideraremos un historial parcial utilizando el ejemplo de posiciones específicas para el caso de un cierre opuesto en una cuenta de cobertura. En primer lugar, puede ver la primera posición por separado: PositionID=1276109280. Se mostrará en su totalidad independientemente del parámetro de entrada *Type*.

```
Positions total: 1
Position: 1 1276109280 Profit:-0.040000
[_1] [deal] [order] [type] [in_out] [volume] [price] [profit]
[0] 2022.03.07 12:20:53 1258725455 1276109280 "BUY" "IN" 0.01000 1.08344 0.00000
[1] 2022.03.07 12:20:58 1258725503 1276109328 "SELL" "OUT_BY" 0.01000 1.08340 -0.0400
Order details:
[_1] [setup] [done] [type] [volume] [open] [current] »
» [sl] [tp] [comment]
[0] 1276109280 2022.03.07 12:20:53 2022.03.07 12:20:53 "BUY" 0.01000 1.08344 1.08344
» 0.00 0.00 ""
[1] 1276109328 2022.03.07 12:20:58 2022.03.07 12:20:58 "CLOSE_BY" 0.01000 1.08340 1.0
» 0.00 0.00 "#1276109280 by #1276109283"
```

También puede ver la segunda: PositionID=1276109283. Sin embargo, si *Type* es igual a «*position*», para seleccionar un fragmento del historial se utiliza la función *HistorySelectByPosition*, y como resultado sólo habrá una orden de salida (a pesar de que hay dos transacciones).

```
Positions total: 1
Position: 1 1276109283 Profit:0.000000
[_1] [deal] [order] [type] [in_out] [volume] [price] [profit]
[0] 2022.03.07 12:20:53 1258725458 1276109283 "SELL" "IN" 0.01000 1.08340 0.00000
[1] 2022.03.07 12:20:58 1258725504 1276109328 "BUY" "OUT_BY" 0.01000 1.08344 0.00000
Order details:
[_1] [setup] [done] [type] [volume] [open] [current] »
» [sl] [tp] [comment]
[0] 1276109283 2022.03.07 12:20:53 2022.03.07 12:20:53 "SELL" 0.01000 1.08340 1.08340
» 0.00 0.00 ""
```

Si configuramos *Type* como «todo el historial», aparecerá una orden «CLOSE_BY».

```
Positions total: 1
Position: 1 1276109283 Profit:0.000000
[_1] [deal] [order] [type] [in_out] [volume] [price] [profit]
[0] 2022.03.07 12:20:53 1258725458 1276109283 "SELL" "IN" 0.01000 1.08340 0.00000
[1] 2022.03.07 12:20:58 1258725504 1276109328 "BUY" "OUT_BY" 0.01000 1.08344 0.00000
Order details:
[_1] [setup] [done] [type] [volume] [open] [current] »
» [sl] [tp] [comment]
[0] 1276109283 2022.03.07 12:20:53 2022.03.07 12:20:53 "SELL" 0.01000 1.08340 1.08340
» 0.00 0.00 ""
[1] 1276109328 2022.03.07 12:20:58 2022.03.07 12:20:58 "CLOSE_BY" 0.01000 1.08340 1.0
» 0.00 0.00 "#1276109280 by #1276109283"
```

Con esta configuración, el historial se selecciona por completo, pero el filtro sólo deja aquellas órdenes en las que el identificador de la posición especificada se encuentra en las propiedades ORDER_POSITION_ID u ORDER_POSITION_BY_ID. Para componer condiciones con un OR lógico, se ha añadido el elemento IS::OR_EQUAL a la clase *TradeFilter*. Puede estudiarlo de forma adicional.

6.4.33 Tipos de transacciones de trading

Además de realizar operaciones de trading, los programas MQL pueden responder a eventos de trading. Es importante tener en cuenta que tales eventos se producen no sólo como resultado de las acciones de los programas, sino también por otras razones; por ejemplo, con la gestión manual por parte del usuario o la realización de acciones automáticas en el servidor (activación de una orden pendiente, *Stop Loss*, *Take Profit*, *Stop Out*, transferencia de posición a un nuevo día, depósito o retirada de fondos de la cuenta, etc.).

Independientemente del iniciador de las acciones, éstas dan lugar a la ejecución de operaciones de trading en la cuenta. Las transacciones de trading son pasos indivisibles que incluyen:

- Procesamiento de una solicitud de operación
- Modificación de la lista de órdenes activas (incluida la adición de una nueva orden, la ejecución y la eliminación de una orden activada)
- Modificación del historial de órdenes
- Modificación del historial de transacciones
- Modificación de posiciones

Dependiendo de la naturaleza de la operación, algunos pasos pueden ser opcionales. Por ejemplo, si se modifican los niveles de protección de una posición, se perderán tres puntos intermedios. Y cuando se envía una orden de compra, el mercado pasará por un ciclo completo: se procesa la solicitud, se crea la orden correspondiente para la cuenta, se ejecuta la orden, se elimina de la lista activa, se añade al historial de órdenes, a continuación se añade la transacción correspondiente al historial y se crea una nueva posición. Todas estas acciones son transacciones de trading.

Para recibir notificaciones sobre tales eventos, la función especial de manejador *OnTradeTransaction* debe ser descrita en un Asesor Experto o un indicador. Lo veremos en detalle en la próxima sección. Y es que uno de sus parámetros, el primero y más importante, tiene el tipo de una estructura predefinida *MqlTradeTransaction*. Así que vamos a hablar en primer lugar de las transacciones como tales.

```
struct MqlTradeTransaction
{
   ulong                     deal;             // Deal ticket
   ulong                     order;            // Order ticket
   string                    symbol;           // Name of the trading instrument
   ENUM_TRADE_TRANSACTION_TYPE type;           // Trade transaction type
   ENUM_ORDER_TYPE           order_type;       // Order type
   ENUM_ORDER_STATE          order_state;      // Order state
   ENUM_DEAL_TYPE            deal_type;        // Deal type
   ENUM_ORDER_TYPE_TIME      time_type;        // Order type by duration
   datetime                  time_expiration;  // Order expiration date
   double                    price;            // Price
   double                    price_trigger;    // Stop limit order trigger price
   double                    price_sl;         // Stop Loss Level
   double                    price_tp;         // Take Profit Level
   double                    volume;           // Volume in lots
   ulong                     position;         // Position ticket
   ulong                     position_by;      // Opposite position ticket
};
```

En la siguiente tabla se describe cada campo de la estructura:

Campo	Descripción
deal	Ticket de la transacción
order	Ticket de la orden
symbol	El nombre del instrumento de trading en el que se realizó la operación
type	Tipo de transacción de trading ENUM_TRADE_TRANSACTION_TYPE (véase más abajo)
order_type	Tipo de orden ENUM_ORDER_TYPE
order_state	Estado de la orden ENUM_ORDER_STATE
deal_type	Tipo de transacción ENUM_DEAL_TYPE
time_type	Tipo de orden por vencimiento ENUM_ORDER_TYPE_TIME
time_expiration	Fechas de vencimiento de órdenes pendientes
price	El precio de una orden, transacción o posición, según la operación
price_trigger	Precio Stop (precio de activación) de una orden Stop Limit
price_sl	Precio *Stop Loss*; puede referirse a una orden, transacción o posición, dependiendo de la operación.
price_tp	Precio *Take Profit*; puede referirse a una orden, transacción o posición, dependiendo de la operación.
volume	Volumen en lotes; puede indicar el volumen actual de la orden, transacción o posición, dependiendo de la operación.

Campo	Descripción
posición	Ticket de la posición afectada por la transacción
position_by	Ticket de posición opuesta

Algunos campos sólo tienen sentido en determinados casos. En particular, el campo *time_expiration* se rellena para las órdenes con *time_type* igual al tipo de vencimiento ORDER_TIME_SPECIFIED o ORDER_TIME_SPECIFIED_DAY. El campo *price_trigger* está reservado únicamente para las órdenes Stop Limit (ORDER_TYPE_BUY_STOP_LIMIT y ORDER_TYPE_SELL_STOP_LIMIT).

También es obvio que las modificaciones de posición operan sobre el ticket de posición (campo *position*), pero no utilizan tickets de orden o transacción. Además, el campo *position_by* está reservado exclusivamente para cerrar una posición opuesta, es decir, la abierta para el mismo instrumento pero en sentido opuesto.

La característica que define el análisis de una transacción es su tipo (campo *type*). Para describirla, la API de MQL5 introduce una enumeración especial ENUM_TRADE_TRANSACTION_TYPE, que contiene todos los tipos posibles de transacciones.

Identificador	Descripción
TRADE_TRANSACTION_ORDER_ADD	Añadir una nueva orden
TRADE_TRANSACTION_ORDER_UPDATE	Modificar una orden activa
TRADE_TRANSACTION_ORDER_DELETE	Eliminar una orden activa
TRADE_TRANSACTION_DEAL_ADD	Añadir una transacción al historial
TRADE_TRANSACTION_DEAL_UPDATE	Cambiar una transacción en el historial
TRADE_TRANSACTION_DEAL_DELETE	Eliminar una transacción del historial
TRADE_TRANSACTION_HISTORY_ADD	Añadir una orden al historial como resultado de su ejecución o cancelación
TRADE_TRANSACTION_HISTORY_UPDATE	Cambiar un orden en el historial
TRADE_TRANSACTION_HISTORY_DELETE	Eliminar una orden del historial
TRADE_TRANSACTION_POSITION	Cambiar una posición
TRADE_TRANSACTION_REQUEST	Notificar que una solicitud de operación ha sido procesada por el servidor y se ha recibido el resultado de su procesamiento.

He aquí algunas explicaciones:

En una transacción del tipo TRADE_TRANSACTION_ORDER_UPDATE, los cambios en la orden incluyen no sólo cambios explícitos por parte del terminal de cliente o del servidor de trading, sino también cambios en su estado (por ejemplo, transición del estado ORDER_STATE_STARTED a ORDER_STATE_PLACED, o de ORDER_STATE_PLACED a ORDER_STATE_PARTIAL, etc.).

Durante la transacción TRADE_TRANSACTION_ORDER_DELETE se puede eliminar una orden como resultado de la correspondiente solicitud explícita o ejecución (fill) en el servidor. En ambos casos, se transferirá al historial y deberá producirse también la transacción TRADE_TRANSACTION_HISTORY_ADD.

La operación TRADE_TRANSACTION_DEAL_ADD se realiza no sólo como resultado de la ejecución de la orden, sino también como resultado de las transacciones con el saldo de la cuenta.

Algunas operaciones, como TRADE_TRANSACTION_DEAL_UPDATE, TRADE_TRANSACTION_DEAL_DELETE, TRADE_TRANSACTION_HISTORY_DELETE son bastante raras porque describen situaciones en las que una transacción u orden del historial se modifica o elimina en el servidor de forma retroactiva. Esto, por regla general, es consecuencia de la sincronización con un sistema de trading externo (bolsa).

Es importante señalar que añadir o liquidar una posición no implica la aparición de la transacción TRADE_TRANSACTION_POSITION. Este tipo de transacción informa de que la posición ha sido modificada en el servidor de operaciones, de forma programática o manual por parte del usuario. En particular, una posición puede experimentar cambios del volumen (cierre parcial opuesto, inversión), del precio de apertura, así como de los niveles *Stop Loss* y *Take Profit*. Algunas acciones, como las recargas, no activan este evento.

Todas las solicitudes de trading emitidas por los programas MQL se reflejan en las transacciones TRADE_TRANSACTION_REQUEST, lo que permite analizar su ejecución de forma diferida. Esto es especialmente importante cuando se utiliza la función *OrderSendAsync*, que devuelve inmediatamente el control al código de llamada, por lo que no se conoce el resultado. Al mismo tiempo, las transacciones se generan del mismo modo cuando se utiliza la función sincrónica *OrderSend*.

Además, mediante las transacciones TRADE_TRANSACTION_REQUEST, puede analizar las acciones de trading del usuario desde la interfaz del terminal.

6.4.34 Evento OnTradeTransaction

Los Asesores Expertos y los indicadores pueden recibir notificaciones sobre eventos de trading si su código contiene una función de procesamiento especial *OnTradeTransaction*.

```
void OnTradeTransaction(const MqlTradeTransaction &trans,
 const MqlTradeRequest &request, const MqlTradeResult &result)
```

El primer parámetro es la estructura *MqlTradeTransaction* descrita en la sección anterior. El segundo y tercer parámetro son estructuras *MqlTradeRequest* y *MqlTradeResult*, que se han presentado anteriormente en las secciones correspondientes.

La estructura *MqlTradeTransaction* que describe la transacción de trading se rellena de forma diferente en función del tipo de transacción especificado en el campo *type*. Por ejemplo, para las transacciones del tipo TRADE_TRANSACTION_REQUEST, todos los demás campos no son importantes, y para obtener información adicional, es necesario analizar el segundo y tercer parámetro de la función (*request* y *result*). Por el contrario, para todos los demás tipos de transacciones, los dos últimos parámetros de la función deben ignorarse.

En el caso de TRADE_TRANSACTION_REQUEST, el campo *request_id* de la variable *result* contiene un identificador (a través del número de serie), con el que la operación *request* queda registrada en el terminal. Este número no tiene nada que ver con los tickets de orden y transacción, ni con los identificadores de posición. En cada sesión con el terminal, la numeración empieza por el principio (1).

La presencia de un identificador de solicitud permite asociar la acción realizada (llamada a las funciones *OrderSend* o *OrderSendAsync*) con el resultado de esta acción pasado a *OnTradeTransaction*. Veremos ejemplos más adelante.

Para las operaciones de trading relacionadas con órdenes activas (TRADE_TRANSACTION_ORDER_ADD, TRADE_TRANSACTION_ORDER_UPDATE y TRADE_TRANSACTION_ORDER_DELETE) y el historial de órdenes (TRADE_TRANSACTION_HISTORY_ADD, TRADE_TRANSACTION_HISTORY_UPDATE, TRADE_TRANSACTION_HISTORY_DELETE), se rellenan los siguientes campos en la estructura *MqlTradeTransaction*:

- orden: ticket de la orden
- symbol: nombre del instrumento financiero de la orden
- type: tipo de operación de trading
- order_type: tipo de orden
- orders_state: estado actual de la orden
- time_type: tipo de vencimiento de la orden
- time_expiration: hora de vencimiento de la orden (para órdenes con tipos de vencimiento ORDER_TIME_SPECIFIED y ORDER_TIME_SPECIFIED_DAY)
- price: precio de la orden especificada por el cliente/programa
- price_trigger: precio stop para activar una orden Stop Limit (sólo para ORDER_TYPE_BUY_STOP_LIMIT y ORDER_TYPE_SELL_STOP_LIMIT)
- price_sl: *Stop Loss* precio de la orden (cumplimentado si se especifica en la orden)
- price_tp: *Take Profit* precio de la orden (cumplimentado si se especifica en la orden)
- volume: volumen actual de la orden (no ejecutada), el volumen inicial de la orden se puede encontrar en el historial de órdenes
- position: ticket de una posición abierta, modificada o cerrada
- position_by: ticket de posición opuesta (sólo para órdenes que se van a cerrar con posición opuesta)

Para operaciones de trading relacionadas con transacciones (TRADE_TRANSACTION_DEAL_ADD, TRADE_TRANSACTION_DEAL_UPDATE y TRADE_TRANSACTION_DEAL_DELETE), se rellenan los siguientes campos en la estructura *MqlTradeTransaction*:

- deal: ticket de transacción
- order: ticket de orden en base al cual se realizó la transacción
- symbol: nombre del instrumento financiero de la transacción
- type: tipo de operación de trading
- deal_type: tipo de transacción
- precio: precio de transacción
- price_sl: *Stop Loss* precio (rellenado si se especifica en la orden en base a la cual se realizó la transacción)
- price_tp: *Take Profit* precio (rellenado si se especifica en la orden en base a la cual se realizó la transacción)
- volume: volumen de transacción
- position: ticket de una posición abierta, modificada o cerrada

- position_by: ticket de posición opuesta (para cerrar transacciones con posición opuesta)

Para las operaciones de trading relacionadas con cambios de posición (TRADE_TRANSACTION_POSITION), se rellenan los siguientes campos en la estructura *MqlTradeTransaction*:

- symbol: nombre del instrumento financiero de la posición
- type: tipo de operación de trading
- deal_type: tipo de posición (DEAL_TYPE_BUY o DEAL_TYPE_SELL)
- price: precio medio ponderado de apertura de la posición
- price_sl: precio *Stop Loss*
- price_tp: precio *Take Profit*
- volume: volumen de posiciones en lotes
- position: ticket de posición

No toda la información disponible sobre órdenes, transacciones y posiciones (por ejemplo, un comentario) se transmite en la descripción de una operación de trading. Para obtener más información, utilice las funciones correspondientes: *OrderGet*, *HistoryOrderGet*, *HistoryDealGet* y *PositionGet*.

Una solicitud de trading enviada desde el terminal manualmente o a través de las funciones de trading *OrderSend/OrderSendAsync* puede generar varias transacciones de trading consecutivas en el servidor de trading. Al mismo tiempo, el orden en que las notificaciones sobre estas operaciones llegan al terminal no está garantizado, por lo que no puede construir su algoritmo de trading esperando unas operaciones después de otras.

Los eventos de trading se procesan de forma asíncrona, es decir, con retraso (en el tiempo) respecto al momento de generación. Cada evento de trading se envía a la cola del programa MQL, y el programa los recoge secuencialmente en el orden de la cola.

Cuando un Asesor Experto está procesando transacciones de trading dentro del procesador *OnTradeTransaction*, el terminal continúa aceptando transacciones de trading entrantes. Por lo tanto, el estado de la cuenta de trading puede cambiar mientras *OnTradeTransaction* está en funcionamiento. En el futuro, el programa será notificado de todos estos eventos en el orden en que aparezcan.

La longitud de la cola de transacciones es de 1024 elementos. Si *OnTradeTransaction* procesa la siguiente transacción durante demasiado tiempo, las transacciones antiguas de la cola pueden ser desbancadas por las más recientes.

Debido al funcionamiento paralelo multihilo del terminal con objetos de trading, en el momento en que se llama al manejador *OnTradeTransaction*, todas las entidades mencionadas en él, incluyendo órdenes, transacciones y posiciones, pueden estar ya en un estado diferente al especificado en las propiedades de la transacción. Para obtener su estado actual, debe seleccionarlos en el entorno actual o en el historial y solicitar sus propiedades utilizando las funciones MQL5 adecuadas.

Comencemos con un ejemplo sencillo de Asesor Experto *TradeTransactions.mq5*, que registra todos los eventos de trading de *OnTradeTransaction*. Su único parámetro *DetailedLog* permite utilizar opcionalmente las clases *OrderMonitor*, *DealMonitor*, *PositionMonitor* para mostrar todas las propiedades. Por defecto, el Asesor Experto muestra sólo el contenido de los campos rellenados de las estructuras *MqlTradeTransaction*, *MqlTradeRequest* y *MqlTradeResult*, que llegan al manejador en forma de parámetros; al mismo tiempo, *request* y *result* se procesan sólo para las transacciones TRADE_TRANSACTION_REQUEST.

```
input bool DetailedLog = false; // DetailedLog ('true' shows order/deal/position deta

void OnTradeTransaction(const MqlTradeTransaction &transaction,
   const MqlTradeRequest &request,
   const MqlTradeResult &result)
{
   static ulong count = 0;
   PrintFormat(">>>% 6d", ++count);
   Print(TU::StringOf(transaction));

   if(transaction.type == TRADE_TRANSACTION_REQUEST)
   {
      Print(TU::StringOf(request));
      Print(TU::StringOf(result));
   }

   if(DetailedLog)
   {
      if(transaction.order != 0)
      {
         OrderMonitor m(transaction.order);
         m.print();
      }
      if(transaction.deal != 0)
      {
         DealMonitor m(transaction.deal);
         m.print();
      }
      if(transaction.position != 0)
      {
         PositionMonitor m(transaction.position);
         m.print();
      }
   }
}
```

Vamos a ejecutarlo en el gráfico EURUSD y a realizar varias acciones manualmente, y las entradas correspondientes aparecerán en el registro (por la pureza del experimento, se supone que nadie ni nada más realiza operaciones en la cuenta de trading; en concreto, que no hay otros Asesores Expertos en ejecución).

Vamos a abrir una posición larga con un lote mínimo.

```
>>> 1
TRADE_TRANSACTION_ORDER_ADD, #=1296991463(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD
» @ 1.10947, V=0.01
>>> 2
TRADE_TRANSACTION_DEAL_ADD, D=1279627746(DEAL_TYPE_BUY), »
» #=1296991463(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10947, V=0.01, P=1296
>>> 3
TRADE_TRANSACTION_ORDER_DELETE, #=1296991463(ORDER_TYPE_BUY/ORDER_STATE_FILLED), EURU
» @ 1.10947, P=1296991463
>>> 4
TRADE_TRANSACTION_HISTORY_ADD, #=1296991463(ORDER_TYPE_BUY/ORDER_STATE_FILLED), EURUS
» @ 1.10947, P=1296991463
>>> 5
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 1.10947, #=12
DONE, D=1279627746, #=1296991463, V=0.01, @ 1.10947, Bid=1.10947, Ask=1.10947, Req=7
```

Venderemos el doble del lote mínimo.

```
>>> 6
TRADE_TRANSACTION_ORDER_ADD, #=1296992157(ORDER_TYPE_SELL/ORDER_STATE_STARTED), EURUS
» @ 1.10964, V=0.02
>>> 7
TRADE_TRANSACTION_DEAL_ADD, D=1279628463(DEAL_TYPE_SELL), »
» #=1296992157(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10964, V=0.02, P=1296
>>> 8
TRADE_TRANSACTION_ORDER_DELETE, #=1296992157(ORDER_TYPE_SELL/ORDER_STATE_FILLED), EUR
» @ 1.10964, P=1296992157
>>> 9
TRADE_TRANSACTION_HISTORY_ADD, #=1296992157(ORDER_TYPE_SELL/ORDER_STATE_FILLED), EURU
» @ 1.10964, P=1296992157
>>> 10
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_SELL, V=0.02, ORDER_FILLING_FOK, @ 1.10964, #=1
DONE, D=1279628463, #=1296992157, V=0.02, @ 1.10964, Bid=1.10964, Ask=1.10964, Req=8
```

Vamos a realizar la operación de cierre del contador.

```
>>> 11
TRADE_TRANSACTION_ORDER_ADD, #=1296992548(ORDER_TYPE_CLOSE_BY/ORDER_STATE_STARTED), E
» @ 1.10964, V=0.01, P=1296991463, b=1296992157
>>> 12
TRADE_TRANSACTION_DEAL_ADD, D=1279628878(DEAL_TYPE_SELL), »
» #=1296992548(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10964, V=0.01, P=1296
>>> 13
TRADE_TRANSACTION_POSITION, EURUSD, @ 1.10947, P=1296991463
>>> 14
TRADE_TRANSACTION_DEAL_ADD, D=1279628879(DEAL_TYPE_BUY), »
» #=1296992548(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10947, V=0.01, P=1296
>>> 15
TRADE_TRANSACTION_ORDER_DELETE, #=1296992548(ORDER_TYPE_CLOSE_BY/ORDER_STATE_FILLED),
» @ 1.10964, P=1296991463, b=1296992157
>>> 16
TRADE_TRANSACTION_HISTORY_ADD, #=1296992548(ORDER_TYPE_CLOSE_BY/ORDER_STATE_FILLED),
» @ 1.10964, P=1296991463, b=1296992157
>>> 17
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_CLOSE_BY, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, #=129699254
» P=1296991463, b=1296992157
DONE, D=1279628878, #=1296992548, V=0.01, @ 1.10964, Bid=1.10961, Ask=1.10965, Req=9
```

Seguimos teniendo una posición corta del lote mínimo. Vamos a cerrarla.

```
>>> 18
TRADE_TRANSACTION_ORDER_ADD, #=1297002683(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD
» @ 1.10964, V=0.01, P=1296992157
>>> 19
TRADE_TRANSACTION_ORDER_DELETE, #=1297002683(ORDER_TYPE_BUY/ORDER_STATE_FILLED), EURU
» @ 1.10964, P=1296992157
>>> 20
TRADE_TRANSACTION_HISTORY_ADD, #=1297002683(ORDER_TYPE_BUY/ORDER_STATE_FILLED), EURUS
» @ 1.10964, P=1296992157
>>> 21
TRADE_TRANSACTION_DEAL_ADD, D=1279639132(DEAL_TYPE_BUY), »
» #=1297002683(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10964, V=0.01, P=1296
>>> 22
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 1.10964, #=12
» P=1296992157
DONE, D=1279639132, #=1297002683, V=0.01, @ 1.10964, Bid=1.10964, Ask=1.10964, Req=10
```

Si lo desea, puede activar la opción *DetailedLog* para registrar todas las propiedades de los objetos de trading en el momento de procesar el evento. En un registro detallado se pueden observar discrepancias entre el estado de los objetos almacenados en la estructura de la transacción (en el momento de su inicio) y el estado actual. Por ejemplo, al añadir una orden para cerrar una posición (opuesta o normal), se especifica un ticket en la transacción, según el cual el objeto monitor ya no podrá leer nada, puesto que la posición se ha borrado. Como resultado, veremos líneas como ésta en el registro:

```
TRADE_TRANSACTION_ORDER_ADD, #=1297777749(ORDER_TYPE_CLOSE_BY/ORDER_STATE_STARTED), E
» @ 1.10953, V=0.01, P=1297774881, b=1297776850
...
Error: PositionSelectByTicket(1297774881) failed: TRADE_POSITION_NOT_FOUND
```

Vamos a reiniciar el Asesor Experto *TradeTransaction.mq5* para restablecer los eventos registrados para la próxima prueba. Esta vez utilizaremos la configuración por defecto (sin detalles).

Ahora vamos a intentar realizar acciones de trading programáticamente en el nuevo Asesor Experto *OrderSendTransaction1.mq5*, y al mismo tiempo describir nuestro manejador *OnTradeTransaction* en él (igual que en el ejemplo anterior).

Este Asesor Experto le permite seleccionar el volumen y la dirección de la operación: si lo deja a cero, se utiliza por defecto el lote mínimo del símbolo actual. También en los parámetros hay una distancia a los niveles de protección en puntos. Se entra en el mercado con los parámetros especificados, hay una pausa de 5 segundos entre el ajuste de *Stop Loss* y *Take Profit*, y a continuación se cierra la posición, de manera que el usuario pueda intervenir (por ejemplo, editar Stop Loss manualmente), aunque esto no es necesario, puesto que ya nos hemos asegurado de que las operaciones manuales sean interceptadas por el programa.

```
enum ENUM_ORDER_TYPE_MARKET
{
   MARKET_BUY = ORDER_TYPE_BUY,     // ORDER_TYPE_BUY
   MARKET_SELL = ORDER_TYPE_SELL    // ORDER_TYPE_SELL
};

input ENUM_ORDER_TYPE_MARKET Type;
input double Volume;                // Volume (0 - minimal lot)
input uint Distance2SLTP = 1000;
```

La estrategia se lanza una vez, para lo cual se utiliza un temporizador de 1 segundo, que se desactiva en su propio manejador.

```
int OnInit()
{
   EventSetTimer(1);
   return INIT_SUCCEEDED;
}

void OnTimer()
{
   EventKillTimer();
   ...
```

Todas las acciones se realizan a través de una estructura *MqlTradeRequestSync* ya familiar con características avanzadas (*MqlTradeSync.mqh*): inicialización implícita de campos con valores correctos, métodos *buy/sell* para órdenes de mercado, *adjust* para niveles de protección y *close* para cerrar la posición.

Paso 1:

```
   MqlTradeRequestSync request;

   const double volume = Volume == 0 ?
      SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN) : Volume;

   Print("Start trade");
   const ulong order = (Type == MARKET_BUY ? request.buy(volume) : request.sell(volum
   if(order == 0 || !request.completed())
   {
      Print("Failed Open");
      return;
   }

   Print("OK Open");
```

Paso 2:

```
   Sleep(5000); // wait 5 seconds (user can edit position)
   Print("SL/TP modification");
   const double price = PositionGetDouble(POSITION_PRICE_OPEN);
   const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);
   TU::TradeDirection dir((ENUM_ORDER_TYPE)Type);
   const double SL = dir.negative(price, Distance2SLTP * point);
   const double TP = dir.positive(price, Distance2SLTP * point);
   if(request.adjust(SL, TP) && request.completed())
   {
      Print("OK Adjust");
   }
   else
   {
      Print("Failed Adjust");
   }
```

Paso 3:

```
   Sleep(5000); // wait another 5 seconds
   Print("Close down");
   if(request.close(request.result.position) && request.completed())
   {
      Print("Finish");
   }
   else
   {
      Print("Failed Close");
   }
}
```

Las esperas intermedias no sólo permiten tener tiempo para considerar el proceso, sino que también demuestran un aspecto importante de la programación MQL5, que es el monohilo. Mientras nuestro Asesor Experto de trading se encuentra dentro de *OnTimer*, los eventos de trading generados por el terminal se acumulan en su cola y serán reenviados al manejador interno de *OnTradeTransaction* en un estilo diferido, sólo después de la salida de *OnTimer*.

Al mismo tiempo, el Asesor Experto de *TradeTransactions* que se ejecuta en paralelo no está ocupado con ningún cálculo y recibirá los eventos de trading lo más rápido posible.

El resultado de la ejecución de dos Asesores Expertos se presenta en el siguiente registro con temporización (por brevedad, *OrderSendTransaction1* se etiqueta como *OS1*, y *Trade Transactions* como *TTs*).

```
19:09:08.078 OS1 Start trade
19:09:08.109 TTs >>> 1
19:09:08.125 TTs TRADE_TRANSACTION_ORDER_ADD, #=1298021794(ORDER_TYPE_BUY/ORDER_STATE
EURUSD, @ 1.10913, V=0.01
19:09:08.125 TTs >>> 2
19:09:08.125 TTs TRADE_TRANSACTION_DEAL_ADD, D=1280661362(DEAL_TYPE_BUY), »
#=1298021794(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10913, V=0.01, »
P=1298021794
19:09:08.125 TTs >>> 3
19:09:08.125 TTs TRADE_TRANSACTION_ORDER_DELETE, #=1298021794(ORDER_TYPE_BUY/ORDER_ST
EURUSD, @ 1.10913, P=1298021794
19:09:08.125 TTs >>> 4
19:09:08.125 TTs TRADE_TRANSACTION_HISTORY_ADD, #=1298021794(ORDER_TYPE_BUY/ORDER_STA
EURUSD, @ 1.10913, P=1298021794
19:09:08.125 TTs >>> 5
19:09:08.125 TTs TRADE_TRANSACTION_REQUEST
19:09:08.125 TTs TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK
D=10, #=1298021794, M=1234567890
19:09:08.125 TTs DONE, D=1280661362, #=1298021794, V=0.01, @ 1.10913, Bid=1.10913, As
Req=9
19:09:08.125 OS1 Waiting for position for deal D=1280661362
19:09:08.125 OS1 OK Open
19:09:13.133 OS1 SL/TP modification
19:09:13.164 TTs >>> 6
19:09:13.164 TTs TRADE_TRANSACTION_POSITION, EURUSD, @ 1.10913, SL=1.09913, TP=1.1191
P=1298021794
19:09:13.164 OS1 OK Adjust
19:09:13.164 TTs >>> 7
19:09:13.164 TTs TRADE_TRANSACTION_REQUEST
19:09:13.164 TTs TRADE_ACTION_SLTP, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK
TP=1.11913, D=10, P=1298021794, M=1234567890
19:09:13.164 TTs DONE, Req=10
19:09:18.171 OS1 Close down
19:09:18.187 OS1 Finish
19:09:18.218 TTs >>> 8
19:09:18.218 TTs TRADE_TRANSACTION_ORDER_ADD, #=1298022443(ORDER_TYPE_SELL/ORDER_STAT
EURUSD, @ 1.10901, V=0.01, P=1298021794
19:09:18.218 TTs >>> 9
19:09:18.218 TTs TRADE_TRANSACTION_DEAL_ADD, D=1280661967(DEAL_TYPE_SELL), »
#=1298022443(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10901, »
SL=1.09913, TP=1.11913, V=0.01, P=1298021794
19:09:18.218 TTs >>> 10
19:09:18.218 TTs TRADE_TRANSACTION_ORDER_DELETE, #=1298022443(ORDER_TYPE_SELL/ORDER_S
EURUSD, @ 1.10901, P=1298021794
19:09:18.218 TTs >>> 11
19:09:18.218 TTs TRADE_TRANSACTION_HISTORY_ADD, #=1298022443(ORDER_TYPE_SELL/ORDER_ST
EURUSD, @ 1.10901, P=1298021794
19:09:18.218 TTs >>> 12
19:09:18.218 TTs TRADE_TRANSACTION_REQUEST
19:09:18.218 TTs TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_SELL, V=0.01, ORDER_FILLING_FO
D=10, #=1298022443, P=1298021794, M=1234567890
19:09:18.218 TTs DONE, D=1280661967, #=1298022443, V=0.01, @ 1.10901, Bid=1.10901, As
Req=11
19:09:18.218 OS1 >>> 1
```

```
19:09:18.218 OS1 TRADE_TRANSACTION_ORDER_ADD, #=1298021794(ORDER_TYPE_BUY/ORDER_STATE
EURUSD, @ 1.10913, V=0.01
19:09:18.218 OS1 >>> 2
19:09:18.218 OS1 TRADE_TRANSACTION_DEAL_ADD, D=1280661362(DEAL_TYPE_BUY), »
#=1298021794(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, »
@ 1.10913, V=0.01, P=1298021794
19:09:18.218 OS1 >>> 3
19:09:18.218 OS1 TRADE_TRANSACTION_ORDER_DELETE, #=1298021794(ORDER_TYPE_BUY/ORDER_ST
EURUSD, @ 1.10913, P=1298021794
19:09:18.218 OS1 >>> 4
19:09:18.218 OS1 TRADE_TRANSACTION_HISTORY_ADD, #=1298021794(ORDER_TYPE_BUY/ORDER_STA
EURUSD, @ 1.10913, P=1298021794
19:09:18.218 OS1 >>> 5
19:09:18.218 OS1 TRADE_TRANSACTION_REQUEST
19:09:18.218 OS1 TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK
D=10, #=1298021794, M=1234567890
19:09:18.218 OS1 DONE, D=1280661362, #=1298021794, V=0.01, @ 1.10913, Bid=1.10913, As
Req=9
19:09:18.218 OS1 >>> 6
19:09:18.218 OS1 TRADE_TRANSACTION_POSITION, EURUSD, @ 1.10913, SL=1.09913, TP=1.1191
P=1298021794
19:09:18.218 OS1 >>> 7
19:09:18.218 OS1 TRADE_TRANSACTION_REQUEST
19:09:18.218 OS1 TRADE_ACTION_SLTP, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK
SL=1.09913, TP=1.11913, D=10, P=1298021794, M=1234567890
19:09:18.218 OS1 DONE, Req=10
19:09:18.218 OS1 >>> 8
19:09:18.218 OS1 TRADE_TRANSACTION_ORDER_ADD, #=1298022443(ORDER_TYPE_SELL/ORDER_STAT
EURUSD, @ 1.10901, V=0.01, P=1298021794
19:09:18.218 OS1 >>> 9
19:09:18.218 OS1 TRADE_TRANSACTION_DEAL_ADD, D=1280661967(DEAL_TYPE_SELL), »
#=1298022443(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10901, »
SL=1.09913, TP=1.11913, V=0.01, P=1298021794
19:09:18.218 OS1 >>> 10
19:09:18.218 OS1 TRADE_TRANSACTION_ORDER_DELETE, #=1298022443(ORDER_TYPE_SELL/ORDER_S
EURUSD, @ 1.10901, P=1298021794
19:09:18.218 OS1 >>> 11
19:09:18.218 OS1 TRADE_TRANSACTION_HISTORY_ADD, #=1298022443(ORDER_TYPE_SELL/ORDER_ST
EURUSD, @ 1.10901, P=1298021794
19:09:18.218 OS1 >>> 12
19:09:18.218 OS1 TRADE_TRANSACTION_REQUEST
19:09:18.218 OS1 TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_SELL, V=0.01, ORDER_FILLING_FO
D=10, #=1298022443, P=1298021794, M=1234567890
19:09:18.218 OS1 DONE, D=1280661967, #=1298022443, V=0.01, @ 1.10901, Bid=1.10901, As
Req=11
```

La numeración de los eventos en los programas es la misma (siempre que se inicien limpiamente, según lo recomendado). Observe que el mismo evento se imprime primero desde *TTs* inmediatamente después de que se ejecute la solicitud, y la segunda vez sólo al final de la prueba, donde, de hecho, todos los eventos se emiten desde la cola a *OS1*.

Si eliminamos los retrasos artificiales, el script, por supuesto, se ejecutará más rápido, pero aún así el manejador *OnTradeTransaction* recibirá notificaciones (múltiples veces) después de los tres pasos, no después de cada solicitud respectiva. ¿Hasta qué punto esto es crítico?

Ahora los ejemplos utilizan nuestra modificación de la estructura *MqlTradeRequestSync*, utilizando a propósito la opción sincrónica *OrderSend*, que también implementa un método universal *completed* que comprueba si la solicitud se ha completado con éxito. Con este control, podemos establecer niveles de protección para una posición, ya que sabemos cómo esperar a que aparezca su ticket. En el marco de este concepto síncrono (adoptado por comodidad), no necesitamos analizar los resultados de las consultas en *OnTradeTransaction*, pero esto no siempre es así.

Cuando un Asesor Experto necesita enviar muchas solicitudes a la vez, como en el caso del ejemplo con el establecimiento de una cuadrícula de órdenes *PendingOrderGrid2.mq5* abordado en la sección sobre propiedades de posiciones, esperar a que cada posición u orden esté «lista» puede reducir el rendimiento global del Asesor Experto. En tales casos se recomienda utilizar la función *OrderSendAsync*. Pero si tiene éxito, sólo rellena el campo *request_id* en *MqlTradeResult*, con el que luego hay que hacer un seguimiento de la aparición de órdenes, transacciones y posiciones en *OnTradeTransaction*.

Uno de los trucos más obvios pero no especialmente elegantes para implementar este esquema es almacenar los identificadores de las solicitudes o estructuras enteras de las solicitudes que se envían en un array, en el contexto global. Estos identificadores pueden buscarse seguidamente en las transacciones entrantes en *OnTradeTransaction*, los tickets pueden encontrarse en el parámetro *MqlTradeResult* y pueden adoptarse además otras medidas. Como resultado, la lógica de trading se separa en diferentes funciones. Por ejemplo, en el contexto del último Asesor Experto *OrderSendTransaction1.mq5*, esta "diversificación" radica en el hecho de que después de enviar la primera orden, los fragmentos de código deben ser transferidos a *OnTradeTransaction* y comprobados para lo siguiente:

- tipo de transacción en *MqlTradeTransaction* (*transaction type*);
- tipo de solicitud en *MqlTradeRequest* (*request action*);
- id de solicitud en *MqlTradeResult* (*result.request_id*);

Todo ello debe completarse con una lógica aplicada específica (por ejemplo, la comprobación de la existencia de una posición), que proporciona la ramificación por estados de la estrategia de trading. Un poco más adelante haremos una modificación similar del Asesor Experto *OrderSendTransaction* bajo un número diferente para mostrar visualmente la cantidad de código fuente adicional. Y luego ofreceremos una forma de organizar el programa de forma más lineal, pero sin abandonar los eventos transaccionales.

Por ahora señalamos únicamente que el desarrollador debe elegir si construye un algoritmo en torno a *OnTradeTransaction* o sin él. En muchos casos, cuando no es necesario el envío masivo de órdenes, es posible permanecer en el paradigma de programación síncrona. Sin embargo, *OnTradeTransaction* es la forma más práctica de controlar la activación de órdenes pendientes y niveles de protección, así como otros eventos generados por el servidor. Tras una pequeña preparación presentaremos dos ejemplos relevantes: la modificación final del Asesor Experto de cuadrícula y la implementación de la popular configuración de dos órdenes OCO (One Cancels Other) (véase la sección *On Trade*).

Una alternativa a la aplicación de *OnTradeTransaction* consiste en el análisis periódico del entorno de trading, es decir, en recordar el número de órdenes y posiciones y buscar cambios entre ellas. Este enfoque es adecuado para estrategias basadas en calendarios o que permiten ciertos retrasos.

Insistimos una vez más en que el uso de *OnTradeTransaction* no significa que el programa deba pasar necesariamente de *OrderSend* a *OrderSendAsync*: Puede utilizar cualquiera de las dos variedades o ambas. Recordemos que la función *OrderSend* tampoco es del todo síncrona, ya que devuelve, en el mejor de los casos, el ticket de la orden y la transacción, pero no la posición. Pronto podremos medir el tiempo de ejecución de un lote de órdenes dentro de la misma estrategia de cuadrícula utilizando ambas variantes de la función: *OrderSend* y *OrderSendAsync*.

Para unificar el desarrollo de programas síncronos y asíncronos sería genial admitir *OrderSendAsync* en nuestra estructura *MqlTradeRequestSync* (a pesar de su nombre), lo cual puede hacerse con sólo un par de correcciones. En primer lugar, debe sustituir todas las llamadas *OrderSend* existentes por su propio método *orderSend*, y en él cambiar la llamada a *OrderSend* o *OrderSendAsync* en función de una bandera.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   ...
   static bool AsyncEnabled;
   ...
private:
   bool orderSend(const MqlTradeRequest &req, MqlTradeResult &res)
   {
      return AsyncEnabled ? ::OrderSendAsync(req, res) : ::OrderSend(req, res);
   }
};
```

Estableciendo la variable pública *AsyncEnabled* en *true* o *false*, puede cambiar de un modo a otro; por ejemplo, en el fragmento de código en el que se envían las órdenes en masa.

En segundo lugar, aquellos métodos de la estructura que devuelvan un ticket (por ejemplo, para entrar en el mercado) deberán devolver el campo *request_id* en lugar de *order*. Por ejemplo, dentro de los métodos *_pending* y *_market* teníamos el siguiente operador:

```
if(OrderSend(this, result)) return result.order;
```

Ahora se sustituye por:

```
if(orderSend(this, result)) return result.order ? result.order :
   (result.retcode == TRADE_RETCODE_PLACED ? result.request_id : 0);
```

Por supuesto, cuando el modo asíncrono está activado, ya no podemos utilizar el método *completed* para esperar a que los resultados de la consulta estén listos inmediatamente después de su envío. Pero este método es, básicamente, opcional: puede abandonarlo incluso cuando trabaje a través de *OrderSend*.

Así pues, teniendo en cuenta la nueva modificación del archivo *MqlTradeSync.mqh*, vamos a crear *OrderSendTransaction2.mq5*.

Este Asesor Experto enviará la solicitud inicial como antes desde *OnTimer*, mientras establece niveles de protección y cierra una posición en *OnTradeTransaction* paso a paso. Aunque esta vez no tendremos un retraso artificial entre las etapas, la secuencia de estados en sí es estándar para muchos Asesores Expertos: se abrió una posición, se modificó, se cerró (si se cumplen ciertas condiciones de mercado, que aquí se dejan entre bastidores).

Dos variables globales le permitirán realizar un seguimiento del estado: *RequestID* con el id de la última solicitud enviada (cuyo resultado esperamos) y *Position Ticket* con un ticket de posición abierta. Cuando allí la posición no ha aparecido todavía, o ya no existe, la entrada es igual a 0.

```
uint RequestID = 0;
ulong PositionTicket = 0;
```

El modo asíncrono está activado en el manejador *OnInit*.

```
int OnInit()
{
   ...
   MqlTradeRequestSync::AsyncEnabled = true;
   ...
}
```

La función *OnTimer* es ahora mucho más corta.

```
void OnTimer()
{
   ...
   // send a request TRADE_ACTION_DEAL (asynchronously!)
   const ulong order = (Type == MARKET_BUY ? request.buy(volume) : request.sell(volum
   if(order) // in asynchronous mode this is now request_id
   {
      Print("OK Open?");
      RequestID = request.result.request_id; // same as order
   }
   else
   {
      Print("Failed Open");
   }
}
```

Una vez completada con éxito la solicitud, sólo obtenemos *request_id* y lo almacenamos en la variable *RequestID*. La impresión de estado contiene ahora un signo de interrogación, como «¿OK Open?», porque aún no se conoce el resultado real.

OnTradeTransaction se complicó considerablemente debido a la verificación de los resultados y la ejecución de las órdenes de trading posteriores según las condiciones. Vamos a verlo poco a poco.

En este caso, toda la lógica de trading se ha trasladado a la rama de operaciones de tipo TRADE_TRANSACTION_REQUEST. Por supuesto, el desarrollador puede utilizar otros tipos si lo desea, pero utilizamos este porque contiene información en forma de una estructura *MqlTradeResult* familiar, es decir, este tipo representa una finalización retrasada de una llamada asíncrona *OrderSendAsync*.

```
void OnTradeTransaction(const MqlTradeTransaction &transaction,
   const MqlTradeRequest &request,
   const MqlTradeResult &result)
{
   static ulong count = 0;
   PrintFormat(">>>% 6d", ++count);
   Print(TU::StringOf(transaction));

   if(transaction.type == TRADE_TRANSACTION_REQUEST)
   {
      Print(TU::StringOf(request));
      Print(TU::StringOf(result));

      ...
      // here is the whole algorithm
   }
}
```

Sólo deberían interesarnos las solicitudes con el ID que esperamos. Así que la siguiente sentencia será *if* anidada. En su bloque, describimos el objeto *MqlTradeRequestSync* con antelación, ya que será necesario enviar solicitudes de trading periódicas de acuerdo con el plan.

```
      if(result.request_id == RequestID)
      {
         MqlTradeRequestSync next;
         next.magic = Magic;
         next.deviation = Deviation;
         ...
      }
```

Sólo tenemos dos tipos de solicitud en funcionamiento, así que añadimos para ellos una *if* anidada más.

```
         if(request.action == TRADE_ACTION_DEAL)
         {
            ... // here is the reaction to opening and closing a position
         }
         else if(request.action == TRADE_ACTION_SLTP)
         {
            ... // here is the reaction to setting SLTP for an open position
         }
```

Hay que tener en cuenta que TRADE_ACTION_DEAL se utiliza tanto para abrir como para cerrar una posición, por lo que se requiere un *if* más, en el que distinguiremos entre estos dos estados dependiendo del valor de la variable *PositionTicket*.

```
if(PositionTicket == 0)
{
   ... // there is no position, so this is an opening notification
}
else
{
   ... // there is a position, so this is a closure
}
```

En la estrategia de trading analizada no hay incrementos de posición (para la compensación) ni posiciones múltiples (para la cobertura), por lo que esta parte es sencilla desde el punto de vista lógico. Los Asesores Expertos reales requerirán estimaciones mucho más diferentes de los estados intermedios.

En el caso de una notificación de apertura de posición, el bloque de código tiene el siguiente aspecto:

```
if(PositionTicket == 0)
{
   // trying to get results from the transaction: select an order by tick
   if(!HistoryOrderSelect(result.order))
   {
      Print("Can't select order in history");
      RequestID = 0;
      return;
   }
   // get position ID and ticket
   const ulong posid = HistoryOrderGetInteger(result.order, ORDER_POSITIC
   PositionTicket = TU::PositionSelectById(posid);
   ...
```

Para simplificar, hemos omitido aquí la comprobación de errores y recotizaciones. Puede ver un ejemplo de su manejo en el código fuente adjunto. Recordemos que todas estas comprobaciones ya se han implementado en los métodos de la estructura *MqlTradeRequestSync*, pero sólo funcionan en modo síncrono, por lo que tenemos que repetirlas explícitamente.

El siguiente fragmento de código para establecer los niveles de protección no ha cambiado mucho.

```
if(PositionTicket == 0)
{
   ...
   const double price = PositionGetDouble(POSITION_PRICE_OPEN);
   const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);
   TU::TradeDirection dir((ENUM_ORDER_TYPE)Type);
   const double SL = dir.negative(price, Distance2SLTP * point);
   const double TP = dir.positive(price, Distance2SLTP * point);
   // sending TRADE_ACTION_SLTP request (asynchronously!)
   if(next.adjust(PositionTicket, SL, TP))
   {
      Print("OK Adjust?");
      RequestID = next.result.request_id;
   }
   else
   {
      Print("Failed Adjust");
      RequestID = 0;
   }
}
```

La única diferencia aquí es: rellenamos la variable *RequestID* con el ID de la nueva solicitud TRADE_ACTION_SLTP.

Recibir una notificación sobre una transacción con un *PositionTicket* distinto de cero implica que la posición se ha cerrado.

```
if(PositionTicket == 0)
{
   ... // see above
}
else
{
   if(!PositionSelectByTicket(PositionTicket))
   {
      Print("Finish");
      RequestID = 0;
      PositionTicket = 0;
   }
}
```

En caso de borrado con éxito, la posición no puede seleccionarse utilizando *PositionSelectByTicket*, por lo que reiniciamos *RequestID* y *PositionTicket*. A continuación, el Asesor Experto vuelve a su estado inicial y está listo para realizar el siguiente ciclo de compra/venta-modificación-cierre.

Nos queda considerar el envío de una solicitud de cierre de la posición. En nuestra estrategia simplificada al mínimo, esto ocurre inmediatamente después de modificar con éxito los niveles de protección.

```
if(request.action == TRADE_ACTION_DEAL)
{
   ... // see above
}
else if(request.action == TRADE_ACTION_SLTP)
{
   // send a TRADE_ACTION_DEAL request to close (asynchronously!)
   if(next.close(PositionTicket))
   {
      Print("OK Close?");
      RequestID = next.result.request_id;
   }
   else
   {
      PrintFormat("Failed Close %lld", PositionTicket);
   }
}
```

Esa es toda la función *OnTradeTransaction*. El Asesor Experto está listo.

Vamos a ejecutar *OrderSendTransaction2.mq5* con la configuración por defecto. He aquí un registro de ejemplo:

```
Start trade
OK Open?
>>> 1
TRADE_TRANSACTION_ORDER_ADD, #=1299508203(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD
» @ 1.10640, V=0.01
>>> 2
TRADE_TRANSACTION_DEAL_ADD, D=1282135720(DEAL_TYPE_BUY), »
» #=1299508203(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10640, V=0.01, P=1299
>>> 3
TRADE_TRANSACTION_ORDER_DELETE, #=1299508203(ORDER_TYPE_BUY/ORDER_STATE_FILLED), EURU
» @ 1.10640, P=1299508203
>>> 4
TRADE_TRANSACTION_HISTORY_ADD, #=1299508203(ORDER_TYPE_BUY/ORDER_STATE_FILLED), EURUS
» @ 1.10640, P=1299508203
>>> 5
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 1.10640, D=10
» #=1299508203, M=1234567890
DONE, D=1282135720, #=1299508203, V=0.01, @ 1.1064, Bid=1.1064, Ask=1.1064, Req=7
OK Adjust?
>>> 6
TRADE_TRANSACTION_POSITION, EURUSD, @ 1.10640, SL=1.09640, TP=1.11640, V=0.01, P=1299
>>> 7
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_SLTP, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, SL=1.09640, TP=
» D=10, P=1299508203, M=1234567890
DONE, Req=8
OK Close?
>>> 8
TRADE_TRANSACTION_ORDER_ADD, #=1299508215(ORDER_TYPE_SELL/ORDER_STATE_STARTED), EURUS
» @ 1.10638, V=0.01, P=1299508203
>>> 9
TRADE_TRANSACTION_ORDER_DELETE, #=1299508215(ORDER_TYPE_SELL/ORDER_STATE_FILLED), EUR
» @ 1.10638, P=1299508203
>>> 10
TRADE_TRANSACTION_HISTORY_ADD, #=1299508215(ORDER_TYPE_SELL/ORDER_STATE_FILLED), EURU
» @ 1.10638, P=1299508203
>>> 11
TRADE_TRANSACTION_DEAL_ADD, D=1282135730(DEAL_TYPE_SELL), »
» #=1299508215(ORDER_TYPE_BUY/ORDER_STATE_STARTED), EURUSD, @ 1.10638, »
» SL=1.09640, TP=1.11640, V=0.01, P=1299508203
>>> 12
TRADE_TRANSACTION_REQUEST
TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_SELL, V=0.01, ORDER_FILLING_FOK, @ 1.10638, D=1
» #=1299508215, P=1299508203, M=1234567890
DONE, D=1282135730, #=1299508215, V=0.01, @ 1.10638, Bid=1.10638, Ask=1.10638, Req=9
Finish
```

La lógica de trading está funcionando como se esperaba, y los eventos de transacción llegan estrictamente después de que se envíe cada orden siguiente. Si ahora ejecutamos nuestro nuevo Asesor Experto y el interceptor de transacciones *TradeTransactions.mq5* en paralelo, los mensajes de registro de dos Asesores Expertos aparecerán de forma sincronizada.

Sin embargo, pasar de la primera versión *OrderSendTransaction1.mq5* directa a una segunda versión *OrderSendTransaction2.mq5* asíncrona requería un código bastante más sofisticado. La pregunta que se plantea es: ¿es posible combinar de algún modo los principios de descripción secuencial de la lógica de trading (transparencia del código) y procesamiento paralelo (velocidad)?

En teoría, esto es posible, pero requerirá en algún momento dedicar tiempo a trabajar en la creación de algún tipo de mecanismo auxiliar.

6.4.35 Solicitudes síncronas y asíncronas

Antes de entrar en detalles, recordemos que cada programa MQL se ejecuta en su propio hilo, y por lo tanto el procesamiento paralelo asíncrono de transacciones (y otros eventos) sólo es posible debido a que otro programa MQL lo estaría haciendo. Al mismo tiempo, es necesario garantizar el intercambio de información entre programas. Ya conocemos un par de formas de hacerlo: variables globales del terminal y archivos. En la Parte 7 del libro exploraremos otras características como recursos gráficos y bases de datos.

De hecho, imagine que un Asesor Experto similar a *TradeTransactions.mq5* se ejecuta en paralelo con el Asesor Experto de trading y guarda las transacciones recibidas (no necesariamente todos los campos, sino sólo los selectivos que afectan a la toma de decisiones) en variables globales. A continuación, el Asesor Experto podría comprobar las variables globales inmediatamente después de enviar la siguiente solicitud y leer los resultados de los mismos sin salir de la función actual. Además, no necesita su propio manejador *OnTradeTransaction*.

Sin embargo, no es fácil organizar la ejecución de un Asesor Experto de terceros. Desde el punto de vista técnico, esto podría hacerse creando un objeto gráfico y aplicando una plantilla con un Asesor Experto de monitor de transacciones predefinido. Pero hay una manera más fácil. La cuestión es que los eventos de *OnTradeTransaction* se traducen no sólo en Asesor Experto, sino también en indicadores. A su vez, un indicador es el tipo de programa MQL más fácil de lanzar: basta con llamar a *iCustom*.

Además, el uso del indicador ofrece otra ventaja: puede describir el búfer del indicador disponible desde programas externos a través de *CopyBuffer*, y disponer en él un *ring buffer* para almacenar las transacciones procedentes del terminal (resultados de solicitudes). Por lo tanto, no hay necesidad de complicarlo con variables globales.

> ¡Atención! El evento *OnTradeTransaction* no se genera para los indicadores en el probador, por lo que sólo puede comprobar el funcionamiento del par Asesor Experto-indicador en línea.

Llamemos a este indicador *TradeTransactionRelay.mq5* y describamos un búfer en él. Podría hacerse invisible porque escribirá datos que no pueden ser renderizados, pero lo dejamos visible para probar el concepto.

```
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots   1

double Buffer[];

void OnInit()
{
   SetIndexBuffer(0, Buffer, INDICATOR_DATA);
}
```

El manejador *OnCalculate* está vacío.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
   return rates_total;
}
```

En el código, necesitamos un convertidor listo de *double* a *ulong* y viceversa, ya que las celdas del búfer pueden corromper valores *ulong* grandes si se escriben allí utilizando una conversión de tipos simple (véase Números reales).

```
#include <MQL5Book/ConverterT.mqh>
Converter<ulong,double> cnv;
```

Aquí está la función *OnTradeTransaction*.

```
#define FIELD_NUM 6 // the most important fields in MqlTradeResult

void OnTradeTransaction(const MqlTradeTransaction &transaction,
   const MqlTradeRequest &request,
   const MqlTradeResult &result)
{
   if(transaction.type == TRADE_TRANSACTION_REQUEST)
   {
      ArraySetAsSeries(Buffer, true);

      // store FIELD_NUM result fields into consecutive buffer cells
      const int offset = (int)((result.request_id * FIELD_NUM)
         % (Bars(_Symbol, _Period) / FIELD_NUM * FIELD_NUM));
      Buffer[offset + 1] = result.retcode;
      Buffer[offset + 2] = cnv[result.deal];
      Buffer[offset + 3] = cnv[result.order];
      Buffer[offset + 4] = result.volume;
      Buffer[offset + 5] = result.price;
      // this assignment must come last,
      // because it is the result ready flag
      Buffer[offset + 0] = result.request_id;
   }
}
```

Hemos decidido conservar únicamente los seis campos más importantes de la estructura *MqlTradeResult*. Si lo desea, puede ampliar el mecanismo a toda la estructura, pero para transferir el campo de cadena *comment* necesitará un array de caracteres para la que tendrá que reservar bastantes elementos.

Así, cada resultado ocupa ahora seis celdas de búfer consecutivas. El índice de la primera celda de estas seis se determina en función del ID de la solicitud: este número simplemente se multiplica por 6. Como puede haber muchas solicitudes, la entrada funciona según el principio de un búfer anular, es decir, el índice resultante se normaliza dividiendo con resto ('%') por el tamaño del búfer indicador, que es el número de barras redondeado a 6. Cuando los números de solicitud superen el tamaño, el registro irá en círculo a partir de los elementos iniciales.

Dado que la numeración de las barras se ve afectada por la formación de nuevas barras, se recomienda poner el indicador en marcos temporales grandes, como D1. Entonces, sólo al principio del día es probable (aunque bastante improbable) la situación en que la numeración de las barras en el indicador cambiará directamente durante el procesamiento de la siguiente transacción, y luego los resultados registrados por el indicador no serán leídos por el Asesor Experto (puede que una transacción se pase por alto).

El indicador está listo. Ahora vamos a empezar a aplicar una nueva modificación de la prueba Asesor Experto *OrderSendTransaction3.mq5* (¡hurra!, esta es su última versión). Vamos a describir la variable *handle* para el controlador del indicador y a crear el indicador en *OnInit*.

```
int handle = 0;

int OnInit()
{
   ...
   const static string indicator = "MQL5Book/p6/TradeTransactionRelay";
   handle = iCustom(_Symbol, PERIOD_D1, indicator);
   if(handle == INVALID_HANDLE)
   {
      Alert("Can't start indicator ", indicator);
      return INIT_FAILED;
   }
   return INIT_SUCCEEDED;
}
```

Para leer los resultados de la consulta desde el búfer del indicador, vamos a preparar una función auxiliar *AwaitAsync*. Como primer parámetro, recibe una referencia a la estructura *MqlTradeRequestSync*. Si tiene éxito, los resultados obtenidos del búfer del indicador con *handle* se escribirán en esta estructura. El identificador de la solicitud que nos interesa ya debería estar en la estructura anidada, en el campo *result.request_id*. Por supuesto, aquí debemos leer los datos según el mismo principio, es decir, en seis barras.

```
#define FIELD_NUM   6  // the most important fields in MqlTradeResult
#define TIMEOUT  1000  // 1 second

bool AwaitAsync(MqlTradeRequestSync &r, const int _handle)
{
   Converter<ulong,double> cnv;
   const int offset = (int)((r.result.request_id * FIELD_NUM)
      % (Bars(_Symbol, _Period) / FIELD_NUM * FIELD_NUM));
   const uint start = GetTickCount();
   // wait for results or timeout
   while(!IsStopped() && GetTickCount() - start < TIMEOUT)
   {
      double array[];
      if((CopyBuffer(_handle, 0, offset, FIELD_NUM, array)) == FIELD_NUM)
      {
         ArraySetAsSeries(array, true);
         // when request_id is found, fill other fields with results
         if((uint)MathRound(array[0]) == r.result.request_id)
         {
            r.result.retcode = (uint)MathRound(array[1]);
            r.result.deal = cnv[array[2]];
            r.result.order = cnv[array[3]];
            r.result.volume = array[4];
            r.result.price = array[5];
            PrintFormat("Got Req=%d at %d ms",
               r.result.request_id, GetTickCount() - start);
            Print(TU::StringOf(r.result));
            return true;
         }
      }
   }
   Print("Timeout for: ");
   Print(TU::StringOf(r));
   return false;
}
```

Ahora que tenemos esta función, vamos a escribir un algoritmo de trading en un estilo asíncrono-síncrono: como una secuencia directa de pasos, cada uno de los cuales espera a que el anterior esté listo debido a las notificaciones del programa indicador paralelo mientras permanece dentro de una función.

```
void OnTimer()
{
   EventKillTimer();

   MqlTradeRequestSync::AsyncEnabled = true;

   MqlTradeRequestSync request;
   request.magic = Magic;
   request.deviation = Deviation;

   const double volume = Volume == 0 ?
      SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN) : Volume;
   ...
```

Paso 1

```
   Print("Start trade");
   ResetLastError();
   if((bool)(Type == MARKET_BUY ? request.buy(volume) : request.sell(volume)))
   {
      Print("OK Open?");
   }

   if(!(AwaitAsync(request, handle) && request.completed()))
   {
      Print("Failed Open");
      return;
   }
   ...
```

Paso 2

```
   Print("SL/TP modification");
   ...
   if(request.adjust(SL, TP))
   {
      Print("OK Adjust?");
   }

   if(!(AwaitAsync(request, handle) && request.completed()))
   {
      Print("Failed Adjust");
   }
```

Paso 3

```
      Print("Close down");
      if(request.close(request.result.position))
      {
         Print("OK Close?");
      }

      if(!(AwaitAsync(request, handle) && request.completed()))
      {
         Print("Failed Close");
      }

      Print("Finish");
   }
```

Tenga en cuenta que las llamadas al método *completed* ahora no se realizan después de enviar la solicitud, sino después de que la función *AwaitAsync* reciba el resultado.

Por lo demás, todo es muy similar a la primera versión de este algoritmo, pero ahora se basa en llamadas a funciones asíncronas y reacciona a eventos asíncronos.

Probablemente no parezca significativo en este ejemplo concreto de una cadena de manipulaciones en una única posición. Sin embargo, podemos utilizar la misma técnica para enviar y controlar un lote de órdenes. Y entonces los beneficios serán evidentes. En un momento demostraremos esto con la ayuda de un Asesor Experto de cuadrícula y al mismo tiempo compararemos el rendimiento de dos funciones: *OrderSend* y *OrderSendAsync*.

Pero ahora mismo, mientras completamos la serie de Asesores expertos *OrderSendTransaction*, vamos a ejecutar la última versión y a ver en el registro de la ejecución regular y lineal de todos los pasos.

```
Start trade
OK Open?
Got Req=1 at 62 ms
DONE, D=1282677007, #=1300045365, V=0.01, @ 1.10564, Bid=1.10564, Ask=1.10564, Order
Waiting for position for deal D=1282677007
SL/TP modification
OK Adjust?
Got Req=2 at 63 ms
DONE, Order placed, Req=2
Close down
OK Close?
Got Req=3 at 78 ms
DONE, D=1282677008, #=1300045366, V=0.01, @ 1.10564, Bid=1.10564, Ask=1.10564, Order
Finish
```

La sincronización con las demoras de la respuesta puede depender en gran medida del servidor, la hora del día y el símbolo. Por supuesto, parte del tiempo aquí no se dedica a una solicitud de operación con confirmación, sino a la ejecución de la función *CopyBuffer*. Según nuestras observaciones, no tarda más de 16 ms (dentro de un ciclo de un temporizador de sistema estándar, quienes lo deseen pueden perfilar programas que utilicen temporizadores de alta precisión *GetMicrosecondCount*).

Ignore la diferencia entre el estado (DONE) y la descripción de la cadena («Order placed»). El hecho es que el comentario (así como los campos *ask/bid*) permanece en la estructura desde el momento en que es enviado por la función *OrderSendAsync*, y el estado final en el campo *retcode* es escrito por nuestra función *AwaitAsync*. Para nosotros es importante que en la estructura con los resultados estén

actualizados los números de los tickets (*deal* y *order*), el precio de ejercicio (*price*) y el volumen (*volume*).

Basándonos en el ejemplo anterior de *OrderSendTransaction3.mq5*, vamos a crear una nueva versión del Asesor Experto de cuadrícula *PendingOrderGrid3.mq5* (la versión anterior se proporciona en la sección Funciones de lectura de propiedades de posición). Podrá establecer una cuadrícula completa de órdenes en modo síncrono o asíncrono, a elección del usuario. También detectaremos los tiempos de ajuste de la cuadrícula completa para comparar.

El modo se controla mediante la variable de entrada *EnableAsyncSetup*. Se asigna la variable *handle* para el manejador del indicador.

```
input bool EnableAsyncSetup = false;

int handle;
```

Durante la inicialización, en el caso del modo asíncrono, creamos una instancia del indicador *TradeTransactionRelay*.

```
int OnInit()
{
   ...
   if(EnableAsyncSetup)
   {
      const uint start = GetTickCount();
      const static string indicator = "MQL5Book/p6/TradeTransactionRelay";
      handle = iCustom(_Symbol, PERIOD_D1, indicator);
      if(handle == INVALID_HANDLE)
      {
         Alert("Can't start indicator ", indicator);
         return INIT_FAILED;
      }
      PrintFormat("Started in %d ms", GetTickCount() - start);
   }
   ...
}
```

Para simplificar la codificación, hemos sustituido el array bidimensional *request* por una unidimensional en la función *SetupGrid*.

```
uint SetupGrid()
{
   ...                                  // prev:
   MqlTradeRequestSyncLog request[];    // MqlTradeRequestSyncLog request[][2];
   ArrayResize(request, GridSize * 2);  // ArrayResize(request, GridSize);
   ...
}
```

Más adelante en el bucle a través del array, en lugar de llamadas del tipo *request[i][1]*, utilizamos el direccionamiento *request[i * 2 + 1]*.

Esta pequeña transformación era necesaria por las siguientes razones. Dado que utilizamos este array de estructuras para las consultas al crear la cuadrícula, y necesitamos esperar todos los resultados, la

función *AwaitAsync* debería ahora tomar como primer parámetro una referencia a un array. Un array unidimensional es más fácil de manejar.

Para cada solicitud, su desplazamiento en el búfer indicador se calcula en función de su *request_id*: todos los desplazamientos se colocan en el array *offset*. A medida que se reciben confirmaciones de solicitudes, los elementos correspondientes del array se marcan como procesados escribiendo en ellos el valor -1. El número de solicitudes ejecutadas se cuenta en la variable *done*. Cuando es igual al tamaño del array, la cuadrícula entera está lista.

```
bool AwaitAsync(MqlTradeRequestSyncLog &r[], const int _handle)
{
   Converter<ulong,double> cnv;
   int offset[];
   const int n = ArraySize(r);
   int done = 0;
   ArrayResize(offset, n);

   for(int i = 0; i < n; ++i)
   {
      offset[i] = (int)((r[i].result.request_id * FIELD_NUM)
         % (Bars(_Symbol, _Period) / FIELD_NUM * FIELD_NUM));
   }

   const uint start = GetTickCount();
   while(!IsStopped() && done < n && GetTickCount() - start < TIMEOUT)
   for(int i = 0; i < n; ++i)
   {
      if(offset[i] == -1) continue; // skip empty elements
      double array[];
      if((CopyBuffer(_handle, 0, offset[i], FIELD_NUM, array)) == FIELD_NUM)
      {
         ArraySetAsSeries(array, true);
         if((uint)MathRound(array[0]) == r[i].result.request_id)
         {
            r[i].result.retcode = (uint)MathRound(array[1]);
            r[i].result.deal = cnv[array[2]];
            r[i].result.order = cnv[array[3]];
            r[i].result.volume = array[4];
            r[i].result.price = array[5];
            PrintFormat("Got Req=%d at %d ms", r[i].result.request_id,
               GetTickCount() - start);
            Print(TU::StringOf(r[i].result));
            offset[i] = -1; // mark processed
            done++;
         }
      }
   }
   return done == n;
}
```

Volviendo a la función *SetupGrid*, vamos a ver cómo se llama a *AwaitAsync* después del bucle de envío de solicitudes.

```
uint SetupGrid()
{
   ...
   const uint start = GetTickCount();
   for(int i = 0; i < (int)GridSize / 2; ++i)
   {
      // calls of buyLimit/sellStopLimit/sellLimit/buyStopLimit
   }

   if(EnableAsyncSetup)
   {
      if(!AwaitAsync(request, handle))
      {
         Print("Timeout");
         return TRADE_RETCODE_ERROR;
      }
   }

   PrintFormat("Done %d requests in %d ms (%d ms/request)",
      GridSize * 2, GetTickCount() - start,
      (GetTickCount() - start) / (GridSize * 2));
   ...
}
```

Si se produce un tiempo de espera al configurar la cuadrícula (no todas las solicitudes recibirán confirmación dentro del tiempo asignado), devolveremos el código TRADE_RETCODE_ERROR, y el Asesor Experto intentará «deshacer» lo que consiguió crear.

Es importante tener en cuenta que el modo asíncrono sólo está pensado para configurar una cuadrícula completa cuando necesitamos enviar un lote de solicitudes. En caso contrario, se seguirá utilizando el modo síncrono. Por lo tanto, debemos poner la bandera *MqlTradeRequestSync::AsyncEnabled* a *true* antes del bucle de envío y volver a ponerla a *false* después. No obstante, preste atención a lo siguiente: pueden producirse errores dentro del bucle, debido a los cuales éste se termina prematuramente, devolviendo el último código del servidor. Por lo tanto, si colocamos un reset asíncrono después del bucle, no hay garantía de que se reinicie.

Para resolver este problema se añade una pequeña clase *AsyncSwitcher* al archivo *MqlTradeSync.mqh*. La clase controla la activación y desactivación del modo asíncrono desde su constructor y destructor. Esto se ajusta al concepto de gestión de recursos RAII que se expone en la sección Gestión de descriptores de archivos.

```
class AsyncSwitcher
{
public:
   AsyncSwitcher(const bool enabled = true)
   {
      MqlTradeRequestSync::AsyncEnabled = enabled;
   }
   ~AsyncSwitcher()
   {
      MqlTradeRequestSync::AsyncEnabled = false;
   }
};
```

Ahora, para la activación temporal segura del modo asíncrono, podemos simplemente describir el objeto local *AsyncSwitcher* en la función *SetupGrid*. El código volverá automáticamente al modo síncrono al salir de la función.

```
uint SetupGrid()
{
   ...
   AsyncSwitcher sync(EnableAsyncSetup);
   ...
   for(int i = 0; i < (int)GridSize / 2; ++i)
   {
      ...
   }
   ...
}
```

El Asesor Experto está listo. Intentemos ejecutarlo dos veces: en modo síncrono y asíncrono para una cuadrícula suficientemente grande (10 niveles, paso de cuadrícula 200).

Para una cuadrícula de 10 niveles, obtendremos 20 consultas, por lo que a continuación se muestran algunos de los registros. En primer lugar, se ha utilizado un modo síncrono. Aclaremos que la inscripción sobre la disponibilidad de las solicitudes se muestra antes que los mensajes sobre las solicitudes porque estos últimos son generados por los destructores de la estructura cuando la función sale. La velocidad de procesamiento es de 51 ms por solicitud.

```
Start setup at 1.10379
Done 20 requests in 1030 ms (51 ms/request)
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.10
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978336, V=0.01, Request executed, Req=1
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK,
» X=1.10400, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978337, V=0.01, Request executed, Req=2
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.10
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978343, V=0.01, Request executed, Req=5
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK,
» X=1.10200, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978344, V=0.01, Request executed, Req=6
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.09
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978348, V=0.01, Request executed, Req=9
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK,
» X=1.10000, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978350, V=0.01, Request executed, Req=10
...
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.1
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978339, V=0.01, Request executed, Req=3
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, @
» X=1.10400, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978340, V=0.01, Request executed, Req=4
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.1
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978345, V=0.01, Request executed, Req=7
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, @
» X=1.10600, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978347, V=0.01, Request executed, Req=8
...
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.1
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978365, V=0.01, Request executed, Req=19
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, @
» X=1.11200, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300978366, V=0.01, Request executed, Req=20
```

El centro de la cuadrícula coincidía con el precio de 1.10400. El sistema asigna números a las solicitudes en el orden en que se reciben, y su numeración en el array se corresponde con el orden en que cursamos las órdenes: desde el nivel base central, nos desviamos gradualmente hacia los lados. Por lo tanto, no se sorprenda de que después de un par de 1 y 2 (para el nivel 1.10200) viene 5 y 6 (1.10000), ya que 3 y 4 (1.10600) se enviaron antes.

En modo asíncrono, los destructores van precedidos de mensajes sobre la disponibilidad de determinadas solicitudes recibidas en *AwaitAsync* en tiempo real, y no necesariamente en el orden en que se enviaron las solicitudes (por ejemplo, las solicitudes 49 y 50 «adelantaron» a las 47 y 48).

```
Started in 16 ms
Start setup at 1.10356
Got Req=41 at 109 ms
DONE, #=1300979180, V=0.01, Order placed, Req=41
Got Req=42 at 109 ms
DONE, #=1300979181, V=0.01, Order placed, Req=42
Got Req=43 at 125 ms
DONE, #=1300979182, V=0.01, Order placed, Req=43
Got Req=44 at 140 ms
DONE, #=1300979183, V=0.01, Order placed, Req=44
Got Req=45 at 156 ms
DONE, #=1300979184, V=0.01, Order placed, Req=45
Got Req=46 at 172 ms
DONE, #=1300979185, V=0.01, Order placed, Req=46
Got Req=49 at 172 ms
DONE, #=1300979188, V=0.01, Order placed, Req=49
Got Req=50 at 172 ms
DONE, #=1300979189, V=0.01, Order placed, Req=50
Got Req=47 at 172 ms
DONE, #=1300979186, V=0.01, Order placed, Req=47
Got Req=48 at 172 ms
DONE, #=1300979187, V=0.01, Order placed, Req=48
Got Req=51 at 172 ms
DONE, #=1300979190, V=0.01, Order placed, Req=51
Got Req=52 at 203 ms
DONE, #=1300979191, V=0.01, Order placed, Req=52
Got Req=55 at 203 ms
DONE, #=1300979194, V=0.01, Order placed, Req=55
Got Req=56 at 203 ms
DONE, #=1300979195, V=0.01, Order placed, Req=56
Got Req=53 at 203 ms
DONE, #=1300979192, V=0.01, Order placed, Req=53
Got Req=54 at 203 ms
DONE, #=1300979193, V=0.01, Order placed, Req=54
Got Req=57 at 218 ms
DONE, #=1300979196, V=0.01, Order placed, Req=57
Got Req=58 at 218 ms
DONE, #=1300979198, V=0.01, Order placed, Req=58
Got Req=59 at 218 ms
DONE, #=1300979199, V=0.01, Order placed, Req=59
Got Req=60 at 218 ms
DONE, #=1300979200, V=0.01, Order placed, Req=60
Done 20 requests in 234 ms (11 ms/request)
...
```

Debido al hecho de que todas las solicitudes se ejecutaron en paralelo, el tiempo total de envío (234 ms) es sólo ligeramente superior al tiempo de una sola solicitud (aquí alrededor de 100 ms, pero usted tendrá su propio cronometraje). Como resultado, obtuvimos una velocidad de 11 ms por solicitud, que es 5 veces más rápida que con el método síncrono. Como las solicitudes se enviaron casi simultáneamente, no podemos saber el tiempo de ejecución de cada una, y los milisegundos indican la llegada del resultado de una solicitud concreta desde el momento del inicio general del envío en grupo.

Otros registros, como en el caso anterior, contienen todos los campos de consulta y resultado que se imprimen desde los destructores de estructuras. La línea «Order placed» no se modificó después de *OrderSendAsync*, ya que nuestro indicador auxiliar *TradeTransactionRelay.mq5* no publica íntegramente la estructura *MqlTradeResult* del mensaje TRADE_TRANSACTION_REQUEST.

```
...
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.10
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979180, V=0.01, Order placed, Req=41
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK,
» X=1.10400, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979181, V=0.01, Order placed, Req=42
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.10
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979184, V=0.01, Order placed, Req=45
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK,
» X=1.10200, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979185, V=0.01, Order placed, Req=46
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.09
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979188, V=0.01, Order placed, Req=49
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK,
» X=1.10000, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979189, V=0.01, Order placed, Req=50
...
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.1
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979182, V=0.01, Order placed, Req=43
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, @
» X=1.10400, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979183, V=0.01, Order placed, Req=44
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.1
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979186, V=0.01, Order placed, Req=47
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, @
» X=1.10600, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979187, V=0.01, Order placed, Req=48
...
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_SELL_LIMIT, V=0.01, ORDER_FILLING_FOK, @ 1.1
» ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979199, V=0.01, Order placed, Req=59
TRADE_ACTION_PENDING, EURUSD, ORDER_TYPE_BUY_STOP_LIMIT, V=0.01, ORDER_FILLING_FOK, @
» X=1.11200, ORDER_TIME_GTC, M=1234567890, G[1.10400]
DONE, #=1300979200, V=0.01, Order placed, Req=60
```

Hasta ahora, nuestro Asesor Experto de cuadrícula tenía un par de órdenes pendientes en cada nivel: Limit y Stop Limit. Para evitar esta duplicación, dejamos sólo las órdenes Limit. Esta será la versión final de *PendingOrderGrid4.mq5*, que también podrá ejecutarse en modo síncrono y asíncrono. No entraremos en detalle en el código fuente, sino que nos limitaremos a señalar las principales diferencias con respecto a la versión anterior.

En la función *SetupGrid* necesitamos un array de estructuras de tamaño igual a *GridSize* y no duplicado. El número de solicitudes también disminuirá 2 veces: los únicos métodos utilizados para ellas son *buyLimit* y *sellLimit*.

La función *CheckGrid* comprueba la integridad de la cuadrícula de una manera diferente. Anteriormente, la ausencia de una orden Stop Limit emparejada en el nivel en el que existe un límite se consideraba un error. Esto podía ocurrir cuando se activaba una orden Stop Limit en el servidor desde un nivel vecino. Sin embargo, este esquema no es capaz de restablecer la cuadrícula si se produce un fuerte movimiento bidireccional del precio (spike) en una barra: eliminará no sólo las órdenes Limit originales, sino también las nuevas generadas a partir de las Stop Limit. Ahora el algoritmo comprueba honestamente los niveles vacantes a ambos lados del precio actual y crea allí órdenes Limit utilizando *RepairGridLevel*. Esta función de ayuda coloca previamente órdenes Stop Limit.

Por último, el manejador *OnTradeTransaction* apareció en *PendingOrderGrid4.mq5*. La activación de una orden pendiente provocará la ejecución de una transacción (y un cambio en la configuración de la cuadrícula que debe corregirse), por lo que controlamos las transacciones por un símbolo y una magia determinados. Cuando se detecta una transacción, se llama instantáneamente a la función *CheckGrid*, además de que se sigue ejecutando al principio de cada barra.

```
void OnTradeTransaction(const MqlTradeTransaction &transaction,
   const MqlTradeRequest &,
   const MqlTradeResult &)
{
   if(transaction.type == TRADE_TRANSACTION_DEAL_ADD)
   {
      if(transaction.symbol == _Symbol)
      {
         DealMonitor dm(transaction.deal); // select the deal
         if(dm.get(DEAL_MAGIC) == Magic)
         {
            CheckGrid();
         }
      }
   }
}
```

Cabe señalar que la presencia de *OnTradeTransaction* no es suficiente para escribir Asesores Expertos que sean resistentes a influencias externas imprevistas. Por supuesto, los eventos le permiten responder rápidamente a la situación, pero no tenemos ninguna garantía de que el Asesor Experto no se apague (o se desconecte) por una razón u otra durante algún tiempo y se salte tal o cual transacción. Por lo tanto, el manejador *OnTradeTransaction* sólo debería ayudar a acelerar los procesos que el programa puede realizar sin él. En concreto, para restablecer correctamente su estado tras el arranque.

Sin embargo, además del evento *OnTradeTransaction*, MQL5 proporciona otro evento más sencillo: *OnTrade*.

6.4.36 Evento OnTrade

El evento *OnTrade* se produce al modificar la lista de órdenes y posiciones abiertas, el historial de órdenes y el historial de transacciones. Cualquier acción de trading (colocar/activar/eliminar una orden pendiente, abrir/cerrar una posición, establecer niveles de protección, etc.) modifica en consecuencia el historial de órdenes y transacciones y/o la lista de posiciones y órdenes actuales. El iniciador de una acción puede ser un usuario, un programa o un servidor.

Para recibir el evento en un programa, debe describir el manejador correspondiente.

void OnTrade(void)

En el caso del envío de solicitudes de trading mediante *OrderSend/OrderSendAsync*, una solicitud desencadenará múltiples eventos *OnTrade*, ya que el procesamiento suele tener lugar en varias etapas y cada operación puede cambiar el estado de las órdenes, las posiciones y el historial de trading.

En general, no existe una proporción exacta en el número de llamadas a *OnTrade* y *OnTradeTransaction*. *OnTrade* se llama después de las correspondientes llamadas a *OnTradeTransaction*.

Dado que el evento *OnTrade* es de naturaleza generalizada y no especifica la esencia de la operación, es menos popular entre los desarrolladores de programas MQL. Por lo general, es necesario comprobar todos los aspectos del estado de la cuenta de trading en el código y compararlo con algún estado guardado, es decir, con la caché aplicada de las entidades de trading utilizadas en la estrategia de trading. En el caso más sencillo, puede, por ejemplo, recordar el ticket de la orden creada en el manejador *OnTrade* para interrogar todas sus propiedades. Sin embargo, esto puede implicar el análisis «innecesario» de un gran número de eventos fortuitos que no están relacionados con una orden específica.

Hablaremos de la posibilidad de aplicar el almacenamiento en caché del entorno de trading y del historial en la sección sobre Asesores Expertos multidivisa.

Para explorar más a fondo *OnTrade*, veamos un Asesor Experto que implementa una estrategia sobre dos órdenes pendientes OCO («One Cancels Other»). Este colocará un par de órdenes stop de ruptura y esperará a que se active una de ellas, tras lo cual se eliminará la segunda. Para mayor claridad, proporcionaremos soporte para ambos tipos de eventos de trading, *OnTrade* y *OnTradeTransaction*, de forma que la lógica de trabajo se ejecutará desde un manejador u otro, según elija el usuario.

El código fuente está disponible en el archivo *OCO2.mq5*. Sus parámetros de entrada incluyen el tamaño del lote *Volume* (por defecto es 0, lo que significa mínimo), la distancia *Distance2SLTP* en puntos para colocar cada una de las órdenes y también determina los niveles de protección, la fecha de vencimiento *Expiration* en segundos desde la hora de configuración, y el conmutador de eventos *ActivationBy* (por defecto, *OnTradeTransaction*). Dado que *Distance2SLTP* establece tanto el desplazamiento desde el precio actual como la distancia hasta el Stop Loss, los Stop Loss de las dos órdenes son los mismos e iguales al precio en el momento de establecerlos.

```
enum EVENT_TYPE
{
   ON_TRANSACTION, // OnTradeTransaction
   ON_TRADE        // OnTrade
};

input double Volume;            // Volume (0 - minimal lot)
input uint Distance2SLTP = 500; // Distance Indent/SL/TP (points)
input ulong Magic = 1234567890;
input ulong Deviation = 10;
input ulong Expiration = 0;     // Expiration (seconds in future, 3600 - 1 hour, etc)
input EVENT_TYPE ActivationBy = ON_TRANSACTION;
```

Para simplificar la inicialización de las estructuras de solicitud, describiremos nuestra propia estructura *MqlTradeRequestSyncOCO* derivada de *MqlTradeRequestSync*.

```
struct MqlTradeRequestSyncOCO: public MqlTradeRequestSync
{
   MqlTradeRequestSyncOCO()
   {
      symbol = _Symbol;
      magic = Magic;
      deviation = Deviation;
      if(Expiration > 0)
      {
         type_time = ORDER_TIME_SPECIFIED;
         expiration = (datetime)(TimeCurrent() + Expiration);
      }
   }
};
```

A nivel global, vamos a introducir varios objetos y variables.

```
OrderFilter orders;        // object for selecting orders
PositionFilter trades;     // object for selecting positions
bool FirstTick = false;    // or single processing of OnTick at start
ulong ExecutionCount = 0;  // counter of trading strategy calls RunStrategy()
```

Toda la lógica de trading, excepto el momento de inicio, se activará mediante eventos de trading. En el manejador *OnInit* configuramos los objetos de filtro y esperamos el primer tick (configuramos *FirstTick* en *true*).

```
int OnInit()
{
   FirstTick = true;

   orders.let(ORDER_MAGIC, Magic).let(ORDER_SYMBOL, _Symbol)
      .let(ORDER_TYPE, (1 << ORDER_TYPE_BUY_STOP) | (1 << ORDER_TYPE_SELL_STOP),
      IS::OR_BITWISE);
   trades.let(POSITION_MAGIC, Magic).let(POSITION_SYMBOL, _Symbol);

   return INIT_SUCCEEDED;
}
```

Sólo nos interesan las órdenes Stop (compra/venta) y las posiciones con un número mágico concreto y el símbolo actual.

En la función *OnTick*, llamamos una vez a la parte principal del algoritmo diseñado como *RunStrategy* (lo describiremos a continuación). Además, esta función sólo se llamará desde *OnTrade* o *OnTradeTransaction*.

```
void OnTick()
{
   if(FirstTick)
   {
      RunStrategy();
      FirstTick = false;
   }
}
```

Por ejemplo, cuando el modo *OnTrade* está activado, este fragmento funciona.

```
void OnTrade()
{
   static ulong count = 0;
   PrintFormat("OnTrade(%d)", ++count);
   if(ActivationBy == ON_TRADE)
   {
      RunStrategy();
   }
}
```

Tenga en cuenta que las llamadas al manejador *OnTrade* se cuentan independientemente de si la estrategia se activa aquí o no. Del mismo modo, los eventos relevantes se cuentan en el manejador *OnTradeTransaction* (incluso si ocurren en vano). Esto se hace para poder ver ambos eventos y sus contadores en el registro al mismo tiempo.

Cuando el modo *OnTradeTransaction* está activado, obviamente, *RunStrategy* empieza desde ahí.

```
void OnTradeTransaction(const MqlTradeTransaction &transaction,
   const MqlTradeRequest &request,
   const MqlTradeResult &result)
{
   static ulong count = 0;
   PrintFormat("OnTradeTransaction(%d)", ++count);
   Print(TU::StringOf(transaction));

   if(ActivationBy != ON_TRANSACTION) return;

   if(transaction.type == TRADE_TRANSACTION_ORDER_DELETE)
   {
      // why not here? for answer, see the text
      /* // this won't work online: m.isReady() == false because order temporarily lo
      OrderMonitor m(transaction.order);
      if(m.isReady() && m.get(ORDER_MAGIC) == Magic && m.get(ORDER_SYMBOL) == _Symbol
      {
         RunStrategy();
      }
      */
   }
   else if(transaction.type == TRADE_TRANSACTION_HISTORY_ADD)
   {
      OrderMonitor m(transaction.order);
      if(m.isReady() && m.get(ORDER_MAGIC) == Magic && m.get(ORDER_SYMBOL) == _Symbol
      {
         // the ORDER_STATE property does not matter - in any case, you need to remov
         // if(transaction.order_state == ORDER_STATE_FILLED
         // || transaction.order_state == ORDER_STATE_CANCELED ...)
         RunStrategy();
      }
   }
}
```

Debe tenerse en cuenta que cuando se negocia en línea, una orden pendiente activada puede desaparecer del entorno de negociación durante algún tiempo debido a que se transfiere de las existentes al historial. Cuando recibimos el evento TRADE_TRANSACTION_ORDER_DELETE, la orden ya ha sido eliminada de la lista activa pero aún no ha aparecido en el historial. Sólo llega cuando recibimos el evento TRADE_TRANSACTION_HISTORY_ADD. Este comportamiento no se observa en el probador, es decir, una orden eliminada se añade inmediatamente al historial y está disponible allí para seleccionar y leer propiedades ya en la fase TRADE_TRANSACTION_ORDER_DELETE.

En ambos manejadores de eventos de trading contamos y registramos el número de llamadas. Para el caso con *OnTrade*, este debe coincidir con *ExecutionCount* que pronto veremos dentro de *RunStrategy*. Sin embargo, para *OnTradeTransaction*, el contador y *ExecutionCount* diferirán significativamente porque la estrategia aquí se llama muy selectivamente, para un tipo de evento. Basándonos en esto, podemos concluir que *OnTradeTransaction* permite un uso más eficiente de los recursos al llamar al algoritmo sólo cuando es apropiado.

El contador *ExecutionCount* se envía al registro cuando se descarga el Asesor Experto.

```
void OnDeinit(const int r)
{
   Print("ExecutionCount = ", ExecutionCount);
}
```

Por último, vamos a introducir la función *RunStrategy*. El contador prometido se incrementa al principio.

```
void RunStrategy()
{
   ExecutionCount++;
   ...
```

A continuación, se describen dos arrays para recibir los tickets de la orden y sus estados desde el objeto de filtro *orders*.

```
   ulong tickets[];
   ulong states[];
```

Para empezar, solicitaremos las órdenes que se ajusten a nuestras condiciones. Si son dos, todo va bien y no hay que hacer nada.

```
   orders.select(ORDER_STATE, tickets, states);
   const int n = ArraySize(tickets);
   if(n == 2) return; // OK - standard state
   ...
```

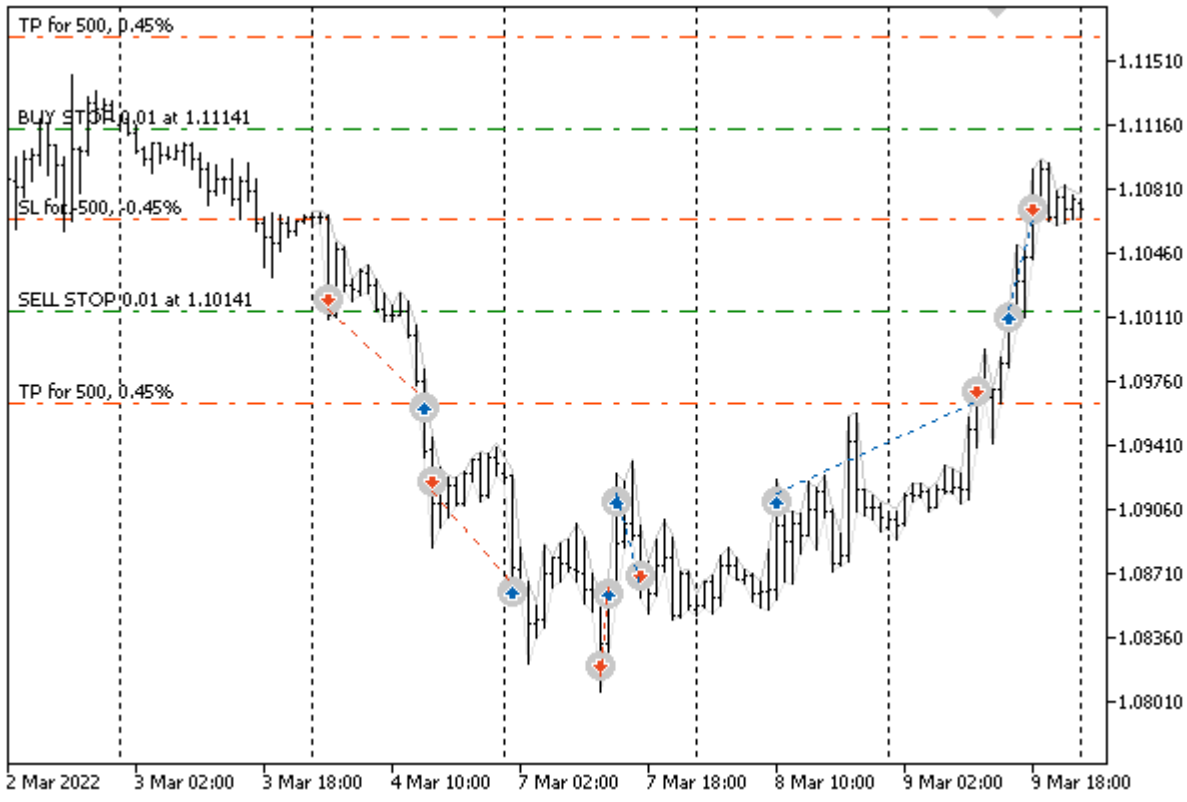
Si queda una orden, entonces es que se ha activado la otra y hay que borrar la que queda.

```
   if(n > 0)          // 1 or 2+ orders is an error, you need to delete everything
   {
      // delete all matching orders, except for partially filled ones
      MqlTradeRequestSyncOCO r;
      for(int i = 0; i < n; ++i)
      {
         if(states[i] != ORDER_STATE_PARTIAL)
         {
            r.remove(tickets[i]) && r.completed();
         }
      }
   }
   ...
```

De lo contrario, no hay órdenes. Por lo tanto, es necesario comprobar si hay alguna posición abierta: para ello, utilizamos otro objeto de filtro *trades* pero los resultados se añaden al mismo array receptor *tickets*. Si no hay posición, colocamos un nuevo par de órdenes.

```
      else // n == 0
      {
         // if there are no open positions, place 2 orders
         if(!trades.select(tickets))
         {
            MqlTradeRequestSyncOCO r;
            SymbolMonitor sm(_Symbol);

            const double point = sm.get(SYMBOL_POINT);
            const double lot = Volume == 0 ? sm.get(SYMBOL_VOLUME_MIN) : Volume;
            const double buy = sm.get(SYMBOL_BID) + point * Distance2SLTP;
            const double sell = sm.get(SYMBOL_BID) - point * Distance2SLTP;

            r.buyStop(lot, buy, buy - Distance2SLTP * point,
               buy + Distance2SLTP * point) && r.completed();
            r.sellStop(lot, sell, sell + Distance2SLTP * point,
               sell - Distance2SLTP * point) && r.completed();
         }
      }
   }
```

Vamos a ejecutar el Asesor Experto en el probador con la configuración por defecto, en el par EURUSD. En la siguiente imagen se muestra el proceso de simulación.



Asesor Experto con un par de órdenes stop pendientes basadas en la estrategia OCO en el probador

En la fase de colocación de un par de órdenes, veremos las siguientes entradas en el registro:

```
buy stop 0.01 EURUSD at 1.11151 sl: 1.10651 tp: 1.11651 (1.10646 / 1.10683)
sell stop 0.01 EURUSD at 1.10151 sl: 1.10651 tp: 1.09651 (1.10646 / 1.10683)
OnTradeTransaction(1)
TRADE_TRANSACTION_ORDER_ADD, #=2(ORDER_TYPE_BUY_STOP/ORDER_STATE_PLACED), ORDER_TIME_
» @ 1.11151, SL=1.10651, TP=1.11651, V=0.01
OnTrade(1)
OnTradeTransaction(2)
TRADE_TRANSACTION_REQUEST
OnTradeTransaction(3)
TRADE_TRANSACTION_ORDER_ADD, #=3(ORDER_TYPE_SELL_STOP/ORDER_STATE_PLACED), ORDER_TIME
» @ 1.10151, SL=1.10651, TP=1.09651, V=0.01
OnTrade(2)
OnTradeTransaction(4)
TRADE_TRANSACTION_REQUEST
```

En cuanto se activa una de las órdenes, ocurre lo siguiente:

```
order [#3 sell stop 0.01 EURUSD at 1.10151] triggered
deal #2 sell 0.01 EURUSD at 1.10150 done (based on order #3)
deal performed [#2 sell 0.01 EURUSD at 1.10150]
order performed sell 0.01 at 1.10150 [#3 sell stop 0.01 EURUSD at 1.10151]
OnTradeTransaction(5)
TRADE_TRANSACTION_DEAL_ADD, D=2(DEAL_TYPE_SELL), #=3(ORDER_TYPE_BUY/ORDER_STATE_START
» EURUSD, @ 1.10150, SL=1.10651, TP=1.09651, V=0.01, P=3
OnTrade(3)
OnTradeTransaction(6)
TRADE_TRANSACTION_ORDER_DELETE, #=3(ORDER_TYPE_SELL_STOP/ORDER_STATE_FILLED), ORDER_T
» EURUSD, @ 1.10151, SL=1.10651, TP=1.09651, V=0.01, P=3
OnTrade(4)
OnTradeTransaction(7)
TRADE_TRANSACTION_HISTORY_ADD, #=3(ORDER_TYPE_SELL_STOP/ORDER_STATE_FILLED), ORDER_TI
» EURUSD, @ 1.10151, SL=1.10651, TP=1.09651, P=3
order canceled [#2 buy stop 0.01 EURUSD at 1.11151]
OnTrade(5)
OnTradeTransaction(8)
TRADE_TRANSACTION_ORDER_DELETE, #=2(ORDER_TYPE_BUY_STOP/ORDER_STATE_CANCELED), ORDER_
» EURUSD, @ 1.11151, SL=1.10651, TP=1.11651, V=0.01
OnTrade(6)
OnTradeTransaction(9)
TRADE_TRANSACTION_HISTORY_ADD, #=2(ORDER_TYPE_BUY_STOP/ORDER_STATE_CANCELED), ORDER_T
» EURUSD, @ 1.11151, SL=1.10651, TP=1.11651, V=0.01
OnTrade(7)
OnTradeTransaction(10)
TRADE_TRANSACTION_REQUEST
```

La orden nº 3 fue eliminada por sí misma, y la orden nº 2 fue eliminada (cancelada) por nuestro Asesor Experto.

Si ejecutamos el Asesor Experto con sólo el modo de funcionamiento a través del evento *OnTrade* cambiado en la configuración, deberíamos obtener resultados financieros completamente similares (ceteris paribus, es decir, por ejemplo, si no se incluyen los retrasos aleatorios en la generación de ticks). Lo único que será diferente es el número de llamadas a la función *RunStrategy*. Por ejemplo, para 4 meses de 2022 en EURUSD, H1 con 88 operaciones, obtendremos las siguientes métricas aproximadas de *ExecutionCount* (lo que importa es el ratio, no los valores absolutos asociados a los ticks de tu bróker):

- OnTradeTransaction – 132
- OnTrade – 438

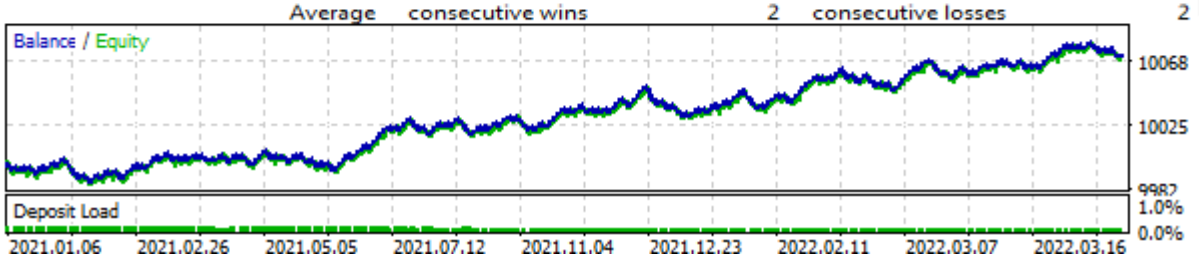
Se trata de una prueba práctica de la posibilidad de construir algoritmos más selectivos basados en *OnTradeTransaction* en comparación con *OnTrade*.

Esta versión *OCO2.mq5* del Asesor Experto reacciona a las acciones con órdenes y posiciones de forma bastante directa. En concreto, en cuanto se cierre la posición anterior mediante Stop Loss o Take Profit, colocará dos nuevas órdenes. Si elimina una de las órdenes manualmente, el Asesor Experto eliminará inmediatamente la segunda y volverá a crear un nuevo par con un desplazamiento respecto al precio actual. Usted puede mejorar el comportamiento mediante la incorporación de un programa similar a lo que se hace en el Asesor Experto de cuadrícula y no reaccionar a las órdenes canceladas en el historial (aunque, por favor, tenga en cuenta que MQL5 no proporciona medios para averiguar si una orden fue cancelada manualmente o mediante programación). Presentaremos una dirección diferente para mejorar este Asesor Experto cuando exploremos la API del calendario económico.

Además, en la versión actual ya está disponible un modo interesante, relacionado con el establecimiento de la fecha de vencimiento de las órdenes pendientes en la variable de entrada *Expiration*. Si un par de órdenes no se activa, inmediatamente después de su vencimiento se coloca un nuevo par en relación con el nuevo precio actual modificado. Como ejercicio independiente, puede intentar optimizar el Asesor Experto en el probador cambiando *Expiration* y *Distance2SLTP*. El trabajo programático con el probador, incluso en el modo de optimización, se abordará en el capítulo siguiente.

A continuación se muestra una de las opciones de ajuste (*Distance2SLTP=250*, *Expiration=5000*) encontradas durante un periodo de 16 meses desde principios de 2021 para el par EURUSD.

History Quality	100%				
Bars	7768	Ticks	26671677	Symbols	1
Initial Deposit	10 000.00				
Total Net Profit	72.57	Balance Drawdow...	13.03	Equity Drawdown ...	13.22
Gross Profit	450.07	Balance Drawdow...	19.20 (0.19%)	Equity Drawdown ...	20.67 (0.21%)
Gross Loss	-377.50	Balance Drawdow...	0.19% (19.20)	Equity Drawdown ...	0.21% (20.67)
Profit Factor	1.19	Expected Payoff	0.22	Margin Level	81061.96%
Recovery Factor	3.51	Sharpe Ratio	0.09	Z-Score	2.53 (98.86%)
AHPR	1.0000 (0.00...	LR Correlation	0.98	OnTester result	0
GHPR	1.0000 (0.00...	LR Standard Error	5.44		
Total Trades	329	Short Trades (won ...	171 (60.23%)	Long Trades (won ...	158 (49.37%)
Total Deals	658	Profit Trades (% of...	181 (55.02%)	Loss Trades (% of t...	148 (44.98%)
	Largest	profit trade	2.67	loss trade	-3.73
	Average	profit trade	2.49	loss trade	-2.55
	Maximum	consecutive wins ($)	7 (17.33)	consecutive losses ...	5 (-12.70)
	Maximal	consecutive profit ...	17.33 (7)	consecutive loss (c...	-12.70 (5)
	Average	consecutive wins	2	consecutive losses	2



Resultados de las pruebas de ejecución del Asesor Experto OCO2

6.4.37 Seguimiento de los cambios en el entorno de trading

En la sección anterior relacionada con el evento *OnTrade* hemos mencionado que algunos enfoques de programación de estrategias de trading pueden requerir que se tomen instantáneas del entorno y se comparen entre sí a lo largo del tiempo. Esta es una práctica común cuando se utiliza *OnTrade* pero también puede activarse en un horario, en cada barra, o incluso tick. Nuestras clases de monitor que pueden leer las propiedades de órdenes, transacciones y posiciones carecían de la capacidad de guardar el estado. En esta sección presentaremos una de las opciones de almacenamiento en caché del entorno de trading.

Las propiedades de todos los objetos de trading se dividen por tipos en tres grupos: entero, real y cadena. Cada clase de objeto tiene sus propios grupos (por ejemplo, para las órdenes, las propiedades de enteros se describen en la enumeración ENUM_ORDER_PROPERTY_INTEGER, y para las posiciones se describen en ENUM_POSITION_PROPERTY_INTEGER), pero la esencia de la división es la misma. Por lo tanto, introduciremos la enumeración PROP_TYPE, con cuya ayuda será posible describir a qué tipo pertenece una propiedad de objeto. Esta generalización surge de forma natural, ya que los mecanismos para almacenar y procesar propiedades del mismo tipo deben ser los mismos, independientemente de si la propiedad pertenece a una orden, posición o transacción.

```
enum PROP_TYPE
{
   PROP_TYPE_INTEGER,
   PROP_TYPE_DOUBLE,
   PROP_TYPE_STRING,
};
```

Los arrays son la forma más sencilla de almacenar valores de propiedades. Obviamente, debido a la presencia de tres tipos de base, necesitaremos tres arrays diferentes. Vamos a describirlos dentro de una nueva clase *TradeState* anidada en *MonitorInterface* (*TradeBaseMonitor.mqh*).

La plantilla básica *MonitorInterface<I,D,S>* constituye la base de todas las clases de monitor aplicadas (*OrderMonitor*, *DealMonitor*, *PositionMonitor*). Los tipos I, D y S corresponden aquí a enumeraciones concretas de propiedades de enteros, reales y cadenas.

Es bastante lógico incluir el mecanismo de almacenamiento en el monitor base, sobre todo porque la caché de propiedades creada se llenará de datos mediante la lectura de propiedades del objeto monitor.

```
template<typename I,typename D,typename S>
class MonitorInterface
{
   ...
   class TradeState
   {
   public:
      ...
      long ulongs[];
      double doubles[];
      string strings[];
      const MonitorInterface *owner;

      TradeState(const MonitorInterface *ptr) : owner(ptr)
      {
         ...
      }
   };
```

Toda la clase *TradeState* se ha hecho pública porque sería necesario acceder a sus campos desde el objeto monitor padre (que se pasa como puntero al constructor), y además *TradeState* sólo se utilizará en la parte protegida del monitor (no se puede acceder a ellos desde el exterior).

Para rellenar tres arrays con valores de propiedades de tres tipos diferentes, primero debe averiguar la distribución de propiedades por tipo e índices en cada array concreto.

Para cada tipo de objeto de trading (órdenes, transacciones y posiciones), los identificadores de las tres enumeraciones correspondientes con propiedades de distintos tipos no se cruzan y forman una numeración continua. Vamos a demostrarlo.

En el capítulo Enumeraciones hemos visto el script *ConversionEnum.mq5* que implementa la función *process* para registrar todos los elementos de una enumeración determinada. Ese script examinó el enum ENUM_APPLIED_PRICE. Ahora podemos crear una copia del script y analizar las otras tres enumeraciones. Por ejemplo, así:

```
void OnStart()
{
   process((ENUM_POSITION_PROPERTY_INTEGER)0);
   process((ENUM_POSITION_PROPERTY_DOUBLE)0);
   process((ENUM_POSITION_PROPERTY_STRING)0);
}
```

Como resultado de su ejecución, obtenemos el siguiente registro. La columna de la izquierda contiene la numeración dentro de las enumeraciones, y los valores de la derecha (después del signo '=') son las constantes integradas (identificadores) de los elementos.

```
ENUM_POSITION_PROPERTY_INTEGER Count=9
0 POSITION_TIME=1
1 POSITION_TYPE=2
2 POSITION_MAGIC=12
3 POSITION_IDENTIFIER=13
4 POSITION_TIME_MSC=14
5 POSITION_TIME_UPDATE=15
6 POSITION_TIME_UPDATE_MSC=16
7 POSITION_TICKET=17
8 POSITION_REASON=18
ENUM_POSITION_PROPERTY_DOUBLE Count=8
0 POSITION_VOLUME=3
1 POSITION_PRICE_OPEN=4
2 POSITION_PRICE_CURRENT=5
3 POSITION_SL=6
4 POSITION_TP=7
5 POSITION_COMMISSION=8
6 POSITION_SWAP=9
7 POSITION_PROFIT=10
ENUM_POSITION_PROPERTY_STRING Count=3
0 POSITION_SYMBOL=0
1 POSITION_COMMENT=11
2 POSITION_EXTERNAL_ID=19
```

Por ejemplo, la propiedad con constante 0 es una cadena POSITION_SYMBOL, las propiedades con constantes 1 y 2 son enteros POSITION_TIME y POSITION_TYPE, la propiedad con constante 3 es un POSITION_VOLUME real, etc.

Así, las constantes son un sistema de índices de extremo a extremo sobre propiedades de todos los tipos, y podemos utilizar el mismo algoritmo (basado en *EnumToArray.mqh*) para obtenerlas.

Para cada propiedad, es necesario recordar su tipo (que determina cuál de los tres arrays almacenará el valor) y el número de serie entre las propiedades del mismo tipo (este será el índice del elemento en el array correspondiente). Por ejemplo, vemos que las posiciones sólo tienen tres propiedades de cadena, por lo que el array *strings* en la instantánea de una posición tendrá que tener el mismo tamaño, y POSITION_SYMBOL (0), POSITION_COMMENT (11) y POSITION_EXTERNAL_ID (19) se escribirán en sus índices 0, 1 y 2.

La conversión de los índices de extremo a extremo de las propiedades en su tipo (uno de PROP_TYPE) y en un número ordinal en un array del tipo correspondiente se puede hacer una vez al inicio del programa, ya que las enumeraciones con propiedades son constantes (integradas en el sistema). Escribimos la tabla de direccionamiento indirecto resultante en un array estático bidimensional *indices*. Su tamaño en la primera dimensión se determinará dinámicamente como el número total de propiedades (de los 3 tipos). Escribiremos el tamaño en la variable estática *limit*. Se asignan un par de celdas para la segunda dimensión: *indices[i][0]* - tipo PROP_TYPE, *indices[i][1]* - índice en uno de los arrays *ulongs*, *doubles* o *strings* (dependiendo de *indices[i][0]*).

```
class TradeState
{
   ...
   static int indices[][2];
   static int j, d, s;
public:
   const static int limit;

   static PROP_TYPE type(const int i)
   {
      return (PROP_TYPE)indices[i][0];
   }

   static int offset(const int i)
   {
      return indices[i][1];
   }
   ...
```

Las variables *j*, *d* y *s* se utilizarán para indexar secuencialmente las propiedades dentro de cada uno de los tres tipos diferentes. Así es como se hace en el método estático *calcIndices*.

```
static int calcIndices()
{
   const int size = fmax(boundary<I>(),
      fmax(boundary<D>(), boundary<S>())) + 1;
   ArrayResize(indices, size);
   j = d = s = 0;
   for(int i = 0; i < size; ++i)
   {
      if(detect<I>(i))
      {
         indices[i][0] = PROP_TYPE_INTEGER;
         indices[i][1] = j++;
      }
      else if(detect<D>(i))
      {
         indices[i][0] = PROP_TYPE_DOUBLE;
         indices[i][1] = d++;
      }
      else if(detect<S>(i))
      {
         indices[i][0] = PROP_TYPE_STRING;
         indices[i][1] = s++;
      }
      else
      {
         Print("Unresolved int value as enum: ", i, " ", typename(TradeState));
      }
   }
   return size;
}
```

El método *boundary* devuelve la constante máxima entre todos los elementos de la enumeración E dada.

```
template<typename E>
static int boundary(const E dummy = (E)NULL)
{
   int values[];
   const int n = EnumToArray(dummy, values, 0, 1000);
   ArraySort(values);
   return values[n - 1];
}
```

El mayor valor de los tres tipos de enumeraciones determina el rango de enteros que deben ordenarse de acuerdo con el tipo de propiedad al que pertenecen.

Aquí utilizamos el método *detect* que devuelve *true* si el entero es un elemento de una enumeración.

```
   template<typename E>
   static bool detect(const int v)
   {
      ResetLastError();
      const string s = EnumToString((E)v); // result is not used
      if(_LastError == 0) // only the absence of an error is important
      {
         return true;
      }
      return false;
   }
```

La última pregunta es cómo ejecutar este cálculo cuando se inicia el programa. Esto se consigue utilizando la naturaleza estática de las variables y el método.

```
template<typename I,typename D,typename S>
static int MonitorInterface::TradeState::indices[][2];
template<typename I,typename D,typename S>
static int MonitorInterface::TradeState::j,
   MonitorInterface::TradeState::d,
   MonitorInterface::TradeState::s;
template<typename I,typename D,typename S>
const static int MonitorInterface::TradeState::limit =
   MonitorInterface::TradeState::calcIndices();
```

Observe que *limit* se inicializa con el resultado de llamar a nuestra función *calcIndices*.

Teniendo una tabla con índices, implementamos el llenado de arrays con valores de propiedades en el método *cache*.

```
class TradeState
{
   ...
   TradeState(const MonitorInterface *ptr) : owner(ptr)
   {
      cache(); // when creating an object, immediately cache the properties
   }

   template<typename T>
   void _get(const int e, T &value) const // overload with record by reference
   {
      value = owner.get(e, value);
   }

   void cache()
   {
      ArrayResize(ulongs, j);
      ArrayResize(doubles, d);
      ArrayResize(strings, s);
      for(int i = 0; i < limit; ++i)
      {
         switch(indices[i][0])
         {
         case PROP_TYPE_INTEGER: _get(i, ulongs[indices[i][1]]); break;
         case PROP_TYPE_DOUBLE: _get(i, doubles[indices[i][1]]); break;
         case PROP_TYPE_STRING: _get(i, strings[indices[i][1]]); break;
         }
      }
   }
};
```

Hacemos un bucle a través de todo el rango de propiedades de 0 a *limit* y, dependiendo del tipo de propiedad en *indices[i][0]*, escribimos su valor en el elemento del array *ulongs*, *doubles* o *strings* bajo el número *indices[i][1]* (el elemento correspondiente del array se pasa por referencia al método *_get*).

Una llamada a *owner.get(e, value)* remite a uno de los métodos estándar de la clase monitor (aquí es visible como puntero abstracto *MonitorInterface*). En concreto, para las posiciones de la clase *PositionMonitor*, esto dará lugar a las llamadas *PositionGetInteger*, *PositionGetDouble,* o *PositionGetString*. El compilador elegirá el tipo correcto. Los monitores de órdenes y transacciones tienen sus propias implementaciones similares, que este código base incluye automáticamente.

Es lógico heredar la descripción de una instantánea de un objeto de trading de la clase monitor. Dado que tenemos que almacenar en caché órdenes, transacciones y posiciones, tiene sentido hacer de la nueva clase una plantilla y recopilar en ella todos los algoritmos comunes adecuados para todos los objetos. Llamémosla *TradeBaseState* (archivo*TradeState.mqh*).

```
template<typename M,typename I,typename D,typename S>
class TradeBaseState: public M
{
   M::TradeState state;
   bool cached;

public:
   TradeBaseState(const ulong t) : M(t), state(&this), cached(ready)
   {
   }

   void passthrough(const bool b)   // enable/disable cache as desired
   {
      cached = b;
   }
   ...
```

Una de las clases de monitor específicas descritas anteriormente se oculta bajo la letra M (*OrderMonitor.mqh*, *PositionMonitor.mqh*, *DealMonitor.mqh*). La base es el objeto de caché *state* de la recién introducida clase *M::TradeState*. Dependiendo de M, se formará en su interior una tabla de índices específica (una para la clase M) y se distribuirán arrays de propiedades (propias para cada instancia de M, es decir, para cada orden, transacción, posición).

La variable *cached* contiene una señal de si los arrays de *state* están llenos de valores de propiedades, y si se deben consultar las propiedades de un objeto para devolver valores de la caché. Esto será necesario más adelante para comparar los estados guardados y actuales.

En otras palabras: cuando *cached* se establece en *false*, el objeto se comportará como un monitor normal, leyendo propiedades del entorno de trading. Cuando *cached* es igual a *true*, el objeto devolverá los valores almacenados previamente de los arrays internos.

```
   virtual long get(const I property) const override
   {
      return cached ? state.ulongs[M::TradeState::offset(property)] : M::get(property
   }

   virtual double get(const D property) const override
   {
      return cached ? state.doubles[M::TradeState::offset(property)] : M::get(propert
   }

   virtual string get(const S property) const override
   {
      return cached ? state.strings[M::TradeState::offset(property)] : M::get(propert
   }
   ...
```

Por defecto, el almacenamiento en caché está, por supuesto, activado.

También debemos proporcionar un método que realice directamente el almacenamiento en caché (rellenado de arrays). Para ello, basta con llamar al método *cache* para el objeto *state*.

```
bool update()
{
   if(refresh())
   {
      cached = false; // disable reading from the cache
      state.cache();  // read real properties and write to cache
      cached = true;  // enable external cache access back
      return true;
   }
   return false;
}
```

¿Qué es el método *refresh*?

Hasta ahora hemos estado utilizando objetos monitor en modo simple: creándolos, leyendo propiedades y borrándolos. Al mismo tiempo, la lectura de propiedades supone que la orden, transacción o posición correspondiente se ha seleccionado en el contexto de trading (dentro del constructor). Como ahora estamos mejorando los monitores para admitir el estado interno, es necesario asegurarse de que el elemento deseado se reasigna para poder leer las propiedades incluso después de un tiempo indefinido (por supuesto, con una comprobación de que el elemento todavía existe). Para implementarlo, hemos añadido el método virtual *refresh* a la plantilla de la clase *MonitorInterface*.

```
// TradeBaseMonitor.mqh
template<typename I,typename D,typename S>
class MonitorInterface
{
   ...
   virtual bool refresh() = 0;
```

Debe devolver *true* al asignar con éxito una orden, transacción o posición. Si el resultado es *false*, la variable integrada *_LastError* debe contener uno de los siguientes errores:

- 4753 ERR_TRADE_POSITION_NOT_FOUND;
- 4754 ERR_TRADE_ORDER_NOT_FOUND;
- 4755 ERR_TRADE_DEAL_NOT_FOUND;

En este caso, la variable miembro *ready*, que señala la disponibilidad del objeto, debe restablecerse a *false* en las implementaciones de este método en las clases derivadas.

Por ejemplo, en el constructor *PositionMonitor*, teníamos y seguimos teniendo una inicialización de este tipo. La situación es similar a la de los monitores de órdenes y transacciones.

```
// PositionMonitor.mqh
   const ulong ticket;
   PositionMonitor(const ulong t): ticket(t)
   {
      if(!PositionSelectByTicket(ticket))
      {
         PrintFormat("Error: PositionSelectByTicket(%lld) failed: %s", ticket,
            E2S(_LastError));
      }
      else
      {
         ready = true;
      }
   }
   ...
```

Ahora añadiremos el método *refresh* a todas las clases específicas de este tipo (véase el ejemplo *PositionMonitor*):

```
// PositionMonitor.mqh
   virtual bool refresh() override
   {
      ready = PositionSelectByTicket(ticket);
      return ready;
   }
```

No obstante, rellenar arrays de caché con los valores de las propiedades es sólo la mitad de la batalla; la segunda parte consiste en comparar estos valores con el estado real de la orden, transacción o posición.

Para identificar las diferencias y escribir los índices de las propiedades modificadas en el array *changes*, la clase generada *TradeBaseState* proporciona el método *getChanges*. El método devuelve *true* cuando se detectan cambios.

```
template<typename M,typename I,typename D,typename S>
class TradeBaseState: public M
{
   ...
   bool getChanges(int &changes[])
   {
      const bool previous = ready;
      if(refresh())
      {
         // element is selected in the trading environment = properties can be read a
         cached = false;    // read directly
         const bool result = M::diff(state, changes);
         cached = true;     // turn cache back on by default
         return result;
      }
      // no longer "ready" = most likely deleted
      return previous != ready; // if just deleted, this is also a change
   }
```

Como puede ver, el trabajo principal se confía a un determinado método *diff* de la clase M. Se trata de un nuevo método: tenemos que escribirlo. Afortunadamente, gracias a la programación orientada a objetos (POO), puede hacer esto una vez en la plantilla base *MonitorInterface* y el método aparecerá inmediatamente para órdenes, transacciones y posiciones.

```
// TradeBaseMonitor.mqh
template<typename I,typename D,typename S>
class MonitorInterface
{
   ...
   bool diff(const TradeState &that, int &changes[])
   {
      ArrayResize(changes, 0);
      for(int i = 0; i < TradeState::limit; ++i)
      {
         switch(TradeState::indices[i][0])
         {
         case PROP_TYPE_INTEGER:
            if(this.get((I)i) != that.ulongs[TradeState::offset(i)])
            {
               PUSH(changes, i);
            }
            break;
         case PROP_TYPE_DOUBLE:
            if(!TU::Equal(this.get((D)i), that.doubles[TradeState::offset(i)]))
            {
               PUSH(changes, i);
            }
            break;
         case PROP_TYPE_STRING:
            if(this.get((S)i) != that.strings[TradeState::offset(i)])
            {
               PUSH(changes, i);
            }
            break;
         }
      }
      return ArraySize(changes) > 0;
   }
```

Así pues, todo está listo para formar clases de caché específicas para órdenes, transacciones y posiciones. Por ejemplo, las posiciones se almacenarán en el monitor ampliado *PositionState* en la base de *PositionMonitor*.

```
class PositionState: public TradeBaseState<PositionMonitor,
   ENUM_POSITION_PROPERTY_INTEGER,
   ENUM_POSITION_PROPERTY_DOUBLE,
   ENUM_POSITION_PROPERTY_STRING>
{
public:
   PositionState(const long t): TradeBaseState(t) { }
};
```

Del mismo modo, en el archivo *TradeState.mqh* se define una clase de caché para las transacciones.

```
class DealState: public TradeBaseState<DealMonitor,
   ENUM_DEAL_PROPERTY_INTEGER,
   ENUM_DEAL_PROPERTY_DOUBLE,
   ENUM_DEAL_PROPERTY_STRING>
{
public:
   DealState(const long t): TradeBaseState(t) { }
};
```

Con las órdenes, las cosas son un poco más complicadas, porque éstas pueden ser activas e históricas. Hasta ahora hemos tenido una clase de monitor genérica para órdenes, *OrderMonitor*, que intenta encontrar el ticket de orden presentado tanto entre las órdenes activas como en el historial. Este enfoque no es adecuado para el almacenamiento en caché, ya que los Asesores Expertos necesitan hacer un seguimiento de la transición de una orden de un estado a otro.

Por esta razón, añadimos 2 clases específicas más al archivo *OrderMonitor.mqh*: *ActiveOrderMonitor* y *HistoryOrderMonitor*.

```
// OrderMonitor.mqh
class ActiveOrderMonitor: public OrderMonitor
{
public:
   ActiveOrderMonitor(const ulong t): OrderMonitor(t)
   {
      if(history) // if the order is in history, then it is already inactive
      {
         ready = false;   // reset ready flag
         history = false; // this object is only for active orders by definition
      }
   }

   virtual bool refresh() override
   {
      ready = OrderSelect(ticket);
      return ready;
   }
};

class HistoryOrderMonitor: public OrderMonitor
{
public:
   HistoryOrderMonitor(const ulong t): OrderMonitor(t) { }

   virtual bool refresh() override
   {
      history = true; // work only with history
      ready = historyOrderSelectWeak(ticket);
      return ready; // readiness is determined by the presence of a ticket in the his
   }
};
```

Cada una de ellas busca un ticket sólo en su zona. Basándose en estos monitores, ya puede crear clases de caché.

```
                                                                    // TradeState.mqh
class OrderState: public TradeBaseState<ActiveOrderMonitor,
   ENUM_ORDER_PROPERTY_INTEGER,
   ENUM_ORDER_PROPERTY_DOUBLE,
   ENUM_ORDER_PROPERTY_STRING>
{
public:
   OrderState(const long t): TradeBaseState(t) { }
};

class HistoryOrderState: public TradeBaseState<HistoryOrderMonitor,
   ENUM_ORDER_PROPERTY_INTEGER,
   ENUM_ORDER_PROPERTY_DOUBLE,
   ENUM_ORDER_PROPERTY_STRING>
{
public:
   HistoryOrderState(const long t): TradeBaseState(t) { }
};
```

El toque final que añadiremos a la clase *TradeBaseState* por comodidad es un método especial para convertir un valor de propiedad en una cadena. Aunque existen varias versiones de los métodos *stringify* en el monitor, todos ellos «imprimirán» valores de la caché (si la variable miembro *cached* es igual a *true*) o valores del objeto original del entorno de trading (si *cached* es igual a *false*). Para visualizar las diferencias entre la caché y el objeto modificado (cuando dichas diferencias se encuentran), necesitamos leer simultáneamente el valor de la caché y pasar por alto la caché. En este sentido, añadimos el método *stringifyRaw* que siempre trabaja con la propiedad directamente (debido a que la variable *cached* se restablece y reinstala temporalmente).

```
   // get the string representation of the property 'i' bypassing the cache
   string stringifyRaw(const int i)
   {
      const bool previous = cached;
      cached = false;
      const string s = stringify(i);
      cached = previous;
   }
```

Vamos a comprobar el rendimiento del monitor de caché utilizando un ejemplo sencillo de un Asesor Experto que monitoriza el estado de una orden activa (*OrderSnapshot.mq5*). Más adelante desarrollaremos esta idea para almacenar en caché cualquier conjunto de órdenes, transacciones o posiciones, es decir, crearemos una caché completa.

El Asesor Experto intentará encontrar la última en la lista de órdenes activas y creará el objeto *OrderState* para ella. Si no hay órdenes, se pedirá al usuario que cree una orden o abra una posición (esto último se asocia a la colocación y ejecución de una orden en el mercado). En cuanto se encuentra una orden se comprueba si el estado de la misma ha cambiado. Esta comprobación se realiza en el manejador *OnTrade*. El Asesor Experto continuará supervisando esta orden hasta que se descargue.

```
int OnInit()
{
   if(OrdersTotal() == 0)
   {
      Alert("Please, create a pending order or open/close a position");
   }
   else
   {
      OnTrade(); // self-invocation
   }
   return INIT_SUCCEEDED;
}

void OnTrade()
{
   static int count = 0;
   // object pointer is stored in static AutoPtr
   static AutoPtr<OrderState> auto;
   // get a "clean" pointer (so as not to dereference auto[] everywhere)
   OrderState *state = auto[];

   PrintFormat(">>> OnTrade(%d)", count++);

   if(OrdersTotal() > 0 && state == NULL)
   {
      const ulong ticket = OrderGetTicket(OrdersTotal() - 1);
      auto = new OrderState(ticket);
      PrintFormat("Order picked up: %lld %s", ticket,
         auto[].isReady() ? "true" : "false");
      auto[].print(); // initial state at the time of "capturing" the order
   }
   else if(state)
   {
      int changes[];
      if(state.getChanges(changes))
      {
         Print("Order properties changed:");
         ArrayPrint(changes);
         ...
      }
      if(_LastError != 0) Print(E2S(_LastError));
   }
}
```

Además de mostrar un array de propiedades modificadas, estaría bien mostrar los cambios en sí. Por lo tanto, en lugar de una elipsis, añadiremos un fragmento de este tipo (nos será útil en futuras clases de cachés completas).

```
for(int k = 0; k < ArraySize(changes); ++k)
{
   switch(OrderState::TradeState::type(changes[k]))
   {
   case PROP_TYPE_INTEGER:
      Print(EnumToString((ENUM_ORDER_PROPERTY_INTEGER)changes[k]), ": ",
         state.stringify(changes[k]), " -> ",
         state.stringifyRaw(changes[k]));
         break;
   case PROP_TYPE_DOUBLE:
      Print(EnumToString((ENUM_ORDER_PROPERTY_DOUBLE)changes[k]), ": ",
         state.stringify(changes[k]), " -> ",
         state.stringifyRaw(changes[k]));
         break;
   case PROP_TYPE_STRING:
      Print(EnumToString((ENUM_ORDER_PROPERTY_STRING)changes[k]), ": ",
         state.stringify(changes[k]), " -> ",
         state.stringifyRaw(changes[k]));
         break;
   }
}
```

Aquí utilizamos el nuevo método *stringifyRaw*. Después de visualizar los cambios, no olvide actualizar el estado de la caché.

```
state.update();
```

Si ejecuta el Asesor Experto en una cuenta sin órdenes activas y coloca una nueva, verá las siguientes entradas en el registro (aquí *buy limit* para EURUSD se crea por debajo del precio de mercado actual).

```
Alert: Please, create a pending order or open/close a position
>>> OnTrade(0)
Order picked up: 1311736135 true
MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PR
ENUM_ORDER_PROPERTY_INTEGER Count=14
0 ORDER_TIME_SETUP=2022.04.11 11:42:39
1 ORDER_TIME_EXPIRATION=1970.01.01 00:00:00
2 ORDER_TIME_DONE=1970.01.01 00:00:00
3 ORDER_TYPE=ORDER_TYPE_BUY_LIMIT
4 ORDER_TYPE_FILLING=ORDER_FILLING_RETURN
5 ORDER_TYPE_TIME=ORDER_TIME_GTC
6 ORDER_STATE=ORDER_STATE_STARTED
7 ORDER_MAGIC=0
8 ORDER_POSITION_ID=0
9 ORDER_TIME_SETUP_MSC=2022.04.11 11:42:39'729
10 ORDER_TIME_DONE_MSC=1970.01.01 00:00:00'000
11 ORDER_POSITION_BY_ID=0
12 ORDER_TICKET=1311736135
13 ORDER_REASON=ORDER_REASON_CLIENT
ENUM_ORDER_PROPERTY_DOUBLE Count=7
0 ORDER_VOLUME_INITIAL=0.01
1 ORDER_VOLUME_CURRENT=0.01
2 ORDER_PRICE_OPEN=1.087
3 ORDER_PRICE_CURRENT=1.087
4 ORDER_PRICE_STOPLIMIT=0.0
5 ORDER_SL=0.0
6 ORDER_TP=0.0
ENUM_ORDER_PROPERTY_STRING Count=3
0 ORDER_SYMBOL=EURUSD
1 ORDER_COMMENT=
2 ORDER_EXTERNAL_ID=
>>> OnTrade(1)
Order properties changed:
10 14
ORDER_PRICE_CURRENT: 1.087 -> 1.09073
ORDER_STATE: ORDER_STATE_STARTED -> ORDER_STATE_PLACED
>>> OnTrade(2)
>>> OnTrade(3)
>>> OnTrade(4)
```

Aquí puede ver cómo el estado de la orden ha cambiado de INICIADA a CURSADA. Si, en lugar de una orden pendiente, hemos abierto en el mercado con un volumen pequeño, es posible que no tengamos tiempo de recibir estos cambios, porque tales órdenes, por regla general, se establecen muy rápidamente, y su estado observado cambia de INICIADO inmediatamente a RELLENADO. Y esto último ya significa que la orden ha pasado al historial. Por lo tanto, es necesario un seguimiento paralelo de historias para rastrearlas. Lo mostraremos en el siguiente ejemplo.

Tenga en cuenta que puede haber muchos eventos *OnTrade* pero no todos están relacionados con nuestra orden.

Intentemos establecer el nivel *Take Profit* y comprobemos el registro.

```
>>> OnTrade(5)
Order properties changed:
10 13
ORDER_PRICE_CURRENT: 1.09073 -> 1.09079
ORDER_TP: 0.0 -> 1.097
>>> OnTrade(6)
>>> OnTrade(7)
```

A continuación, cambie la fecha de vencimiento: de GTC a un día.

```
>>> OnTrade(8)
Order properties changed:
10
ORDER_PRICE_CURRENT: 1.09079 -> 1.09082
>>> OnTrade(9)
>>> OnTrade(10)
Order properties changed:
2 6
ORDER_TIME_EXPIRATION: 1970.01.01 00:00:00 -> 2022.04.11 00:00:00
ORDER_TYPE_TIME: ORDER_TIME_GTC -> ORDER_TIME_DAY
>>> OnTrade(11)
```

Aquí, en el proceso de cambio de nuestra orden, el precio tuvo tiempo suficiente de cambiar, y por lo tanto hemos «enganchado» una notificación intermedia sobre el nuevo valor en ORDER_PRICE_CURRENT. Y sólo después de eso, los cambios esperados en ORDER_TYPE_TIME y ORDER_TIME_EXPIRATION entraron en el registro.

A continuación, eliminamos la orden.

```
>>> OnTrade(12)
TRADE_ORDER_NOT_FOUND
```

Ahora, para cualquier acción con la cuenta que lleve a eventos *OnTrade*, nuestro Asesor Experto mostrará TRADE_ORDER_NOT_FOUND, ya que está diseñado para seguir una sola orden. Si el Asesor Experto se reinicia, «atrapará» otra orden si la hay. Pero dejaremos el Asesor Experto y empezaremos a prepararnos para una tarea más urgente.

Por regla general, el almacenamiento en caché y el control de los cambios no son necesarios para una sola orden o posición, sino para todas o un conjunto de ellas, seleccionadas en función de determinadas condiciones. Para ello, desarrollaremos una clase de plantilla base *TradeCache* (*TradeCache.mqh*) y, a partir de ella, crearemos clases aplicadas para listas de órdenes, transacciones y posiciones.

```
template<typename T,typename F,typename E>
class TradeCache
{
   AutoPtr<T> data[];
   const E property;
   const int NOT_FOUND_ERROR;

public:
   TradeCache(const E id, const int error): property(id), NOT_FOUND_ERROR(error) { }

   virtual string rtti() const
   {
      return typename(this); // will be redefined in derived classes for visual outpu
   }
   ...
```

En esta plantilla, la letra T indica una de las clases de la familia *TradeState*. Como puede ver, se reserva un array de tales objetos en forma de punteros automáticos con el nombre *data*.

La letra F describe el tipo de una de las clases de filtro (*OrderFilter.mqh*, incluyendo *HistoryOrderFilter*, *DealFilter.mqh*, *PositionFilter.mqh*) utilizado para seleccionar elementos en caché. En el caso más sencillo, cuando el filtro no contiene condiciones *let*, todos los elementos se almacenarán en caché (con respecto al historial de muestreo para objetos del historial).

La letra E corresponde a la enumeración en la que se encuentra la *property* que identifica los objetos. Dado que esta propiedad suele ser SOME_TICKET, se supone que la enumeración es un entero ENUM_SOMETHING_PROPERTY_INTEGER.

La variable NOT_FOUND_ERROR está destinada al código de error que se produce al intentar asignar un objeto inexistente para su lectura, por ejemplo, ERR_TRADE_POSITION_NOT_FOUND para posiciones.

En parámetros, el método de la clase principal *scan* recibe una referencia al filtro configurado (debe ser configurado por el código llamante).

```
   void scan(F &f)
   {
      const int existedBefore = ArraySize(data);

      ulong tickets[];
      ArrayResize(tickets, existedBefore);
      for(int i = 0; i < existedBefore; ++i)
      {
         tickets[i] = data[i][].get(property);
      }
      ...
```

Al principio del método recopilamos los identificadores de los objetos ya almacenados en caché en el array *tickets*. Obviamente, en la primera ejecución, éste estará vacío.

A continuación, rellenamos el array *objects* con entradas de objetos relevantes utilizando un filtro. Para cada nuevo ticket, creamos un objeto monitor de caché T y lo añadimos al array *data*. Para los objetos antiguos, analizamos la presencia de cambios llamando a *data[j][].getChanges(changes)* y luego actualizamos la caché llamando a *data[j][].update()*.

```
      ulong objects[];
      f.select(objects);
      for(int i = 0, j; i < ArraySize(objects); ++i)
      {
         const ulong ticket = objects[i];
         for(j = 0; j < existedBefore; ++j)
         {
            if(tickets[j] == ticket)
            {
               tickets[j] = 0; // mark as found
               break;
            }
         }

         if(j == existedBefore) // this is not in the cache, you need to add
         {
            const T *ptr = new T(ticket);
            PUSH(data, ptr);
            onAdded(*ptr);
         }
         else
         {
            ResetLastError();
            int changes[];
            if(data[j][].getChanges(changes))
            {
               onUpdated(data[j][], changes);
               data[j][].update();
            }
            if(_LastError) PrintFormat("%s: %lld (%s)", rtti(), ticket, E2S(_LastErro
         }
      }
      ...
```

Como puede ver, en cada fase del cambio, es decir, cuando se añade un objeto o después de modificarlo, se llama a los métodos *onAdded* y *onUpdated*. Se trata de métodos stub virtuales que el escáner puede utilizar para notificar al programa los eventos apropiados. Se espera que el código de la aplicación implemente una clase derivada con versiones anuladas de estos métodos. Abordaremos esta cuestión un poco más adelante, pero por ahora seguiremos considerando el método *scan*.

En el bucle anterior, todos los tickets encontrados en el array *tickets* se ponen a cero y, por lo tanto, los elementos restantes corresponden a los objetos que faltan del entorno de trading. A continuación, se comprueban llamando a *getChanges* y comparando el código de error con NOT_FOUND_ERROR. Si esto es cierto, se llama al método virtual *onRemoved*. Devuelve una bandera booleana (proporcionada por el código de la aplicación) que indica si el elemento debe eliminarse de la caché.

```
      for(int j = 0; j < existedBefore; ++j)
      {
         if(tickets[j] == 0) continue; // skip processed elements

         // this ticket was not found, most likely deleted
         int changes[];
         ResetLastError();
         if(data[j][].getChanges(changes))
         {
            if(_LastError == NOT_FOUND_ERROR) // for example, ERR_TRADE_POSITION_NOT_FOUND
            {
               if(onRemoved(data[j][]))
               {
                  data[j] = NULL;              // release the object and array element
               }
               continue;
            }

            // NB! Usually we shouldn't fall here
            PrintFormat("Unexpected ticket: %lld (%s) %s", tickets[j],
               E2S(_LastError), rtti());
            onUpdated(data[j][], changes, true);
            data[j][].update();
         }
         else
         {
            PrintFormat("Orphaned element: %lld (%s) %s", tickets[j],
               E2S(_LastError), rtti());
         }
      }
   }
```

Al final del método *scan*, el array *data* se despeja de elementos nulos, pero este fragmento se omite aquí por brevedad.

La clase base proporciona implementaciones estándar de los métodos *onAdded*, *onRemoved* y *onUpdated* que muestran la esencia de los eventos en el registro. Definiendo la macro PRINT_DETAILS en su código antes de incluir el archivo de encabezado *TradeCache.mqh* puede ordenar la impresión de todas las propiedades de cada objeto nuevo.

```
virtual void onAdded(const T &state)
{
   Print(rtti(), " added: ", state.get(property));
   #ifdef PRINT_DETAILS
   state.print();
   #endif
}

virtual bool onRemoved(const T &state)
{
   Print(rtti(), " removed: ", state.get(property));
   return true; // allow the object to be removed from the cache (false to save)
}

virtual void onUpdated(T &state, const int &changes[],
   const bool unexpected = false)
{
   ...
}
```

No presentaremos el método *onUpdated*, ya que prácticamente repite el código para la salida de cambios del Asesor Experto *OrderSnapshot.mq5* mostrado anteriormente.

Por supuesto, la clase base tiene facilidades para obtener el tamaño de la caché y acceder a un objeto específico por número.

```
int size() const
{
   return ArraySize(data);
}

T *operator[](int i) const
{
   return data[i][]; // return pointer (T*) from AutoPtr object
}
```

A partir de la clase base *TradeCache* podemos crear fácilmente ciertas clases para almacenar en caché listas de posiciones, órdenes activas y órdenes del historial. El almacenamiento en caché de las transacciones se deja como tarea independiente.

```
class PositionCache: public TradeCache<PositionState,PositionFilter,
   ENUM_POSITION_PROPERTY_INTEGER>
{
public:
   PositionCache(const ENUM_POSITION_PROPERTY_INTEGER selector = POSITION_TICKET,
      const int error = ERR_TRADE_POSITION_NOT_FOUND): TradeCache(selector, error) {
};

class OrderCache: public TradeCache<OrderState,OrderFilter,
   ENUM_ORDER_PROPERTY_INTEGER>
{
public:
   OrderCache(const ENUM_ORDER_PROPERTY_INTEGER selector = ORDER_TICKET,
      const int error = ERR_TRADE_ORDER_NOT_FOUND): TradeCache(selector, error) { }
};

class HistoryOrderCache: public TradeCache<HistoryOrderState,HistoryOrderFilter,
   ENUM_ORDER_PROPERTY_INTEGER>
{
public:
   HistoryOrderCache(const ENUM_ORDER_PROPERTY_INTEGER selector = ORDER_TICKET,
      const int error = ERR_TRADE_ORDER_NOT_FOUND): TradeCache(selector, error) { }
};
```

Para resumir el proceso de desarrollo de la funcionalidad presentada, ofrecemos un diagrama de las clases principales. Se trata de una versión simplificada de los diagramas de UML que puede resultar útil a la hora de diseñar programas complejos en MQL5.

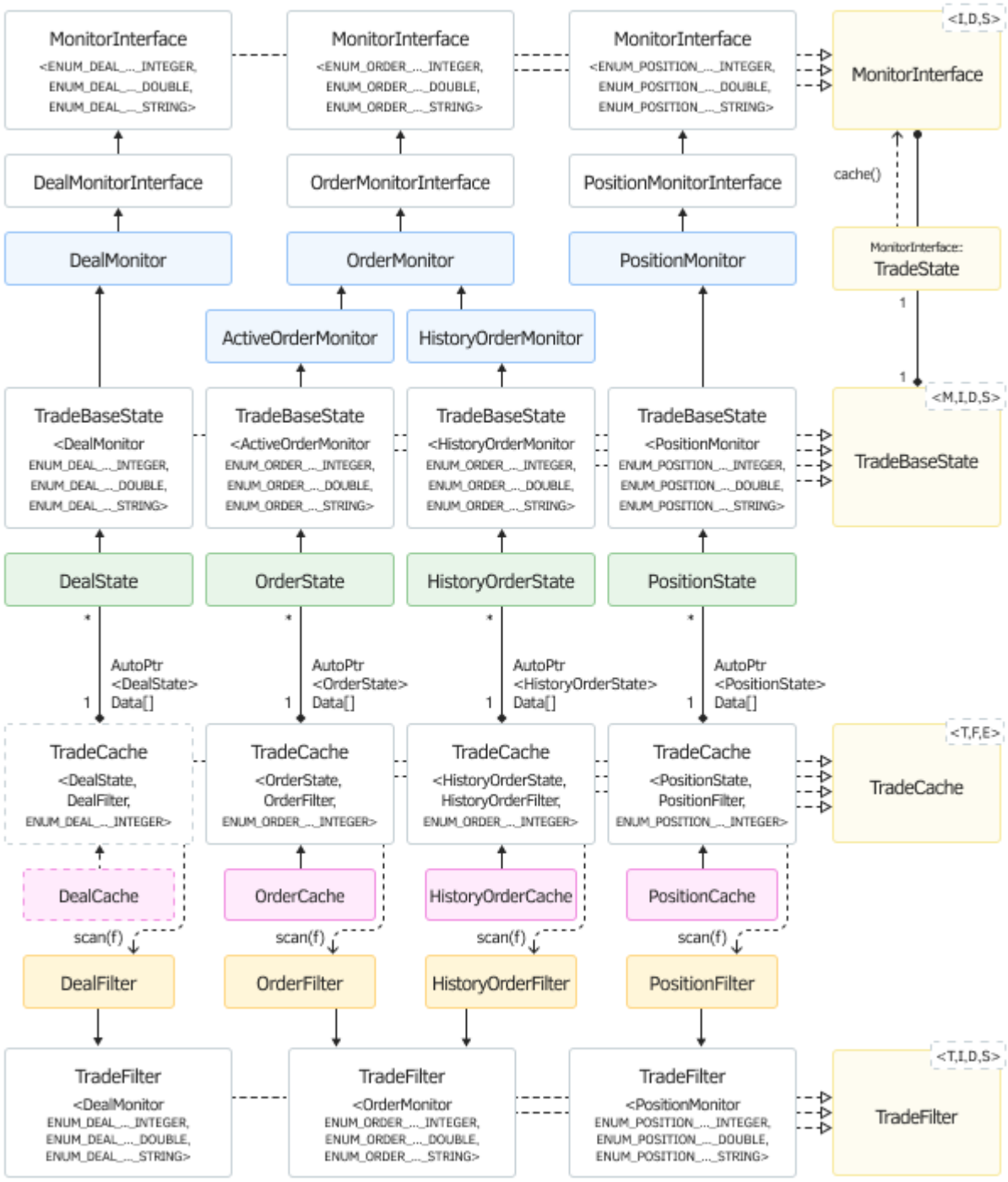


Diagrama de clases de monitores, filtros y cachés de objetos de trading

Las plantillas se marcan en amarillo, las clases abstractas se dejan en blanco y ciertas implementaciones se muestran en color. Las flechas sólidas con puntas rellenas indican herencia, y las flechas punteadas con puntas huecas indican clasificación de plantillas. Las flechas punteadas con puntas abiertas indican el uso de los métodos especificados entre sí por las clases. Las conexiones con diamantes son una composición (inclusión de unos objetos en otros).

Como ejemplo de uso de la caché, vamos a crear un Asesor Experto *TradeSnapshot.mq5*, que responderá a cualquier cambio en el entorno de trading desde el manejador *OnTrade*. Para el filtrado y el almacenamiento en caché, el código describe 6 objetos, 2 (filtro y caché) para cada tipo de elemento: posiciones, órdenes activas y órdenes históricas.

```
PositionFilter filter0;
PositionCache positions;

OrderFilter filter1;
OrderCache orders;

HistoryOrderFilter filter2;
HistoryOrderCache history;
```

No se establecen condiciones para los filtros a través de las llamadas al método *let* para que todos los objetos en línea descubiertos entren en la caché. Existe un ajuste adicional para las órdenes del historial.

Opcionalmente, al inicio, puede cargar en la caché órdenes anteriores con una profundidad de historial determinada. Esto puede hacerse a través de la variable de entrada *HistoryLookup*. En esta variable, puede seleccionar el último día, la última semana (por duración, no por calendario), el mes (30 días) o el año (360 días). Por defecto, el historial pasado no se carga (más concretamente, sólo se carga en 1 segundo). Dado que la macro PRINT_DETAILS está definida en el Asesor Experto, tenga cuidado con las cuentas con un gran historial: pueden generar un gran registro si no se limita el periodo.

```
enum ENUM_HISTORY_LOOKUP
{
   LOOKUP_NONE = 1,
   LOOKUP_DAY = 86400,
   LOOKUP_WEEK = 604800,
   LOOKUP_MONTH = 2419200,
   LOOKUP_YEAR = 29030400,
   LOOKUP_ALL = 0,
};

input ENUM_HISTORY_LOOKUP HistoryLookup = LOOKUP_NONE;

datetime origin;
```

En el manejador *OnInit*, reiniciamos las cachés (en caso de que el Asesor Experto se reinicie con nuevos parámetros), calculamos la fecha de inicio del historial en la variable *origin*, y llamamos a *OnTrade* por primera vez.

```
int OnInit()
{
   positions.reset();
   orders.reset();
   history.reset();
   origin = HistoryLookup ? TimeCurrent() - HistoryLookup : 0;

   OnTrade(); // self start
   return INIT_SUCCEEDED;
}
```

El manejador *OnTrade* es minimalista, ya que todas las complejidades están ahora ocultas dentro de las clases.

```
void OnTrade()
{
   static int count = 0;

   PrintFormat(">>> OnTrade(%d)", count++);
   positions.scan(filter0);
   orders.scan(filter1);
   // make a history selection just before using the filter
   // inside the 'scan' method
   HistorySelect(origin, LONG_MAX);
   history.scan(filter2);
   PrintFormat(">>> positions: %d, orders: %d, history: %d",
      positions.size(), orders.size(), history.size());
}
```

Inmediatamente después de lanzar el Asesor Experto en una cuenta limpia, veremos el siguiente mensaje:

```
>>> OnTrade(0)
>>> positions: 0, orders: 0, history: 0
```

Vamos a intentar ejecutar el caso de prueba más sencillo: vamos a comprar o vender en una cuenta «vacía» que no tiene posiciones abiertas ni órdenes pendientes. El registro incluirá los siguientes eventos (que ocurren casi instantáneamente).

En primer lugar, se detectará una orden activa.

```
>>> OnTrade(1)
OrderCache added: 1311792104
MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PR
ENUM_ORDER_PROPERTY_INTEGER Count=14
0 ORDER_TIME_SETUP=2022.04.11 12:34:51
1 ORDER_TIME_EXPIRATION=1970.01.01 00:00:00
2 ORDER_TIME_DONE=1970.01.01 00:00:00
3 ORDER_TYPE=ORDER_TYPE_BUY
4 ORDER_TYPE_FILLING=ORDER_FILLING_FOK
5 ORDER_TYPE_TIME=ORDER_TIME_GTC
6 ORDER_STATE=ORDER_STATE_STARTED
7 ORDER_MAGIC=0
8 ORDER_POSITION_ID=0
9 ORDER_TIME_SETUP_MSC=2022.04.11 12:34:51'096
10 ORDER_TIME_DONE_MSC=1970.01.01 00:00:00'000
11 ORDER_POSITION_BY_ID=0
12 ORDER_TICKET=1311792104
13 ORDER_REASON=ORDER_REASON_CLIENT
ENUM_ORDER_PROPERTY_DOUBLE Count=7
0 ORDER_VOLUME_INITIAL=0.01
1 ORDER_VOLUME_CURRENT=0.01
2 ORDER_PRICE_OPEN=1.09218
3 ORDER_PRICE_CURRENT=1.09218
4 ORDER_PRICE_STOPLIMIT=0.0
5 ORDER_SL=0.0
6 ORDER_TP=0.0
ENUM_ORDER_PROPERTY_STRING Count=3
0 ORDER_SYMBOL=EURUSD
1 ORDER_COMMENT=
2 ORDER_EXTERNAL_ID=
```

A continuación, esta orden pasará al historial (al mismo tiempo, cambiarán al menos el estado, el tiempo de ejecución y el ID de posición).

```
HistoryOrderCache added: 1311792104
MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PR
ENUM_ORDER_PROPERTY_INTEGER Count=14
0 ORDER_TIME_SETUP=2022.04.11 12:34:51
1 ORDER_TIME_EXPIRATION=1970.01.01 00:00:00
2 ORDER_TIME_DONE=2022.04.11 12:34:51
3 ORDER_TYPE=ORDER_TYPE_BUY
4 ORDER_TYPE_FILLING=ORDER_FILLING_FOK
5 ORDER_TYPE_TIME=ORDER_TIME_GTC
6 ORDER_STATE=ORDER_STATE_FILLED
7 ORDER_MAGIC=0
8 ORDER_POSITION_ID=1311792104
9 ORDER_TIME_SETUP_MSC=2022.04.11 12:34:51'096
10 ORDER_TIME_DONE_MSC=2022.04.11 12:34:51'097
11 ORDER_POSITION_BY_ID=0
12 ORDER_TICKET=1311792104
13 ORDER_REASON=ORDER_REASON_CLIENT
ENUM_ORDER_PROPERTY_DOUBLE Count=7
0 ORDER_VOLUME_INITIAL=0.01
1 ORDER_VOLUME_CURRENT=0.0
2 ORDER_PRICE_OPEN=1.09218
3 ORDER_PRICE_CURRENT=1.09218
4 ORDER_PRICE_STOPLIMIT=0.0
5 ORDER_SL=0.0
6 ORDER_TP=0.0
ENUM_ORDER_PROPERTY_STRING Count=3
0 ORDER_SYMBOL=EURUSD
1 ORDER_COMMENT=
2 ORDER_EXTERNAL_ID=
>>> positions: 0, orders: 1, history: 1
```

Obsérvese que estas modificaciones se han producido dentro de la misma llamada a *OnTrade*. En otras palabras: mientras nuestro programa analizaba las propiedades de la nueva orden (llamando a *orders.scan*), la orden era procesada por el terminal en paralelo, y para cuando se comprobó el historial (llamando a *history.scan*), ya había descendido en el historial. Por eso aparece tanto aquí como allí de acuerdo con la última línea de este fragmento de registro. Este comportamiento es normal en los programas multihilo y debe tenerse en cuenta a la hora de diseñarlos, pero no siempre tiene por qué ser así. Aquí simplemente llamamos la atención sobre ello. Cuando se ejecuta un programa MQL rápidamente, esta situación no suele ocurrir.

Si comprobáramos primero el historial y luego las órdenes en línea, en la primera fase podríamos encontrarnos con que la orden aún no está en el historial, y en la segunda fase con que la orden ya no está en línea. Es decir, teóricamente podría perderse por un momento. Una situación más realista es saltarse una orden en su fase activa debido a la sincronización del historial, es decir, fijarla inmediatamente por primera vez en el historial.

Recuerde que MQL5 no permite sincronizar el entorno de trading en su conjunto, sino sólo por partes:

- Entre las órdenes activas, la información es relevante para la orden para la que se acaba de llamar a la función *OrderSelect* o *OrderGetTicket*.
- Entre las posiciones, la información es relevante para la posición para la que se acaba de llamar a la función *PositionSelect*, *PositionSelectByTicket* o *PositionGetTicket*.
- Para las órdenes y transacciones del historial, la información está disponible en el contexto de la última llamada de *HistorySelect*, *HistorySelectByPosition HistoryOrderSelect*. *HistoryDealSelect*

Además, recordemos que los eventos de trading (como cualquier evento MQL5) son mensajes sobre cambios que se han producido, puestos en cola, y recuperado de la cola de forma diferida, y no inmediatamente en el momento de los cambios. Además, el evento *OnTrade* se produce después de los eventos *OnTradeTransaction* relevantes.

Pruebe diferentes configuraciones del programa, depure y genere registros detallados para elegir el algoritmo más fiable para su sistema de trading.

Volvamos a nuestro registro. En la siguiente activación de *OnTrade*, la situación ya se ha arreglado: la caché de órdenes activas ha detectado la eliminación de la orden. Por el camino, la caché de posiciones vio una posición abierta.

```
>>> OnTrade(2)
PositionCache added: 1311792104
MonitorInterface<ENUM_POSITION_PROPERTY_INTEGER,ENUM_POSITION_PROPERTY_DOUBLE,ENUM_PO
ENUM_POSITION_PROPERTY_INTEGER Count=9
0 POSITION_TIME=2022.04.11 12:34:51
1 POSITION_TYPE=POSITION_TYPE_BUY
2 POSITION_MAGIC=0
3 POSITION_IDENTIFIER=1311792104
4 POSITION_TIME_MSC=2022.04.11 12:34:51'097
5 POSITION_TIME_UPDATE=2022.04.11 12:34:51
6 POSITION_TIME_UPDATE_MSC=2022.04.11 12:34:51'097
7 POSITION_TICKET=1311792104
8 POSITION_REASON=POSITION_REASON_CLIENT
ENUM_POSITION_PROPERTY_DOUBLE Count=8
0 POSITION_VOLUME=0.01
1 POSITION_PRICE_OPEN=1.09218
2 POSITION_PRICE_CURRENT=1.09214
3 POSITION_SL=0.00000
4 POSITION_TP=0.00000
5 POSITION_COMMISSION=0.0
6 POSITION_SWAP=0.00
7 POSITION_PROFIT=-0.04
ENUM_POSITION_PROPERTY_STRING Count=3
0 POSITION_SYMBOL=EURUSD
1 POSITION_COMMENT=
2 POSITION_EXTERNAL_ID=
OrderCache removed: 1311792104
>>> positions: 1, orders: 0, history: 1
```

Al cabo de un tiempo, cerramos la posición. Como en nuestro código la caché de posición se comprueba primero (*positions.scan*), los cambios en la posición cerrada se incluyen en el registro.

```
>>> OnTrade(8)
PositionCache changed: 1311792104
POSITION_PRICE_CURRENT: 1.09214 -> 1.09222
POSITION_PROFIT: -0.04 -> 0.04
```

Más adelante, en la misma llamada de *OnTrade*, detectamos la aparición de una orden de cierre y su transferencia instantánea al historial (de nuevo, debido a su rápido procesamiento paralelo por parte del terminal).

```
OrderCache added: 1311796883
MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PR
ENUM_ORDER_PROPERTY_INTEGER Count=14
0 ORDER_TIME_SETUP=2022.04.11 12:39:55
1 ORDER_TIME_EXPIRATION=1970.01.01 00:00:00
2 ORDER_TIME_DONE=1970.01.01 00:00:00
3 ORDER_TYPE=ORDER_TYPE_SELL
4 ORDER_TYPE_FILLING=ORDER_FILLING_FOK
5 ORDER_TYPE_TIME=ORDER_TIME_GTC
6 ORDER_STATE=ORDER_STATE_STARTED
7 ORDER_MAGIC=0
8 ORDER_POSITION_ID=1311792104
9 ORDER_TIME_SETUP_MSC=2022.04.11 12:39:55'710
10 ORDER_TIME_DONE_MSC=1970.01.01 00:00:00'000
11 ORDER_POSITION_BY_ID=0
12 ORDER_TICKET=1311796883
13 ORDER_REASON=ORDER_REASON_CLIENT
ENUM_ORDER_PROPERTY_DOUBLE Count=7
0 ORDER_VOLUME_INITIAL=0.01
1 ORDER_VOLUME_CURRENT=0.01
2 ORDER_PRICE_OPEN=1.09222
3 ORDER_PRICE_CURRENT=1.09222
4 ORDER_PRICE_STOPLIMIT=0.0
5 ORDER_SL=0.0
6 ORDER_TP=0.0
ENUM_ORDER_PROPERTY_STRING Count=3
0 ORDER_SYMBOL=EURUSD
1 ORDER_COMMENT=
2 ORDER_EXTERNAL_ID=
HistoryOrderCache added: 1311796883
MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PR
ENUM_ORDER_PROPERTY_INTEGER Count=14
0 ORDER_TIME_SETUP=2022.04.11 12:39:55
1 ORDER_TIME_EXPIRATION=1970.01.01 00:00:00
2 ORDER_TIME_DONE=2022.04.11 12:39:55
3 ORDER_TYPE=ORDER_TYPE_SELL
4 ORDER_TYPE_FILLING=ORDER_FILLING_FOK
5 ORDER_TYPE_TIME=ORDER_TIME_GTC
6 ORDER_STATE=ORDER_STATE_FILLED
7 ORDER_MAGIC=0
8 ORDER_POSITION_ID=1311792104
9 ORDER_TIME_SETUP_MSC=2022.04.11 12:39:55'710
10 ORDER_TIME_DONE_MSC=2022.04.11 12:39:55'711
11 ORDER_POSITION_BY_ID=0
12 ORDER_TICKET=1311796883
13 ORDER_REASON=ORDER_REASON_CLIENT
ENUM_ORDER_PROPERTY_DOUBLE Count=7
0 ORDER_VOLUME_INITIAL=0.01
1 ORDER_VOLUME_CURRENT=0.0
2 ORDER_PRICE_OPEN=1.09222
3 ORDER_PRICE_CURRENT=1.09222
4 ORDER_PRICE_STOPLIMIT=0.0
5 ORDER_SL=0.0
6 ORDER_TP=0.0
```

```
ENUM_ORDER_PROPERTY_STRING Count=3
0 ORDER_SYMBOL=EURUSD
1 ORDER_COMMENT=
2 ORDER_EXTERNAL_ID=
>>> positions: 1, orders: 1, history: 2
```

Ya hay dos órdenes en la caché del historial, pero las cachés de posición y de órdenes activas que se analizaron antes de la caché del historial aún no han aplicado estos cambios.

No obstante, en el siguiente evento *OnTrade*, vemos que la posición está cerrada, y la orden de mercado ha desaparecido.

```
>>> OnTrade(9)
PositionCache removed: 1311792104
OrderCache removed: 1311796883
>>> positions: 0, orders: 0, history: 2
```

Si monitorizamos las cachés en cada tick (o una vez por segundo, pero no sólo para eventos OnTrade), veremos cambios en las propiedades ORDER_PRICE_CURRENT y POSITION_PRICE_CURRENT sobre la marcha. POSITION_PROFIT también cambiará.

Nuestras clases no tienen *persistence*, es decir, viven sólo en RAM y no saben cómo guardar y restaurar su estado en ningún almacenamiento a largo plazo, como archivos. Esto significa que el programa puede pasar por alto un cambio ocurrido entre sesiones de terminal. Si necesita esa funcionalidad, deberá implementarla usted mismo. En el futuro, en la Parte 7 del libro, veremos el soporte de base de datos SQLite integrado en MQL5, que proporciona la forma más eficiente y conveniente para almacenar la caché del entorno de trading y datos tabulares similares.

6.4.38 Crear Asesores Expertos multisímbolo

Hasta ahora, en el marco del libro, hemos analizado principalmente ejemplos de Asesores Expertos que operan en el símbolo de trabajo actual del gráfico. Sin embargo, MQL5 permite generar órdenes de trading para cualquier símbolo de *Observación de Mercado*, independientemente del símbolo de trabajo del gráfico.

De hecho, muchos de los ejemplos de las secciones anteriores tenían un parámetro de entrada *symbol*, en el que se puede especificar un símbolo arbitrario. Por defecto, hay una cadena vacía, que se trata como el símbolo actual del gráfico. Así, ya hemos visto los siguientes ejemplos:

- *CustomOrderSend.mq5* en Enviar una solicitud de trading
- *MarketOrderSend.mq5* en Operaciones de compraventa
- *MarketOrderSendMonitor.mq5* en Funciones para leer las propiedades de órdenes activas
- *PendingOrderSend.mq5* en Configurar una orden pendiente
- *PendingOrderModify.mq5* en Modificar una orden pendiente
- *PendingOrderDelete.mq5* en Borrar una orden pendiente

Puede intentar ejecutar estos ejemplos con un símbolo diferente y asegurarse de que las operaciones de trading se realizan exactamente igual que con el nativo.

Además, como vimos en la descripción de *OnBookEven* y *OnTradeTransaction*, estos eventos son universales e informan de los cambios en el entorno de trading relativos a símbolos arbitrarios. Pero esto no es cierto para el evento *OnTick* que sólo se genera cuando se produce un cambio en los nuevos

precios del símbolo actual. Normalmente, esto no es un problema, pero el trading multidivisa de alta frecuencia requiere que se tomen algunas medidas técnicas adicionales, como suscribirse a eventos *OnBookEvent* para otros símbolos o establecer un temporizador de alta frecuencia. Otra opción para eludir esta limitación en forma de indicador espía *EventTickSpy.mq5* se presentó en la sección Generación de eventos personalizados.

En el contexto de hablar sobre la compatibilidad del trading multisímbolo, hay que señalar que un concepto similar de Asesores Expertos de marco temporal múltiple no es del todo correcto. Operar en las nuevas horas de apertura de las barras es sólo un caso especial de agrupación de ticks por periodos arbitrarios, no necesariamente estándar. Por supuesto, el análisis de la aparición de una nueva barra en un marco temporal específico se simplifica por el núcleo del sistema debido a funciones como *iTime(_Symbol, PERIOD_XX, 0)*, pero este análisis se basa en ticks de todos modos.

Puede construir barras virtuales dentro de su Asesor Experto por el número de ticks (equivolume), por el rango de precios (renko, rango) y así sucesivamente. En algunos casos, incluso para mayor claridad, tiene sentido generar tales «marcos temporales» explícitamente fuera del Asesor Experto, en forma de símbolos personalizados. Pero este enfoque tiene sus limitaciones: hablaremos de ellas en la siguiente parte del libro.

No obstante, si el sistema de trading sigue requiriendo el análisis de las cotizaciones basada en la apertura de barras o utiliza un indicador multidivisa, habrá que esperar de algún modo a la sincronización de las barras de todos los instrumentos implicados. Proporcionamos un ejemplo de una clase que realiza esta tarea en la sección Seguimiento de formación de barras.

Al desarrollar un Asesor Experto multisímbolo, la tarea imperativa consiste en segregar un algoritmo de trading universal en bloques distintos. Estos bloques pueden aplicarse posteriormente a varios símbolos con distintos ajustes. El enfoque más lógico para lograrlo es articular una o varias clases en el marco del concepto de programación orientada a objetos (POO).

Ilustremos esta metodología con un ejemplo de Asesor Experto que emplea la conocida estrategia de martingala. Como se suele entender, la estrategia de martingala es intrínsecamente arriesgada, dada su práctica de duplicar los lotes después de cada operación perdedora en previsión de recuperar las pérdidas anteriores. Mitigar este riesgo es esencial, y un enfoque eficaz consiste en operar simultáneamente con varios símbolos, preferiblemente aquellos con correlaciones débiles. De este modo, las pérdidas temporales en un instrumento pueden compensarse, en potencia, con ganancias en otros.

La incorporación de una variedad de instrumentos (o diversas configuraciones dentro de un único sistema de trading, o incluso distintos sistemas de trading) dentro del Asesor Experto sirve para disminuir el impacto global de los fallos de componentes individuales. En esencia, cuanto mayor sea la diversidad de instrumentos o sistemas, menos dependerá el resultado final de los contratiempos aislados de sus partes constituyentes.

Llamemos a un nuevo Asesor Experto *MultiMartingale.mq5*. Los ajustes del algoritmo de trading incluyen:

- *UseTime* - bandera lógica para activar/desactivar el trading programado
- *HourStart* y *Hour End* - el intervalo de horas dentro del cual se permite el trading, si *UseTime* es igual a *true*
- *Lots* - el volumen de la primera transacción de la serie
- *Factor* - coeficiente de aumento del volumen de las transacciones posteriores a una pérdida

- *Limit* - el número máximo de operaciones en una serie perdedora con multiplicación de volúmenes (tras ella, vuelta al lote inicial)
- *Stop Loss* y *Take Profit* - distancia a los niveles de protección en puntos
- *StartType* - tipo de la primera transacción (compra o venta)
- *Trailing* - indicación de stop loss trailing

En el código fuente se describen de esta manera:

```
input bool UseTime = true;      // UseTime (hourStart and hourEnd)
input uint HourStart = 2;       // HourStart (0...23)
input uint HourEnd = 22;        // HourEnd (0...23)
input double Lots = 0.01;       // Lots (initial)
input double Factor = 2.0;      // Factor (lot multiplication)
input uint Limit = 5;           // Limit (max number of multiplications)
input uint StopLoss = 500;      // StopLoss (points)
input uint TakeProfit = 500;    // TakeProfit (points)
input ENUM_POSITION_TYPE StartType = 0; // StartType (first order type: BUY or SELL)
input bool Trailing = true;     // Trailing
```

En teoría, es lógico establecer niveles de protección no en puntos, sino en términos de acciones del indicador rango medio verdadero (Average True Range, ATR). Sin embargo, en la actualidad no es una tarea primordial.

Además, el Asesor Experto incorpora un mecanismo para detener temporalmente las operaciones de trading durante un tiempo especificado por el usuario (controlado por el parámetro *SkipTimeOnError*) en caso de errores. Omitiremos aquí una discusión detallada de este aspecto, ya que puede consultarse en los códigos fuente.

Para consolidar todo el conjunto de configuraciones en una entidad unificada, se define una estructura denominada *Settings*. Esta estructura tiene campos que reflejan variables de entrada. Además, la estructura incluye el campo *symbol*, que aborda la naturaleza multidivisa de la estrategia. En otras palabras: el símbolo puede ser arbitrario y difiere del símbolo de trabajo del gráfico.

```
struct Settings
{
   bool useTime;
   uint hourStart;
   uint hourEnd;
   double lots;
   double factor;
   uint limit;
   uint stopLoss;
   uint takeProfit;
   ENUM_POSITION_TYPE startType;
   ulong magic;
   bool trailing;
   string symbol;
   ...
};
```

En la fase inicial de desarrollo, rellenamos la estructura con variables de entrada. Sin embargo, esto sólo es suficiente para operar con un único símbolo. Posteriormente, a medida que ampliemos el

algoritmo para abarcar múltiples símbolos, tendremos que leer varios conjuntos de configuraciones (utilizando un enfoque diferente) y añadirlos a un array de estructuras.

La estructura también engloba varios métodos beneficiosos. En concreto, el método *validate* verifica la corrección de la configuración, confirmando la existencia del símbolo especificado, y devuelve un indicador de éxito (*true*).

```
struct Settings
{
   ...
   bool validate()
   {
 ...// checking the lot size and protective levels (see the source code)

      double rates[1];
      const bool success = CopyClose(symbol, PERIOD_CURRENT, 0, 1, rates) > -1;
      if(!success)
      {
         Print("Unknown symbol: ", symbol);
      }
      return success;
   }
   ...
};
```

La llamada a *CopyClose* no sólo comprueba si el símbolo está en línea en *Observación de Mercado*, sino que también inicia la carga de sus cotizaciones (del marco temporal deseado) y ticks en el probador. Si no se hace esto, sólo las cotizaciones y los ticks (en el modo de ticks reales) del instrumento y marco temporal actualmente seleccionados estarán disponibles en el probador por defecto. Como estamos escribiendo un Asesor Experto multidivisa, necesitaremos cotizaciones y ticks de terceros.

```
struct Settings
{
   ...
   void print() const
   {
      Print(symbol, (startType == POSITION_TYPE_BUY ? "+" : "-"), (float)lots,
        "*", (float)factor,
        "^", limit,
        "(", stopLoss, ",", takeProfit, ")",
        useTime ? "[" + (string)hourStart + "," + (string)hourEnd + "]": "");
   }
};
```

El método *print* envía todos los campos al registro de forma abreviada en una sola línea. Por ejemplo:

```
EURUSD+0.01*2.0^5(500,1000)[2,22]
|      | |   |   |  |    |   |  |
|      | |   |   |  |    |   |  `until this hour trading is allowed
|      | |   |   |  |    |   `from this hour trading is allowed
|      | |   |   |  |    `take profit in points
|      | |   |   |  `stop loss in points
|      | |   |   `maximum size of a series of losing trades (after '^')
|      | |   `lot multiplication factor (after '*')
|      | `initial lot in series
|      `+ start with Buy
|      `- start with Sell
`instrument
```

Necesitaremos otros métodos en la estructura *Settings* cuando pasemos a la multidivisa. Por ahora, imaginemos una versión simplificada de lo que podría ser el manejador *OnInit* del Asesor Experto que opera con un símbolo.

```
int OnInit()
{
   Settings settings =
   {
      UseTime, HourStart, HourEnd,
      Lots, Factor, Limit,
      StopLoss, TakeProfit,
      StartType, Magic, SkipTimeOnError, Trailing, _Symbol
   };

   if(settings.validate())
   {
      settings.print();
      ...
      // here you will need to initialize the trading algorithm with these settings
   }
   ...
}
```

De acuerdo con la programación orientada a objetos (POO), el sistema de trading debe describirse de forma generalizada como una interfaz de software. De nuevo, para simplificar el ejemplo, sólo utilizaremos un método en esta interfaz: *trade*.

```
interface TradingStrategy
{
   virtual bool trade(void);
};
```

La tarea principal del algoritmo es operar, y ni siquiera importa desde dónde decidamos llamar a este método: en cada tick desde *OnTick*, en la apertura de la barra, o posiblemente en el temporizador.

Lo más probable es que sus Asesores Expertos en funcionamiento necesiten métodos de interfaz adicionales para configurar y admitir varios modos. Pero en este ejemplo no son necesarios.

Vamos a empezar a crear una clase de un sistema de trading específico basado en la interfaz. En nuestro caso, todas las instancias serán de la clase *SimpleMartingale*. Sin embargo, también es posible implementar muchas clases diferentes que hereden la interfaz dentro de un Asesor Experto y luego

utilizarlas de manera uniforme en una combinación arbitraria. Una cartera de estrategias (preferiblemente de naturaleza muy diferente) suele caracterizarse por una mayor estabilidad del comportamiento financiero.

```
class SimpleMartingale: public TradingStrategy
{
protected:
   Settings settings;
   SymbolMonitor symbol;
   AutoPtr<PositionState> position;
   AutoPtr<TrailingStop> trailing;
   ...
};
```

Dentro de la clase, vemos una estructura *Settings* familiar y el monitor del símbolo de trabajo *SymbolMonitor*. Además, necesitaremos controlar la presencia de posiciones y seguir el nivel de stop-loss de las mismas, para lo que hemos introducido variables con punteros automáticos a objetos *PositionState* y *TrailingStop*. Los punteros automáticos nos permiten en nuestro código no preocuparnos por el borrado explícito de objetos ya que esto se hará automáticamente cuando el control salga del ámbito, o cuando un nuevo puntero sea asignado al puntero automático.

> La clase *TrailingStop* es una clase base, con la implementación más sencilla de seguimiento de precios, de la que se puede heredar una gran cantidad de algoritmos más complejos, un ejemplo de los cuales consideramos como un derivado *TrailingStopByMA*. Por lo tanto, para dar flexibilidad al programa en el futuro, es conveniente garantizar que el código de llamada pueda pasar su propio objeto de seguimiento específico y personalizado, derivado de *TrailingStop*. Esto se puede hacer, por ejemplo, pasando un puntero al constructor o convirtiendo *SimpleMartingale* en una clase de plantilla (entonces la clase de trailing será establecida por el parámetro de plantilla).
>
> Este principio de la programación orientada a objetos (POO) se denomina *dependency injection* y se utiliza ampliamente junto con muchos otros que hemos mencionado brevemente en la sección Fundamentos teóricos de la programación orientada a objetos: composición.

La configuración se pasa a la clase de estrategia como parámetro del constructor. A partir de ella, asignamos todas las variables internas.

```
class SimpleMartingale: public TradingStrategy
{
   ...
   double lotsStep;
   double lotsLimit;
   double takeProfit, stopLoss;
public:
   SimpleMartingale(const Settings &state) : symbol(state.symbol)
   {
      settings = state;
      const double point = symbol.get(SYMBOL_POINT);
      takeProfit = settings.takeProfit * point;
      stopLoss = settings.stopLoss * point;
      lotsLimit = settings.lots;
      lotsStep = symbol.get(SYMBOL_VOLUME_STEP);

      // calculate the maximum lot in the series (after a given number of multiplicat
      for(int pos = 0; pos < (int)settings.limit; pos++)
      {
         lotsLimit = MathFloor((lotsLimit * settings.factor) / lotsStep) * lotsStep;
      }

      double maxLot = symbol.get(SYMBOL_VOLUME_MAX);
      if(lotsLimit > maxLot)
      {
         lotsLimit = maxLot;
      }
      ...
```

A continuación, utilizamos el objeto *PositionFilter* para buscar las posiciones «propias» existentes (por el símbolo y número mágico). Si se encuentra una posición de este tipo, creamos para ella el objeto *PositionState* y, si es necesario, el objeto *TrailingStop*.

```
      PositionFilter positions;
      ulong tickets[];
      positions.let(POSITION_MAGIC, settings.magic).let(POSITION_SYMBOL, settings.sym
         .select(tickets);
      const int n = ArraySize(tickets);
      if(n > 1)
      {
         Alert(StringFormat("Too many positions: %d", n));
      }
      else if(n > 0)
      {
         position = new PositionState(tickets[0]);
         if(settings.stopLoss && settings.trailing)
         {
           trailing = new TrailingStop(tickets[0], settings.stopLoss,
              ((int)symbol.get(SYMBOL_SPREAD) + 1) * 2);
         }
      }
   }
```

Las operaciones de programación se dejarán por ahora «entre bastidores» en el método *trade* (campos de parámetros *useTime*, *hourStart* y *hourEnd*). Pasemos directamente al algoritmo de trading.

Si todavía no hay ni ha habido ninguna posición, el puntero *PositionState* será cero, y tendremos que abrir una posición larga o corta de acuerdo con la dirección seleccionada *startType*.

```
   virtual bool trade() override
   {
      ...
      ulong ticket = 0;

      if(position[] == NULL)
      {
         if(settings.startType == POSITION_TYPE_BUY)
         {
            ticket = openBuy(settings.lots);
         }
         else
         {
            ticket = openSell(settings.lots);
         }
      }
      ...
```

Aquí se utilizan los métodos de ayuda *openBuy* y *openSell*. Hablaremos de ellos en un par de párrafos. Por ahora, sólo necesitamos saber que devuelven el número de ticket en caso de éxito o 0 en caso de fallo.

Si el objeto *position* ya contiene información sobre la posición rastreada, comprobamos si está activa llamando a *refresh*. En caso de éxito (*true*), actualice la información de la posición llamando a *update* y rastree también el stop loss, si fue solicitado por los ajustes.

```
else // position[] != NULL
{
   if(position[].refresh()) // does position still exists?
   {
      position[].update();
      if(trailing[]) trailing[].trail();
   }
   ...
```

Si se cierra la posición, *refresh* devolverá *false*, y estaremos en otra rama *if* para abrir una nueva posición: en la misma dirección, si se fijó un beneficio, o en dirección opuesta, si se produjo una pérdida. Tenga en cuenta que todavía tenemos una instantánea de la posición anterior en la caché.

```
   else // the position is closed - you need to open a new one
   {
      if(position[].get(POSITION_PROFIT) >= 0.0)
      {
        // keep the same direction:
        // BUY in case of profitable previous BUY
        // SELL in case of profitable previous SELL
         if(position[].get(POSITION_TYPE) == POSITION_TYPE_BUY)
            ticket = openBuy(settings.lots);
         else
            ticket = openSell(settings.lots);
      }
      else
      {
         // increase the lot within the specified limits
         double lots = MathFloor((position[].get(POSITION_VOLUME) * settings.fa

         if(lotsLimit < lots)
         {
            lots = settings.lots;
         }

         // change the trade direction:
         // SELL in case of previous unprofitable BUY
         // BUY in case of previous unprofitable SELL
         if(position[].get(POSITION_TYPE) == POSITION_TYPE_BUY)
            ticket = openSell(lots);
         else
            ticket = openBuy(lots);
      }
   }
}
...
```

La presencia de un ticket distinto de cero en esta etapa final significa que debemos empezar a controlarlo con los nuevos objetos *PositionState* y *TrailingStop*.

```
      if(ticket > 0)
      {
         position = new PositionState(ticket);
         if(settings.stopLoss && settings.trailing)
         {
            trailing = new TrailingStop(ticket, settings.stopLoss,
               ((int)symbol.get(SYMBOL_SPREAD) + 1) * 2);
         }
      }

      return true;
   }
```

A continuación presentamos, con algunas abreviaturas, el método *openBuy* (*openSell* es lo mismo), que consta de tres pasos:

- Preparación de la estructura *MqlTradeRequestSync* usando el método *prepare* (no se muestra aquí, rellena *deviation* y *magic*);
- Envío de una orden mediante una llamada al método *request.buy*;
- Comprobación del resultado con el método *postprocess* (no se muestra aquí, llama a *request.completed* y, en caso de error, comienza el periodo de suspensión del trading a la espera de mejores condiciones).

```
ulong openBuy(double lots)
{
   const double price = symbol.get(SYMBOL_ASK);

   MqlTradeRequestSync request;
   prepare(request);
   if(request.buy(settings.symbol, lots, price,
      stopLoss ? price - stopLoss : 0,
      takeProfit ? price + takeProfit : 0))
   {
      return postprocess(request);
   }
   return 0;
}
```

Normalmente, las posiciones se cierran mediante stop loss o take profit. No obstante, admitimos las operaciones programadas que puedan provocar cierres. Volvamos al principio del método *trade* para programar el trabajo.

```
virtual bool trade() override
{
   if(settings.useTime && !scheduled(TimeCurrent())) // time out of schedule?
   {
      // if there is an open position, close it
      if(position[] && position[].isReady())
      {
         if(close(position[].get(POSITION_TICKET)))
         {
                              // at the request of the designer:
            position = NULL; // clear the cache or we could...
            // do not do this zeroing, that is, save the position in the cache,
            // to transfer the direction and lot of the next trade to a new series
         }
         else
         {
            position[].refresh(); // guaranteeing reset of the 'ready' flag
         }
      }
      return false;
   }
   ...// opening positions (given above)
}
```

El método de trabajo *close* es muy similar al de *openBuy*, por lo que no lo consideraremos aquí. Otro método, *scheduled*, sólo devuelve *true* o *false*, dependiendo de si la hora actual cae dentro del rango de horas de trabajo especificado (*hourStart*, *hourEnd*).

Así pues, la clase de trading está lista. No obstante, para trabajar con varias divisas, tendrá que crear varias copias de la misma, que gestionará la clase *TradingStrategyPool*, en la que describimos un array de punteros a *TradingStrategy* y métodos para reponerla: constructor paramétrico y *push*.

```
class TradingStrategyPool: public TradingStrategy
{
private:
   AutoPtr<TradingStrategy> pool[];
public:
   TradingStrategyPool(const int reserve = 0)
   {
      ArrayResize(pool, 0, reserve);
   }

   TradingStrategyPool(TradingStrategy *instance)
   {
      push(instance);
   }

   void push(TradingStrategy *instance)
   {
      int n = ArraySize(pool);
      ArrayResize(pool, n + 1);
      pool[n] = instance;
   }

   virtual bool trade() override
   {
      for(int i = 0; i < ArraySize(pool); i++)
      {
         pool[i][].trade();
      }
      return true;
   }
};
```

No es necesario hacer el pool derivado de la interfaz *TradingStrategy*, pero si lo hacemos, esto permite el futuro empaquetamiento de pools de estrategias en otros pools de estrategias más grandes, y así sucesivamente. El método *trade* simplemente llama al mismo método en todos los objetos de array.

En el contexto global, vamos a añadir un puntero automático al pool de trading, y en el manejador *OnInit* nos aseguraremos de su llenado. Podemos empezar con una única estrategia (abordaremos la multidivisa un poco más adelante).

```
AutoPtr<TradingStrategyPool> pool;

int OnInit()
{
   ... // settings initialization was given earlier
   if(settings.validate())
   {
      settings.print();
      pool = new TradingStrategyPool(new SimpleMartingale(settings));
      return INIT_SUCCEEDED;
   }
   else
   {
      return INIT_FAILED;
   }
   ...
}
```

Para empezar a operar, sólo tenemos que escribir el pequeño manejador *OnTick* siguiente.

```
void OnTick()
{
   if(pool[] != NULL)
   {
      pool[].trade();
   }
}
```

Pero, ¿qué hay de la compatibilidad multidivisa?

El conjunto actual de parámetros de entrada está diseñado para un solo instrumento. Podemos utilizar esto para probar y optimizar el Asesor Experto en un único símbolo, pero después de que los ajustes óptimos se encuentran para todos los símbolos, tienen que combinarse y pasarse al algoritmo de alguna manera.

En este caso, aplicamos la solución más sencilla. El código anterior contenía una línea con los ajustes formados por el método *print* generado por las estructuras *Settings*. Implementamos el método en la estructura *parse* que realiza la operación inversa: restaura el estado de los campos por la descripción de la línea. Además, como necesitamos concatenar varias configuraciones para distintos caracteres, acordaremos que se concatenen en una sola cadena larga mediante un carácter delimitador especial, por ejemplo ';'. Entonces es fácil escribir el método estático *parseAll* para leer el conjunto fusionado de configuraciones, que llamará a *parse* para llenar el array de estructuras *Settings* pasadas por referencia. El código fuente completo de los métodos se encuentra en el archivo adjunto.

```
struct Settings
{
   ...
   bool parse(const string &line);
   void static parseAll(const string &line, Settings &settings[])
   ...
};
```

Por ejemplo, la siguiente cadena concatenada contiene ajustes para tres símbolos:

```
EURUSD+0.01*2.0^7(500,500)[2,22];AUDJPY+0.01*2.0^8(300,500)[2,22];GBPCHF+0.01*1.7^8(1
```

El método *parseAll* puede analizar este tipo de líneas. Para introducir una cadena de este tipo en el Asesor Experto, describimos la variable de entrada *WorkSymbols*.

```
input string WorkSymbols = ""; // WorkSymbols (name±lots*factor^limit(sl,tp)[start,stop];...)
```

Si está vacía, el Asesor Experto trabajará con los ajustes de las variables de entrada individuales presentadas anteriormente. Si se especifica la cadena, el manejador de *OnInit* rellenará el pool de sistemas de trading basándose en los resultados del análisis de esta línea.

```
int OnInit()
{
   if(WorkSymbols == "")
   {
      ...// work with the current single character, as before
   }
   else
   {
      Print("Parsed settings:");
      Settings settings[];
      Settings::parseAll(WorkSymbols, settings);
      const int n = ArraySize(settings);
      pool = new TradingStrategyPool(n);
      for(int i = 0; i < n; i++)
      {
         settings[i].trailing = Trailing;
         // support multiple systems on one symbol for hedging accounts
         settings[i].magic = Magic + i;  // different magic numbers for each subsyste
         pool[].push(new SimpleMartingale(settings[i]));
      }
   }
   return INIT_SUCCEEDED;
}
```

Es importante tener en cuenta que en MQL5, la longitud de la cadena de entrada está restringida a 250 caracteres. Además, durante la optimización en el probador, las cadenas se truncan aún más hasta un máximo de 63 caracteres. En consecuencia, para optimizar el trading concurrente a través de numerosos símbolos, se hace imperativo idear un método alternativo para cargar las configuraciones, como recuperarlas de un archivo de texto. Esto puede lograrse fácilmente utilizando la misma variable de entrada, siempre que se designe con un nombre de archivo en lugar de una cadena que contenga ajustes.

Este enfoque se aplica en el método *Settings::parseAll* mencionado. El nombre del archivo de texto en el que una cadena de entrada se pasará al Asesor Experto sin limitación de longitud se establece de acuerdo con el principio universal adecuado para todos los casos similares: el nombre del archivo comienza con el nombre del Asesor Experto, y luego, después del guion, debe haber el nombre de la variable cuyos datos contiene el archivo. Por ejemplo, en nuestro caso, en la variable de entrada *WorkSymbols*, puede especificar opcionalmente el nombre de archivo «MultiMartingale-WorkSymbols.txt». A continuación, el método *parseAll* intentará leer el texto del archivo (debe estar en la «sandbox» *MQL5/Files* estándar).

> El paso de nombres de archivos en los parámetros de entrada requiere que se tomen medidas adicionales para las simulaciones posteriores y la optimización de dicho Asesor Experto: la directiva

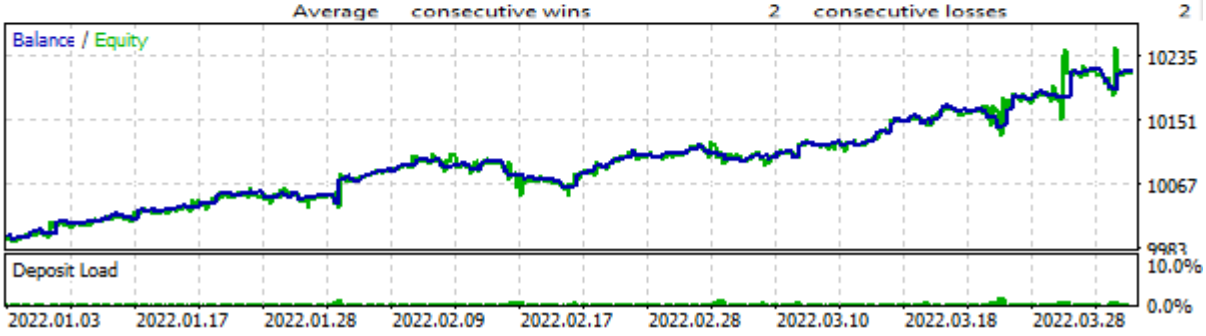
#property tester_file "MultiMartingale-WorkSymbols.txt" debe añadirse al código fuente. Esto se abordará en detalle en la sección Directivas del preprocesador para el probador. Cuando se añada esta directiva, el Asesor Experto requerirá la presencia del archivo y no se iniciará sin él en el probador.

El Asesor Experto está listo. Podemos probarlo en diferentes símbolos por separado, elegir la mejor configuración para cada uno y construir una cartera de trading. En el próximo capítulo, estudiaremos la API del probador, incluida la optimización, y este Asesor Experto nos resultará muy útil. Mientras tanto, vamos a comprobar su funcionamiento multidivisa.

```
WorkSymbols=EURUSD+0.01*1.2^4(300,600)[9,11];GBPCHF+0.01*2.0^7(300,400)[14,16];AUDJPY
```

En el primer trimestre de 2022, recibiremos el siguiente informe (los informes de MetaTrader 5 no proporcionan estadísticas desglosadas por símbolos, por lo que es posible distinguir un informe de una sola divisa de uno de varias divisas sólo por la tabla de transacciones/órdenes/posiciones).

History Quality	100%				
Bars	1534	Ticks	6281503	Symbols	3
Initial Deposit	10 000.00				
Total Net Profit	214.73	Balance Drawdow...	4.01	Equity Drawdown ...	3.97
Gross Profit	548.08	Balance Drawdow...	38.72 (0.38%)	Equity Drawdown ...	58.27 (0.57%)
Gross Loss	-333.35	Balance Drawdow...	0.38% (38.72)	Equity Drawdown ...	0.57% (58.27)
Profit Factor	1.64	Expected Payoff	0.74	Margin Level	7590.27%
Recovery Factor	3.69	Sharpe Ratio	0.15	Z-Score	0.09 (7.17%)
AHPR	1.0001 (0.01...	LR Correlation	0.96	OnTester result	0
GHPR	1.0001 (0.01...	LR Standard Error	15.45		
Total Trades	291	Short Trades (won ...	66 (53.03%)	Long Trades (won ...	225 (53.78%)
Total Deals	582	Profit Trades (% of...	156 (53.61%)	Loss Trades (% of t...	135 (46.39%)
	Largest	profit trade	35.23	loss trade	-13.34
	Average	profit trade	3.51	loss trade	-2.47
	Maximum	consecutive wins ($)	7 (7.21)	consecutive losses...	7 (-23.19)
	Maximal	consecutive profit ...	36.11 (4)	consecutive loss (c...	-26.37 (5)
	Average	consecutive wins	2	consecutive losses	2



Informe del probador para un Asesor Experto de estrategia de martingala multidivisa

Debe tenerse en cuenta que, debido al hecho de que la estrategia se lanza desde el manejador *OnTick*, las ejecuciones en diferentes símbolos principales (es decir, los seleccionados en la lista desplegable de ajustes del probador) darán resultados ligeramente diferentes. En nuestra prueba, nos limitamos a utilizar EURUSD como el instrumento más líquido y con mayor frecuencia de ticks, lo que es suficiente para la mayoría de las aplicaciones. No obstante, si desea reaccionar a los ticks de todos los instrumentos, puede utilizar un indicador como *EventTickSpy.mq5*. Opcionalmente, puede ejecutar la lógica de trading en un temporizador sin estar vinculada a los ticks de un instrumento específico.

Y este es el aspecto de la estrategia de trading para un solo símbolo, en este caso AUDJPY.

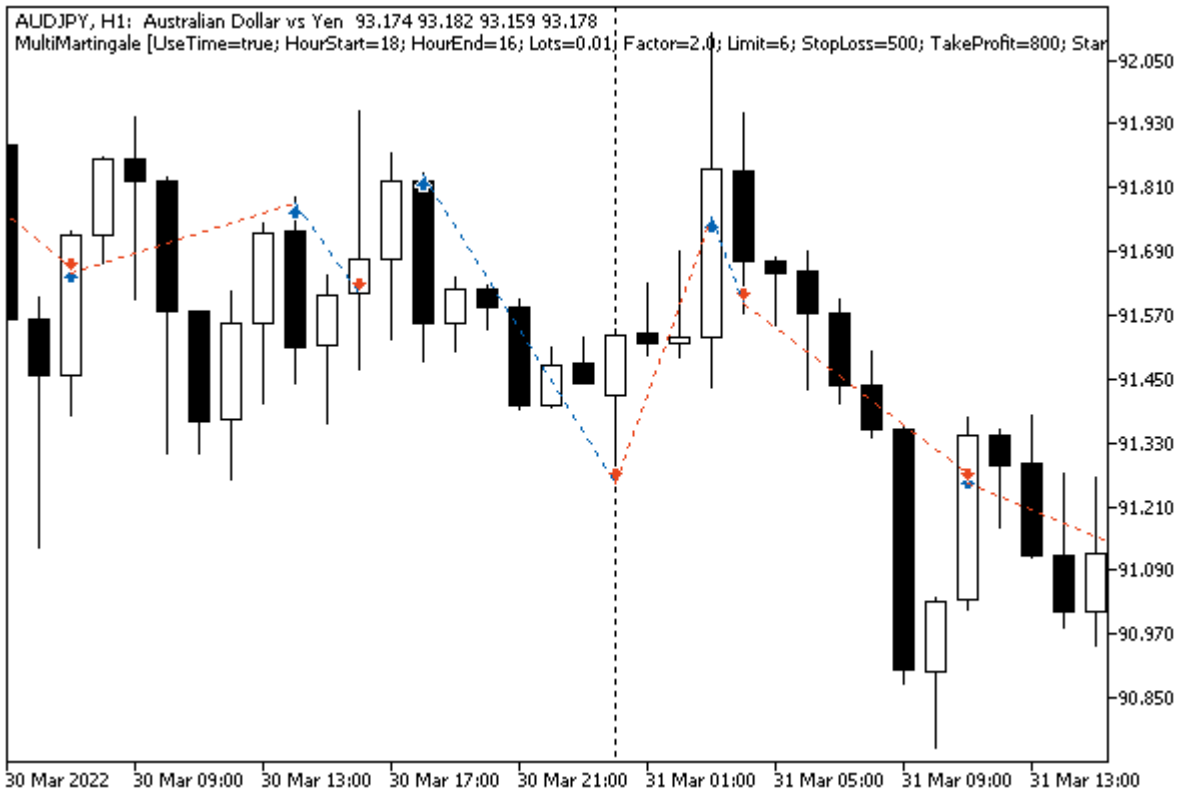


Gráfico con una prueba de un Asesor Experto en estrategia de martingala multidivisa

Por cierto, para todos los Asesores Expertos multidivisa, hay otra cuestión importante que queda desatendida aquí. Nos referimos al método de selección del tamaño del lote, por ejemplo, en función de la carga del depósito o del riesgo. Anteriormente, mostramos ejemplos de este tipo de cálculos en un Asesor Experto no de trading *LotMarginExposureTable.mq5*. En *MultiMartingale.mq5* hemos simplificado la tarea eligiendo un lote fijo y mostrándolo en los ajustes de cada símbolo. Sin embargo, en los Asesores Expertos operacionales multidivisa, tiene sentido elegir lotes en proporción al valor de los instrumentos (por margen o volatilidad).

Para concluir, me gustaría señalar que las estrategias multidivisa pueden requerir diferentes principios de optimización. La estrategia considerada permite encontrar por separado los parámetros de los símbolos y luego combinarlos. No obstante, algunas estrategias de arbitraje y agrupación (por ejemplo, el trading de pares) se basan en el análisis simultáneo de todas las herramientas para tomar decisiones de trading. En este caso, los ajustes asociados a todos los símbolos deben incluirse por separado en los parámetros de entrada.

6.4.39 Limitaciones y ventajas de los Asesores Expertos

Debido a su funcionamiento específico, los Asesores Expertos tienen algunas limitaciones, así como ventajas sobre otros tipos de programas MQL. En concreto, todas las funciones destinadas a los indicadores están prohibidas en los Asesores Expertos:

- *SetIndexBuffer*
- *IndicatorSetDouble*
- *IndicatorSetInteger*
- *IndicatorSetString*
- *PlotIndexSetDouble*

- *PlotIndexSetInteger*
- *PlotIndexSetString*
- *PlotIndexGetInteger*

Además, los Asesores Expertos no deben describir manejadores de eventos que son típicos para otros tipos de programas: *OnStart* (scripts y servicios) y *OnCalculate* (indicadores).

A diferencia de los indicadores, sólo se puede colocar un Asesor Experto en cada gráfico.

Al mismo tiempo, los Asesores Expertos son el único tipo de programas MQL que además de probarse (lo que ya hemos hecho tanto para los indicadores como para los Asesores Expertos), también pueden optimizarse. El optimizador permite encontrar los mejores parámetros de entrada en función de varios criterios, tanto de trading como matemáticos abstractos. Para estos fines, la API incluye funciones adicionales y varios manejadores de eventos específicos. Estudiaremos este material en el próximo capítulo.

Además, en los Asesores Expertos (así como en los scripts y servicios, es decir, en todos los tipos de programas excepto los indicadores) están disponibles grupos de funciones MQL5 integradas para trabajar con la red a nivel de socket y varios protocolos de Internet (HTTP, FTP, SMTP). Las estudiaremos en la séptima parte del libro.

6.4.40 Crear Asesores Expertos en el Asistente MQL

Por lo tanto, estamos completando el estudio de las API de trading para el desarrollo de Asesores Expertos. A lo largo de este capítulo hemos considerado varios ejemplos, que puede utilizar como punto de partida para su propio proyecto. Sin embargo, si quiere empezar un Asesor Experto desde cero, no tiene que hacerlo literalmente «desde cero». El MetaEditor proporciona el Asistente MQL integrado que, entre otras cosas, permite la creación de plantillas de Asesor Experto. Además, en el caso del Asesor Experto, este Asistente ofrece dos formas diferentes de generar el código fuente.

Ya conocimos el primer paso del Asistente en la sección Asistente MQL y borrador del programa. Obviamente, en el primer paso, seleccionamos el tipo de proyecto que se va a crear. En el capítulo anterior creamos una plantilla de script. Más adelante, en el capítulo dedicado a los indicadores, hicimos un recorrido por la creación de un modelo de indicador. Ahora consideraremos las dos opciones siguientes:

- Asesor Experto (plantilla)
- Asesor Experto (generar)

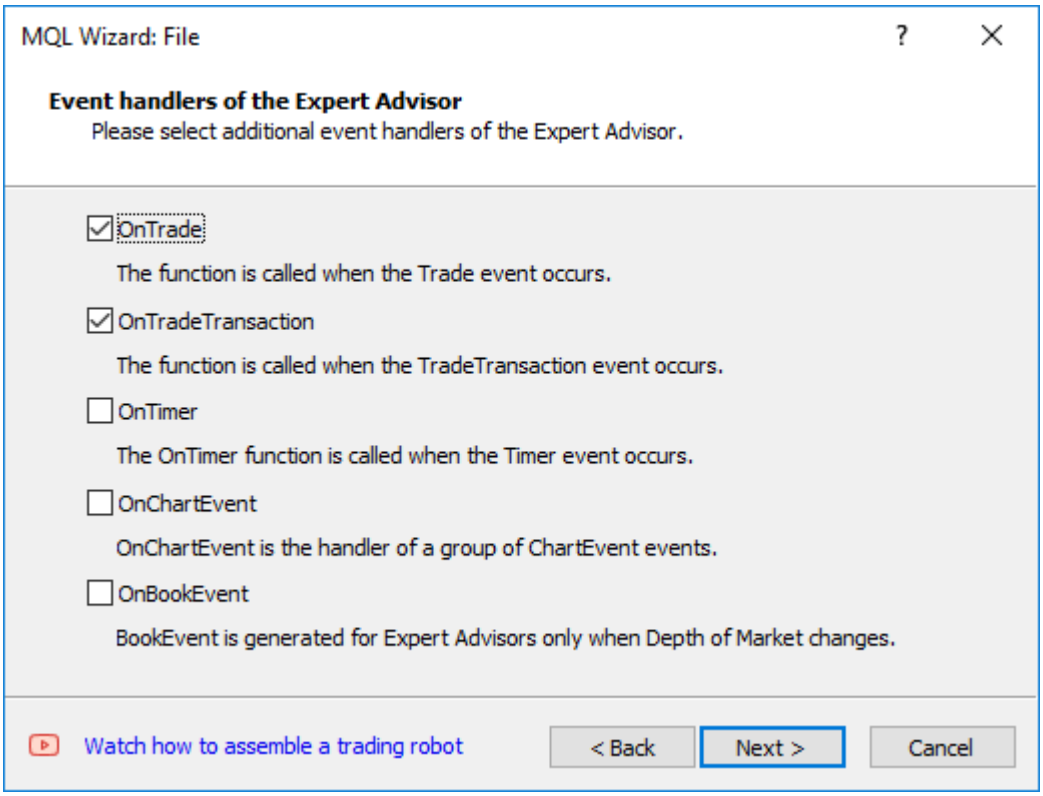
La primera es más sencilla. Puede seleccionar un nombre, parámetros de entrada y manejadores de eventos requeridos, como se muestra en las capturas de pantalla que se muestran a continuación, pero no habrá lógica de trading ni algoritmos ya hechos en el archivo fuente resultante.

La segunda opción es más complicada. Dará como resultado un Asesor Experto ya hecho basado en la biblioteca estándar que proporciona un conjunto de clases en archivos de encabezado disponibles en el paquete estándar de MetaTrader 5. Los archivos se encuentran en las carpetas *MQL5/Include/Expert/*, *MQL5/Include/Trade*, *MQL5/Include/Indicators* y varias más. Las clases de la biblioteca implementan las señales de los indicadores más populares, mecanismos para realizar operaciones de trading basadas en combinaciones de señales, así como algoritmos de gestión de fondos y trailing stop. El estudio detallado de la biblioteca estándar queda fuera del alcance de este libro.

Independientemente de las opciones que seleccione, en el segundo paso del Asistente deberá introducir el nombre del Asesor Experto y los parámetros de entrada. El aspecto de este paso es similar al que ya

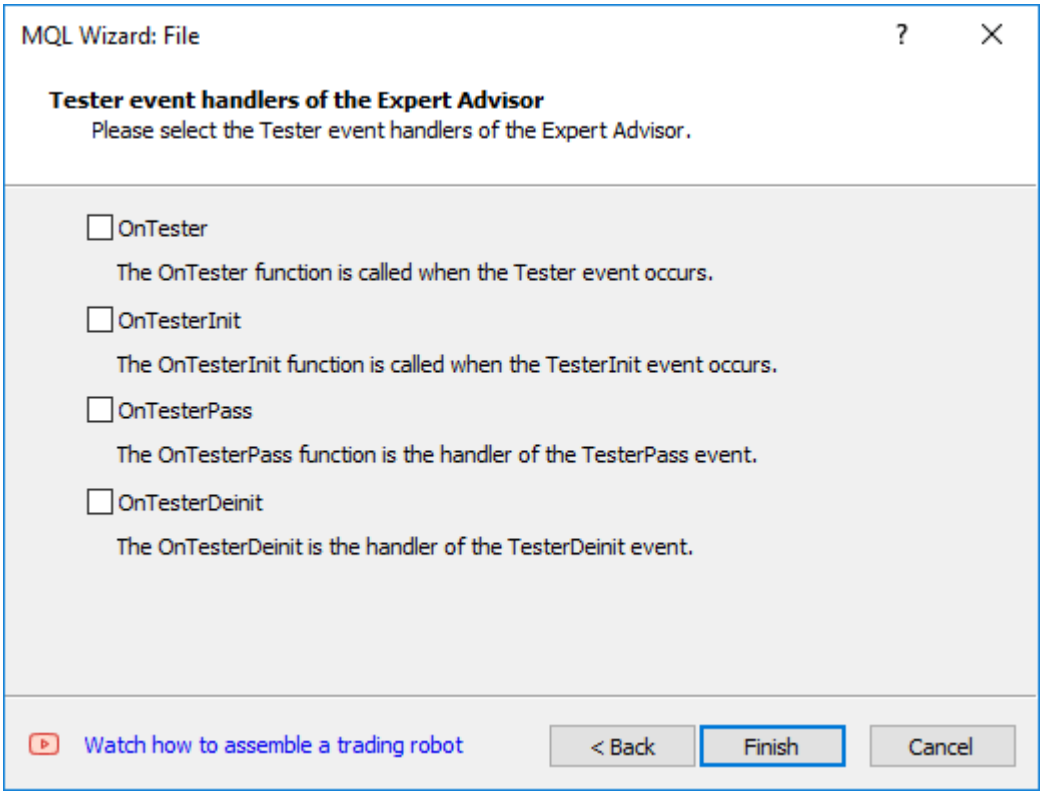
se mostró en la sección Asistente MQL y borrador del programa. La única advertencia es que los Asesores Expertos basados en la biblioteca estándar deben tener dos parámetros obligatorios (no extraíbles): *Symbol* y *TimeFrame*.

Para una plantilla simple, en el 3er paso, se propone seleccionar los manejadores de eventos adicionales que se añadirán al código fuente, además de *OnTick* (*OnTick* siempre insertado).



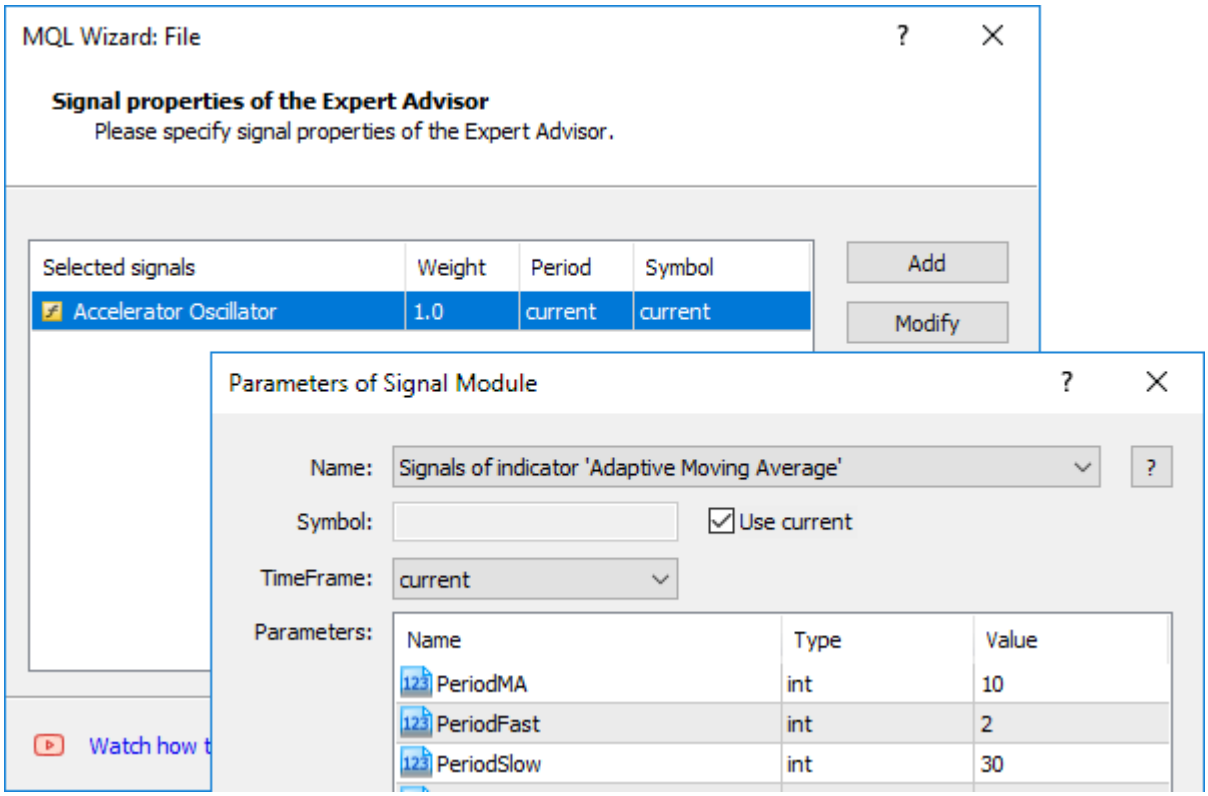
Creación de una plantilla de Asesor Experto. Paso 3. Manejadores de eventos adicionales

El cuarto y último paso le permite especificar uno o más manejadores de eventos opcionales para el probador. De ellos hablaremos en el próximo capítulo.



Creación de una plantilla de Asesor Experto. Paso 4. Controladores de eventos del probador

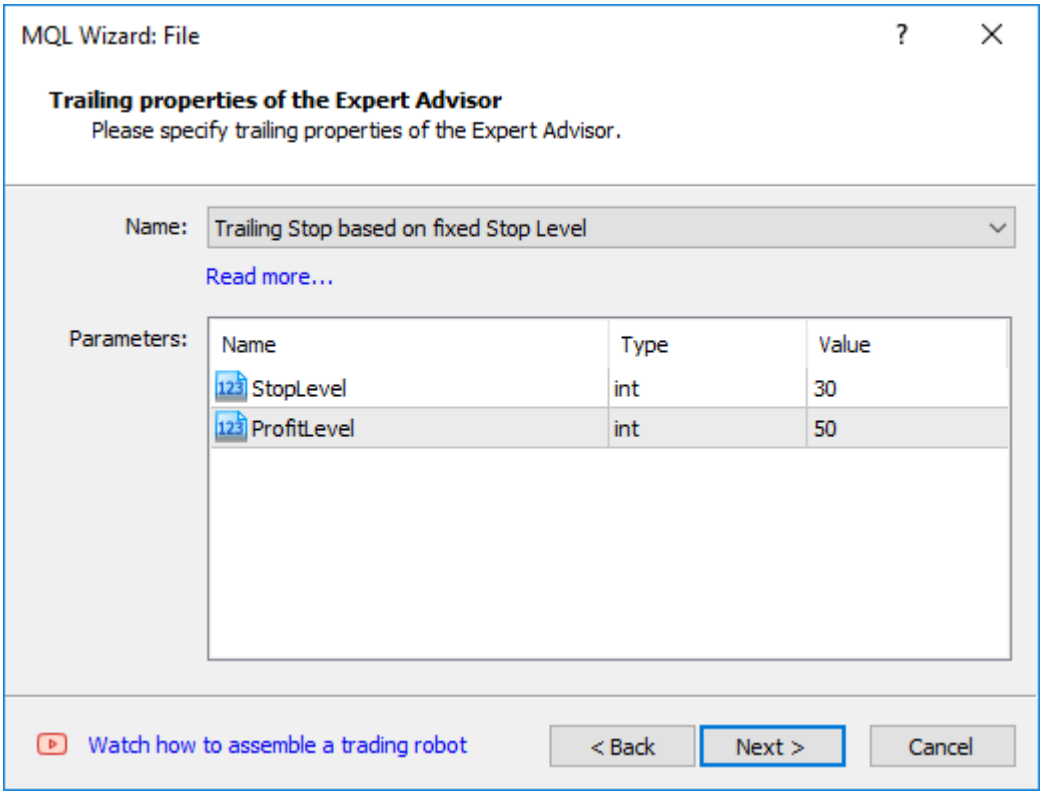
Si el usuario elige generar un programa basado en la biblioteca estándar en el primer paso del Asistente, entonces el 3er paso consiste en configurar las señales de trading.



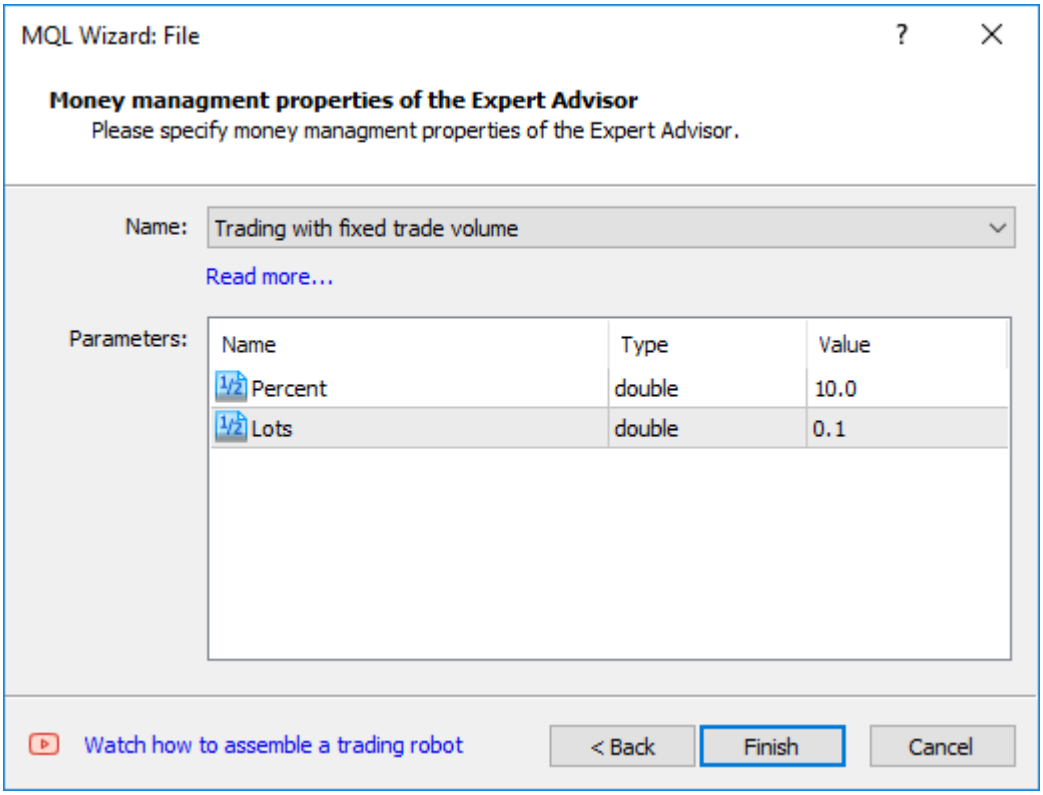
Generación de un Asesor Experto listo. Paso 3. Configuración de señales de trading

Puede obtener más información en la documentación.

Los pasos 4 y 5 están diseñados para incluir trailing en el Asesor Experto y seleccionar automáticamente los lotes según uno de los métodos predefinidos.



Generación de un Asesor Experto listo. Paso 4. Elección de un método de trailing stop



Generación de un Asesor Experto listo. Paso 5. Selección de lotes

El Asistente, por supuesto, no es una herramienta universal, y el prototipo de programa resultante, por regla general, necesita mejorarse. No obstante, los conocimientos adquiridos en este capítulo le permitirán confiar más en los códigos fuente generados y ampliarlos según sea necesario.

6.5 Pruebas y optimización de Asesores Expertos

El desarrollo de Asesores Expertos implica, no sólo y no tanto la implementación de una estrategia de trading en MQL5, sino en mayor medida probar su rendimiento financiero, encontrar los ajustes óptimos y depurar (buscar y corregir errores) en diversas situaciones. Todo esto se puede hacer en el probador integrado de MetaTrader 5.

El probador funciona para varias divisas y admite varios modos de generación de ticks: basados en los precios de apertura del marco temporal seleccionado, en los precios OHLC del marco temporal M1, en ticks generados artificialmente y en el historial de ticks reales. De este modo, puede elegir la relación óptima entre velocidad y precisión de la simulación de trading.

Los ajustes del probador le permiten establecer el intervalo de tiempo de simulación en el pasado, el tamaño del depósito y el apalancamiento; se utilizan para emular recotizaciones y características específicas de la cuenta (incluido el tamaño de las comisiones, los márgenes, los horarios de las sesiones y la limitación del número de lotes). Todos los detalles del trabajo con el probador desde el punto de vista del usuario se encuentran en la documentación del terminal.

Anteriormente comentamos brevemente el trabajo con el probador, en concreto, en la sección Simulación de indicadores. Recordemos que las funciones de control del probador y su optimización no están disponibles para los indicadores, pero sí para los Asesores Expertos. Sin embargo, personalmente, me gustaría ver una opción de autoajuste adaptativo de los indicadores: todo lo que se necesita es admitir el manejador *OnTester* en ellos, de lo que hablaremos en una sección aparte.

Como sabe, existen varios modos de optimización, como la enumeración directa de combinaciones de parámetros de entrada del Asesor Experto, el algoritmo genético acelerado, los cálculos matemáticos o las ejecuciones secuenciales a través de símbolos en *Observación de Mercado*. Como criterio de optimización, puede utilizar tanto métricas bien conocidas como la rentabilidad, el ratio de Sharpe, el factor de recuperación y la rentabilidad esperada, así como variables «personalizadas» integradas en el código fuente por el desarrollador del Asesor Experto. En el contexto de este libro, se supone que el lector ya está familiarizado con los principios de configuración, ejecución e interpretación de los resultados de optimización, porque en este capítulo comenzaremos a estudiar la API de control del probador. Los interesados pueden refrescar sus conocimientos con la ayuda de la sección correspondiente de la documentación.

Una función especialmente importante del comprobador es la optimización multihilo, que puede realizarse mediante programas agentes locales y distribuidos (en red), incluidos en la nube en MQL5 Cloud Network. Una sola ejecución de simulación (con parámetros de entrada específicos) lanzada manualmente por el usuario, o una de las muchas ejecuciones solicitadas durante la optimización (cuando implementamos la enumeración de los valores de los parámetros en rangos determinados) se realiza en un programa independiente: el agente. Técnicamente, se trata de un archivo metatester64.exe, y las copias de sus procesos pueden verse en el Administrador de tareas de Windows durante la simulación y optimización. Por ello, el probador es multihilo.

El terminal es un despachador que distribuye tareas a agentes locales y remotos. Si es necesario, lanza agentes locales. Al optimizar, por defecto, se lanzan varios agentes; su cantidad corresponde al número de núcleos del procesador. Después de ejecutar la siguiente tarea para probar un Asesor Experto con los parámetros especificados, el agente devuelve los resultados al terminal.

Cada agente crea su propio entorno de trading y de software. Todos los agentes están aislados entre sí y del terminal de cliente.

En concreto, el agente tiene sus propias variables globales y su propia sandbox de archivos, incluyendo la carpeta donde se escriben los registros detallados del agente, *Tester/Agent-IPaddress-Port/Logs*. Aquí *Tester* es el directorio de instalación del probador (durante una instalación estándar junto con MetaTrader 5, esta es la subcarpeta donde se instala el terminal). El nombre del directorio *Agent-IPaddress-Port*, en lugar de *IPaddress* y *Port*, contendrá la dirección de red y los valores de puerto específicos que se utilizan para comunicarse con el terminal. Para los agentes locales, esta es la dirección 127.0.0.1 y el rango de puertos, por defecto, a partir de 3000 (por ejemplo, en un ordenador con 4 núcleos, veremos agentes en los puertos 3000, 3001, 3002, 3003).

Al probar un Asesor Experto, todas las operaciones de archivo se realizan en la carpeta *Tester/Agent-IPaddress-Port/MQL5/Files*. Sin embargo, es posible implementar la interacción entre agentes locales y el terminal de cliente (así como entre diferentes copias del terminal en el mismo ordenador) a través de una carpeta compartida. Para ello, al abrir un archivo con la función *FileOpen*, debe especificarse la bandera FILE_COMMON. Otra forma de transferir datos de los agentes al terminal es la proporcionada por el mecanismo frames.

La sandbox local del agente se borra automáticamente antes de cada prueba por razones de seguridad (para evitar que diferentes Asesores Expertos lean los datos de los demás).

Se crea una carpeta con el historial de cotizaciones junto a la sandbox de archivos para cada agente, *Tester/Agent-IPaddress-Port/bases/ServerName/Symbol/*. En la siguiente sección le recordamos brevemente cómo se forma.

El terminal almacena los resultados de cada una de las ejecuciones de prueba y optimizaciones en una caché especial que se encuentra en el directorio de instalación, en la subcarpeta *Tester/cache/*. Los resultados de las pruebas se guardan en archivos con la extensión *tst*, y los resultados de la optimización se guardan en archivos *opt*. Ambos formatos son de código abierto para los desarrolladores de MetaQuotes, por lo que puede implementar su propio procesamiento de datos analíticos por lotes, o utilizar códigos fuente ya listos para usar de la base de código del sitio web mql5.com.

En este capítulo, en primer lugar, consideraremos los principios básicos del funcionamiento de los programas MQL en el probador y, a continuación, aprenderemos a interactuar con él en la práctica.

6.5.1 Generar ticks en el probador

La presencia del manejador *OnTick* en el Asesor Experto no es obligatoria para que se pruebe en el probador. El Asesor Experto puede utilizar una o varias de las otras funciones conocidas:

- *OnTick* - manejador de eventos para la llegada de un nuevo tick
- *OnTrade* - manejador de eventos de trading
- *OnTradeTransaction* - manejador de operaciones de trading
- *OnTimer* - manejador de señales de temporizador
- *OnChartEvent* - manejador de eventos en el gráfico, incluidos los gráficos personalizados

Al mismo tiempo, dentro del probador, el principal equivalente del curso temporal es un hilo de ticks, que contiene no sólo los cambios de precio, sino también el tiempo con precisión de milisegundos. Por lo tanto, para probar los Asesores Expertos, es necesario generar secuencias de ticks. El probador de MetaTrader 5 tiene 4 modos de generación de ticks:

- Ticks reales (si el bróker facilita su historial)
- Cada tick (emulación basada en las cotizaciones disponibles en el marco temporal M1)
- Precios OHLC de barras de un minuto (1 minuto OHLC)
- Sólo precios de apertura (1 tick por barra)

Más adelante analizaremos otro modo de funcionamiento, los cálculos matemáticos, ya que no está relacionado con las cotizaciones y los ticks.

Cualquiera de los cuatro modos que elija el usuario, el terminal carga los datos históricos disponibles para la simulación. Si se ha seleccionado el modo de ticks reales, y el bróker no dispone de ellos para este instrumento, entonces se utiliza el modo «Todos los ticks». El probador indica la naturaleza de la generación de ticks en su informe de forma gráfica y en porcentaje (donde 100 % significa que todos los ticks son reales).

El historial del instrumento seleccionado en los ajustes del probador se sincroniza y el terminal lo descarga del servidor de trading antes de iniciar el proceso de simulación. Al mismo tiempo, por primera vez, el terminal descarga el historial del servidor de trading hasta la profundidad requerida (con un cierto margen, en función del marco temporal, al menos 1 año antes del inicio de la prueba), para no solicitarlo posteriormente. En el futuro sólo se producirá la descarga de nuevos datos. Todo ello va acompañado de los correspondientes mensajes en el registro del probador.

El agente de simulación recibe el historial del instrumento probado del terminal de cliente inmediatamente después de iniciarse la simulación. Si el proceso de simulación utiliza datos de otros instrumentos (por ejemplo, se trata de un Asesor Experto multidivisa), en este caso el agente de simulación solicita el historial necesario al terminal de cliente en la primera llamada. Si el terminal dispone de datos históricos, éstos se transfieren inmediatamente a los agentes de simulación. Si faltan datos, el terminal los solicitará y descargará del servidor, y luego los transferirá a los agentes de simulación.

También se utilizan instrumentos adicionales cuando se calcula el precio de los tipos cruzados durante las operaciones de trading. Por ejemplo, al probar una estrategia en EURCHF con una divisa de depósito en dólares estadounidenses, antes de procesar la primera operación de trading, el agente de simulación solicitará el historial de EURUSD y USDCHF al terminal de cliente, aunque la estrategia no se refiera directamente a estos instrumentos.

En este sentido, antes de probar una estrategia multidivisa, se recomienda descargar primero todos los datos históricos necesarios en el terminal de cliente. Esto ayudará a evitar retrasos en la simulación/optimización asociados a la reanudación de los datos. Puede descargar el historial, por ejemplo, abriendo los gráficos correspondientes y desplazándolos hasta el principio del historial.

Veamos ahora con más detalle los modos de generación de ticks.

Ticks reales del historial

La simulación y la optimización en ticks reales son lo más parecido a las condiciones reales. Se trata de los ticks de bolsas y proveedores de liquidez.

> Si hay una barra de minutos en el historial del símbolo, pero no hay datos de ticks para ese minuto, el probador generará ticks en el modo «Todos los ticks» (ver más adelante). Esto le permite construir el gráfico correcto en el probador en caso de datos de tick incompletos del bróker. Además, los datos de ticks pueden no coincidir con las barras de minutos por diversas razones. Por ejemplo, debido a desconexiones u otros fallos en la transmisión de datos de la fuente al terminal de cliente. En la simulación, los datos por minutos se consideran más fiables.

Los ticks se almacenan en la caché de símbolos del probador de estrategias. El tamaño de la caché no es superior a 128,000 ticks. Cuando llegan nuevos ticks, se expulsan los datos más antiguos. Sin embargo, utilizando la función *CopyTicks*, se pueden obtener ticks fuera de la caché (sólo cuando se realizan una simulación utilizando ticks reales). En este caso, los datos se solicitarán a la base de datos de ticks del probador, que se corresponde plenamente con la base de datos similar del terminal de cliente. En esta base no se realizan ajustes por barras de minutos. Por lo tanto, los ticks que aparecen en él pueden diferir de los que aparecen en la caché.

Todos los ticks (emulación)

Si el historial de ticks reales no está disponible o si necesita minimizar el tráfico de red (porque el archivo de ticks reales puede consumir recursos importantes), puede optar por generar artificialmente ticks basados en las cotizaciones disponibles del marco temporal M1.

El historial de cotizaciones de los instrumentos financieros se transmite desde el servidor de trading al terminal de cliente de MetaTrader 5 en forma de bloques de barras de minutos muy apretados. El procedimiento de consulta del historial y las construcciones de los marcos temporales requeridos se estudiaron detalladamente en la sección Aspectos técnicos de la organización y el almacenamiento de series temporales.

El elemento mínimo del historial de precios es una barra de minutos, de la que puede obtener información sobre cuatro valores de precios OHLC: Open, High, Low y Close.

Una nueva barra de minutos se abre no en el momento en que comienza un nuevo minuto (el número de segundos pasa a ser 0), sino cuando se produce un tick, es decir, un cambio de precio de al menos un punto. Del mismo modo, no podemos determinar a partir de la barra con precisión de un segundo cuándo llegó el tick correspondiente al precio de cierre de esta barra de un minuto: sólo conocemos el último precio de la barra de un minuto, que se registró como precio de Cierre.

Así, para cada barra de minutos, conocemos 4 puntos de control, que podemos decir con seguridad que el precio ha estado allí. Si la barra tiene sólo 4 ticks, entonces esta información es suficiente para la simulación, pero por lo general, el volumen de ticks es de más de 4. Esto significa que es necesario generar puntos de control adicionales para los ticks que se produjeron entre los precios *Open*, *High*, *Low* y *Close*. Los fundamentos de la generación de ticks en el modo «Todos los ticks» se describen en la documentación.

Al realizar la simulación en el modo «Todos los ticks», la función *OnTick* del Asesor Experto será llamada en todos los ticks que se generen. El Asesor Experto recibirá los precios *Ask/Bid/Last* y precio de la misma manera que cuando trabaja en línea.

El modo de simulación «Todos los ticks» es el más preciso (después del modo de ticks reales), pero también el que requiere más tiempo. Para la evaluación primaria de la mayoría de las estrategias de trading suele bastar con utilizar uno de los dos modos de simulación simplificados: en los precios OHLC M1 o en la apertura de las barras del marco temporal seleccionado.

1 minuto OHLC

En el modo «1 minuto OHLC», la secuencia de ticks se construye sólo por los precios OHLC de las barras de minutos, el número de llamadas a la función *OnTick* se reduce significativamente; por lo tanto, el tiempo de simulación también se reduce. Se trata de un modo muy eficaz y útil que ofrece un compromiso entre la precisión de las pruebas y la velocidad. No obstante, hay que tener cuidado cuando se trata del Asesor Experto de otra persona.

La negativa a generar ticks intermedios adicionales entre los precios *Open*, *High*, *Low* y *Close* conduce a la aparición de un determinismo rígido en la evolución de los precios a partir del momento en que se define el precio *Open*. Esto hace posible crear un «Grial de simulación» que muestre un bonito gráfico de balance con tendencia al alza cuando se realiza la simulación.

> Para una barra de un minuto, se conocen 4 precios, de los cuales el primero es *Open*, y el último es *Close*. Los precios registrados entre ellos son *High* y *Low*, y se pierde la información sobre el orden en que se producen, pero sabemos que el precio *High* es mayor o igual que *Open*, y *Low* es menor o igual que *Open*.
>
> Tras recibir el precio *Open*, tenemos que analizar únicamente el siguiente tick para determinar si es *High* o *Low*. Si el precio está por debajo de *Open*, esto es *Low*: comprar en este tick, ya que el siguiente tick corresponderá al precio *High*, en el cual cerramos la operación de compra y abrimos una de venta. El siguiente tick es el último de la barra, *Close*, en el que cerramos venta.
>
> Si un tick con un precio superior al precio de apertura viene después de nuestro precio, entonces la secuencia de transacciones se invierte. Aparentemente, se podría operar en cada barra de este modo. Cuando se prueba un Asesor Experto de este tipo en el historial, todo va perfectamente, pero en línea fallará.

Un efecto similar puede producirse involuntariamente, debido a una combinación de características del algoritmo de cálculo (por ejemplo, el cálculo de estadísticas) y la generación de ticks.

Por lo tanto, siempre es importante probarlo en el modo «Todos los ticks» o, mejor, basado en ticks reales después de encontrar la configuración óptima del Asesor Experto en los modos de simulación aproximados («1 minuto OHLC» y «Sólo precios de apertura»).

Sólo precios de apertura

En este modo, los ticks se generan utilizando los precios OHLC del marco temporal seleccionado para la simulación. En este caso, la función *OnTick* sólo se ejecuta una vez, al principio de cada barra. Debido a esta característica, los niveles de stop y las órdenes pendientes pueden activarse a un precio diferente del solicitado (especialmente cuando se realiza la simulación en marcos temporales superiores). A cambio de esto, tenemos la oportunidad de realizar rápidamente una simulación de evaluación del Asesor Experto.

Por ejemplo, el Asesor Experto se prueba en EURUSD H1 en el modo «Sólo precios de apertura». En este caso, el número total de ticks (puntos de control) será 4 veces superior al número de barras de hora que caen dentro del intervalo analizado. Pero en este caso, el manejador *OnTick* sólo será llamado en la apertura de las barras de hora. Para el resto de ticks («ocultos» para el Asesor Experto), se realizan las siguientes comprobaciones necesarias para una correcta simulación:

- cálculo de los requisitos de margen
- activación de *Stop Loss* y *Take Profit*
- activación de órdenes pendientes
- eliminación de órdenes pendientes a su vencimiento

Si no hay posiciones abiertas ni órdenes pendientes, entonces no hay necesidad de estas comprobaciones en ticks ocultos, y el aumento de velocidad puede ser significativo.

Una excepción cuando se generan ticks en el modo «Sólo precios de apertura» son los periodos W1 y MN1: para estos marcos temporales, los ticks se generan para los precios OHLC de cada día, no semanales ni mensuales, respectivamente.

Este modo «Sólo precios de apertura» es muy adecuado para probar estrategias que realizan operaciones sólo en la apertura de la barra y no utilizan órdenes pendientes, y no utilizan los niveles *Stop Loss* y *Take Profit*. Para la clase de estrategias de este tipo se conserva toda la precisión de simulación necesaria.

La API de MQL5 no permite al programa averiguar en qué modo se está ejecutando en el probador. Al mismo tiempo, esto puede ser importante para los Asesores Expertos o los indicadores que utilizan, que no están diseñados, por ejemplo, para funcionar correctamente a precios de apertura u OHLC. A este respecto, aplicamos un sencillo mecanismo de detección de modos. El código fuente se adjunta en el archivo *TickModel.mqh*.

Vamos a declarar nuestra enumeración con los modos existentes.

```
enum TICK_MODEL
{
   TICK_MODEL_UNKNOWN = -1,    /*Unknown (any)*/    // unknown/not yet defined
   TICK_MODEL_REAL = 0,        /*Real ticks*/       // best quality
   TICK_MODEL_GENERATED = 1,   /*Generated ticks*/  // good quality
   TICK_MODEL_OHLC_M1 = 2,     /*OHLC M1*/          // acceptable quality and fast
   TICK_MODEL_OPEN_PRICES = 3, /*Open prices*/      // worse quality, but very fast
   TICK_MODEL_MATH_CALC = 4,   /*Math calculations*/// no ticks (not defined)
};
```

Excepto el primer elemento, que se reserva para el caso en que el modo aún no se ha determinado o no se puede determinar por alguna razón, todos los demás elementos están dispuestos en orden descendente de calidad de simulación, empezando por real y terminando con los precios de apertura (para ellos, el desarrollador debe comprobar la estrategia de compatibilidad con el hecho de que su trading se lleva a cabo sólo en la apertura de una nueva barra). El último modo TICK_MODEL_MATH_CALC funciona sin ticks en absoluto; lo consideraremos por separado.

El principio de detección del modo se basa en la comprobación de la disponibilidad de ticks y sus tiempos en los dos primeros ticks al iniciar la prueba. La comprobación en sí está envuelta en la función *getTickModel*, que el Asesor Experto debe llamar desde el manejador *OnTick*. Dado que la comprobación se realiza una vez, el modelo de variable estática se describe dentro de la función inicialmente establecida en TICK_MODEL_UNKNOWN. Almacenará y conmutará el estado actual de la comprobación, que será necesaria para distinguir entre los modos OHLC y los precios de apertura.

```
TICK_MODEL getTickModel()
{
   static TICK_MODEL model = TICK_MODEL_UNKNOWN;
   ...
```

En el primer tick analizado, el modelo es igual a TICK_MODEL_UNKNOWN, y se intenta obtener ticks reales llamando a *CopyTicks*.

```
if(model == TICK_MODEL_UNKNOWN)
{
   MqlTick ticks[];
   const int n = CopyTicks(_Symbol, ticks, COPY_TICKS_ALL, 0, 10);
   if(n == -1)
   {
      switch(_LastError)
      {
      case ERR_NOT_ENOUGH_MEMORY:    // emulate ticks
         model = TICK_MODEL_GENERATED;
         break;

      case ERR_FUNCTION_NOT_ALLOWED: // prices of opening and OHLC
         if(TimeCurrent() != iTime(_Symbol, _Period, 0))
         {
            model = TICK_MODEL_OHLC_M1;
         }
         else if(model == TICK_MODEL_UNKNOWN)
         {
            model = TICK_MODEL_OPEN_PRICES;
         }
         break;
      }

      Print(E2S(_LastError));
   }
   else
   {
      model = TICK_MODEL_REAL;
   }
}
...
```

Si tiene éxito, la detección termina inmediatamente con el ajuste del modelo a TICK_MODEL_REAL. Si no se dispone de ticks reales, el sistema devolverá un determinado código de error, según el cual podemos sacar las siguientes conclusiones. El código de error ERR_NOT_ENOUGH_MEMORY corresponde al modo de emulación de ticks. No está del todo claro por qué el código es así, pero se trata de un rasgo característico, y aquí lo utilizamos. En los otros dos modos de generación de ticks, obtendremos el error ERR_FUNCTION_NOT_ALLOWED.

Puede distinguir un modo del otro por el tiempo de tick. Si resulta ser un no-múltiplo del marco temporal para un tick, entonces estamos hablando del modo OHLC. Sin embargo, el problema aquí es que el primer tick en ambos modos puede estar alineado con la hora de apertura de la barra. Así, obtendremos el valor TICK_MODEL_OPEN_PRICES, pero es necesario especificarlo. Por lo tanto, para la conclusión final, debe analizarse un tick más (volver a llamar a la función sobre él si antes se recibió TICK_MODEL_OPEN_PRICES). Para este caso, se proporciona la siguiente rama *if* dentro de la función.

```
   else if(model == TICK_MODEL_OPEN_PRICES)
   {
      if(TimeCurrent() != iTime(_Symbol, _Period, 0))
      {
         model = TICK_MODEL_OHLC_M1;
      }
   }
   return model;
}
```

Comprobemos el funcionamiento del detector en un sencillo Asesor Experto *TickModel.mq5*. En el parámetro de entrada *TickCount*, especificamos el número máximo de ticks analizados, es decir, cuántas veces se llamará a la función *getTickModel*. Sabemos que con dos es suficiente, pero para asegurarnos de que el modelo no cambia después, se sugieren 5 ticks por defecto. También proporcionamos el parámetro *RequireTickModel* que indica al Asesor Experto que finalice la operación si el nivel de simulación es inferior al solicitado. Por defecto, su valor es TICK_MODEL_UNKNOWN, lo que significa que no hay restricción de modo.

```
input int TickCount = 5;
input TICK_MODEL RequireTickModel = TICK_MODEL_UNKNOWN;
```

En el manejador *OnTick*, ejecutamos nuestro código sólo si funciona en el probador.

```
void OnTick()
{
   if(MQLInfoInteger(MQL_TESTER))
   {
      static int count = 0;
      if(count++ < TickCount)
      {
         // output tick information for reference
         static MqlTick tick[1];
         SymbolInfoTick(_Symbol, tick[0]);
         ArrayPrint(tick);
         // define and display the model (preliminarily)
         const TICK_MODEL model = getTickModel();
         PrintFormat("%d %s", count, EnumToString(model));
         // if the tick counter is 2+, the conclusion is final and we act based on it
         if(count >= 2)
         {
            if(RequireTickModel != TICK_MODEL_UNKNOWN
            && RequireTickModel < model) // quality less than requested
            {
               PrintFormat("Tick model is incorrect (%s %sis required), terminating",
                  EnumToString(RequireTickModel),
                  (RequireTickModel != TICK_MODEL_REAL ? "or better " : ""));
               ExpertRemove(); // end operation
            }
         }
      }
   }
}
```

Intentemos ejecutar el Asesor Experto en el probador con diferentes modos de generación de ticks eligiendo una combinación común de EURUSD H1.

El parámetro *RequireTickModel* en el Asesor Experto se establece en OHLC M1. Si el modo de prueba es «Todos los ticks», recibiremos el mensaje correspondiente en el registro, y el Asesor Experto seguirá funcionando.

```
[time] [bid] [ask] [last] [volume] [time_msc] [flags] [volume_real]
[0] 2022.04.01 00:00:30 1.10656 1.10679 1.10656 0 1648771230000 14 0.00000
NOT_ENOUGH_MEMORY
1 TICK_MODEL_GENERATED
[time] [bid] [ask] [last] [volume] [time_msc] [flags] [volume_real]
[0] 2022.04.01 00:01:00 1.10656 1.10680 1.10656 0 1648771260000 12 0.00000
2 TICK_MODEL_GENERATED
[time] [bid] [ask] [last] [volume] [time_msc] [flags] [volume_real]
[0] 2022.04.01 00:01:30 1.10608 1.10632 1.10608 0 1648771290000 14 0.00000
3 TICK_MODEL_GENERATED
```

Los modos OHLC M1 y ticks reales también son adecuados, y en este último caso, no habrá código de error.

```
[time] [bid] [ask] [last] [volume] [time_msc] [flags] [volume_real]
[0] 2022.04.01 00:00:00 1.10656 1.10687 0.0000 0 1648771200122 134 0.00000
1 TICK_MODEL_REAL
[time] [bid] [ask] [last] [volume] [time_msc] [flags] [volume_real]
[0] 2022.04.01 00:00:00 1.10656 1.10694 0.0000 0 1648771200417 4 0.00000
2 TICK_MODEL_REAL
[time] [bid] [ask] [last] [volume] [time_msc] [flags] [volume_real]
[0] 2022.04.01 00:00:00 1.10656 1.10691 0.0000 0 1648771200816 4 0.00000
3 TICK_MODEL_REAL
```

Sin embargo, si cambia el modo en el probador a «Sólo precios de apertura», el Asesor Experto se detendrá después del segundo tick.

```
[time] [bid] [ask] [last] [volume] [time_msc] [flags] [volume_real]
[0] 2022.04.01 00:00:00 1.10656 1.10679 1.10656 0 1648771200000 14 0.00000
FUNCTION_NOT_ALLOWED
1 TICK_MODEL_OPEN_PRICES
[time] [bid] [ask] [last] [volume] [time_msc] [flags] [volume_real]
[0] 2022.04.01 01:00:00 1.10660 1.10679 1.10660 0 1648774800000 14 0.00000
2 TICK_MODEL_OPEN_PRICES
Tick model is incorrect (TICK_MODEL_OHLC_M1 or better is required), terminating
ExpertRemove() function called
```

Este método requiere ejecutar una prueba y esperar un par de ticks para determinar el modo. En otras palabras: no podemos detener la prueba antes de tiempo devolviendo un error de *OnInit*. Es más: al iniciar una optimización con el tipo de generación de ticks incorrecto, no podremos detener la optimización, cosa que sólo se puede hacer desde la función *OnTesterInit*. Así, el probador intentará completar todas las pasadas durante la optimización, aunque se detendrán al principio. Esta es la limitación actual de la plataforma.

6.5.2 Gestión del tiempo en el probador: temporizador, Sleep, GMT

A la hora de desarrollar Asesores Expertos, hay que tener en cuenta que el probador dispone de unas especificaciones de simulación del paso del tiempo basadas en ticks generados y operación de las funciones relacionadas con el tiempo.

En la simulación, la hora local devuelta por la función *TimeLocal* es siempre igual a la hora del servidor según *TimeTradeServer*. A su vez, la hora del servidor es siempre igual a *TimeGMT* GMT. Así, todas estas funciones, cuando se prueban, dan la misma hora. Se trata de una característica técnica de la plataforma, que se produce porque se decidió no almacenar localmente la información sobre la hora del servidor, sino tomarla siempre del servidor, con el que puede no haber conexión en un momento determinado.

Esta característica crea dificultades en la implementación de estrategias relacionadas con el tiempo global, en particular, con referencia a los comunicados de prensa. En tales casos, es necesario especificar la zona horaria de las cotizaciones en los ajustes del Asesor Experto que se está probando o inventar métodos para la autodetección de la zona horaria (véase la sección Horario de verano).

Pasemos ahora a otras funciones para trabajar con el tiempo.

Como sabemos, es posible procesar eventos de temporizador en MQL5. El manejador *OnTimer* se llama independientemente del modo de simulación. Esto significa que si la simulación se lanza en el modo «Sólo precios de apertura» en el período H4, y se establece un temporizador dentro del Asesor Experto con una llamada cada segundo, entonces el manejador *OnTick* será llamado una vez en la apertura de

cada barra H4 y luego, dentro de la barra, el manejador *OnTimer* será llamado 14400 veces (3600 segundos * 4 horas). La medida en que el tiempo de simulación del Asesor Experto aumentará en este caso depende de su algoritmo.

Otra función que influye en el transcurso del tiempo dentro de un programa es la función *Sleep*, que permite suspender la ejecución de un Asesor Experto durante un tiempo. Esto puede ser necesario cuando se solicita algún dato que aún no está listo en el momento de la solicitud, y es necesario esperar hasta que esté listo.

Es importante entender que *Sleep* sólo afecta al programa que lo llama y no retrasa el proceso de simulación. De hecho, al llamar a *Sleep*, los ticks generados se «reproducen» dentro del retardo especificado, como resultado de lo cual pueden activarse órdenes pendientes, niveles de stop, etc. Tras llamar a *Sleep*, el tiempo simulado en el comprobador se incrementa en el intervalo especificado en el parámetro de la función.

Más adelante, en la sección sobre simulación de Asesores Expertos multidivisa, mostraremos cómo puede utilizar el temporizador y la función *Sleep* para sincronizar barras.

6.5.3 Simulación de visualización: gráfico, objetos, indicadores

El probador permite realizar simulación de dos formas distintas: con y sin visualización. El método se selecciona eligiendo la opción correspondiente en la pestaña de ajustes principales del comprobador.

Cuando la visualización está activada, el probador abre una ventana independiente en la que reproduce las operaciones de trading y muestra indicadores y objetos. Aunque es visual, no necesitamos verlo en todos los casos, sino sólo en los programas con una interfaz de usuario (por ejemplo, paneles de trading o marcas controladas por objetos gráficos). Para otros Asesores Expertos, sólo es importante la ejecución del algoritmo según la estrategia establecida. Esto puede comprobarse sin visualización, lo que puede acelerar considerablemente el proceso. Por cierto: es en este modo en el que se realizan las pruebas durante la optimización.

Durante esta simulación y optimización «en segundo plano» no se construye ningún objeto gráfico. Por lo tanto, al acceder a las propiedades de los objetos, el Asesor Experto recibirá valores cero. Así, podrá comprobar el trabajo con los objetos y el gráfico sólo cuando realice la simulación en el modo visual.

Anteriormente, en la sección Simulación de indicadores, hemos visto el comportamiento específico de los indicadores en el probador. Para aumentar la eficacia de la optimización y simulación no visual de Asesores Expertos (mediante indicadores), los indicadores pueden calcularse no en cada tick, sino sólo cuando les solicitemos datos. El recálculo en cada tick sólo se produce si hay funciones *EventChartCustom*, *OnChartEvent*, *OnTimer* o directivas *tester_everytick_calculate* en el indicador (véase Directivas del preprocesador para el probador). En la ventana del comprobador visual, los indicadores en línea siempre reciben eventos *OnCalculate* en cada tick.

Si la simulación se lleva a cabo en un modo no visual, después de su finalización, el gráfico de símbolos se abre automáticamente en el terminal, que muestra las transacciones completadas y los indicadores que se utilizaron en el Asesor Experto. Esto ayuda a correlacionar los momentos de entrada y salida del mercado con los valores de los indicadores. Sin embargo, aquí nos referimos sólo a los indicadores que funcionan en el símbolo y el marco temporal de la simulación. Si el Asesor Experto ha creado indicadores en otros símbolos o marcos temporales, no se mostrarán.

> Es importante tener en cuenta que los indicadores que aparecen en el gráfico que se abre automáticamente una vez finalizada la simulación se vuelven a calcular una vez finalizada ésta. Esto

sucede incluso si estos indicadores se utilizaron en el Asesor Experto probado y se calcularon previamente «sobre la marcha», a medida que se formaban las barras.

En algunos casos, el programador puede necesitar ocultar información sobre qué indicadores se utilizan en el algoritmo de trading, y por lo tanto su visualización en el gráfico no es deseable. Para ello puede utilizarse la función *IndicatorRelease*.

La función *IndicatorRelease* está pensada originalmente para liberar la parte calculada del indicador si ya no se necesita. Esto ahorra memoria y recursos del procesador. Su segundo propósito es prohibir la visualización del indicador en el gráfico de simulación después de completar una sola ejecución.

Para desactivar la visualización del indicador en el gráfico al final de la simulación, basta con llamar a *IndicatorRelease* con el manejador del indicador en el manejador *OnDeinit*. La función *OnDeinit* siempre se llama en Asesores Expertos después de la finalización y antes de mostrar el gráfico de prueba. Ni OnDeinit ni los destructores de objetos globales y estáticos se llaman en los propios indicadores en el probador; esto es lo que acordaron los desarrolladores de MetaTrader 5.

Además, la API de MQL5 incluye una función especial, *TesterHideIndicators*, con una finalidad similar, que estudiaremos más adelante.

Al mismo tiempo, debe tenerse en cuenta que las plantillas tpl (si se crean) pueden influir adicionalmente en la representación externa del gráfico de simulación.

Así, si existe una plantilla *tester.tpl* en el directorio *MQL5/Profiles/Templates*, se aplicará al gráfico abierto. Si el Asesor Experto utilizó otros indicadores en su trabajo y no prohibió su visualización, entonces los indicadores de la plantilla y del Asesor Experto se combinarán en el gráfico.

Cuando *tester.tpl* está ausente, se aplica la plantilla por defecto (*default.tpl*).

Si la carpeta *MQL5/Profiles/Templates* contiene una plantilla tpl con el mismo nombre que el Asesor Experto (por ejemplo, *ExpertMACD.tpl*), entonces durante la simulación visual o en el gráfico abierto después de la simulación, sólo se mostrarán los indicadores de esta plantilla. En este caso, no se mostrará ningún indicador utilizado en el Asesor Experto probado.

6.5.4 Pruebas multidivisa

Como usted sabe, el probador de MetaTrader 5 le permite probar estrategias que operan con múltiples instrumentos financieros. Desde un punto de vista puramente técnico, en función de los recursos de hardware del ordenador, es posible simular el trading simultáneo de todos los instrumentos disponibles.

Simular este tipo de estrategias impone al probador varios requisitos técnicos adicionales:

- Generación de secuencias de ticks para todos los instrumentos
- Cálculo de indicadores para todos los instrumentos
- Cálculo de los requisitos de margen y emulación de otras condiciones de trading para todos los instrumentos

Al acceder al historial por primera vez, el probador descarga automáticamente del terminal el historial de los instrumentos necesarios. Si el terminal no contiene el historial requerido, lo solicitará a su vez al servidor de trading. Por lo tanto, antes de probar un Asesor Experto multidivisa, se recomienda seleccionar los instrumentos necesarios en la página *Observación de Mercado* del terminal y descargar la cantidad de datos deseada.

El agente carga el historial que falta con un pequeño margen para proporcionar los datos necesarios para el cálculo de indicadores o la copia por el Asesor Experto en el momento de la prueba. La cantidad mínima de historial descargado del servidor de trading depende del marco temporal. Por ejemplo, para plazos de D1 y menos, es de un año. En otras palabras: el historial preliminar se descarga desde el principio del año anterior en relación con la fecha de inicio del probador. Esto da al menos 1 año de historial si la simulación se solicita a partir del 1 de enero, y un máximo de casi dos años si se pide a partir de diciembre. Para un marco temporal semanal, se solicita un historial de 100 barras, es decir, aproximadamente dos años (un año tiene 52 semanas). Para la simulación en un marco temporal mensual, el agente solicitará 100 meses (lo que equivale al historial de unos 8 años: 12 meses * 8 años = 96). En cualquier caso, en marcos temporales inferiores al de trabajo, se dispondrá de un número de barras proporcionalmente mayor. Si los datos existentes no son suficientes para la profundidad predefinida del historial preliminar, este hecho se registrará en el registro de pruebas.

No se puede configurar (cambiar) este comportamiento. Por lo tanto, si necesita proporcionar un número especificado de barras históricas del marco temporal actual desde el principio, debe establecer una fecha de inicio anterior para la prueba y luego «esperar» en el código del Asesor Experto a la fecha de inicio del trading requerida o a un número suficiente de barras. Antes de eso, debería saltarse todos los eventos.

El probador también emula su propia *Observación de Mercado*, de la que el programa puede obtener información sobre los instrumentos. Por defecto, al inicio de la simulación, el probador *Market Watch* contiene un solo símbolo: el símbolo sobre el que se inicia la prueba. Todos los símbolos adicionales se añaden automáticamente al probador *Observación de Mercado* cuando se accede a ellos a través de las funciones de la API. En el primer acceso a un símbolo de «terceros» desde un programa MQL, el agente de simulación sincronizará los datos del símbolo con el terminal.

Se puede acceder a los datos de los símbolos adicionales en los siguientes casos:

- Utilización de indicadores técnicos, *iCustom*, o *IndicatorCreate* para el par símbolo/marco temporal
- Consulta de otro símbolo *Observación de Mercado*:
  - SeriesInfoInteger
  - Bars
  - SymbolSelect
  - SymbolIsSynchronized
  - SymbolInfoDouble
  - SymbolInfoInteger
  - SymbolInfoString
  - SymbolInfoTick
  - SymbolInfoSessionQuote
  - SymbolInfoSessionTrade
  - MarketBookAdd
  - MarketBookGet
- Consulta de las series temporales del par símbolo/marco temporal mediante las siguientes funciones:
  - CopyBuffer
  - CopyRates
  - CopyTime

- CopyOpen
- CopyHigh
- CopyLow
- CopyClose
- CopyTickVolume
- CopyRealVolume
- CopySpread

Además, puede solicitar explícitamente el historial de los símbolos deseados llamando a la función *SymbolSelect* en el manejador *OnInit*. El historial se cargará por adelantado antes de que comience la simulación del Asesor Experto.

En el momento en que se accede a otro símbolo por primera vez, el proceso de simulación se detiene y el historial del par símbolo/período se descarga del terminal al agente de simulación. La generación de secuencias de ticks también está activada en este momento.

Cada instrumento genera su propia secuencia de ticks según el modo de generación de ticks establecido.

La sincronización de barras de diferentes símbolos es de particular importancia cuando se implementan Asesores Expertos multidivisa, ya que la corrección de los cálculos depende de ello. Se considera que un estado está sincronizado cuando las últimas barras de todos los símbolos utilizados tienen la misma hora de apertura.

El probador genera y reproduce su secuencia de ticks para cada instrumento. Al mismo tiempo, se abre una nueva barra en cada instrumento, independientemente de cómo se abran las barras en otros instrumentos. Esto significa que al probar un Asesor Experto multidivisa, es posible que se produzca una situación (y la mayoría de las veces sucede) en la que una nueva barra ya se haya abierto en un instrumento, pero aún no en otro.

Por ejemplo, si estamos probando un Asesor Experto utilizando datos del símbolo EURUSD y una nueva vela horaria se ha abierto para este símbolo, recibiremos el evento *OnTick*. Pero al mismo tiempo, no hay garantía de que se haya abierto una nueva vela en GBPUSD, que también podríamos estar utilizando.

Por lo tanto, el algoritmo de sincronización implica que debe comprobar las cotizaciones de todos los instrumentos y esperar a que se igualen las horas de apertura de las últimas barras.

Esto no plantea ninguna duda mientras se utilicen los modos de simulación de ticks reales, emulación de todos los ticks u OHLC M1. Con estos modos se genera un número suficiente de ticks dentro de una vela para esperar el momento de la sincronización de barras de diferentes símbolos. Sólo tiene que completar la función *OnTick* y comprobar la aparición de una nueva barra en GBPUSD en el siguiente tick. Pero cuando se prueba en el modo «Sólo precios de apertura», no habrá ningún otro tick, ya que el Asesor Experto es llamado sólo una vez por barra, y puede parecer que este modo no es adecuado para probar Asesores Expertos multidivisa. De hecho, el probador permite detectar el momento en que se abre una nueva barra en otro símbolo utilizando la función *Sleep* (en un bucle) o un temporizador.

En primer lugar, consideremos un ejemplo de Asesor Experto *SyncBarsBySleep.mq5*, que demuestra la sincronización de barras a través de *Sleep*.

Un par de parámetros de entrada le permiten establecer el tamaño de *Pause* en segundos para esperar las barras de otro símbolo, así como el nombre de ese otro símbolo (*OtherSymbol*), que debe ser diferente del símbolo del gráfico.

```
input uint Pause = 1;                    // Pause (seconds)
input string OtherSymbol = "USDJPY";
```

Para identificar patrones en el retraso de los horarios de apertura de las barras, describimos una clase simple *BarTimeStatistics* que contiene un campo para contar el número total de barras (*total*) y el número de barras en las que no hubo sincronización inicialmente (*late*), es decir, el otro símbolo llegó tarde.

```
class BarTimeStatistics
{
public:
   int total;
   int late;

   BarTimeStatistics(): total(0), late(0) { }

   ~BarTimeStatistics()
   {
      PrintFormat("%d bars on %s was late among %d total bars on %s (%2.1f%%)",
         late, OtherSymbol, total, _Symbol, late * 100.0 / total);
   }
};
```

El objeto de esta clase imprime las estadísticas recibidas en su destructor. Dado que vamos a hacer que este objeto sea estático, el informe se imprimirá al final de la prueba.

Si el modo de generación de ticks seleccionado en el probador difiere de los precios de apertura, lo detectaremos utilizando la función *getTickModel* que vimos con anterioridad y devolverá una advertencia.

```
void OnTick()
{
   const TICK_MODEL model = getTickModel();
   if(model != TICK_MODEL_OPEN_PRICES)
   {
      static bool shownOnce = false;
      if(!shownOnce)
      {
         Print("This Expert Advisor is intended to run in \"Open Prices\" mode");
         shownOnce = true;
      }
   }
```

A continuación, *OnTick* proporciona el algoritmo de sincronización de trabajo.

```
   // time of the last known bar for _Symbol
   static datetime lastBarTime = 0;
   // attribute of synchronization
   static bool synchronized = false;
   // bar counters
   static BarTimeStatistics stats;

   const datetime currentTime = iTime(_Symbol, _Period, 0);

   // if it is executed for the first time or the bar has changed, save the bar
   if(lastBarTime != currentTime)
   {
      stats.total++;
      lastBarTime = currentTime;
      PrintFormat("Last bar on %s is %s", _Symbol, TimeToString(lastBarTime));
      synchronized = false;
   }

   // time of the last known bar for another symbol
   datetime otherTime;
   bool late = false;

   // wait until the times of two bars become the same
   while(currentTime != (otherTime = iTime(OtherSymbol, _Period, 0)))
   {
      late = true;
      PrintFormat("Wait %d seconds...", Pause);
      Sleep(Pause * 1000);
   }
   if(late) stats.late++;

   // here we are after synchronization, save the new status
   if(!synchronized)
   {
      // use TimeTradeServer() because TimeCurrent() does not change in the absence o
      Print("Bars are in sync at ", TimeToString(TimeTradeServer(),
         TIME_DATE | TIME_SECONDS));
      // no longer print a message until the next out of sync
      synchronized = true;
   }
   // here is your synchronous algorithm
   // ...
}
```

Vamos a configurar el probador para ejecutar el Asesor Experto en EURUSD, H1, que es el instrumento más líquido. Vamos a utilizar los parámetros por defecto del Asesor Experto, es decir, USDJPY será el «otro» símbolo.

Como resultado de la prueba, el registro contendrá las siguientes entradas (mostramos intencionadamente los registros relacionados con la descarga del historial del USDJPY, que se produjo durante la primera llamada a *iTime*).

```
2022.04.15 00:00:00 Last bar on EURUSD is 2022.04.15 00:00
USDJPY: load 27 bytes of history data to synchronize in 0:00:00.001
USDJPY: history synchronized from 2020.01.02 to 2022.04.20
USDJPY,H1: history cache allocated for 8109 bars and contains 8006 bars from 2021.01.
USDJPY,H1: 1 bar from 2022.04.15 00:00 added
USDJPY,H1: history begins from 2021.01.04 00:00
2022.04.15 00:00:00 Bars are in sync at 2022.04.15 00:00:00
2022.04.15 01:00:00 Last bar on EURUSD is 2022.04.15 01:00
2022.04.15 01:00:00 Wait 1 seconds...
2022.04.15 01:00:01 Bars are in sync at 2022.04.15 01:00:01
2022.04.15 02:00:00 Last bar on EURUSD is 2022.04.15 02:00
2022.04.15 02:00:00 Wait 1 seconds...
2022.04.15 02:00:01 Bars are in sync at 2022.04.15 02:00:01
...
2022.04.20 23:59:59 95 bars on USDJPY was late among 96 total bars on EURUSD (99.0%)
```

Puede ver que las barras del USDJPY se retrasan regularmente. Si selecciona USDJPY, H1 en la configuración del probador y EURUSD en los parámetros del Asesor Experto, obtendrá la imagen opuesta.

```
2022.04.15 00:00:00 Last bar on USDJPY is 2022.04.15 00:00
EURUSD: load 27 bytes of history data to synchronize in 0:00:00.002
EURUSD: history synchronized from 2018.01.02 to 2022.04.20
EURUSD,H1: history cache allocated for 8109 bars and contains 8006 bars from 2021.01.
EURUSD,H1: 1 bar from 2022.04.15 00:00 added
EURUSD,H1: history begins from 2021.01.04 00:00
2022.04.15 00:00:00 Bars are in sync at 2022.04.15 00:00:00
2022.04.15 01:00:00 Last bar on USDJPY is 2022.04.15 01:00
2022.04.15 01:00:00 Wait 1 seconds...
2022.04.15 01:00:01 Bars are in sync at 2022.04.15 01:00:01
2022.04.15 02:00:00 Last bar on USDJPY is 2022.04.15 02:00
2022.04.15 02:00:00 Wait 1 seconds...
2022.04.15 02:00:01 Bars are in sync at 2022.04.15 02:00:01
...
2022.04.20 23:59:59 23 bars on EURUSD was late among 96 total bars on USDJPY (24.0%)
```

Aquí, en la mayoría de los casos, no hubo necesidad de esperar: las barras de EURUSD ya existían en el momento en que se formó la barra de USDJPY.

Hay otra forma de sincronizar las barras: utilizando un temporizador. En el libro se incluye un ejemplo de este tipo de Asesor Experto, *SyncBarsByTimer.mq5*. Tenga en cuenta que los eventos del temporizador, por regla general, se producen dentro de la barra (porque la probabilidad de acertar exactamente el comienzo es muy baja). Por ello, las barras están casi siempre sincronizadas.

También podríamos recordarle la posibilidad de sincronizar las barras mediante el indicador espía *EventTickSpy.mq5*, pero este se basa en eventos personalizados que sólo funcionan cuando se realizan simulaciones visuales. Además, para este tipo de indicadores que requieren una respuesta para cada tick, es importante utilizar la directiva *#property tester_everytick_calculate*. Ya hemos hablado de ello en la sección Simulación de indicadores y se lo recordaremos una vez más en la sección dedicada a las directivas del probador específicas.

6.5.5 Criterios de optimización

Un criterio de optimización es una métrica determinada que define la calidad del conjunto de parámetros de entrada sometidos a prueba. Cuanto mayor sea el valor del criterio de optimización, mejor se estimará el resultado de la prueba con un conjunto determinado de parámetros. El parámetro se selecciona en la pestaña «Ajustes», a la derecha del campo «Optimización».

El criterio es importante no sólo para que el usuario pueda comparar los resultados. Sin un criterio de optimización, es imposible utilizar un algoritmo genético, ya que en función del criterio «decide» cómo seleccionar a los candidatos para las nuevas generaciones. El criterio no se utiliza durante la optimización completa con una iteración completa de todas las variantes posibles.

El probador dispone de los siguientes criterios de optimización integrados:

- Balance máximo
- Máxima rentabilidad
- Ganancia máxima esperada (ganancia/pérdida media por operación)
- Reducción mínima como porcentaje del capital
- Factor de recuperación máximo
- Ratio de Sharpe máximo
- Criterio de optimización personalizado

Al elegir esta última opción se tendrá en cuenta como criterio de optimización el valor de la función *OnTester* implementada en el Asesor Experto: la consideraremos más adelante. Este parámetro permite al programador utilizar cualquier índice personalizado para la optimización.

En MetaTrader 5 también existe un «criterio complejo» especial. Se trata de una métrica integral de la calidad de la pasada de prueba, que tiene en cuenta varios parámetros a la vez:

- Número de transacciones
- Reducción
- Factor de recuperación
- Expectativa matemática de ganar
- Ratio de Sharpe

Los desarrolladores no revelan la fórmula, pero se sabe que los valores posibles van de 0 a 100. Es importante que los valores del parámetro complejo afecten al color de las celdas de la columna *Result* en la tabla de optimización con independencia del criterio, es decir, el resaltado siguiendo este esquema funciona incluso cuando se elige otro criterio para mostrar en la columna *Result*. Las combinaciones débiles con valores inferiores a 20 se resaltan en rojo, las combinaciones fuertes por encima de 80 se resaltan en verde oscuro.

La búsqueda de un criterio universal del factor de calidad del sistema de trading es una tarea urgente y difícil para la mayoría de los operadores, ya que la elección de la configuración basada en el valor máximo de un criterio (por ejemplo, el beneficio) está, por regla general, lejos de ser la mejor opción en términos de comportamiento estable y predecible del Asesor Experto en un futuro previsible.

La presencia de un indicador complejo permite nivelar los puntos débiles de cada métrica individual (y necesariamente están disponibles y son ampliamente conocidos) y proporciona una pauta a la hora de desarrollar sus propias variables personalizadas para su cálculo en *OnTester*. Nos ocuparemos de esto pronto.

6.5.6 Obtener estadísticas financieras de prueba: TesterStatistics

Solemos evaluar la calidad de un Asesor Experto basándonos en un informe de trading, que es similar a un informe de simulación cuando se trata de un probador. Contiene un gran número de variables que caracterizan el estilo de trading, la estabilidad y, por supuesto, la rentabilidad. Todas estas métricas, con algunas excepciones, están disponibles para el programa MQL a través de una función especial *TesterStatistics*. Así, el desarrollador del Asesor Experto tiene la capacidad de analizar variables individuales en el código y construir sus propios criterios combinados de calidad de optimización a partir de ellas.

double TesterStatistics(ENUM_STATISTICS statistic)

La función *TesterStatistics* devuelve el valor de la variable estadística especificada, calculada en base a los resultados de una ejecución separada del Asesor Experto en el probador. Se puede llamar a una función en el manejador *OnDeinit* o *OnTester*, que aún está por discutir.

Todas las variables estadísticas disponibles se resumen en la enumeración ENUM_STATISTICS. Algunas de ellas son características cualitativas, es decir, números reales (normalmente beneficios totales, reducciones, ratios, etc.), y las otras son cuantitativas, es decir, números enteros (por ejemplo, el número de transacciones). Sin embargo, ambos grupos están controlados por la misma función con el resultado *double*.

En la siguiente tabla se muestran los indicadores reales (coeficientes e importes monetarios). Todos los importes monetarios se expresan en la divisa del depósito.

Identificador	Descripción
STAT_INITIAL_DEPOSIT	Depósito inicial
STAT_WITHDRAWAL	Importe de los fondos retirados de la cuenta
STAT_PROFIT	Beneficio o pérdida neta al final de la simulación, la suma de STAT_GROSS_PROFIT y STAT_GROSS_LOSS
STAT_GROSS_PROFIT	Beneficio total, la suma de todas las operaciones rentables (mayor o igual a cero)
STAT_GROSS_LOSS	Pérdida total, la suma de todas las operaciones perdedoras (menor o igual a cero)
STAT_MAX_PROFITTRADE	Beneficio máximo: el mayor valor de entre todas las operaciones rentables (mayor o igual a cero)
STAT_MAX_LOSSTRADE	Pérdida máxima: el valor más pequeño de entre todas las operaciones perdedoras (menor o igual a cero)
STAT_CONPROFITMAX	Beneficio máximo total en una serie de operaciones rentables (mayor o igual a cero)
STAT_MAX_CONWINS	Beneficio total en la serie más larga de operaciones rentables
STAT_CONLOSSMAX	Pérdida máxima total en una serie de operaciones perdedoras (menor o igual a cero)
STAT_MAX_CONLOSSES	Pérdida total en la serie más larga de operaciones perdedoras

Identificador	Descripción
STAT_BALANCEEMIN	Valor del saldo mínimo
STAT_BALANCE_DD	Disposición máxima del saldo en dinero
STAT_BALANCEDD_PERCENT	Disposición de saldo en porcentaje, que se registró en el momento de la máxima disposición de saldo en dinero (STAT_BALANCE_DD)
STAT_BALANCE_DDREL_PERCENT	Disposición máxima del saldo en porcentaje
STAT_BALANCE_DD_RELATIVE	Disposición de saldo en equivalente monetario, que se registró en el momento de la máxima disposición de saldo en porcentaje (STAT_BALANCE_DDREL_PERCENT)
STAT_EQUITYMIN	Valor mínimo de los fondos propios
STAT_EQUITY_DD	Disposición máxima en dinero
STAT_EQUITYDD_PERCENT	Disposición en porcentaje, que se registró en el momento de la disposición máxima de fondos en el dinero (STAT_EQUITY_DD)
STAT_EQUITY_DDREL_PERCENT	Disposición máxima en porcentaje
STAT_EQUITY_DD_RELATIVE	Disposición en dinero que se registró en el momento de la disposición máxima en porcentaje (STAT_EQUITY_DDREL_PERCENT)
STAT_EXPECTED_PAYOFF	Expectativa matemática de ganancias (media aritmética del beneficio total y el número de transacciones)
STAT_PROFIT_FACTOR	Rentabilidad, que es el cociente STAT_GROSS_PROFIT/STAT_GROSS_LOSS (si STAT_GROSS_LOSS = 0; la rentabilidad toma el valor DBL_MAX).
STAT_RECOVERY_FACTOR	Factor de recuperación: el ratio de STAT_PROFIT/STAT_BALANCE_DD
STAT_SHARPE_RATIO	Ratio de Sharpe
STAT_MIN_MARGINLEVEL	Nivel de margen mínimo alcanzado
STAT_CUSTOM_ONTESTER	El valor del criterio de optimización personalizado devuelto por la función OnTester.

En la siguiente tabla muestra los indicadores enteros (importes).

Identificador	Descripción
STAT_DEALS	Número total de transacciones completadas
STAT_TRADES	Número de operaciones (transacciones de salida del mercado)
STAT_PROFIT_TRADES	Operaciones rentables
STAT_LOSS_TRADES	Operaciones perdedoras

Identificador	Descripción
STAT_SHORT_TRADES	Operaciones cortas
STAT_LONG_TRADES	Operaciones largas
STAT_PROFIT_SHORTTRADES	Operaciones cortas rentables
STAT_PROFIT_LONGTRADES	Operaciones largas rentables
STAT_PROFITTRADES_AVGCON	Duración media de una serie de operaciones rentables
STAT_LOSSTRADES_AVGCON	Duración media de una serie de operaciones perdedoras
STAT_CONPROFITMAX_TRADES	Número de operaciones que formaron STAT_CONPROFITMAX (beneficio máximo en la secuencia de operaciones rentables)
STAT_MAX_CONPROFIT_TRADES	Número de operaciones en la serie más larga de operaciones rentables STAT_MAX_CONWINS
STAT_CONLOSSMAX_TRADES	Número de operaciones que formaron STAT_CONLOSSMAX (pérdida máxima en la secuencia de operaciones perdedoras)
STAT_MAX_CONLOSS_TRADES	Número de operaciones en la serie más larga de operaciones perdedoras STAT_MAX_CONLOSSES

Intentemos utilizar las métricas presentadas para crear nuestro propio criterio complejo de calidad de Asesor Experto. Para ello, necesitamos algún tipo de ejemplo «experimental» de un programa MQL. Tomemos el Asesor Experto *MultiMartingale.mq5* como punto de partida, pero lo simplificaremos: eliminaremos la multidivisa, el tratamiento de errores integrados y la programación. Además, elegiremos para ella una estrategia de trading de señales con un único cálculo en la barra, es decir, a los precios de apertura. Esto acelerará la optimización y ampliará el campo de experimentación.

La estrategia se basará en las condiciones de sobrecompra y sobreventa determinadas por el indicador OsMA. El indicador de Bandas de Bollinger superpuesto a OsMA le ayudará a encontrar dinámicamente los límites del exceso de volatilidad, lo que se traduce en señales de trading.

Cuando OsMA regrese dentro del corredor, cruzando el borde inferior de abajo hacia arriba, abriremos una operación de compra. Cuando OsMA cruce el límite superior de la misma forma de arriba a abajo, venderemos. Para salir de las posiciones, utilizamos la media móvil, también aplicada a OsMA. Si OsMA muestra un movimiento inverso (hacia abajo para una posición larga o hacia arriba para una posición corta) y toca MA, la posición se cerrará. Esta estrategia se ilustra en la siguiente captura de pantalla.

Estrategia de trading basada en los indicadores OsMA, BBands y MA

La línea vertical azul corresponde a la barra en la que se abre la compra, ya que en las dos barras anteriores, la banda inferior de Bollinger fue atravesada por el histograma OsMA de abajo arriba (este lugar está marcado con una flecha azul hueca en la subventana). La línea vertical roja es la ubicación de la señal inversa, por lo que se cerró la compra y se abrió la venta. En la subventana, en este lugar (o mejor dicho, en las dos barras anteriores, donde se encuentra la flecha roja hueca), el histograma OsMA cruza la banda de Bollinger superior de arriba a abajo. Por último, la línea verde indica el cierre de la venta, debido a que el histograma comenzó a subir por encima de la MA roja.

Llamemos al Asesor Experto *BandOsMA.mq5*. Los ajustes generales incluirán un número mágico, un lote fijo y una distancia de Stop Loss en puntos. Para el Stop Loss, utilizaremos *TrailingStop* del ejemplo anterior. Aquí no se utiliza el Take Profit.

```
input group "C O M M O N   S E T T I N G S"
sinput ulong Magic = 1234567890;
input double Lots = 0.01;
input int StopLoss = 1000;
```

Hay tres grupos de ajustes destinados a los indicadores.

```
input group "O S M A   S E T T I N G S"
input int FastOsMA = 12;
input int SlowOsMA = 26;
input int SignalOsMA = 9;
input ENUM_APPLIED_PRICE PriceOsMA = PRICE_TYPICAL;

input group "B B A N D S   S E T T I N G S"
input int BandsMA = 26;
input int BandsShift = 0;
input double BandsDeviation = 2.0;

input group "M A   S E T T I N G S"
input int PeriodMA = 10;
input int ShiftMA = 0;
input ENUM_MA_METHOD MethodMA = MODE_SMA;
```

En el Asesor Experto de *MultiMartingale.mq5* no teníamos señales de trading, mientras que la dirección de apertura la fijaba el usuario. Aquí tenemos señales de trading, y tiene sentido organizarlas como una clase separada. En primer lugar, describamos la interfaz abstracta *TradingSignal*.

```
interface TradingSignal
{
   virtual int signal(void);
};
```

Es tan sencillo como nuestra otra interfaz *TradingStrategy*. Y esto es bueno. Cuanto más sencillas sean las interfaces y los objetos, más probable es que hagan una sola cosa, lo cual es un buen estilo de programación porque minimiza los errores y hace más comprensibles los grandes proyectos de software. Debido a la abstracción en cualquier programa que utilice *TradingSignal*, será posible sustituir una señal por otra. También podemos sustituir la estrategia. Nuestras estrategias se encargan ahora de preparar y enviar las órdenes, y las señales las inician basándose en el análisis del mercado.

En nuestro caso, vamos a empaquetar la implementación específica de *TradingSignal* en la clase *BandOsMaSignal*. Por supuesto, necesitamos variables para almacenar los descriptores de los 3 indicadores. Las instancias del indicador se crean y eliminan en el constructor y el destructor, respectivamente. Todos los parámetros se pasarán desde las variables de entrada. Tenga en cuenta que *iBands* y *iMA* se basan en el manejador *hOsMA*.

```
class BandOsMaSignal: public TradingSignal
{
   int hOsMA, hBands, hMA;
   int direction;
public:
   BandOsMaSignal(const int fast, const int slow, const int signal,
      const ENUM_APPLIED_PRICE price,
      const int bands, const int shift, const double deviation,
      const int period, const int x, ENUM_MA_METHOD method)
   {
      hOsMA = iOsMA(_Symbol, _Period, fast, slow, signal, price);
      hBands = iBands(_Symbol, _Period, bands, shift, deviation, hOsMA);
      hMA = iMA(_Symbol, _Period, period, x, method, hOsMA);
      direction = 0;
   }

   ~BandOsMaSignal()
   {
      IndicatorRelease(hMA);
      IndicatorRelease(hBands);
      IndicatorRelease(hOsMA);
   }
   ...
```

La dirección de la señal de trading actual se coloca en la variable *direction*: 0 - sin señales (situación indefinida), +1 - compra, -1 - venta. Rellenaremos esta variable en el método *signal*. Su código repite la descripción verbal anterior de las señales en MQL5.

```
      virtual int signal(void) override
      {
         double osma[2], upper[2], lower[2], ma[2];
         // get two values of each indicator on bars 1 and 2
         if(CopyBuffer(hOsMA, 0, 1, 2, osma) != 2) return 0;
         if(CopyBuffer(hBands, UPPER_BAND, 1, 2, upper) != 2) return 0;
         if(CopyBuffer(hBands, LOWER_BAND, 1, 2, lower) != 2) return 0;
         if(CopyBuffer(hMA, 0, 1, 2, ma) != 2) return 0;

         // if there was a signal already, check if it has ended
         if(direction != 0)
         {
            if(direction > 0)
            {
               if(osma[0] >= ma[0] && osma[1] < ma[1])
               {
                  direction = 0;
               }
            }
            else
            {
               if(osma[0] <= ma[0] && osma[1] > ma[1])
               {
                  direction = 0;
               }
            }
         }

         // in any case, check if there is a new signal
         if(osma[0] <= lower[0] && osma[1] > lower[1])
         {
            direction = +1;
         }
         else if(osma[0] >= upper[0] && osma[1] < upper[1])
         {
            direction = -1;
         }

         return direction;
      }
   };
```

Como puede ver, los valores del indicador se leen para las barras 1 y 2, ya que trabajaremos en la apertura de una barra, y la barra 0 acaba de abrirse cuando llamamos al método *signal*.

La nueva clase que implemente la interfaz *TradingStrategy* se llamará *SimpleStrategy*.

La clase proporciona algunas características nuevas, al tiempo que utiliza algunas partes ya existentes. En concreto, ha conservado los punteros automáticos para *PositionState* y *TrailingStop* y tiene un nuevo puntero automático para la señal *TradingSignal*. Además, como vamos a operar sólo en la apertura de barras, necesitamos la variable *lastBar*, que almacenará la hora de la última barra procesada.

```
class SimpleStrategy: public TradingStrategy
{
protected:
   AutoPtr<PositionState> position;
   AutoPtr<TrailingStop> trailing;
   AutoPtr<TradingSignal> command;

   const int stopLoss;
   const ulong magic;
   const double lots;

   datetime lastBar;
   ...
```

Los parámetros globales se pasan al constructor *SimpleStrategy*. También pasamos un puntero al objeto *TradingSignal*: en este caso, será *BandOsMaSignal* el que tendrá que crear el código de llamada. A continuación, el constructor intenta encontrar entre las posiciones existentes aquellas que tengan el número y el símbolo *magic* requeridos y, si lo consigue, añade un trailing stop. Esto será útil si el Asesor Experto tiene una interrupción por una razón u otra, y la posición ya ha sido abierta.

```
public:
   SimpleStrategy(TradingSignal *signal, const ulong m, const int sl, const double v)
      command(signal), magic(m), stopLoss(sl), lots(v), lastBar(0)
   {
 // select "our" position among the existing ones (if there is a suitable one)
      PositionFilter positions;
      ulong tickets[];
      positions.let(POSITION_MAGIC, magic).let(POSITION_SYMBOL, _Symbol).select(ticke
      const int n = ArraySize(tickets);
      if(n > 1)
      {
         Alert(StringFormat("Too many positions: %d", n));
 // TODO: close extra positions - this is not allowed by the strategy
      }
      else if(n > 0)
      {
         position = new PositionState(tickets[0]);
         if(stopLoss)
         {
           trailing = new TrailingStop(tickets[0], stopLoss, stopLoss / 50);
         }
      }
   }
```

La implementación del método *trade* es similar al ejemplo de la martingala. No obstante, hemos eliminado las multiplicaciones de lotes y hemos añadido la llamada al método *signal*.

```
virtual bool trade() override
{
   // we work only once when a new bar appears
   if(lastBar == iTime(_Symbol, _Period, 0)) return false;

   int s = command[].signal(); // getting a signal

   ulong ticket = 0;

   if(position[] != NULL)
   {
      if(position[].refresh()) // position exists
      {
         // the signal has changed to the opposite or disappeared
         if((position[].get(POSITION_TYPE) == POSITION_TYPE_BUY && s != +1)
         || (position[].get(POSITION_TYPE) == POSITION_TYPE_SELL && s != -1))
         {
            PrintFormat("Signal lost: %d for position %d %lld",
               s, position[].get(POSITION_TYPE), position[].get(POSITION_TICKET));
            if(close(position[].get(POSITION_TICKET)))
            {
               position = NULL;
            }
            else
            {
              // update internal flag 'ready'
              // according to whether or not there was a closure
               position[].refresh();
            }
         }
         else
         {
            position[].update();
            if(trailing[]) trailing[].trail();
         }
      }
      else // position is closed
      {
         position = NULL;
      }
   }

   if(position[] == NULL && s != 0)
   {
      ticket = (s == +1) ? openBuy() : openSell();
   }

   if(ticket > 0) // new position just opened
   {
      position = new PositionState(ticket);
      if(stopLoss)
```

```
         {
            trailing = new TrailingStop(ticket, stopLoss, stopLoss / 50);
         }
      }
      // store the current bar
      lastBar = iTime(_Symbol, _Period, 0);

      return true;
   }
```

Los métodos auxiliares *openBuy*, *openSell* y otros han sufrido cambios mínimos, por lo que no los enumeraremos (se adjunta el código fuente completo).

Dado que siempre tenemos una sola estrategia en este Asesor Experto, en contraste con la martingala multidivisa en la que cada símbolo requería su propia configuración, vamos a excluir el grupo de estrategias y gestionar el objeto estrategia directamente.

```
AutoPtr<TradingStrategy> strategy;

int OnInit()
{
   if(FastOsMA >= SlowOsMA) return INIT_PARAMETERS_INCORRECT;
   strategy = new SimpleStrategy(
      new BandOsMaSignal(FastOsMA, SlowOsMA, SignalOsMA, PriceOsMA,
         BandsMA, BandsShift, BandsDeviation,
         PeriodMA, ShiftMA, MethodMA),
         Magic, StopLoss, Lots);
   return INIT_SUCCEEDED;
}

void OnTick()
{
   if(strategy[] != NULL)
   {
      strategy[].trade();
   }
}
```

Ahora tenemos un Asesor Experto listo que podemos utilizar como herramienta para estudiar el probador. En primer lugar, vamos a crear una estructura auxiliar *TesterRecord* para consultar y almacenar todos los datos estadísticos.

```
struct TesterRecord
{
   string feature;
   double value;

   static void fill(TesterRecord &stats[])
   {
      ResetLastError();
      for(int i = 0; ; ++i)
      {
         const double v = TesterStatistics((ENUM_STATISTICS)i);
         if(_LastError) return;
         TesterRecord t = {EnumToString((ENUM_STATISTICS)i), v};
         PUSH(stats, t);
      }
   }
};
```

En este caso, el campo de cadena *feature* sólo es necesario para la salida de registro informativo. Para guardar todos los indicadores (por ejemplo, para poder generar más tarde su propio formulario de informe), basta con un simple array del tipo *double* de longitud adecuada.

Utilizando la estructura del manejador *OnDeinit*, nos aseguramos de que la API de MQL5 devuelva los mismos valores que el informe del probador.

```
void OnDeinit(const int)
{
   TesterRecord stats[];
   TesterRecord::fill(stats);
   ArrayPrint(stats, 2);
}
```

Por ejemplo, al ejecutar el Asesor Experto en EURUSD, H1 con un depósito de 10000 y sin ninguna optimización (con la configuración por defecto), obtendremos aproximadamente los siguientes valores para 2021 (fragmento):

```
                          [feature]  [value]
[ 0] "STAT_INITIAL_DEPOSIT"       10000.00
[ 1] "STAT_WITHDRAWAL"                0.00
[ 2] "STAT_PROFIT"                    6.01
[ 3] "STAT_GROSS_PROFIT"            303.63
[ 4] "STAT_GROSS_LOSS"             -297.62
[ 5] "STAT_MAX_PROFITTRADE"          15.15
[ 6] "STAT_MAX_LOSSTRADE"           -10.00
...
[27] "STAT_DEALS"                   476.00
[28] "STAT_TRADES"                  238.00
...
[37] "STAT_CONLOSSMAX_TRADES"         8.00
[38] "STAT_MAX_CONLOSS_TRADES"        8.00
[39] "STAT_PROFITTRADES_AVGCON"       2.00
[40] "STAT_LOSSTRADES_AVGCON"         2.00
```

Conociendo todos estos valores, podemos inventar nuestra propia fórmula para la métrica combinada de la calidad del Asesor Experto y, al mismo tiempo, la función de optimización objetivo. Pero, en cualquier caso, el valor de este indicador deberá comunicarse al probador. Y eso es lo que hace la función *OnTester*.

6.5.7 Evento OnTester

El evento *OnTester* se genera al finalizar la simulación del Asesor Experto en datos históricos (tanto una ejecución de probador separada iniciada por el usuario como una de las múltiples ejecuciones lanzadas automáticamente por el probador durante la optimización). Para manejar el evento *OnTester*, un programa MQL debe tener una función correspondiente en su código fuente, pero esto no es necesario. Incluso sin la función *OnTester*, los Asesores Expertos pueden optimizarse con éxito basándose en criterios estándar.

La función sólo puede utilizarse en Asesores Expertos.

double OnTester()

La función está diseñada para calcular algún valor de tipo *double*, utilizado como criterio de optimización personalizado (*Custom max*). La selección de criterios es importante sobre todo para el éxito de la optimización genética, al tiempo que permite también al usuario evaluar y comparar los efectos de diferentes ajustes.

En la optimización genética, los resultados se ordenan en una generación según el criterio descendente. Es decir, los resultados con el valor más alto se consideran los mejores desde el punto de vista del criterio de optimización. Los peores valores de esta clasificación se descartan posteriormente y no participan en la formación de la siguiente generación.

> Tenga en cuenta que los valores devueltos por la función *OnTester* sólo se tienen en cuenta cuando se selecciona un criterio personalizado en la configuración del probador. La disponibilidad de la función *OnTester* no significa automáticamente su utilización por el algoritmo genético.
>
> La API de MQL5 no proporciona los medios para averiguar mediante programación qué criterio de optimización ha seleccionado el usuario en la configuración del probador. A veces es muy importante saberlo para poder aplicar algoritmos analíticos propios para postprocesar los resultados de la optimización.

La función es llamada por el núcleo sólo en el probador, justo antes de la llamada de la función *OnDeinit*.

Para calcular el valor de retorno, podemos utilizar tanto las estadísticas estándar disponibles a través de la función *TesterStatistics* como sus cálculos arbitrarios.

En el Asesor Experto *BandOsMA.mq5*, creamos el manejador *OnTester* que tiene en cuenta varias métricas: el beneficio, la rentabilidad, el número de operaciones y el ratio de Sharpe. A continuación, multiplicamos todas las métricas tras sacar la raíz cuadrada de cada una. Por supuesto, cada desarrollador puede tener sus propias preferencias e ideas para construir esos criterios de calidad generalizados.

```
double sign(const double x)
{
   return x > 0 ? +1 : (x < 0 ? -1 : 0);
}

double OnTester()
{
   const double profit = TesterStatistics(STAT_PROFIT);
   return sign(profit) * sqrt(fabs(profit))
      * sqrt(TesterStatistics(STAT_PROFIT_FACTOR))
      * sqrt(TesterStatistics(STAT_TRADES))
      * sqrt(fabs(TesterStatistics(STAT_SHARPE_RATIO)));
}
```

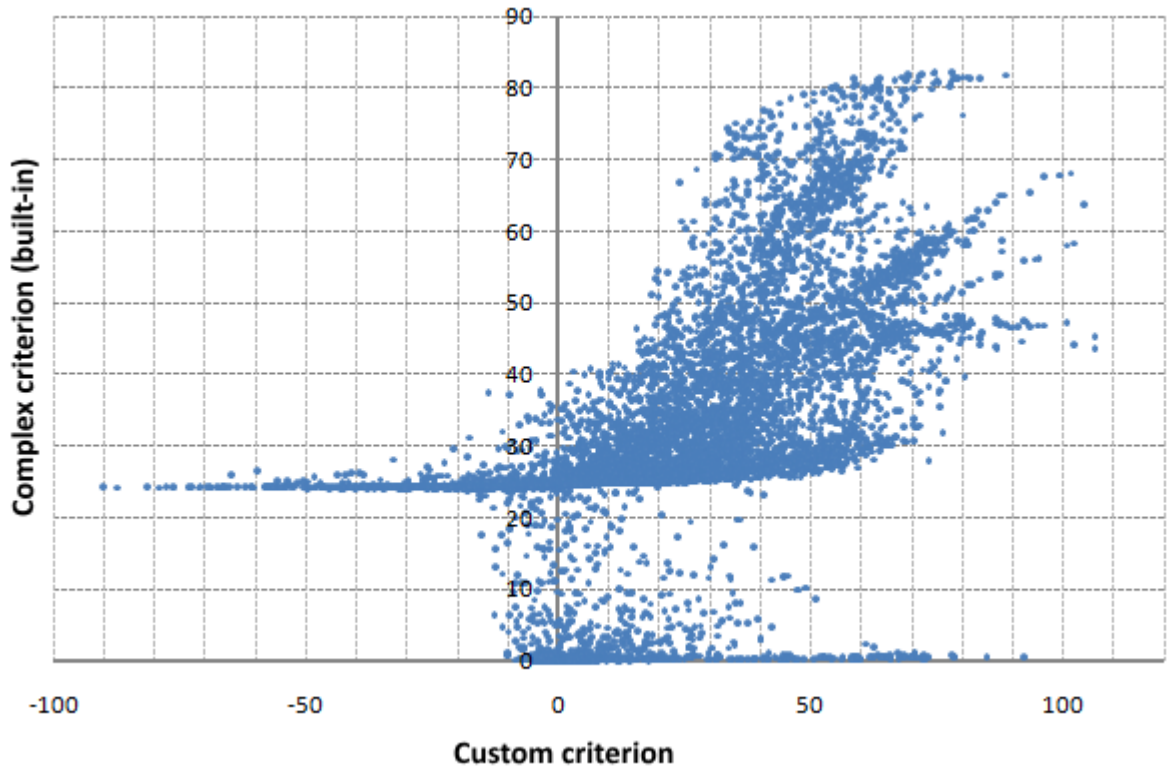
El registro de la prueba unitaria muestra una línea con el valor de la función *OnTester*.

Vamos a lanzar la optimización genética del Asesor Experto para 2021 en EURUSD, H1 con la selección de los parámetros del indicador y el tamaño de Stop Loss (el archivo *MQL5/Presets/MQL5Book/BandOsMA.set* se proporciona con el libro). Para comprobar la calidad de la optimización incluiremos también pruebas forward desde principios de 2022 (5 meses).

Primero, optimicemos según nuestro criterio.

Como usted sabe, MetaTrader 5 guarda todos los criterios estándar en los resultados de la optimización, además de la actual utilizada durante la optimización. Esto permite, una vez finalizada la optimización, analizar los resultados desde distintos puntos seleccionando determinados criterios en la lista desplegable de la esquina superior derecha del panel con la tabla. Así pues, aunque realizamos la optimización según nuestro propio criterio, también disponemos del criterio complejo integrado más interesante.

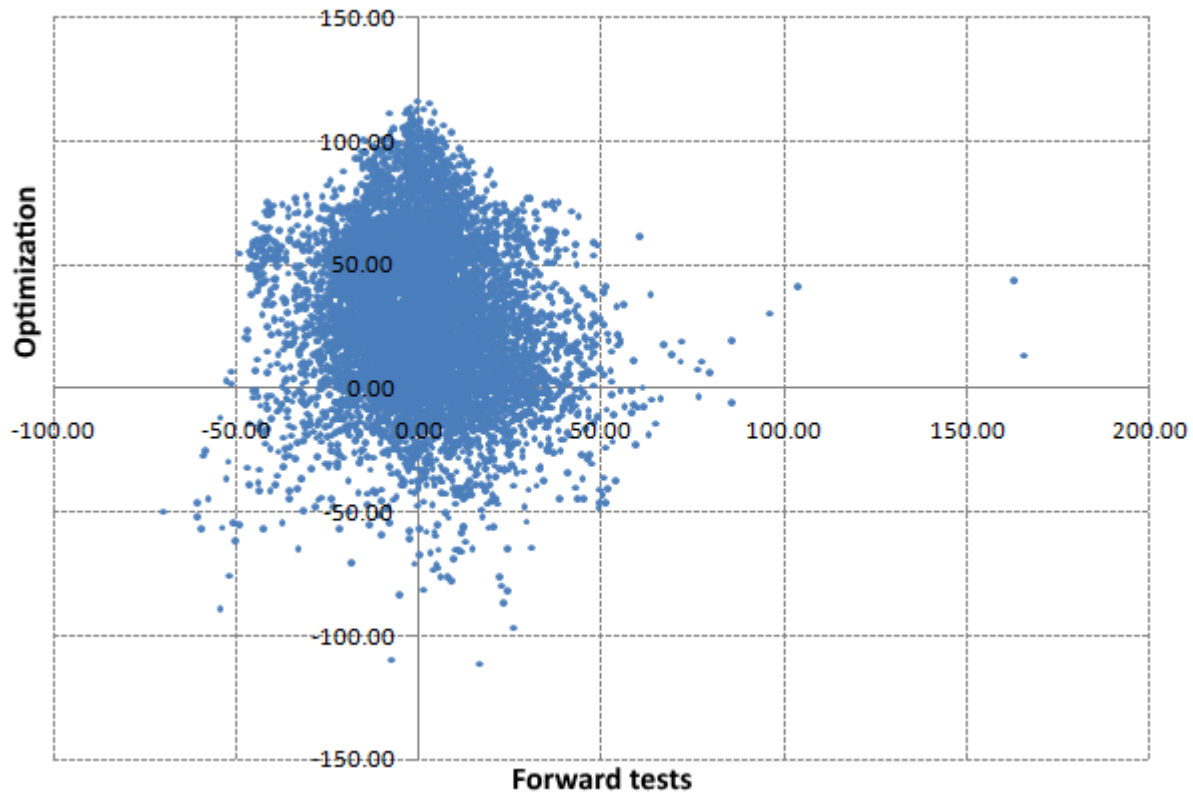
Podemos exportar la tabla de optimización a un archivo XML, primero con nuestros criterios seleccionados, y después con un criterio complejo dando un nuevo nombre al archivo (desafortunadamente, sólo se escribe un criterio en el archivo de exportación; es importante no cambiar la clasificación entre dos exportaciones). Esto permite combinar dos tablas en un programa externo y construir un diagrama en el que se trazan dos criterios a lo largo de los ejes; cada punto indica una combinación de criterios en una ejecución.



Comparación de criterios de optimización personalizados y complejos

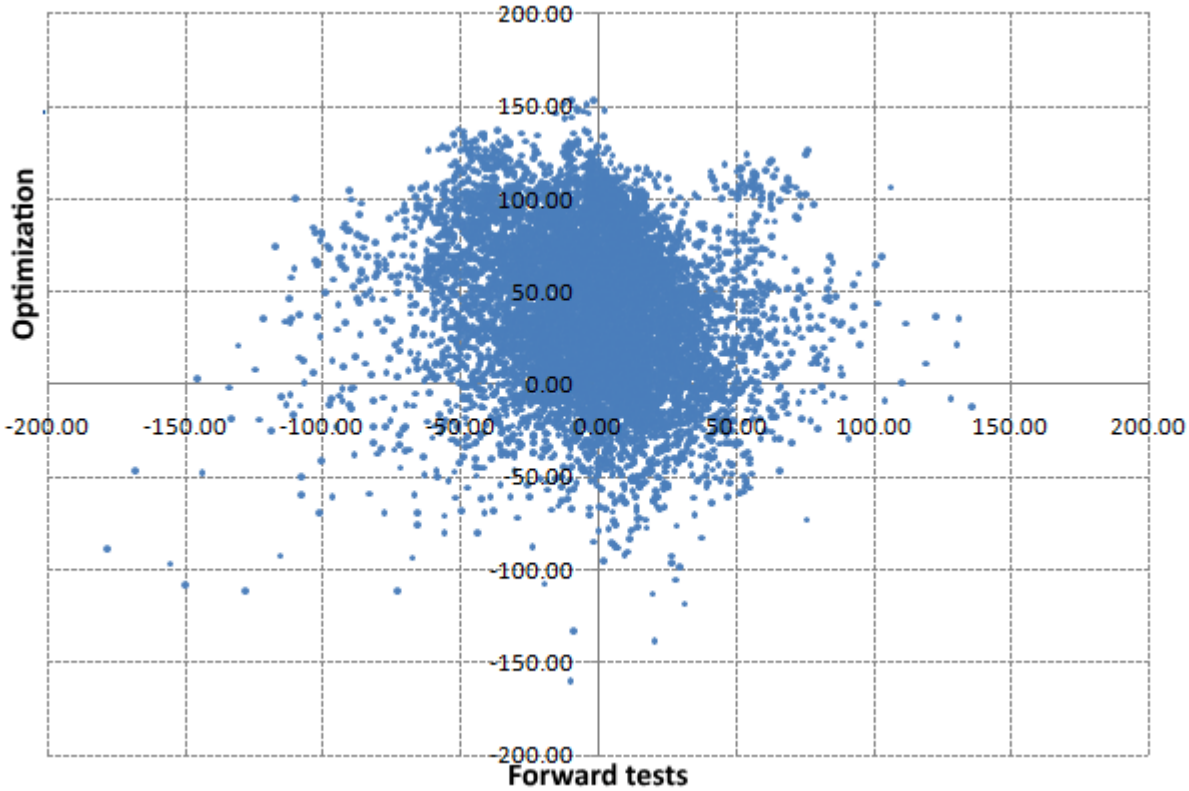
En un criterio complejo, observamos una estructura multinivel, ya que se calcula según una fórmula con condiciones: en algún lugar funciona una rama y en otro lugar funciona otra. Nuestros criterios personalizados se calculan siempre con la misma fórmula. También observamos la presencia de valores negativos en nuestro criterio (esto es de esperar) y el rango declarado de 0-100 para el criterio complejo.

Vamos a comprobar lo bueno que es nuestro criterio analizando sus valores para el periodo forward.



Valores del criterio personalizado en periodos de optimización y pruebas forward

Como era de esperar, sólo una parte de los buenos indicadores de optimización se mantuvieron en forward. Sin embargo, estamos más interesados, no en el criterio, sino en el beneficio. Veamos su distribución en el enlace optimización-forward.



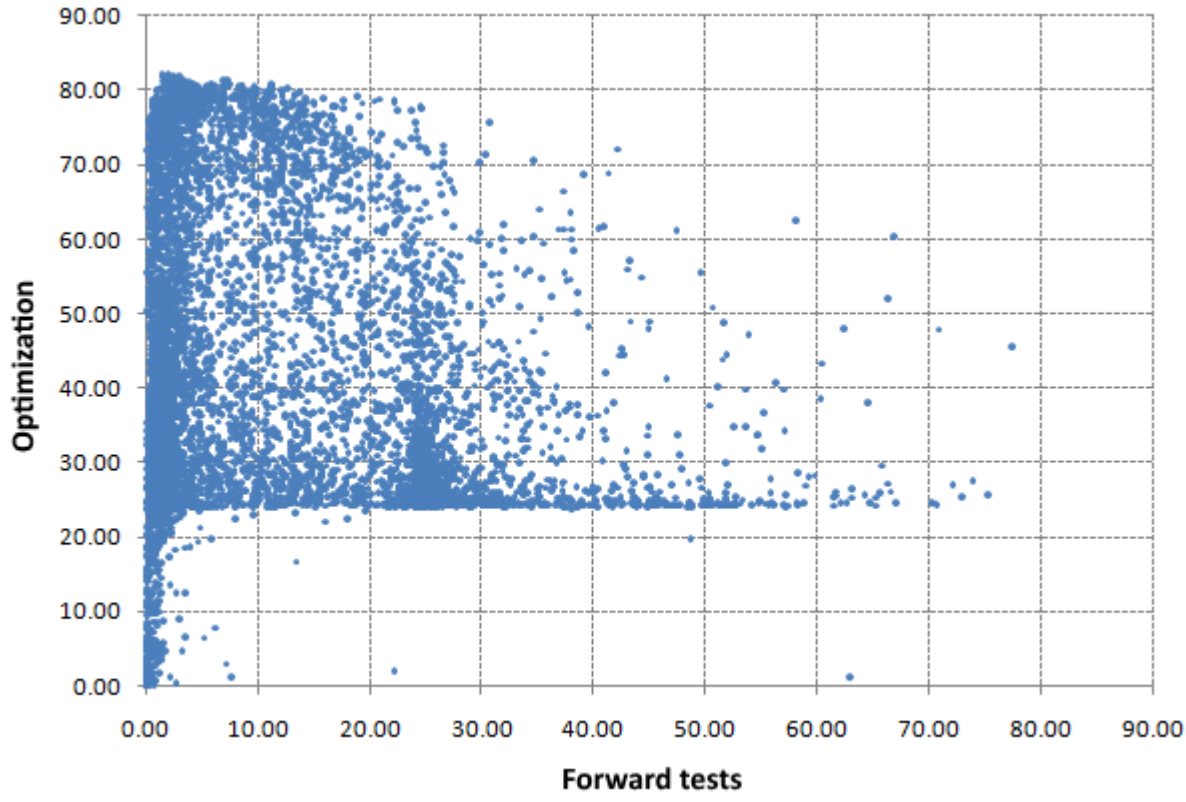
Beneficio en periodos de optimización y pruebas forward

El panorama aquí es similar. De los 6850 pases con beneficio en el periodo de optimización, 3123 resultaron ser rentables también a plazo (45 %). Y de las 1000 mejores, sólo 323 fueron rentables, lo que no es suficiente. Por lo tanto, este Asesor Experto necesitará mucho trabajo para identificar configuraciones rentables estables. ¿Pero tal vez sea el problema de los criterios de optimización?

Repitamos la optimización, esta vez utilizando el criterio complejo integrado.

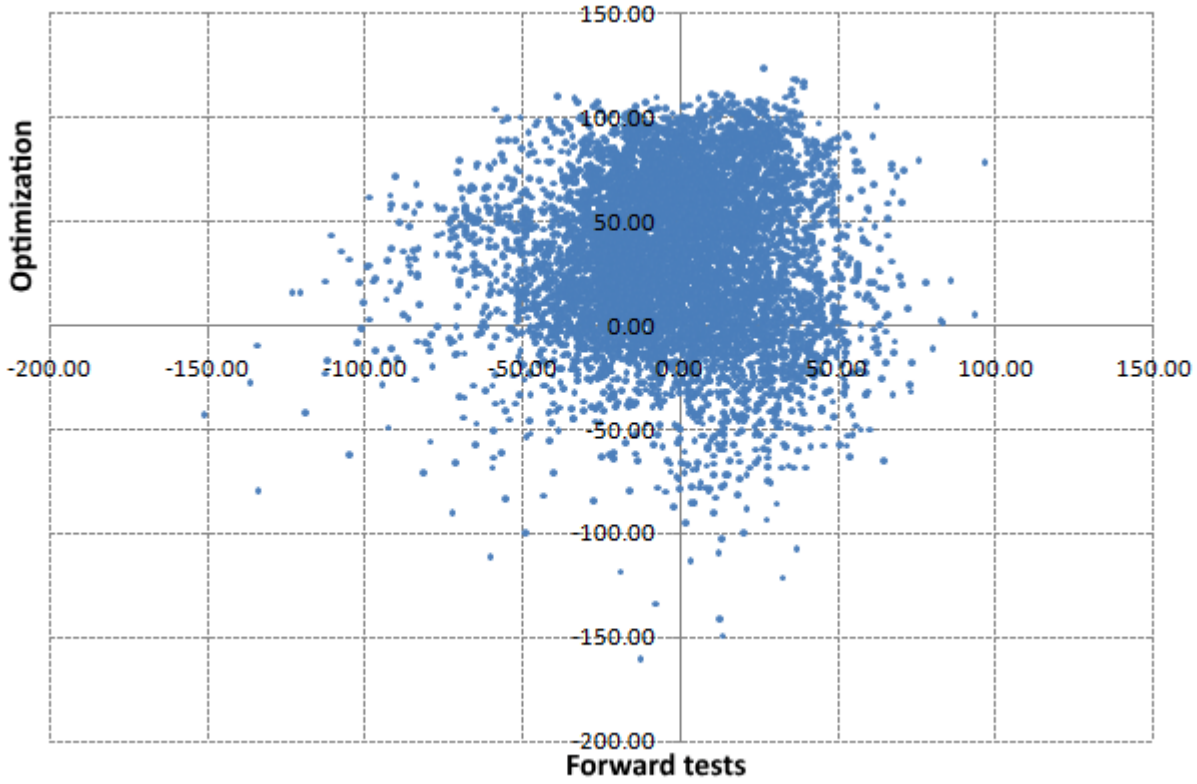
> ¡Atención! MetaTrader 5 genera cachés de optimización durante las optimizaciones: archivos opt en *Tester/cache*. Al iniciar la siguiente optimización, busca cachés adecuados para continuar la optimización. Si existe un archivo de caché con la configuración anterior, el proceso no empieza desde el principio, sino que tiene en cuenta los resultados anteriores. Esto le permite construir optimizaciones genéticas en cadena, asumiendo que encuentra los mejores resultados (al fin y al cabo, cada optimización genética es un proceso aleatorio).
>
> MetaTrader 5 no tiene en cuenta el criterio de optimización como factor distintivo en la configuración. Esto puede ser útil en algunos casos, basándonos en lo anterior, pero interferirá con nuestra tarea actual. Para llevar a cabo un experimento puro, necesitamos una optimización desde cero. Por lo tanto, inmediatamente después de la primera optimización utilizando nuestro criterio, no podemos lanzar la segunda utilizando el criterio complejo.
>
> No hay forma de desactivar el comportamiento actual desde la interfaz del terminal. Por lo tanto, deberá eliminar o renombrar (cambiar la extensión) el archivo opt anterior manualmente en cualquier gestor de archivos. Un poco más adelante nos familiarizaremos con la directiva de preprocesador para el probador *tester_no_cache*, que se puede especificar en el código fuente de un Asesor Experto en particular, lo que le permite desactivar la lectura de la caché.

La comparación de los valores del criterio complejo en los periodos de optimización y el periodo futuro adopta la siguiente forma:



Criterio complejo para periodos de optimización y pruebas forward

Aquí está la estabilidad de los beneficios a plazo.



Beneficio en periodos de optimización y pruebas forward

De los 5952 resultados positivos de la historia, sólo 2655 (también alrededor del 45 %) se mantuvieron en negro. Pero de los 1000 primeros, 581 resultaron tener éxito en forward.

Así pues, hemos visto que es bastante sencillo utilizar *OnTester* desde el punto de vista técnico, pero nuestro criterio funciona peor que el integrado (ceteris paribus), aunque dista mucho de ser ideal. Así pues, desde el punto de vista de la búsqueda de la fórmula del propio criterio y la posterior elección razonable de los parámetros sin mirar hacia el futuro, hay más preguntas sobre el contenido de *OnTester* que respuestas.

Aquí, la programación desemboca sin problemas en la investigación y la actividad científica, y queda fuera del alcance de este libro. Pero daremos un ejemplo de criterio calculado con nuestra propia métrica, y no con métricas ya hechas: *TesterStatistics*. Hablaremos del criterio R2, también conocido como coeficiente de determinación (*RSquared.mqh*).

Vamos a crear una función para calcular R2 a partir de la curva de equilibrio. Se sabe que cuando se negocia con un lote permanente, un sistema de trading ideal debería mostrar el saldo en forma de línea recta. Ahora utilizamos un lote permanente, por lo que nos vendrá bien. En cuanto a R2 en el caso de lotes variables, lo trataremos un poco más adelante.

Al final, R2 es una medida inversa de la varianza de los datos en relación con la regresión lineal construida sobre ellos. El rango de valores de R2 va de menos infinito a +1 (aunque en nuestro caso es muy improbable que haya grandes valores negativos). Es obvio que la línea encontrada se caracteriza simultáneamente por una pendiente; por lo tanto, para universalizar el código, guardaremos tanto R2 como la tangente del ángulo en la estructura R2A como resultado intermedio.

```
struct R2A
{
   double r2;    // square of correlation coefficient
   double angle; // tangent of the slope
   R2A(): r2(0), angle(0) { }
};
```

El cálculo de los indicadores se realiza en la función *RSquared* que toma un array de datos como entrada y devuelve una estructura R2A.

```
R2A RSquared(const double &data[])
{
   int size = ArraySize(data);
   if(size <= 2) return R2A();
   double x, y, div;
   int k = 0;
   double Sx = 0, Sy = 0, Sxy = 0, Sx2 = 0, Sy2 = 0;
   for(int i = 0; i < size; ++i)
   {
      if(data[i] == EMPTY_VALUE
      || !MathIsValidNumber(data[i])) continue;
      x = i + 1;
      y = data[i];
      Sx  += x;
      Sy  += y;
      Sxy += x * y;
      Sx2 += x * x;
      Sy2 += y * y;
      ++k;
   }
   size = k;
   const double Sx22 = Sx * Sx / size;
   const double Sy22 = Sy * Sy / size;
   const double SxSy = Sx * Sy / size;
   div = (Sx2 - Sx22) * (Sy2 - Sy22);
   if(fabs(div) < DBL_EPSILON) return R2A();
   R2A result;
   result.r2 = (Sxy - SxSy) * (Sxy - SxSy) / div;
   result.angle = (Sxy - SxSy) / (Sx2 - Sx22);
   return result;
}
```

Para la optimización necesitamos un valor de criterio, y aquí el ángulo es importante porque una curva de equilibrio descendente suave con una pendiente negativa también puede obtener una buena estimación R2. Por lo tanto, escribiremos una función más que «sumará menos» a cualquier estimación de R2 con un ángulo negativo. Tomamos el valor de R2 módulo porque puede ser negativo en el caso de datos muy malos (dispersos) que no encajan en nuestro modelo lineal. Por lo tanto, debemos evitar una situación en la que un menos por un menos dé un más.

```
double RSquaredTest(const double &data[])
{
   const R2A result = RSquared(data);
   const double weight = 1.0 - 1.0 / sqrt(ArraySize(data) + 1);
   if(result.angle < 0) return -fabs(result.r2) * weight;
   return result.r2 * weight;
}
```

Además, nuestro criterio tiene en cuenta el tamaño de la serie, que corresponde al número de operaciones. Por ello, un aumento del número de transacciones incrementará el indicador.

Teniendo esta herramienta a nuestra disposición, implementaremos la función de calcular la línea de balance en el Asesor Experto y encontraremos R2 para ello. Al final, multiplicamos el valor por 100, convirtiendo así la escala al rango del criterio complejo integrado.

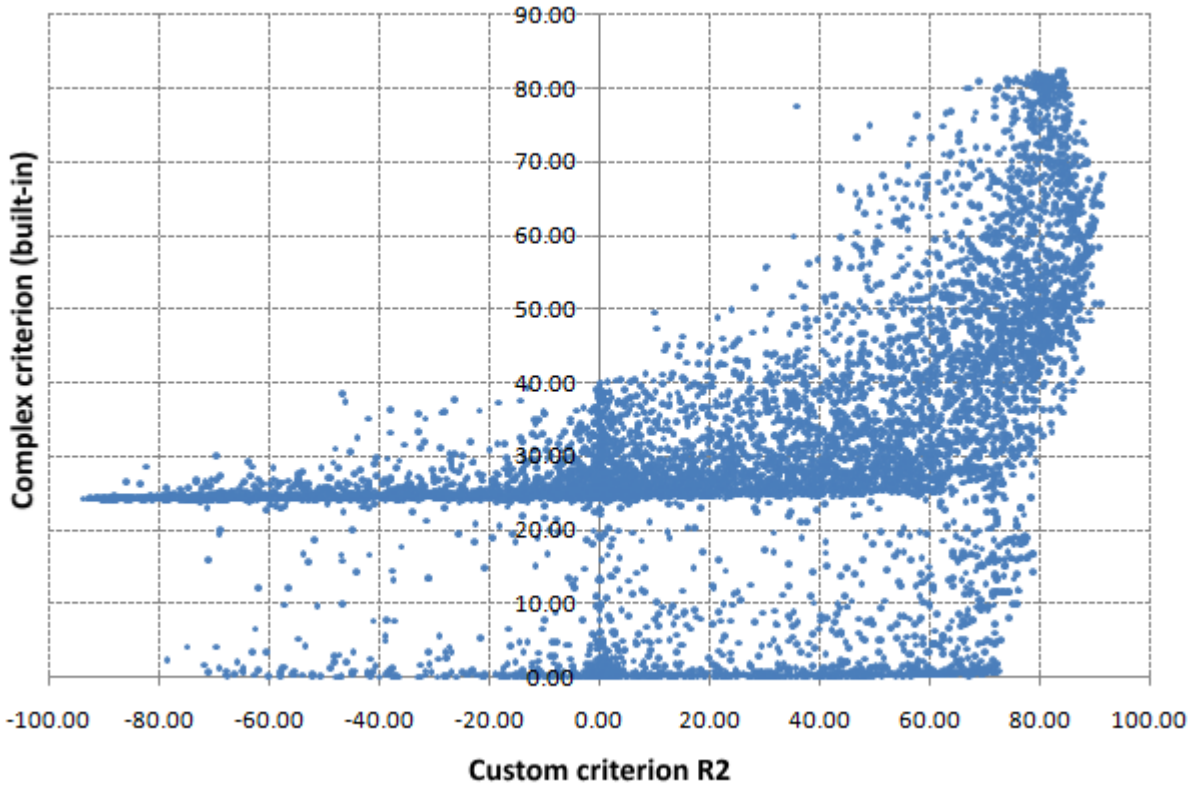
```
#define STAT_PROPS 4

double GetR2onBalanceCurve()
{
   HistorySelect(0, LONG_MAX);

   const ENUM_DEAL_PROPERTY_DOUBLE props[STAT_PROPS] =
   {
      DEAL_PROFIT, DEAL_SWAP, DEAL_COMMISSION, DEAL_FEE
   };
   double expenses[][STAT_PROPS];
   ulong tickets[]; // only needed because of the 'select' prototype, but useful for

   DealFilter filter;
   filter.let(DEAL_TYPE, (1 << DEAL_TYPE_BUY) | (1 << DEAL_TYPE_SELL), IS::OR_BITWISE
      .let(DEAL_ENTRY,
      (1 << DEAL_ENTRY_OUT) | (1 << DEAL_ENTRY_INOUT) | (1 << DEAL_ENTRY_OUT_BY),
      IS::OR_BITWISE)
      .select(props, tickets, expenses);

   const int n = ArraySize(tickets);

   double balance[];

   ArrayResize(balance, n + 1);
   balance[0] = TesterStatistics(STAT_INITIAL_DEPOSIT);

   for(int i = 0; i < n; ++i)
   {
      double result = 0;
      for(int j = 0; j < STAT_PROPS; ++j)
      {
         result += expenses[i][j];
      }
      balance[i + 1] = result + balance[i];
   }
   const double r2 = RSquaredTest(balance);
   return r2 * 100;
}
```

En el manejador *OnTester*, utilizaremos el nuevo criterio bajo la directiva de compilación condicional, por lo que necesitamos descomentar la directiva *#define USE_R2_CRITERION* al principio del código fuente.

```
double OnTester()
{
#ifdef USE_R2_CRITERION
   return GetR2onBalanceCurve();
#else
   const double profit = TesterStatistics(STAT_PROFIT);
   return sign(profit) * sqrt(fabs(profit))
      * sqrt(TesterStatistics(STAT_PROFIT_FACTOR))
      * sqrt(TesterStatistics(STAT_TRADES))
      * sqrt(fabs(TesterStatistics(STAT_SHARPE_RATIO)));
#endif
}
```

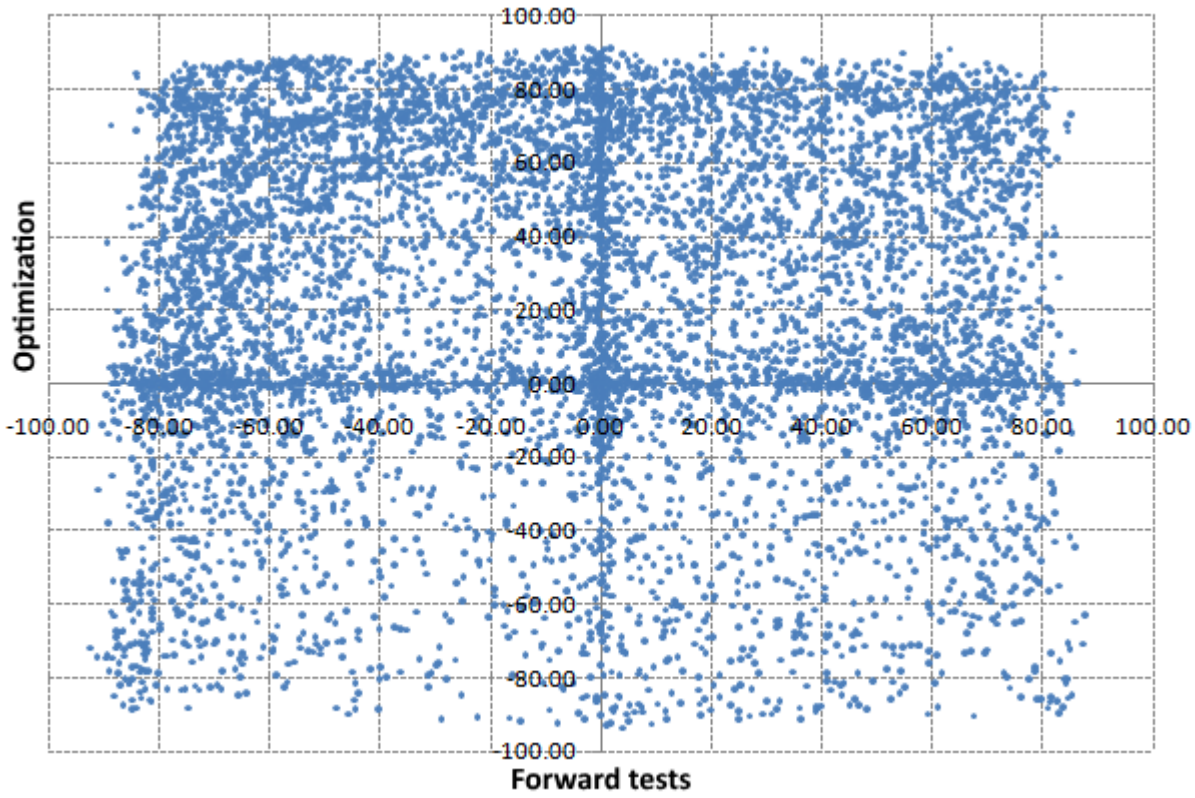
Vamos a borrar los resultados anteriores de las optimizaciones (archivos opt con caché) y a lanzar una nueva optimización del Asesor Experto: por el criterio R2.

Al comparar los valores del criterio R2 con los del criterio complejo, podemos afirmar que ha aumentado la «convergencia» entre ambos.



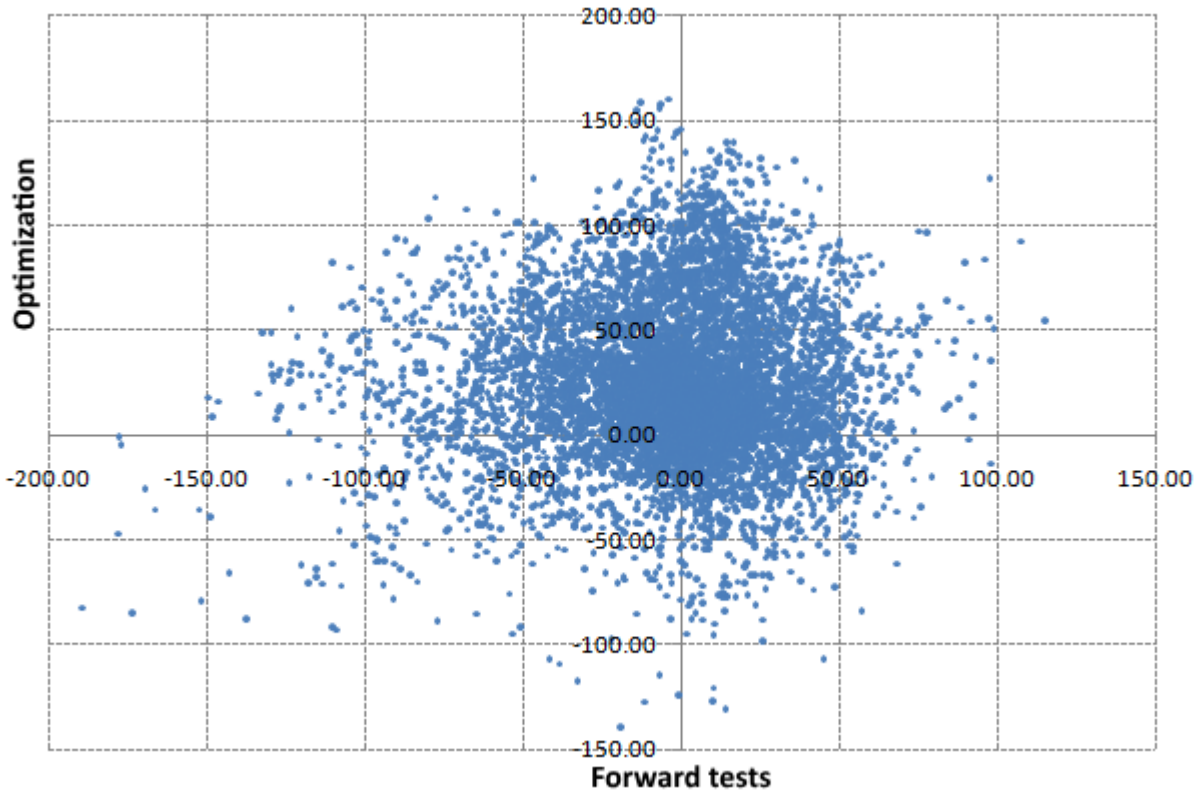
Comparación del criterio personalizado R2 y el criterio integrado complejo

Los valores del criterio R2 en la ventana de optimización y en el periodo anterior para los conjuntos de parámetros correspondientes tienen el siguiente aspecto:



Criterio R2 sobre los periodos de optimización y las pruebas forward

Y así es como se combinan los beneficios en el pasado y en el futuro:



Beneficio en periodos de optimización y pruebas forward para R2

Las estadísticas son las siguientes: de los últimos 5582 pases rentables, 2638 (47 %) siguieron siendo rentables, y de los 1000 primeros pases más rentables hay 566 que siguieron siendo rentables, lo que es comparable con el criterio de complejidad integrado.

Como ya se ha dicho, las estadísticas proporcionan materia prima para las siguientes fases de optimización, más inteligentes, que son algo más que una tarea de programación. Nos centraremos en otros aspectos puramente programáticos de la optimización.

6.5.8 Sintonización automática: ParameterGetRange y ParameterSetRange

En la sección anterior aprendimos a pasar un criterio de optimización al probador. Sin embargo, hemos pasado por alto un punto importante. Si echa un vistazo a nuestros registros de optimización, podrá ver muchos mensajes de error, como los que aparecen a continuación:

```
...
Best result 90.61004580175876 produced at generation 25. Next generation 26
genetic pass (26, 388) tested with error "incorrect input parameters" in 0:00:00.021
genetic pass (26, 436) tested with error "incorrect input parameters" in 0:00:00.007
genetic pass (26, 439) tested with error "incorrect input parameters" in 0:00:00.007
genetic pass (26, 363) tested with error "incorrect input parameters" in 0:00:00.008
genetic pass (26, 365) tested with error "incorrect input parameters" in 0:00:00.008
...
```

En otras palabras: cada pocas pasadas de prueba, algo falla en los parámetros de entrada y la pasada en cuestión no se realiza. El manejador *OnInit* contiene la siguiente comprobación:

```
if(FastOsMA >= SlowOsMA) return INIT_PARAMETERS_INCORRECT;
```

Por nuestra parte, es bastante lógico imponer la restricción de que el periodo de la media móvil (MA) lenta sea mayor que el periodo de la media móvil rápida. Sin embargo, el probador no sabe esas cosas sobre nuestro algoritmo, por lo que intenta clasificar una gran variedad de combinaciones de periodos, incluidas las incorrectas. Esta podría ser una situación común para la optimización que, sin embargo, tiene una consecuencia negativa.

Como aplicamos la optimización genética, en cada generación hay varias muestras rechazadas que no participan en mutaciones posteriores. El optimizador de MetaTrader 5 no compensa estas pérdidas, es decir, no genera un sustituto para ellas. Un menor tamaño de la población puede afectar entonces negativamente a la calidad. Por lo tanto, es necesario idear una forma de garantizar que las configuraciones de entrada se enumeran sólo en las combinaciones correctas. Y aquí vienen en nuestra ayuda dos funciones de la API de MQL5: *ParameterGetRange* y *ParameterSetRange*.

Ambas funciones tienen dos prototipos sobrecargados que difieren en los tipos de parámetros: *long* y *double*. Así se describen las dos variantes de la función *ParameterGetRange*:

bool ParameterGetRange(const string name, bool &enable, long &value, long &start, long &step, long &stop)
bool ParameterGetRange(const string name, bool &enable, double &value, double &start, double &step, double &stop)

Para la variable de entrada especificada por nombre, la función recibe información sobre su valor actual (*value*), rango de valores (*start*, *stop*) y paso de cambio (*step*) durante la optimización. Además, se escribe un atributo en la variable *enable* de si la optimización está activada para la variable de entrada llamada 'nombre'.

La función devuelve un indicador de éxito (*true*) o de error (*false*).

La función sólo puede invocarse desde tres manejadores especiales relacionados con la optimización: *OnTesterInit*, *OnTesterPass* y *OnTesterDeinit*. Hablaremos de ellos en la sección siguiente. Como se puede adivinar por los nombres, *OnTesterInit* se llama antes de que comience la optimización, *OnTesterDeinit* después de la finalización de la optimización, y *OnTesterPass* después de cada pasada en el proceso de optimización. Por ahora, sólo nos interesa *OnTesterInit*. Al igual que las otras dos funciones, no tiene parámetros y puede declararse con el tipo *void*, es decir, no devuelve nada.

Dos versiones de la función *ParameterSetRange* tienen prototipos similares y realizan la acción opuesta: establecen las propiedades de optimización del parámetro de entrada del Asesor Experto.

```
bool ParameterSetRange(const string name, bool enable, long value, long start, long step, long stop)
bool ParameterSetRange(const string name, bool enable, double value, double start, double step,
double stop)
```

La función establece las reglas de modificación de la variable *input* con el nombre *name* al realizar la optimización: valor, paso de modificación, valores inicial y final.

Esta función sólo puede ser llamada desde el manejador *OnTesterInit* cuando se inicia la optimización en el probador de estrategias.

Así, utilizando las funciones *ParameterGetRange* y *ParameterSetRange*, puede analizar y establecer nuevos valores de rango y paso, así como excluir completamente o, viceversa, incluir ciertos parámetros de la optimización, a pesar de los ajustes en el probador de estrategias. Esto le permite crear sus propios scripts para gestionar el espacio de parámetros de entrada durante la optimización.

La función permite utilizar en la optimización incluso aquellas variables que se declaran con el modificador *sinput* (no están disponibles para su inclusión en la optimización por parte del usuario).

> ¡Atención! Después de la llamada de *ParameterSetRange* con un cambio en los ajustes de una variable de entrada específica, las llamadas posteriores de *ParameterGetRange* no «verán» estos cambios y seguirán volviendo a los ajustes originales. Esto imposibilita el uso conjunto de funciones en productos de software complejos, en los que las configuraciones pueden ser gestionadas por diferentes clases y bibliotecas de desarrolladores independientes.

Mejoremos el Asesor Experto *BandOsMA* utilizando las nuevas funciones. La versión actualizada se denomina *BandOsMApro.mq5* («pro» puede descodificarse condicionalmente como «optimización del rango de parámetros»).

Así, tenemos el manejador *OnTesterInit*, en el que leemos la configuración de los parámetros *FastOsMA* y *SlowOsMA*, y comprobamos si están incluidos en la optimización. Si es así, hay que desactivarlos y ofrecer algo a cambio.

```
void OnTesterInit()
{
   bool enabled1, enabled2;
   long value1, start1, step1, stop1;
   long value2, start2, step2, stop2;
   if(ParameterGetRange("FastOsMA", enabled1, value1, start1, step1, stop1)
   && ParameterGetRange("SlowOsMA", enabled2, value2, start2, step2, stop2))
   {
      if(enabled1 && enabled2)
      {
         if(!ParameterSetRange("FastOsMA", false, value1, start1, step1, stop1)
         || !ParameterSetRange("SlowOsMA", false, value2, start2, step2, stop2))
         {
            Print("Can't disable optimization by FastOsMA and SlowOsMA: ",
               E2S(_LastError));
            return;
         }
         ...
      }
   }
   else
   {
      Print("Can't adjust optimization by FastOsMA and SlowOsMA: ", E2S(_LastError));
   }
}
```

Por desgracia, debido a la adición de *OnTesterInit*, el compilador también requiere que añada *OnTesterDeinit*, aunque no necesitamos esta función, pero nos vemos obligados a aceptar y añadir un manejador vacío.

```
void OnTesterDeinit()
{
}
```

La presencia de las funciones *OnTesterInit/OnTesterDeinit* en el código conducirá al hecho de que cuando se inicie la optimización, se abrirá un gráfico adicional en el terminal con una copia de nuestro Asesor Experto ejecutándose en él. Funciona en un modo especial que le permite recibir datos adicionales (los llamados *frames*) de copias probadas en agentes, pero exploraremos esta posibilidad más adelante. Por ahora, es importante que tengamos en cuenta que todas las operaciones con archivos, registros, gráficos y objetos funcionan en esta copia auxiliar del Asesor Experto directamente en el terminal, como de costumbre (y no en el agente). En concreto, todos los mensajes de error y las llamadas a *Print* se mostrarán en el registro de la pestaña *Experts* del terminal.

Tenemos información sobre los rangos de cambio y los pasos de estos parámetros, podemos literalmente recalcular todas las combinaciones correctas. Esta tarea se asigna a una función separada *Iterate* porque una operación similar tendrá que ser reproducida por copias del Asesor Experto en agentes, en el manejador *OnInit*.

En la función *Iterate* tenemos dos bucles anidados sobre los periodos de MA rápida y lenta en los que contamos el número de combinaciones válidas, es decir, cuando el periodo *i* es inferior a *j*. Necesitamos el parámetro opcional *find* al llamar a *Iterate* desde *OnInit* para devolver el par por el número de secuencia de la combinación *i* y *j*. Dado que se requiere devolver 2 números, declaramos la estructura *PairOfPeriods* para ellos.

```
struct PairOfPeriods
{
   int fast;
   int slow;
};

PairOfPeriods Iterate(const long start1, const long stop1, const long step1,
   const long start2, const long stop2, const long step2,
   const long find = -1)
{
   int count = 0;
   for(int i = (int)start1; i <= (int)stop1; i += (int)step1)
   {
      for(int j = (int)start2; j <= (int)stop2; j += (int)step2)
      {
         if(i < j)
         {
            if(count == find)
            {
               PairOfPeriods p = {i, j};
               return p;
            }
            ++count;
         }
      }
   }
   PairOfPeriods p = {count, 0};
   return p;
}
```

Al llamar a *Iterate* desde *OnTesterInit*, no utilizamos el parámetro *find* y seguimos contando hasta el final, y devolvemos la cantidad resultante en el primer campo de la estructura. Este será el rango de valores de algún nuevo parámetro de sombra, para el que debemos activar la optimización. Vamos a llamarlo *FastSlowCombo4Optimization* y a añadirlo al nuevo grupo de parámetros auxiliares de entrada. Pronto añadiremos más.

```
input group "A U X I L I A R Y"
sinput int FastSlowCombo4Optimization = 0;   // (reserved for optimization)
...
```

Volvamos a *OnTesterInit* y organicemos una optimización de MQL5 por el parámetro *FastSlowCombo4Optimization* en el rango deseado utilizando *ParameterSetRange*.

```
void OnTesterInit()
{
   ...
         PairOfPeriods p = Iterate(start1, stop1, step1, start2, stop2, step2);
         const int count = p.fast;
         ParameterSetRange("FastSlowCombo4Optimization", true, 0, 0, 1, count);
         PrintFormat("Parameter FastSlowCombo4Optimization is enabled with maximum: %
            count);
   ...
}
```

Tenga en cuenta que el número de iteraciones resultante para el nuevo parámetro debe aparecer en el registro del terminal.

Cuando realice pruebas en el agente, utilice el número de *FastSlowCombo4Optimization* para obtener un par de períodos llamando de nuevo a *Iterate*, esta vez con el parámetro *find* lleno. Pero el problema es que para esta operación se requiere conocer los rangos iniciales y el paso de cambio de parámetros *FastOsMA* y *SlowOsMA*. Esta información sólo está presente en el terminal, por lo que tenemos que transferirla de alguna manera al agente.

Ahora aplicaremos la única solución que conocemos por el momento: añadiremos otros 3 parámetros de optimización de sombras y estableceremos algunos valores para ellos. En el futuro nos familiarizaremos con la tecnología de transferencia de archivos a los agentes (véase Directivas del preprocesador para el probador). Entonces podremos escribir en el archivo todo el array de índices calculado por la función *Iterate* y enviarlo a los agentes. Esto evitará tres parámetros adicionales de optimización de sombras.

Por lo tanto, vamos a añadir tres parámetros de entrada:

```
sinput ulong FastShadow4Optimization = 0;    // (reserved for optimization)
sinput ulong SlowShadow4Optimization = 0;    // (reserved for optimization)
sinput ulong StepsShadow4Optimization = 0;   // (reserved for optimization)
```

Utilizamos el tipo *ulong* para ser más económicos: para empaquetar 2 números *int* en cada valor. Así se rellenan en *OnTesterInit*.

```
void OnTesterInit()
{
   ...
         const ulong fast = start1 | (stop1 << 16);
         const ulong slow = start2 | (stop2 << 16);
         const ulong step = step1 | (step2 << 16);
         ParameterSetRange("FastShadow4Optimization", false, fast, fast, 1, fast);
         ParameterSetRange("SlowShadow4Optimization", false, slow, slow, 1, slow);
         ParameterSetRange("StepsShadow4Optimization", false, step, step, 1, step);
   ...
}
```

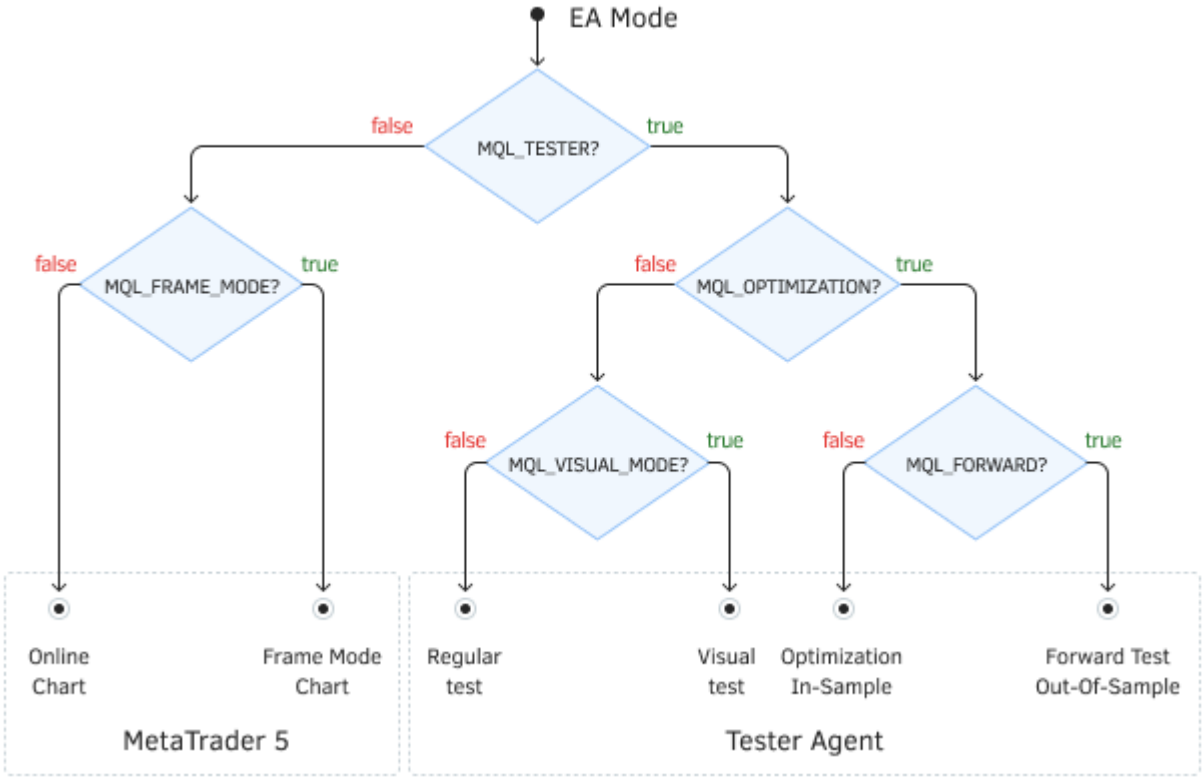
Los 3 parámetros no son optimizables (*false* en el segundo argumento).

Con esto concluyen nuestras operaciones con la función *OnTesterInit*. Pasemos al lado receptor: el manejador *OnInit*.

```
int OnInit()
{
   // keep the check for single tests
   if(FastOsMA >= SlowOsMA) return INIT_PARAMETERS_INCORRECT;

   // when optimizing, we require the presence of shadow parameters
   if(MQLInfoInteger(MQL_OPTIMIZATION) && StepsShadow4Optimization == 0)
   {
      return INIT_PARAMETERS_INCORRECT;
   }

   PairOfPeriods p = {FastOsMA, SlowOsMA}; // by default we work with normal paramete
   if(FastShadow4Optimization && SlowShadow4Optimization && StepsShadow4Optimization)
   {
      // if the shadow parameters are full, decode them into periods
      int FastStart = (int)(FastShadow4Optimization & 0xFFFF);
      int FastStop = (int)((FastShadow4Optimization >> 16) & 0xFFFF);
      int SlowStart = (int)(SlowShadow4Optimization & 0xFFFF);
      int SlowStop = (int)((SlowShadow4Optimization >> 16) & 0xFFFF);
      int FastStep = (int)(StepsShadow4Optimization & 0xFFFF);
      int SlowStep = (int)((StepsShadow4Optimization >> 16) & 0xFFFF);

      p = Iterate(FastStart, FastStop, FastStep,
         SlowStart, SlowStop, SlowStep, FastSlowCombo4Optimization);
      PrintFormat("MA periods are restored from shadow: FastOsMA=%d SlowOsMA=%d",
         p.fast, p.slow);
   }

   strategy = new SimpleStrategy(
      new BandOsMaSignal(p.fast, p.slow, SignalOsMA, PriceOsMA,
         BandsMA, BandsShift, BandsDeviation,
         PeriodMA, ShiftMA, MethodMA),
         Magic, StopLoss, Lots);
   return INIT_SUCCEEDED;
}
```

Utilizando la función *MQLInfoInteger* podemos determinar todos los modos del Asesor Experto, incluidos los relacionados con el probador y la optimización. Habiendo especificado uno de los elementos de la enumeración ENUM_MQL_INFO_INTEGER como parámetro, obtendremos como resultado un signo lógico (*true/false*):

- MQL_TESTER - el programa funciona en el probador
- MQL_VISUAL_MODE - el probador se ejecuta en modo visual
- MQL_OPTIMIZATION - la pasada de prueba se realiza durante la optimización (no por separado)
- MQL_FORWARD - la pasada de prueba se realiza en el período forward después de la optimización (si se especifica por la configuración de optimización)
- MQL_FRAME_MODE - el Asesor Experto se está ejecutando en un modo de servicio especial en el gráfico del terminal (y no en el agente) para controlar la optimización (encontrará más información al respecto en la sección siguiente)



Modos del probador de programas MQL

Todo está listo para iniciar la optimización. En cuanto se inicie, con la configuración mencionada *Presets/MQL5Book/BandOsMA.set*, veremos un mensaje en el registro *Experts* del terminal:

```
Parameter FastSlowCombo4Optimization is enabled with maximum: 698
```

Esta vez no debería haber errores en el registro de optimización y todas las generaciones se realizan sin bloquearse.

```
...
Best result 91.02452934181422 produced at generation 39. Next generation 42
Best result 91.56338892567393 produced at generation 42. Next generation 43
Best result 91.71026391877101 produced at generation 43. Next generation 44
Best result 91.71026391877101 produced at generation 43. Next generation 45
Best result 92.48460871443507 produced at generation 45. Next generation 46
...
```

Esto puede determinarse incluso por el aumento del tiempo total de optimización: antes, algunas pasadas se rechazaban en una fase temprana, y ahora todas se procesan en su totalidad.

No obstante, nuestra solución tiene un inconveniente: ahora, los ajustes de trabajo del Asesor Experto incluyen no sólo un par de períodos en los parámetros *FastOsMA* y *SlowOsMA*, sino también el número ordinal de su combinación entre todas las posibles (*FastSlowCombo4Optimization*). Lo único que podemos hacer es dar salida a los periodos descodificados en la función *OnInit*, que se ha demostrado anteriormente.

Así, una vez encontrados unos buenos ajustes con la ayuda de la optimización, el usuario, como es habitual, realizará una única ejecución para afinar el comportamiento del sistema de trading. Al principio del registro de pruebas debe aparecer una inscripción del siguiente tenor:

```
MA periods are restored from shadow: FastOsMA=27 SlowOsMA=175
```

A continuación, puede introducir los periodos especificados en los parámetros del mismo nombre y restablecer todos los parámetros de sombra.

6.5.9 Grupo de eventos OnTester para el control de la optimización

Existen tres eventos especiales en MQL5 para gestionar el proceso de optimización y transferir los resultados arbitrarios aplicados (además de los indicadores de trading) de los agentes al terminal: *OnTesterInit*, *OnTesterDeinit* y *OnTesterPass*. Habiendo descrito los manejadores para ellos en el código, el programador podrá realizar las acciones que necesite antes de iniciar la optimización, después de que la optimización se haya completado, y al final de cada una de las pasadas individuales de optimización (si se han recibido datos de la aplicación desde el agente; abajo encontrará más información al respecto).

Todos los manejadores son opcionales. Como hemos visto, la optimización funciona sin ellos. Debe entenderse también que los tres eventos funcionan sólo durante la optimización, pero no en una sola prueba.

El Asesor Experto con estos manejadores se carga automáticamente en un gráfico independiente del terminal con el símbolo y periodo especificados en el probador. Este Asesor Experto de instancia no negocia, sino que sólo realiza acciones de servicio. Todos los demás manejadores de eventos, como *OnInit*, *OnDeinit* y *OnTick*, no funcionan en él.

Para averiguar si un Asesor Experto se ejecuta en el modo de trading normal en el agente o en el modo de servicio en el terminal, llame a la función *MQLInfoInteger(MQL_FRAME_MODE)* en su código y obtenga *true* o *false*. Este modo de servicio se conoce también como modo «frames» y se aplica a los paquetes de datos que se pueden enviar al terminal desde las instancias del Asesor Experto en los agentes. Veremos un poco más adelante cómo se hace.

Durante la optimización, sólo una instancia del Asesor Experto trabaja en el terminal y, si es necesario, recibe los frames entrantes. No olvide que tal instancia se lanza sólo si el código del Asesor Experto contiene uno de los tres manejadores de eventos descritos.

El evento *OnTesterInit* se genera cuando se lanza la optimización en el probador de estrategias antes de la primera pasada. El manejador tiene dos versiones: con tipo de retorno *int* y *void*.

```
int OnTesterInit(void)
void OnTesterInit(void)
```

En la versión de retorno *int*, un valor cero (INIT_SUCCEEDED) significa una inicialización exitosa del Asesor Experto lanzado en el gráfico en el terminal, lo que permite iniciar la optimización. Cualquier otro valor significa un código de error, y la optimización no se iniciará.

La segunda versión de la función siempre implica una preparación exitosa del Asesor Experto para la optimización.

Se proporciona un tiempo limitado para la ejecución de *OnTesterInit*, después del cual el Asesor Experto se verá obligado a terminar, y la propia optimización se cancelará. En este caso, se mostrará el mensaje correspondiente en el registro del probador.

En la sección anterior vimos un ejemplo de cómo se utilizaba el manejador *OnTesterInit* para modificar los parámetros de optimización utilizando las funciones *ParameterGetRange/ParameterSetRange*.

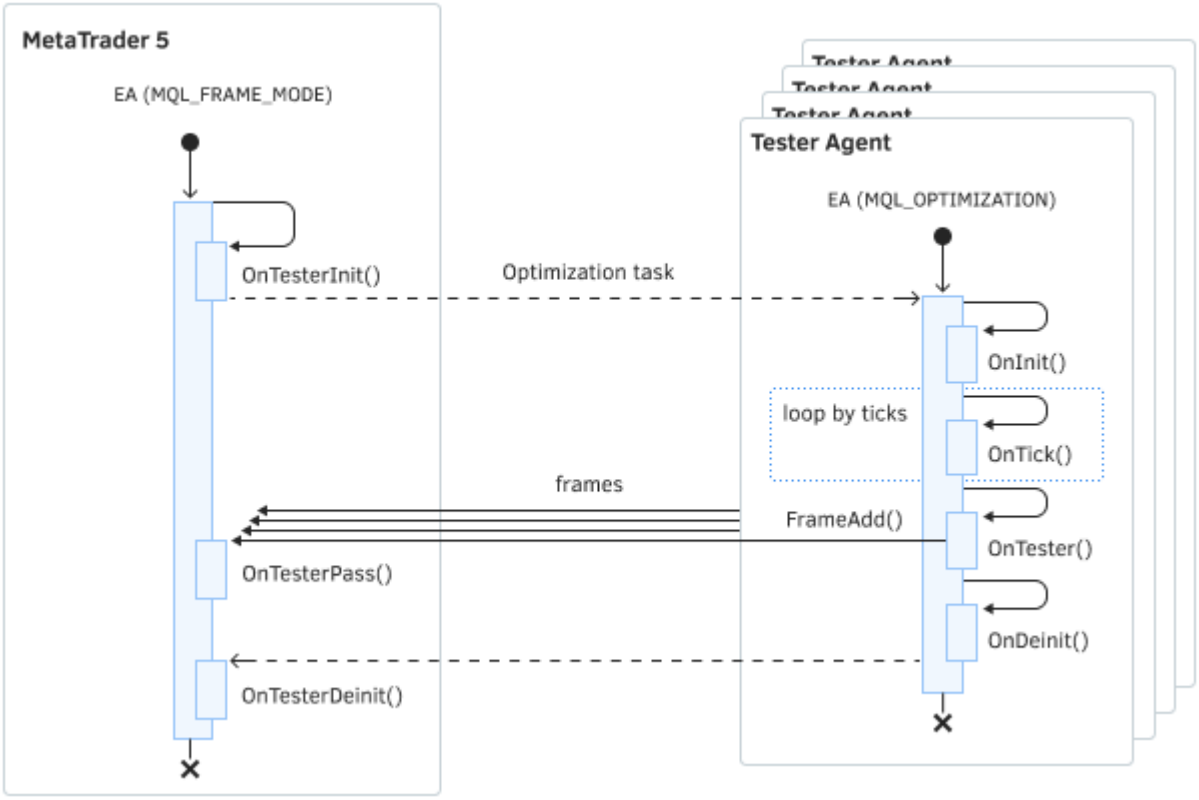
void OnTesterDeinit(void)

La función *OnTesterDeinit* se llama al finalizar la optimización del Asesor Experto.

La función está pensada para el tratamiento final de los resultados de optimización aplicados. Por ejemplo, si se abrió un archivo en *OnTesterInit* para escribir el contenido de los frames, habrá que cerrarlo en *OnTesterDeinit*.

void OnTesterPass(void)

El evento *OnTesterPass* se genera automáticamente cuando llega un frame de datos durante la optimización. La función permite procesar los datos de aplicación recibidos de las instancias del Asesor Experto que se ejecutan en los agentes durante la optimización. Un frame del agente de pruebas debe enviarse desde el manejador *OnTester* utilizando la función *FrameAdd*.



En el diagrama se muestra la secuencia de eventos al optimizar Asesores Expertos

Los agentes envían automáticamente al terminal un conjunto estándar de estadísticas financieras sobre cada pasada de prueba. El Asesor Experto no está obligado a enviar nada utilizando *FrameAdd* si no lo necesita. Si no se utilizan frames, no se llamará al manejador *OnTesterPass*.

Utilizando *OnTesterPass* puede procesar dinámicamente los resultados de la optimización «sobre la marcha»; por ejemplo, mostrarlos en un gráfico en el terminal o añadirlos a un archivo para su posterior procesamiento por lotes.

Para demostrar las capacidades de los manejadores de eventos de *OnTester*, primero necesitamos conocer las funciones para trabajar con frames. Se presentan en las secciones siguientes.

6.5.10 Enviar frames de datos de los agentes al terminal

MQL5 proporciona un grupo de funciones para organizar la transferencia y el procesamiento de sus propios resultados de optimización (aplicados), además de estadísticas e indicadores financieros estándar. Uno de ellos, *FrameAdd*, está diseñado para enviar datos de los agentes de simulación. Otras funciones están destinadas a recibir datos en el terminal.

El formato de intercambio de datos se basa en frames. Ésta es una estructura interna especial que un Asesor Experto puede rellenar en el comprobador basándose en un array de tipo simple (que no contiene cadenas, objetos de clase o arrays dinámicos) o utilizando un archivo con un nombre especificado (el archivo debe crearse primero en el sandbox del agente). Llamando a la función *FrameAdd* varias veces, el Asesor Experto puede enviar una serie de frames al terminal. No hay límites en el número de frames.

Existen dos versiones de la función *FrameAdd*.

bool FrameAdd(const string name, ulong id, double value, const string filename)
bool FrameAdd(const string name, ulong id, double value, const void &data[])

La función añade un frame de datos al búfer para enviarlo al terminal. Los parámetros *name* y *id* son etiquetas públicas que se pueden utilizar para filtrar frames en la función *FrameFilter*. El parámetro *value* permite pasar un valor numérico arbitrario que se puede utilizar cuando un valor es suficiente. Los datos más voluminosos se indican en el array *data* (puede ser un array de estructuras simples) o en un archivo denominado *filename*.

Si no hay datos voluminosos que transferir (por ejemplo, sólo necesita transferir el estado del proceso), utilice la primera forma de la función y especifique NULL en lugar de una cadena con el nombre del archivo o la segunda forma con un array ficticio de tamaño cero.

La función devuelve *true* en caso de éxito.

La función sólo puede invocarse en el manejador *OnTester*.

La función no tiene ningún efecto cuando se llama durante una prueba simple, es decir, fuera de la optimización.

> Sólo se pueden enviar datos de los agentes al terminal. No hay mecanismos en MQL5 para enviar datos en la dirección opuesta durante la optimización. Todos los datos que el Asesor Experto quiera enviar a los agentes deben estar preparados y disponibles (en forma de parámetros de entrada o archivos conectados por directivas) antes de iniciar la optimización.

Veremos un ejemplo de uso de *FrameAdd* después de familiarizarnos con las funciones del anfitrión en la siguiente sección.

6.5.11 Obtener frames de datos en terminal

Los frames enviados desde los agentes de simulación por la función *FrameAdd* se entregan en el terminal y se escriben en el orden de recepción en un archivo mqd con el nombre del Asesor Experto en la carpeta *terminal_directory/MQL5/Files/Tester*. La llegada de uno o varios frames a la vez genera el evento *OnTesterPass*.

La API de MQL5 proporciona 4 funciones para analizar y leer frames: *FrameFirst*, *FrameFilter*, *FrameNext* y *FrameInputs*. Todas las funciones devuelven un valor booleano con una indicación de éxito (*true*) o error (*false*).

Para acceder a los frames existentes, el núcleo mantiene la metáfora de un puntero interno al frame actual. El puntero avanza automáticamente cuando se lee el siguiente frame mediante la función *FrameNext*, pero puede volver al principio de todos los frames con *FrameFirst* o *FrameFilter*. Así, un programa MQL puede organizar la iteración de frames en un bucle hasta que haya revisado todos los frames. Este proceso puede repetirse si es necesario, por ejemplo, aplicando distintos filtros en *OnTesterDeinit*.

bool FrameFirst()

La función *FrameFirst* establece el puntero de lectura de frame interno al principio y restablece el filtro (si se estableció previamente utilizando la función *FrameFilter*).

En teoría, para una única recepción y procesamiento de todos los frames, no es necesario llamar a *FrameFirst*, puesto que el puntero ya está al principio cuando comienza la optimización.

bool FrameFilter(const string name, ulong id)

Establece el filtro de lectura de frames y fija el puntero interno del frame al principio. El filtro afectará a los frames que se incluyan en las siguientes llamadas a *FrameNext*.

Si se pasa una cadena vacía como primer parámetro, el filtro sólo funcionará mediante un parámetro numérico, es decir, todos los frames con el *id* especificado. Si el valor del segundo parámetro es igual a ULONG_MAX, sólo funciona el filtro de texto.

Llamar a *FrameFilter("", ULONG_MAX)* equivale a llamar a *FrameFirst()*, lo que equivale a la ausencia de filtro.

> Si llama a *FrameFirst* o *FrameFilter* en *OnTesterPass*, asegúrese de que es realmente lo que necesita: probablemente el código contenga un error lógico, ya que es posible que se produzca un bucle, que se lea el mismo frame o que aumente exponencialmente la carga computacional.

bool FrameNext(ulong &pass, string &name, ulong &id, double &value)
bool FrameNext(ulong &pass, string &name, ulong &id, double &value, void &data[])

La función *FrameNext* lee un frame y mueve el puntero al siguiente. El parámetro *pass* tendrá registrado el número de pase de optimización. Los parámetros *name*, *id* y *value* recibirán los valores pasados en los parámetros correspondientes de la función *FrameAdd*.

Es importante tener en cuenta que la función puede devolver *false* mientras opera normalmente cuando no hay más frames para leer. En este caso, la variable integrada *_LastError* contiene el valor 4000 (no tiene notación integrada).

Independientemente de la forma de la función *FrameAdd* que se haya utilizado para enviar los datos, el contenido del archivo o del array se colocará en el array *data* receptor. El tipo del array receptor debe coincidir con el tipo del array enviado, y hay ciertos matices en el caso del envío de un archivo.

Un archivo binario (FILE_BIN) debe aceptarse preferiblemente en un array de bytes *uchar* para garantizar la compatibilidad con cualquier tamaño (porque otros tipos más grandes pueden no ser múltiplos del tamaño del archivo). Si el tamaño del archivo (de hecho, el tamaño del bloque de datos en el frame recibido) no es múltiplo del tamaño del tipo de array receptor, la función *FrameNext* no leerá los datos y devolverá un error INVALID_ARRAY (4006).

Un archivo de texto *Unicode* (FILE_TXT o FILE_CSV sin modificador FILE_ANSI) debe aceptarse en un array de tipo *ushort* y convertirse después en una cadena llamando a *ShortArrayToString*. Un archivo de texto ANSI debe recibirse en un array *uchar* y convertirse utilizando *CharArrayToString*.

bool FrameInputs(ulong pass, string &parameters[], uint &count)

La función *FrameInputs* le permite obtener descripciones y valores de los parámetros del Asesor Experto *input* sobre los que se forma el pase con el número de pase especificado. El array de cadenas *parameters* se llenará con líneas como «ParameterNameN=ValueParameterN». El parámetro *count* se rellenará con el número de elementos del array *parameters*.

Las llamadas a estas cuatro funciones sólo están permitidas dentro de los manejadores *OnTesterPass* y *OnTesterDeinit*.

Los frames pueden llegar a la terminal en lotes, en cuyo caso se tarda tiempo en entregarlos. Por lo tanto, no es necesario que todos ellos tengan tiempo de generar el evento *OnTesterPass* y se procesarán hasta el final de la optimización. En este sentido, para garantizar la recepción de todos los frames tardíos, es necesario colocar un bloque de código con su procesamiento (utilizando la función *FrameNext*) en *OnTesterDeinit*.

Veamos un ejemplo sencillo de *FrameTransfer.mq5*.

El Asesor Experto tiene cuatro parámetros de prueba. Todos ellos, excepto la última cadena, pueden incluirse en la optimización.

```
input bool Parameter0;
input long Parameter1;
input double Parameter2;
input string Parameter3;
```

No obstante, para simplificar el ejemplo, el número de pasos para los parámetros *Parameter1* y *Parameter2* se limita a 10 (para cada uno). Así, si no utiliza *Parameter0*, el número máximo de pasadas es 121. *Parameter3* es un ejemplo de parámetro que no puede incluirse en la optimización.

El Asesor Experto no negocia, sino que genera datos aleatorios que imitan los datos arbitrarios de la aplicación. No utilice la aleatoriedad de este modo en sus proyectos de trabajo: sólo es adecuada para demostraciones.

```
ulong startup; // track the time of one run (just like demo data)

int OnInit()
{
   startup = GetMicrosecondCount();
   MathSrand((int)startup);
   return INIT_SUCCEEDED;
}
```

Los datos se envían en dos tipos de frames: desde un archivo y desde un array. Cada tipo tiene su propio identificador.

```
#define MY_FILE_ID 100
#define MY_TIME_ID 101

double OnTester()
{
 // send file in one frame
   const static string filename = "binfile";
   int h = FileOpen(filename, FILE_WRITE | FILE_BIN | FILE_ANSI);
   FileWriteString(h, StringFormat("Random: %d", MathRand()));
   FileClose(h);
   FrameAdd(filename, MY_FILE_ID, MathRand(), filename);

 // send array in another frame
   ulong dummy[1];
   dummy[0] = GetMicrosecondCount() - startup;
   FrameAdd("timing", MY_TIME_ID, 0, dummy);

   return (Parameter2 + 1) * (Parameter1 + 2);
}
```

El archivo se escribe como binario, con cadenas simples. El resultado (criterio) de *OnTester* es una expresión aritmética simple en la que intervienen *Parameter1* y *Parameter2*.

En el lado receptor, en la instancia del Asesor Experto que se ejecuta en el modo de servicio en el gráfico terminal, recogemos los datos de todos los frames con archivos y los ponemos en un archivo CSV común. El archivo se abre en el manejador *OnTesterInit*.

```
int handle; // file for collecting applied results
void OnTesterInit()
{
   handle = FileOpen("output.csv", FILE_WRITE | FILE_CSV | FILE_ANSI, ",");
}
```

Como ya se ha mencionado, es posible que no dé tiempo a que todos los frames entren en el manejador *OnTesterPass*, por lo que es necesario comprobarlos adicionalmente en *OnTesterDeinit*. Por lo tanto, hemos implementado una función de ayuda *ProcessFileFrames*, que llamaremos desde *OnTesterPass*, y desde *OnTesterDeinit*.

Dentro de *ProcessFileFrames* guardamos nuestro contador interno de frames procesados, *framecount*. Utilizándolo como ejemplo, nos aseguraremos de que el orden de llegada de los frames y la numeración de las pasadas de prueba no suelan coincidir.

```
void ProcessFileFrames()
{
   static ulong framecount = 0;
   ...
```

Para recibir frames en la función se describen las variables necesarias según el prototipo *FrameNext*. El array de datos receptor se describe aquí como *uchar*. Si escribiéramos algunas estructuras en nuestro archivo binario, podríamos llevarlas directamente a un array de estructuras del mismo tipo.

```
   ulong   pass;
   string  name;
   long    id;
   double  value;
   uchar   data[];
   ...
```

A continuación se describen las variables para obtener las entradas del Asesor Experto para el pase actual al que pertenece el frame.

```
   string  params[];
   uint    count;
   ...
```

A continuación, leemos los frames en un bucle con *FrameNext*. Recuerde que varios frames pueden entrar en el manejador a la vez, por lo que se necesita un bucle. Para cada frame enviamos al registro del terminal el número de pase, el nombre del frame y el valor *double* resultante. Omitimos los frames con un ID distinto de MY_FILE_ID y las procesaremos más tarde.

```
   ResetLastError();

   while(FrameNext(pass, name, id, value, data))
   {
      PrintFormat("Pass: %lld Frame: %s Value:%f", pass, name, value);
      if(id != MY_FILE_ID) continue;
      ...
   }

   if(_LastError != 4000 && _LastError != 0)
   {
      Print("Error: ", E2S(_LastError));
   }
}
```

Para los frames con MY_FILE_ID, hacemos lo siguiente: consultamos las variables de entrada, averiguamos cuáles están incluidas en la optimización y guardamos sus valores en un archivo CSV común junto con la información del frame. Cuando el recuento de frames es 0, formamos el encabezado del archivo CSV en la variable *header*. En todos los frames, el registro actual (nuevo) del archivo CSV se forma en la variable *record*.

```
void ProcessFileFrames()
{
   ...
      if(FrameInputs(pass, params, count))
      {
         string header, record;
         if(framecount == 0) // prepare CSV header
         {
            header = "Counter,Pass ID,";
         }
         record = (string)framecount + "," + (string)pass + ",";
         // collect optimized parameters and their values
         for(uint i = 0; i < count; i++)
         {
            string name2value[];
            int n = StringSplit(params[i], '=', name2value);
            if(n == 2)
            {
               long pvalue, pstart, pstep, pstop;
               bool enabled = false;
               if(ParameterGetRange(name2value[0],
                  enabled, pvalue, pstart, pstep, pstop))
               {
                  if(enabled)
                  {
                     if(framecount == 0) // prepare CSV header
                     {
                        header += name2value[0] + ",";
                     }
                     record += name2value[1] + ","; // data field
                  }
               }
            }
         }
         if(framecount == 0) // prepare CSV header
         {
            FileWriteString(handle, header + "Value,File Content\n");
         }
         // write data to CSV
         FileWriteString(handle, record + DoubleToString(value) + ","
            + CharArrayToString(data) + "\n");
      }
      framecount++;
   ...
}
```

La llamada a *ParameterGetRange* también podría hacerse de forma más eficiente, sólo que con un valor cero de *framecount*: pruebe a hacerlo.

En el manejador *OnTesterPass*, simplemente llamamos a *ProcessFileFrames*.

```
void OnTesterPass()
{
   ProcessFileFrames(); // standard processing of frames on the go
}
```

Además, llamamos a la misma función desde *OnTesterDeinit* y cerramos el archivo CSV.

```
void OnTesterDeinit()
{
   ProcessFileFrames(); // pick up late frames
   FileClose(handle);   // close the CSV file
   ..
}
```

En *OnTesterDeinit*, procesamos frames con MY_TIME_ID. La duración de las pasadas de prueba se entrega en estos frames, y aquí se calcula la duración media de una pasada. En teoría, tiene sentido hacer esto sólo para el análisis en su programa, ya que para el usuario la duración de los pases la muestra ya el probador en el registro.

```
void OnTesterDeinit()
{
   ...
   ulong   pass;
   string  name;
   long    id;
   double  value;
   ulong   data[]; // same array type as sent

   FrameFilter("timing", MY_TIME_ID); // rewind to the first frame

   ulong count = 0;
   ulong total = 0;
   // cycle through 'timing' frames only
   while(FrameNext(pass, name, id, value, data))
   {
      if(ArraySize(data) == 1)
      {
         total += data[0];
      }
      else
      {
         total += (ulong)value;
      }
      ++count;
   }
   if(count > 0)
   {
      PrintFormat("Average timing: %lld", total / count);
   }
}
```

El Asesor Experto está listo. Vamos a activar la optimización completa para ello (porque el número total de opciones está limitado artificialmente y es demasiado pequeño para el algoritmo genético). Sólo

podemos elegir precios abiertos ya que el Asesor Experto no negocia. Por ello, debe elegir un criterio personalizado (todos los demás criterios darán 0). Por ejemplo, fijemos el rango *Parameter1* de 1 a 10 en pasos simples, y *Parameter2* se fija de -0.5 a +0.5 en pasos de 0.1.

Vamos a ejecutar la optimización. En el registro de expertos del terminal, veremos entradas sobre los frames recibidos del formulario:

```
Pass: 0 Frame: binfile Value:5105.000000
Pass: 0 Frame: timing Value:0.000000
Pass: 1 Frame: binfile Value:28170.000000
Pass: 1 Frame: timing Value:0.000000
Pass: 2 Frame: binfile Value:17422.000000
Pass: 2 Frame: timing Value:0.000000
...
Average timing: 1811
```

En el archivo output.csv aparecerán las líneas correspondientes con los números de pase, los valores de los parámetros y el contenido de los frames:

```
Counter,Pass ID,Parameter1,Parameter2,Value,File Content
0,0,0,-0.5,5105.00000000,Random: 87
1,1,1,-0.5,28170.00000000,Random: 64
2,2,2,-0.5,17422.00000000,Random: 61
...
37,35,2,-0.2,6151.00000000,Random: 68
38,62,7,0.0,17422.00000000,Random: 61
39,36,3,-0.2,16899.00000000,Random: 71
40,63,8,0.0,17422.00000000,Random: 61
...
117,116,6,0.5,27648.00000000,Random: 74
118,117,7,0.5,16899.00000000,Random: 71
119,118,8,0.5,17422.00000000,Random: 61
120,119,9,0.5,28170.00000000,Random: 64
```

Obviamente, nuestra numeración interna (columna *Count*) va en orden, y los números de pase *Pass ID* pueden mezclarse (esto depende de muchos factores del procesamiento paralelo de los lotes de trabajos por parte de los agentes). En concreto, el lote de tareas puede ser el primero en terminar el agente al que se asignaron las tareas con números de secuencia superiores: en este caso, la numeración en el archivo comenzará a partir de los pases superiores.

En el registro del comprobador, puede consultar las estadísticas de servicio por frames.

```
242 frames (42.78 Kb total, 181 bytes per frame) received
local 121 tasks (100%), remote 0 tasks (0%), cloud 0 tasks (0%)
121 new records saved to cache file 'tester\cache\FrameTransfer.EURUSD.H1. »
  » 20220101.20220201.20.9E2DE099D4744A064644F6BB39711DE8.opt'
```

Es importante señalar que durante la optimización genética, los números de ejecución se presentan en el informe de optimización como un par *(generation number, copy number)*, mientras que el número de paso obtenido en la función *FrameNext* es *ulong*. De hecho, este es el número de paso en los trabajos por lotes en el contexto de la ejecución de optimización actual. MQL5 no proporciona ningún medio para hacer coincidir la numeración de pases con un informe genético. Para ello, deben calcularse las sumas de comprobación de los parámetros de entrada de cada pasada. Los archivos Opt con una caché de optimización contienen ya un campo de este tipo con un hash MD5.

6.5.12 Directivas del preprocesador para el probador

En la sección sobre propiedades generales de los programas, primero nos familiarizamos con las directivas *#property* en los programas MQL. Luego nos encontramos con directivas pensadas para scripts, servicios e indicadores. También hay un grupo de directivas para el probador. Ya hemos mencionado algunas de ellas. Por ejemplo, *tester_everytick_calculate* afecta al cálculo de los indicadores.

En la siguiente tabla se enumeran todas las directivas del comprobador con sus explicaciones.

Directiva	Descripción
tester_indicator "string"	El nombre del indicador personalizado con el formato «nombre_indicador.ex5»
tester_file «string»	Nombre de archivo con el formato «nombre_archivo.extensión» con los datos iniciales necesarios para la prueba del programa.
tester_library «string»	Nombre de la biblioteca con una extensión como «library.ex5» o «library.dll».
tester_set «string»	Nombre de archivo con el formato «nombre_archivo.set» con los ajustes para los valores y rangos de optimización de los parámetros de entrada del programa.
tester_no_cache	Desactivación de la lectura de la caché existente de optimizaciones anteriores (archivos opt)
tester_everytick_calculate	Desactivación del modo de ahorro de recursos para el cálculo de indicadores en el comprobador

Las dos últimas directivas no tienen argumentos. Todos los demás esperan una cadena entre comillas dobles con el nombre de un archivo de un tipo u otro. También se deduce de esto que las directivas se pueden repetir con distintos archivos, es decir, se pueden incluir varios archivos de configuración o varios indicadores.

La directiva *tester_indicator* es necesaria para conectar al proceso de simulación aquellos indicadores que no se mencionan en el código fuente del programa sometido a prueba en forma de cadenas constantes (literales). Por regla general, el compilador puede determinar automáticamente el indicador necesario a partir de las llamadas a *iCustom* si su nombre se especifica explícitamente en el parámetro correspondiente, por ejemplo, *iCustom(symbol, period, "indicator_name",...)*. Sin embargo, esto no siempre es así.

Digamos que estamos escribiendo un Asesor Experto universal que puede utilizar diferentes indicadores de media móvil, no sólo los estándar integrados. A continuación, podemos crear una variable de entrada para especificar el nombre del indicador por el usuario. Entonces, la llamada a *iCustom* se convertirá en *iCustom(symbol, period, CustomIndicatorName,...)*, donde *CustomIndicatorName* es una variable de entrada del Asesor Experto, cuyo contenido se desconoce en el momento de la compilación. Además, en este caso es probable que el desarrollador aplique *IndicatorCreate* en lugar de *iCustom*, ya que también hay que configurar el número y los tipos de parámetros del indicador. En estos casos, para depurar el programa o demostrarlo con un indicador específico, debemos proporcionar el nombre al probador utilizando la directiva *tester_indicator*.

> La necesidad de informar de los nombres de los indicadores en el código fuente limita considerablemente la capacidad de probar esos programas universales que pueden conectar varios indicadores en línea.

Sin la directiva *tester_indicator*, el terminal no podrá enviar al agente un indicador que no esté explícitamente declarado en el código fuente, por lo que el programa dependiente perderá parte o toda su funcionalidad.

La directiva *tester_file* permite especificar un archivo que se transferirá a los agentes y se colocará en el sandbox antes de realizar la simulación. El contenido y el tipo de archivo no están regulados. Por ejemplo, estos pueden ser los pesos de una red neuronal preentrenada, datos de Profundidad de Mercado recopilados previamente (porque el probador no puede reproducirlos), etc.

> Tenga en cuenta que el archivo de la directiva *tester_file* sólo se lee si existía en tiempo de compilación. Si el código fuente se compiló cuando no existía el archivo correspondiente, su aparición en el futuro ya no servirá de nada: el programa compilado se enviará al agente sin archivo auxiliar. Por lo tanto, por ejemplo, si el archivo especificado en *tester_file* se genera en *OnTesterInit*, debe asegurarse de que el archivo con el nombre dado ya existía en el tiempo de compilación, aun cuando estuviera vacío. Se lo demostraremos a continuación.

Tenga en cuenta que el compilador no genera advertencias si el archivo especificado en la directiva *tester_file* no existe.

Los archivos conectados deben estar en el sandbox del terminal *MQL5/Files/*.

La directiva *tester_library* informa al probador de la necesidad de transferir la librería, que es un programa auxiliar que sólo puede funcionar en el contexto de otro programa MQL, a los agentes. Hablaremos de las bibliotecas en detalle en una sección aparte.

Las bibliotecas necesarias para la simulación se determinan automáticamente mediante las directivas *#import* del código fuente. Sin embargo, si alguna biblioteca es utilizada por un indicador externo, entonces esta propiedad debe estar habilitada. La biblioteca puede tener tanto la extensión *dll* como la extensión *ex5*.

La directiva *tester_set* funciona con archivos *set* con ajustes del programa MQL. El archivo especificado en la directiva estará disponible en el menú contextual del probador y permitirá al usuario aplicar rápidamente los ajustes.

Si se especifica el nombre sin una ruta, el archivo *set* debe estar en el mismo directorio que el Asesor Experto. Esto es algo inesperado, porque el directorio por defecto para los archivos de *set* es *Presets*, y ahí es donde se guardan mediante comandos desde la interfaz del terminal. Para conectar el archivo *set* desde el directorio dado, debe especificarlo explícitamente en la directiva y escribir delante de él una barra, que indica la ruta absoluta dentro de la carpeta MQL5.

```
#property tester_set "/Presets/xyz.set"
```

Cuando no hay barra oblicua inicial, la ruta es relativa al lugar en el que se colocó el texto de origen.

> Inmediatamente después de añadir el archivo y recompilar el programa, debe volver a seleccionar el Asesor Experto en el probador; de lo contrario, el archivo no se recogerá.

Si especifica el nombre y el número de versión del Asesor Experto como «*<expert_name>* _*<number>* *.set*» en el nombre del archivo *set*, éste se añadirá automáticamente al menú de descarga de versiones de parámetros con el número de versión *<number>*. Por ejemplo, el nombre «*MACD Sample_4.set*» significa que se trata de un archivo *set* para el Asesor Experto «*MACD Sample.mq5*» con número de versión 4.

Los interesados pueden estudiar el formato de los archivos *set*: para ello, guarde manualmente los ajustes de simulación/optimización en el probador de estrategias y, a continuación, abra el archivo creado de este modo en un editor de texto.

Veamos ahora la directiva *tester_no_cache*. Al realizar la optimización, el probador de estrategias guarda todos los resultados de las pasadas realizadas en la caché de optimización (archivos con la extensión *opt*), en la que se almacena el resultado de la prueba para cada conjunto de parámetros de entrada. Esto permite, cuando se vuelve a optimizar con los mismos parámetros, obtener resultados listos sin necesidad de volver a calcular y sin perder tiempo.

Sin embargo, para algunas tareas, como los cálculos matemáticos, puede ser necesario realizar cálculos independientemente de la presencia de resultados listos en la caché de optimización. En este caso, en el código fuente, debe incluir la propiedad *tester_no_cache*. Al mismo tiempo, los propios resultados de las pruebas se seguirán almacenando en la memoria caché para que pueda ver todos los datos sobre las pasadas completadas en el probador de estrategias.

La directiva *tester_everytick_calculate* está diseñada para activar el modo de cálculo del indicador en cada tick en el probador.

Por defecto, los indicadores se calculan en el probador sólo cuando se accede a ellos para obtener datos, es decir, cuando se solicitan los valores de los búferes de indicadores. Esto permite acelerar considerablemente la simulación y la optimización si no es necesario obtener los valores de los indicadores en cada tick.

Sin embargo, algunos programas pueden requerir que los indicadores se vuelvan a calcular en cada tick. Es en estos casos cuando resulta útil la propiedad *tester_everytick_calculate*.

Los indicadores del probador de estrategias también se ven obligados a calcularse en cada tick en los siguientes casos:

- al realizar simulación en modo visual;
- si existen las funciones *EventChartCustom*, *OnChartEvent* o *OnTimer* en el indicador.

Esta propiedad sólo se aplica a las operaciones del probador de estrategias. En el terminal, los indicadores se calculan siempre en cada tick entrante.

De hecho, la directiva se ha utilizado en el Asesor Experto *FrameTransfer.mq5*:

```
#property tester_set "FrameTransfer.set"
```

Simplemente no nos centramos en ello. El archivo «*FrameTransfer.set*» se encuentra junto al código fuente. En el mismo Asesor Experto, también necesitamos otra directiva de la tabla anterior:

```
#property tester_no_cache
```

Además, veamos un ejemplo de directiva *tester_file*. Anteriormente, en la sección sobre ajuste automático de los parámetros del Asesor Experto a la hora de optimizar, introdujimos *BandOsMApro.mq5*, en el que fue necesario introducir varios parámetros sombra para pasar rangos de optimización a nuestro código fuente que se ejecuta en los agentes.

La directiva *tester_file* nos permitirá deshacernos de estos parámetros adicionales. Llamemos a la nueva versión *BandOsMAprofile.mq5*.

Ya que estamos familiarizados con la directiva *tester_set*, vamos a añadir a la nueva versión el archivo */Presets/MQL5Book/BandOsMA.set* anteriormente mencionado.

```
#property tester_set "/Presets/MQL5Book/BandOsMA.set"
```

La información sobre el rango y el paso de los periodos de cambio de *FastOsMA* y *SlowOsMA* se guardará en el archivo *BandOsMAprofile.csv* en lugar de tres parámetros de entrada adicionales *FastShadow4Optimization*, *SlowShadow4Optimization*, *StepsShadow4Optimization*.

```
#define SETTINGS_FILE "BandOsMAprofile.csv"
#property tester_file SETTINGS_FILE

const string SettingsFile = SETTINGS_FILE;
```

El ajuste de sombra *FastSlowCombo4Optimization* sigue siendo necesario para una enumeración completa de las combinaciones de periodos permitidas.

```
input group "A U X I L I A R Y"
sinput int FastSlowCombo4Optimization = 0;   // (reserved for optimization)
```

Recordemos que encontramos su rango de optimización en la función *Iterate*. La primera vez la llamamos en *OnTesterInit* con una enumeración completa de combinaciones de periodos rápidos y lentos.

Básicamente, podríamos almacenar todas las combinaciones válidas en el array de estructuras *PairOfPeriods* y escribirla en un archivo binario para su transmisión a los agentes. A continuación, en los agentes, nuestro Asesor Experto podría leer el array listo del archivo y por el índice *FastSlowCombo4Optimization* extraer el par correspondiente de *FastOsMA* y *SlowOsMA* del array.

En lugar de ello, nos centraremos en un cambio mínimo en la lógica de funcionamiento del programa: seguiremos restaurando un par de periodos debido a la segunda llamada *Iterate* en el manejador *OnInit*. Esta vez obtendremos el rango y el paso de enumeración de los valores del periodo no de los parámetros de sombra, sino del archivo CSV.

He aquí los cambios en *OnTesterInit*:

```
int OnTesterInit()
{
   ...
        // check if the file already exists before compiling
        // - if not, the tester will not be able to send it to agents
         const bool preExisted = FileIsExist(SettingsFile);

         // write the settings to a file for transfer to copy programs on agents
         int handle = FileOpen(SettingsFile, FILE_WRITE | FILE_CSV | FILE_ANSI, ",");
         FileWrite(handle, "FastOsMA", start1, step1, stop1);
         FileWrite(handle, "SlowOsMA", start2, step2, stop2);
         FileClose(handle);

         if(!preExisted)
         {
            PrintFormat("Required file %s is missing. It has been just created."
               " Please restart again.",
               SettingsFile);
            ChartClose();
            return INIT_FAILED;
         }
   ...
   return INIT_SUCCEEDED;
}
```

Nótese que hemos hecho el manejador *OnTesterInit* con el tipo de retorno *int*, lo que hace posible cancelar la optimización si el archivo no existe. Sin embargo, en cualquier caso, los datos reales se escriben en el archivo, por lo que si no existía, ahora se crea, y el posterior inicio de la optimización será sin duda un éxito.

Si desea omitir este paso, puede crear previamente un archivo *MQL5/Files/BandOsMAprofile.csv* vacío.

El manejador *OnInit* se ha modificado como sigue:

```
int OnInit()
{
   if(FastOsMA >= SlowOsMA) return INIT_PARAMETERS_INCORRECT;

   PairOfPeriods p = {FastOsMA, SlowOsMA}; // default initial parameters
   int handle = FileOpen(SettingsFile, FILE_READ | FILE_TXT | FILE_ANSI);

   // during optimization, a file with shadow parameters is needed
   if(MQLInfoInteger(MQL_OPTIMIZATION) && handle == INVALID_HANDLE)
   {
      return INIT_PARAMETERS_INCORRECT;
   }

   if(handle != INVALID_HANDLE)
   {
      if(FastSlowCombo4Optimization != -1)
      {
         // if there is a shadow copy, read the period values from it
         const string line1 = FileReadString(handle);
         string settings[];
         if(StringSplit(line1, ',', settings) == 4)
         {
            int FastStart = (int)StringToInteger(settings[1]);
            int FastStep = (int)StringToInteger(settings[2]);
            int FastStop = (int)StringToInteger(settings[3]);
            const string line2 = FileReadString(handle);
            if(StringSplit(line2, ',', settings) == 4)
            {
               int SlowStart = (int)StringToInteger(settings[1]);
               int SlowStep = (int)StringToInteger(settings[2]);
               int SlowStop = (int)StringToInteger(settings[3]);
               p = Iterate(FastStart, FastStop, FastStep,
                  SlowStart, SlowStop, SlowStep, FastSlowCombo4Optimization);
               PrintFormat("MA periods are restored from shadow: FastOsMA=%d SlowOsMA
                  p.fast, p.slow);
            }
         }
      }
      FileClose(handle);
   }
```

Al ejecutar pruebas individuales tras la optimización, veremos valores de periodo descodificados en el registro *FastOsMA* y *SlowOsMA* basados en el valor optimizado *FastSlowCombo4Optimization*. En el futuro, podemos sustituir estos valores en los parámetros del periodo y eliminar el archivo csv. También hemos previsto que el archivo no se tenga en cuenta si *FastSlowCombo4Optimization* se establece en -1.

6.5.13 Gestionar la visibilidad de los indicadores: TesterHideIndicators

Por defecto, el gráfico de simulación visual muestra todos los indicadores creados en el Asesor Experto que se está probando. Además, estos indicadores se muestran en el gráfico, que se abre

automáticamente al final de la simulación. Todo esto se aplica sólo a aquellos indicadores que se crean directamente en su código: los indicadores anidados que se pueden utilizar en el cálculo de los indicadores principales no se aplican aquí.

La visibilidad de los indicadores no siempre es deseable desde el punto de vista del desarrollador, que puede querer ocultar los detalles de implementación de un Asesor Experto. En tales casos, la función *TesterHideIndicators* desactivará la visualización de los indicadores utilizados en el gráfico.

void TesterHideIndicators(bool hide)

El parámetro booleano *hide* indica si se deben ocultar (por valor *true*) o mostrar (por valor *false*) los indicadores. El estado establecido es recordado por el entorno de ejecución del programa MQL hasta que se cambie llamando de nuevo a la función con el valor inverso del parámetro. El estado actual de esta configuración afecta a todos los indicadores de nueva creación.

En otras palabras, la función *TesterHideIndicators* con el valor de bandera requerido *hide* debe llamarse antes de crear los descriptores de los indicadores correspondientes. En concreto, después de llamar a la función con el parámetro *true*, los nuevos indicadores se marcarán con una bandera oculta y no se mostrarán durante la simulación visual ni en el gráfico, que se abre automáticamente cuando finaliza la simulación.

Para desactivar el modo de ocultar los indicadores recién creados, llame a *TesterHideIndicators* con *false*.

La función sólo es aplicable en el probador.

La función tiene algunas especificidades relacionadas con su rendimiento, siempre que se creen plantillas tpl especiales para el probador o Asesor Experto en la carpeta */MQL5/Profiles/Templates*.

Si hay alguna plantilla especial en la carpeta *<expert_name>.tpl*, entonces durante la simulación visual y en el gráfico de simulación, sólo se mostrarán los indicadores de esta plantilla. En este caso, no se mostrará ningún indicador utilizado en el Asesor Experto probado, aun cuando la función se llamase en el código del Asesor Experto *TesterHideIndicators* con *false*.

Si hay una plantilla en la carpeta *tester.tpl*, entonces durante la simulación visual y en el gráfico de simulación se mostrarán los indicadores de la plantilla *tester.tpl*, además de aquellos indicadores del Asesor Experto que no estén prohibidos por la llamada a *TesterHideIndicators*. La función *TesterHideIndicators* no afecta a los indicadores de la plantilla.

Si no hay plantilla *tester.tpl*, pero sí una plantilla *default.tpl*, entonces los indicadores de la misma se procesan según un principio similar.

Demostraremos cómo trabaja la función con el ejemplo de Gran Asesor Experto un poco más adelante.

6.5.14 Emulación de operaciones de depósito y retirada

El probador de MetaTrader 5 le permite emular operaciones de depósito y retirada, lo que le da la posibilidad de experimentar con algunos sistemas de gestión de dinero.

bool TesterDeposit(double money)

La función *TesterDeposit* repone la cuenta en el proceso de simulación del tamaño de la cantidad depositada en el parámetro de dinero. El importe se indica en la moneda del depósito de prueba.

bool TesterWithdrawal(double money)

La función *TesterWithdrawal* realiza extracciones iguales a *money*.

Ambas funciones devuelven *true* como señal de éxito.

Como ejemplo, consideremos un Asesor Experto basado en la estrategia «carry trade». Para ello, debemos seleccionar un símbolo con grandes swaps positivos en una de las direcciones de trading, por ejemplo, comprar AUDUSD. El Asesor Experto abrirá una o más posiciones en la dirección especificada. Las posiciones no rentables se mantendrán para acumular swaps en ellas. Las posiciones rentables se cerrarán al alcanzar una cantidad predeterminada de beneficios por lote. Los swaps ganados se retirarán de la cuenta. El código fuente está disponible en el archivo *CrazyCarryTrade.mq5*.

En los parámetros de entrada, el usuario puede seleccionar la dirección de la operación, el tamaño de la misma (0 por defecto, que significa el lote mínimo) y el beneficio mínimo por lote, al que se cerrará una posición rentable.

```
enum ENUM_ORDER_TYPE_MARKET
{
   MARKET_BUY = ORDER_TYPE_BUY,
   MARKET_SELL = ORDER_TYPE_SELL
};

input ENUM_ORDER_TYPE_MARKET Type;
input double Volume;
input double MinProfitPerLot = 1000;
```

En primer lugar, probemos en el manejador *OnInit* el rendimiento de las funciones *TesterWithdrawal* y *TesterDeposit*. En concreto, el intento de retirar un saldo doble dará lugar al error 10019.

```
int OnInit()
{
   PRTF(TesterWithdrawal(AccountInfoDouble(ACCOUNT_BALANCE) * 2));
   /*
   not enough money for 20 000.00 withdrawal (free margin: 10 000.00)
   TesterWithdrawal(AccountInfoDouble(ACCOUNT_BALANCE)*2)=false / MQL_ERROR::10019(10
   */
   ...
```

Sin embargo, las siguientes retiradas y acreditaciones de 100 unidades de la divisa de la cuenta se realizarán correctamente.

```
   PRTF(TesterWithdrawal(100));
   /*
   deal #2 balance -100.00 [withdrawal] done
   TesterWithdrawal(100)=true / ok
   */
   PRTF(TesterDeposit(100)); // return the money
   /*
   deal #3 balance 100.00 [deposit] done
   TesterDeposit(100)=true / ok
   */
   return INIT_SUCCEEDED;
}
```

En el manejador *OnTick*, vamos a comprobar la disponibilidad de las posiciones utilizando *PositionFilter* y a llenar el array *values* con sus ganancias/pérdidas actuales y swaps acumulados.

```
void OnTick()
{
   const double volume = Volume == 0 ?
      SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN) : Volume;
   ENUM_POSITION_PROPERTY_DOUBLE props[] = {POSITION_PROFIT, POSITION_SWAP};
   double values[][2];
   ulong tickets[];
   PositionFilter pf;
   pf.select(props, tickets, values, true);
   ...
```

Cuando no hay posiciones, abrimos una en una dirección predefinida.

```
   if(ArraySize(tickets) == 0) // no positions
   {
      MqlTradeRequestSync request1;
      (Type == MARKET_BUY ? request1.buy(volume) : request1.sell(volume));
   }
   else
   {
      ... // there are positions - see the next box
   }
```

Cuando hay posiciones, las recorremos en un ciclo y cerramos aquellas para las que hay beneficios suficientes (ajustados a los swaps). Al hacerlo, sumamos también los intercambios de posiciones cerradas y las pérdidas totales. Dado que los swaps crecen en proporción al tiempo, los utilizamos como factor amplificador para cerrar posiciones «antiguas». Así, es posible cerrar con pérdidas.

```
      double loss = 0, swaps = 0;
      for(int i = 0; i < ArraySize(tickets); ++i)
      {
         if(values[i][0] + values[i][1] * values[i][1] >= MinProfitPerLot * volume)
         {
            MqlTradeRequestSync request0;
            if(request0.close(tickets[i]) && request0.completed())
            {
               swaps += values[i][1];
            }
         }
         else
         {
            loss += values[i][0];
         }
      }
      ...
```

Si las pérdidas totales aumentan, periódicamente abrimos posiciones adicionales, pero lo hacemos con menos frecuencia cuando hay más posiciones, para controlar de alguna manera los riesgos.

```
      if(loss / ArraySize(tickets) <= -MinProfitPerLot * volume * sqrt(ArraySize(tick
      {
         MqlTradeRequestSync request1;
         (Type == MARKET_BUY ? request1.buy(volume) : request1.sell(volume));
      }
      ...
```

Por último, eliminamos los swaps de la cuenta.

```
      if(swaps >= 0)
      {
         TesterWithdrawal(swaps);
      }
```

En el manejador *OnDeinit*, mostramos estadísticas sobre las deducciones.

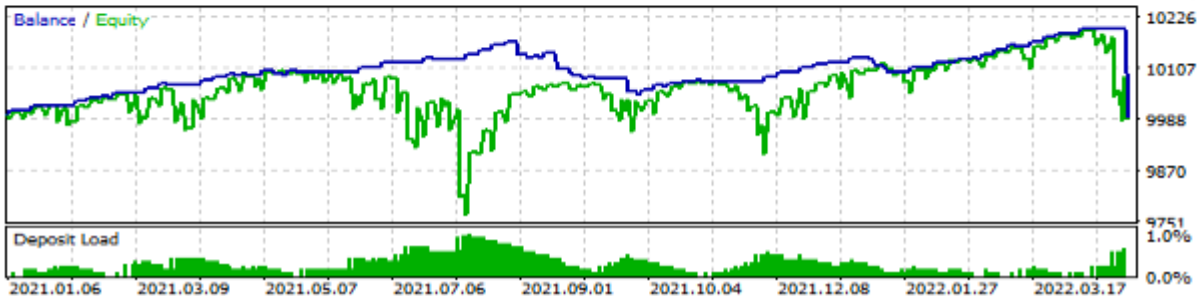
```
void OnDeinit(const int)
{
   PrintFormat("Deposit: %.2f Withdrawals: %.2f",
      TesterStatistics(STAT_INITIAL_DEPOSIT),
      TesterStatistics(STAT_WITHDRAWAL));
}
```

Por ejemplo, al ejecutar el Asesor Experto con la configuración predeterminada para el período comprendido entre 2021 y principios de 2022, obtenemos el siguiente resultado para AUDUSD:

```
final balance 10091.19 USD
Deposit: 10000.00 Withdrawals: 197.42
```

Este es el aspecto del informe y el gráfico:

Bars	8270	Ticks	1956175	Symbols	1
Initial Deposit	10 000.00				
Total Net Profit	91.19	Balance Drawdow...	0.00	Equity Drawdown ...	227.74
Gross Profit	420.17	Balance Drawdow...	202.97 (1.97...	Equity Drawdown ...	270.51 (2.69...
Gross Loss	-328.98	Balance Drawdow...	1.97% (202....	Equity Drawdown ...	2.69% (270....
Withdrawal	197.42				
Profit Factor	1.28	Expected Payoff	1.45	Margin Level	10964.05%
Recovery Factor	0.34	Sharpe Ratio	0.10	Z-Score	-3.30 (99.74...
AHPR	1.0000 (-0.0...	LR Correlation	0.52	OnTester result	0
GHPR	1.0000 (-0.0...	LR Standard Error	41.26		
Total Trades	63	Short Trades (won ...	0 (0.00%)	Long Trades (won ...	63 (76.19%)
Total Deals	126	Profit Trades (% of...	48 (76.19%)	Loss Trades (% of t...	15 (23.81%)



Informe pericial con las retiradas de fondos de la cuenta

Así, operando con un lote mínimo y cargando un depósito de no más del 1 % durante algo más de un año, conseguimos retirar unos 200 USD.

6.5.15 Parada forzada de la prueba: TesterStop

Si es necesario, dependiendo de las condiciones observadas, el desarrollador puede dejar de probar el Asesor Experto antes. Por ejemplo, esto puede hacerse cuando se alcanza un número determinado de transacciones con pérdidas o un nivel de reducción. Para ello, la API de MQL5 proporciona la función *TesterStop*.

void TesterStop()

La función da la orden de terminar el probador, es decir, la parada se producirá sólo después de que el programa devuelva el control al entorno de ejecución.

Llamar a *TesterStop* se considera un final de simulación normal, por lo que se llamará a la función *OnTester* y devolverá todas las estadísticas de trading acumuladas y el valor del criterio de optimización al probador de estrategias.

También existe una forma regular alternativa de interrumpir la simulación: utilizando la función *ExpertRemove* que ya vimos anteriormente. La llamada de *ExpertRemove* también devuelve las estadísticas de trading recopiladas en el momento en que se llama a la función. Sin embargo, existen algunas diferencias.

Como resultado de la llamada a *ExpertRemove*, el Asesor Experto se descarga de la memoria del agente. Por lo tanto, si usted necesita ejecutar un nuevo pase con un nuevo conjunto de parámetros, se necesitará algún tiempo para volver a cargar el programa MQL. Cuando se utiliza *TesterStop*, esto no ocurre, y este método es preferible en términos de rendimiento.

Por otro lado, la llamada a *ExpertRemove* establece la bandera de parada *IsStopped* en el programa MQL, que se puede utilizar de forma estándar en diferentes partes del programa para finalizar («limpiar» los recursos). Pero llamar a *TesterStop* no establece esta bandera, y por lo tanto el desarrollador puede necesitar introducir su propia variable global para indicar la terminación anticipada y manejarla de una manera específica.

> Es importante tener en cuenta que *TesterStop* está diseñado para detener sólo una pasada del probador.
>
> MQL5 no proporciona funciones para la terminación anticipada de la optimización. Por lo tanto, por ejemplo, si su Asesor Experto detecta que la optimización ha sido lanzada en el modelo de generación de ticks equivocado, y esto puede detectarse sólo después de que la optimización haya sido lanzada (*OnTesterInit* no ayuda aquí), entonces las llamadas a *TesterStop* o *ExpertRemove* interrumpirán nuevos pases, pero los pases mismos continuarán siendo iniciados, generando resultados nulos en masa. Lo veremos en la sección de ejemplo de Gran Asesor Experto, que utilizará la protección del lanzamiento a precios abiertos.
>
> Se podría suponer que la llamada a *ExpertRemove* en la instancia del Asesor Experto que se ejecuta en el terminal y que realmente sirve a un gestor de optimización detendría la optimización, pero no es así. Incluso cerrando el gráfico con este Asesor Experto trabajando en el modo frame no detiene la optimización.

Le sugerimos que pruebe usted mismo estas funciones en acción.

6.5.16 Ejemplo de Gran Asesor Experto

Para generalizar y consolidar los conocimientos sobre las capacidades del probador, vamos a considerar paso a paso un amplio ejemplo de un Asesor Experto. En este ejemplo, resumiremos los siguientes aspectos:

- Utilización de varios símbolos, incluida la sincronización de barras
- Utilización de un indicador de un Asesor Experto
- Utilización de eventos
- Cálculo independiente de las principales estadísticas de trading
- Cálculo del criterio de optimización personalizado R2 ajustado a lotes variables
- Envío y tratamiento de frames con datos de aplicación (informes de trading desglosados por símbolos)

Utilizaremos *MultiMartingale.mq5* como base técnica del Asesor Experto, pero lo haremos menos arriesgado pasando a operar con señales de sobrecompra/sobreventa multidivisa y aumentando los lotes sólo como añadido opcional. Anteriormente, en *BandOsMA.mq5* vimos ya cómo operar basándose en señales de trading de indicadores. Esta vez utilizaremos *UseUnityPercentPro.mq5* como indicador de la señal. Sin embargo, primero tenemos que modificarlo. Llamemos a la nueva versión *UnityPercentEvent.mq5*.

UnityPercentEvent.mq5

Recordemos la esencia del indicador *Unity*. Calcula la fuerza relativa de las divisas o tickers incluidos en un conjunto de instrumentos dados (se supone que todos los instrumentos tienen una divisa común a través de la cual es posible la conversión). En cada barra se forman lecturas para todas las divisas: algunas serán más caras, otras más baratas, y los dos elementos extremos se encuentran en un estado límite. Más adelante se pueden considerar dos estrategias esencialmente opuestas para ellas:

- Más desglose (confirmación y continuación de un fuerte movimiento lateral)
- Retroceso (inversión del movimiento hacia el centro debido a sobrecompra y sobreventa)

Para operar cualquiera de estas señales, debemos hacer un símbolo de trabajo de dos divisas (o tickers en general), si hay algo adecuado para esta combinación en la Observación de Mercado. Por ejemplo, si la línea superior del indicador pertenece a EUR y la inferior a USD, corresponden al par EURUSD, y según la estrategia de desglose deberíamos comprarlo pero según la estrategia de rebote, deberíamos venderlo.

> En un caso más general, por ejemplo, cuando se indican CFD o materias primas con una divisa de cotización común en la cesta de instrumentos de trabajo del indicador, no siempre es posible crear un instrumento real. Para estos casos sería necesario complicar más el Asesor Experto introduciendo sintéticos de trading (posiciones compuestas), pero no lo haremos aquí y nos limitaremos al mercado Forex, donde casi todos los tipos cruzados suelen estar disponibles.

Por lo tanto, el Asesor Experto no sólo debe leer todos los búferes de los indicadores, sino también averiguar los nombres de las divisas, que corresponden a los valores máximos y mínimos. Y aquí tenemos un pequeño obstáculo.

MQL5 no permite leer los nombres de los búferes indicadores de terceros y, en general, cualquier propiedad de línea que no sea entera. Existen tres funciones para configurar las propiedades: *PlotIndexSetInteger*, *PlotIndexSetDouble* y *PlotIndexSetString*, pero sólo existe una función para leerlos, *PlotIndexGetInteger*.

En teoría, cuando los programas MQL compilados en un único complejo de trading son creados por el mismo desarrollador, esto no representa un gran problema. En concreto, podríamos separar una parte del código fuente del indicador en un archivo de encabezado e incluirlo no sólo en el indicador, sino también en el Asesor Experto. A continuación, en el Asesor Experto, sería posible repetir el análisis de los parámetros de entrada del indicador y restaurar la lista de divisas, completamente similar a la creada por el indicador. Duplicar los cálculos no es muy bonito, pero funcionaría. No obstante, también se requiere una solución más universal cuando el indicador tiene un desarrollador diferente, y éste no quiere revelar el algoritmo ni planea cambiarlo en el futuro (entonces las versiones compiladas del indicador y del Asesor Experto serán incompatibles). Este tipo de «acoplamiento» de indicadores ajenos con los propios, o de un Asesor Experto encargado a un servicio independiente, es una práctica muy común. Por lo tanto, el desarrollador del indicador debe facilitar al máximo su integración.

Una de las posibles soluciones es que el indicador envíe mensajes con los números y nombres de los búferes después de la inicialización.

Así es como se hace en el manejador *OnInit* del indicador *UnityPercentEvent.mq5* (el código de abajo se muestra de forma abreviada ya que casi nada ha cambiado).

```
int OnInit()
{
   // find the common currency for all pairs
   const string common = InitSymbols();
   ...
   // set up the displayed lines in the currency cycle
   int replaceIndex = -1;
   for(int i = 0; i <= SymbolCount; i++)
   {
      string name;
      // change the order so that the base (common) currency goes under index 0,
      // the rest depends on the order in which the pairs are entered by the user
      if(i == 0)
      {
         name = common;
         if(name != workCurrencies.getKey(i))
         {
            replaceIndex = i;
         }
      }
      else
      {
         if(common == workCurrencies.getKey(i) && replaceIndex > -1)
         {
            name = workCurrencies.getKey(replaceIndex);
         }
         else
         {
            name = workCurrencies.getKey(i);
         }
      }

      // set up rendering of buffers
      PlotIndexSetString(i, PLOT_LABEL, name);
      ...
      // send indexes and buffer names to programs where they are needed
      EventChartCustom(0, (ushort)BarLimit, i, SymbolCount + 1, name);
   }
   ...
}
```

En comparación con la versión original, aquí sólo se ha añadido una línea. Contiene la llamada *EventChartCustom*. La variable de entrada *BarLimit* se utiliza como identificador de la copia del indicador (de las que potencialmente puede haber varias). Como el indicador se llamará desde el Asesor Experto y no se mostrará al usuario, basta con indicar un número positivo pequeño, al menos 1, pero tendremos, por ejemplo, 10.

Ahora el indicador está listo y sus señales se pueden utilizar en Asesores Expertos de terceros. Empecemos a desarrollar el Asesor Experto *UnityMartingale.mq5*. Para simplificar la presentación, la dividiremos en 4 etapas, añadiendo gradualmente nuevos bloques. Tendremos tres versiones preliminares y una versión final.

UnityMartingaleDraft1.mq5

En la primera etapa, para la versión *UnityMartingaleDraft1.mq5*, utilicemos *MultiMartingale.mq5* como base y modifiquémosla.

Cambiaremos el nombre de la antigua variable de entrada *StartType*, que determinaba la dirección de la primera transacción de la serie, por *SignalType*. Se utilizará para elegir entre las estrategias consideradas BREAKOUT y PULLBACK.

```
enum SIGNAL_TYPE
{
   BREAKOUT,
   PULLBACK
};
...
input SIGNAL_TYPE StartType = 0; // SignalType
```

Para configurar el indicador, necesitamos un grupo separado de variables de entrada.

```
input group "U N I T Y   S E T T I N G S"
input string UnitySymbols = "EURUSD,GBPUSD,USDCHF,USDJPY,AUDUSD,USDCAD,NZDUSD";
input int UnityBarLimit = 10;
input ENUM_APPLIED_PRICE UnityPriceType = PRICE_CLOSE;
input ENUM_MA_METHOD UnityPriceMethod = MODE_EMA;
input int UnityPricePeriod = 1;
```

Tenga en cuenta que el parámetro *UnitySymbols* contiene una lista de instrumentos de clúster para construir un indicador, y normalmente difiere de la lista de instrumentos de trabajo con los que queremos operar. Los instrumentos negociados se siguen configurando en el parámetro *WorkSymbols*.

Por ejemplo, por defecto, pasamos un conjunto de los principales pares de divisas *Forex* al indicador, y por lo tanto podemos indicar como trading no sólo los pares principales, sino también cualquier cruce. Suele tener sentido limitar este conjunto a los instrumentos con las mejores condiciones de trading (en concreto, diferenciales pequeños o moderados). Además, es deseable evitar distorsiones, es decir, mantener una cantidad igual de cada divisa en todos los pares, neutralizando así estadísticamente los riesgos potenciales de elegir una dirección poco acertada para una de las divisas.

A continuación, envolvemos el control del indicador en la clase *UnityController*. Además del indicador *handle*, los campos de clase almacenan los siguientes datos:

- El número de *buffers* del indicador, que se recibirán de los mensajes del indicador después de su inicialización.
- El número de *bar* del que se están leyendo los datos (normalmente el incompleto actual es 0, o el último completado es 1)
- El array *data* con los valores leídos de los búferes de los indicadores en la barra especificada
- La última hora de lectura *lastRead*
- Bandera de funcionamiento por ticks o barras *tickwise*

Además, la clase utiliza el objeto *MultiSymbolMonitor* para sincronizar las barras de todos los símbolos implicados.

```
class UnityController
{
   int handle;
   int buffers;
   const int bar;
   double data[];
   datetime lastRead;
   const bool tickwise;
   MultiSymbolMonitor sync;
   ...
```

En el constructor, que acepta todos los parámetros del indicador como argumentos, creamos el indicador y configuramos el objeto *sync*.

```
public:
   UnityController(const string symbolList, const int offset, const int limit,
      const ENUM_APPLIED_PRICE type, const ENUM_MA_METHOD method, const int period):
      bar(offset), tickwise(!offset)
   {
      handle = iCustom(_Symbol, _Period, "MQL5Book/p6/UnityPercentEvent",
         symbolList, limit, type, method, period);
      lastRead = 0;

      string symbols[];
      const int n = StringSplit(symbolList, ',', symbols);
      for(int i = 0; i < n; ++i)
      {
         sync.attach(symbols[i]);
      }
   }

   ~UnityController()
   {
      IndicatorRelease(handle);
   }
   ...
```

El número de búferes se establece mediante el método *attached*. Lo llamaremos al recibir un mensaje del indicador.

```
   void attached(const int b)
   {
      buffers = b;
      ArrayResize(data, buffers);
   }
```

Un método especial *isReady* devuelve *true* cuando las últimas barras de todos los símbolos tienen la misma hora. Sólo en el estado de dicha sincronización obtendremos los valores correctos del indicador. Cabe señalar que aquí se supone el mismo calendario de sesiones de trading para todos los instrumentos. Si no es así, habrá que modificar el análisis de los tiempos.

```
bool isReady()
{
   return sync.check(true) == 0;
}
```

Definimos la hora actual de diferentes maneras dependiendo del modo de funcionamiento del indicador: cuando se recalcula en cada tick (*tickwise* es igual a *true*), utilizamos la hora del servidor, y cuando se recalcula una vez por barra, utilizamos la hora de apertura de la última barra.

```
datetime lastTime() const
{
   return tickwise ? TimeTradeServer() : iTime(_Symbol, _Period, 0);
}
```

La presencia de este método nos permitirá excluir la lectura del indicador si la hora actual no ha cambiado y, en consecuencia, los últimos datos leídos almacenados en el búfer *data* siguen siendo relevantes. Y así es como se organiza la lectura de los búferes indicadores en el método *read*. Sólo necesitamos un valor de cada búfer para la barra con el índice *bar*.

```
bool read()
{
   if(!buffers) return false;
   for(int i = 0; i < buffers; ++i)
   {
      double temp[1];
      if(CopyBuffer(handle, i, bar, 1, temp) == 1)
      {
         data[i] = temp[0];
      }
      else
      {
         return false;
      }
   }
   lastRead = lastTime();
   return true;
}
```

Al final, sólo guardamos el tiempo de lectura en la variable *lastRead*. Si está vacía o no es igual a la nueva hora actual, el acceso a los datos del controlador en los siguientes métodos hará que se lean los búferes indicadores utilizando *read*.

Los principales métodos externos del controlador son *getOuterIndices* para obtener los índices de los valores máximo y mínimo y el operador '[]' para leer los valores.

```
   bool isNewTime() const
   {
      return lastRead != lastTime();
   }

   bool getOuterIndices(int &min, int &max)
   {
      if(isNewTime())
      {
         if(!read()) return false;
      }
      max = ArrayMaximum(data);
      min = ArrayMinimum(data);
      return true;
   }

   double operator[](const int buffer)
   {
      if(isNewTime())
      {
         if(!read())
         {
            return EMPTY_VALUE;
         }
      }
      return data[buffer];
   }
};
```

Anteriormente, el Asesor Experto *BandOsMA.mq5* introdujo el concepto de interfaz *TradingSignal*.

```
interface TradingSignal
{
   virtual int signal(void);
};
```

Basándonos en ella, describiremos la implementación de la señal utilizando el indicador *UnityPercentEvent*. El objeto controlador *UnityController* se pasa al constructor. También indica los índices de divisas (búferes), de cuyas señales queremos hacer un seguimiento. Podremos crear un conjunto arbitrario de señales diferentes para los símbolos de trabajo seleccionados.

```
class UnitySignal: public TradingSignal
{
   UnityController *controller;
   const int currency1;
   const int currency2;

public:
   UnitySignal(UnityController *parent, const int c1, const int c2):
      controller(parent), currency1(c1), currency2(c2) { }

   virtual int signal(void) override
   {
      if(!controller.isReady()) return 0; // waiting for bars synchronization
      if(!controller.isNewTime()) return 0; // waitng for time to change

      int min, max;
      if(!controller.getOuterIndices(min, max)) return 0;

      // overbought
      if(currency1 == max && currency2 == min) return +1;
      // oversold
      if(currency2 == max && currency1 == min) return -1;
      return 0;
   }
};
```

El método *signal* devuelve 0 en situación de incertidumbre y +1 o -1 en estados de sobrecompra y sobreventa de dos divisas concretas.

Para formalizar las estrategias de trading utilizamos la interfaz *TradingStrategy*.

```
interface TradingStrategy
{
   virtual bool trade(void);
};
```

En este caso, se crea sobre su base la clase *UnityMartingale*, que coincide en gran medida con *SimpleMartingale* de *MultiMartingale.mq5*. Sólo mostraremos las diferencias.

```
class UnityMartingale: public TradingStrategy
{
protected:
   ...
   AutoPtr<TradingSignal> command;

public:
   UnityMartingale(const Settings &state, TradingSignal *signal)
   {
      ...
      command = signal;
   }
   virtual bool trade() override
   {
      ...
      int s = command[].signal(); // get controller signal
      if(s != 0)
      {
         if(settings.startType == PULLBACK) s *= -1; // reverse logic for bounce
      }
      ulong ticket = 0;
      if(position[] == NULL) // clean start - there were (and is) no positions
      {
         if(s == +1)
         {
            ticket = openBuy(settings.lots);
         }
         else if(s == -1)
         {
            ticket = openSell(settings.lots);
         }
      }
      else
      {
         if(position[].refresh()) // position exists
         {
            if((position[].get(POSITION_TYPE) == POSITION_TYPE_BUY && s == -1)
            || (position[].get(POSITION_TYPE) == POSITION_TYPE_SELL && s == +1))
            {
               // signal in the other direction - we need to close
               PrintFormat("Opposite signal: %d for position %d %lld",
                  s, position[].get(POSITION_TYPE), position[].get(POSITION_TICKET));
               if(close(position[].get(POSITION_TICKET)))
               {
                  // position = NULL; - save the position in the cache
               }
               else
               {
                  position[].refresh(); // control possible closing errors
               }
            }
```

```
               else
               {
                  // the signal is the same or absent - "trailing"
                  position[].update();
                  if(trailing[]) trailing[].trail();
               }
            }
            else // no position - open a new one
            {
               if(s == 0) // no signals
               {
                  // here is the full logic of the old Expert Advisor:
                  // - reversal for martingale loss
                  // - continuation by the initial lot in a profitable direction
                  ...
               }
               else // there is a signal
               {
                  double lots;
                  if(position[].get(POSITION_PROFIT) >= 0.0)
                  {
                     lots = settings.lots; // initial lot after profit
                  }
                  else // increase the lot after the loss
                  {
                     lots = MathFloor((position[].get(POSITION_VOLUME) * settings.factor

                     if(lotsLimit < lots)
                     {
                        lots = settings.lots;
                     }
                  }

                  ticket = (s == +1) ? openBuy(lots) : openSell(lots);
               }
            }
         }
      }
      ...
   }
```

La parte de trading está lista; queda por considerar la inicialización. En el nivel global se describen un puntero automático al objeto *UnityController* y el array con los nombres de las divisas. El conjunto de sistemas de trading es completamente similar a los desarrollos anteriores.

```
AutoPtr<TradingStrategyPool> pool;
AutoPtr<UnityController> controller;

int currenciesCount;
string currencies[];
```

En el manejador *OnInit*, creamos el objeto *UnityController* y esperamos a que el indicador envíe la distribución de divisas por índices del búfer.

```
int OnInit()
{
   currenciesCount = 0;
   ArrayResize(currencies, 0);

   if(!StartUp(true)) return INIT_PARAMETERS_INCORRECT;

   const bool barwise = UnityPriceType == PRICE_CLOSE && UnityPricePeriod == 1;
   controller = new UnityController(UnitySymbols, barwise,
      UnityBarLimit, UnityPriceType, UnityPriceMethod, UnityPricePeriod);
   // waiting for messages from the indicator on currencies in buffers
   return INIT_SUCCEEDED;
}
```

Si en los parámetros de entrada del indicador se selecciona el tipo de precio PRICE_CLOSE y un período único, el cálculo en el controlador se realizará una vez por barra. En todos los demás casos, las señales se actualizarán por ticks, pero no más a menudo que una vez por segundo (recuerde la implementación del método *lastTime* en el controlador).

El método de ayuda *StartUp* generalmente hace lo mismo que el antiguo manejador *OnInit* en el Asesor Experto *MultiMartingale*. Rellena la estructura *Settings* con opciones de configuración, comprobando que sean correctas y creando un conjunto de sistemas de trading *TradingStrategyPool* consistente en objetos de la clase *UnityMartingale* para diferentes símbolos de trading *WorkSymbols*. Sin embargo, ahora este proceso se divide en dos etapas debido a que tenemos que esperar información sobre la distribución de divisas entre los búferes. Por lo tanto, la función *StartUp* tiene un parámetro de entrada que denota una llamada de *OnInit* y posteriormente de *OnChartEvent*.

Al analizar el código fuente de *StartUp*, es importante recordar que la inicialización es diferente para los casos en que operamos con un solo instrumento que coincide con el gráfico actual y cuando se especifica una cesta de instrumentos. El primer modo está activo cuando *WorkSymbols* es una línea vacía. Es conveniente para optimizar un Asesor Experto para un instrumento específico. Una vez encontrados los ajustes para varios instrumentos, podemos combinarlos en *WorkSymbols*.

```
bool StartUp(const bool init = false)
{
   if(WorkSymbols == "")
   {
      Settings settings =
      {
         UseTime, HourStart, HourEnd,
         Lots, Factor, Limit,
         StopLoss, TakeProfit,
         StartType, Magic, SkipTimeOnError, Trailing, _Symbol
      };

      if(settings.validate())
      {
         if(init)
         {
            Print("Input settings:");
            settings.print();
         }
      }
      else
      {
         if(init) Print("Wrong settings, please fix");
         return false;
      }
      if(!init)
      {
         ...// creating a trading system based on the indicator
      }
   }
   else
   {
      Print("Parsed settings:");
      Settings settings[];
      if(!Settings::parseAll(WorkSymbols, settings))
      {
         if(init) Print("Settings are incorrect, can't start up");
         return false;
      }
      if(!init)
      {
         ...// creating a trading system based on the indicator
      }
   }
   return true;
}
```

La función *StartUp* de *OnInit* se llama con el parámetro *true*, lo que significa que sólo se comprueba la corrección de los ajustes. La creación de un objeto del sistema de trading se retrasa hasta que se recibe un mensaje del indicador en *OnChartEvent*.

```
void OnChartEvent(const int id,
   const long &lparam, const double &dparam, const string &sparam)
{
   if(id == CHARTEVENT_CUSTOM + UnityBarLimit)
   {
      PrintFormat("%lld %f '%s'", lparam, dparam, sparam);
      if(lparam == 0) ArrayResize(currencies, 0);
      currenciesCount = (int)MathRound(dparam);
      PUSH(currencies, sparam);
      if(ArraySize(currencies) == currenciesCount)
      {
         if(pool[] == NULL)
         {
            start up(); // indicator readiness confirmation
         }
         else
         {
            Alert("Repeated initialization!");
         }
      }
   }
}
```

Aquí recordamos el número de divisas en la variable global *currenciesCount* y las almacenamos en el array *currencies*, tras lo cual llamamos a *StartUp* con el parámetro *false* (valor por defecto, por tanto omitido). Los mensajes llegan desde la cola en el orden en que existen en los búferes del indicador. Así, obtenemos una coincidencia entre el índice y el nombre de la divisa.

Cuando se vuelve a llamar a *StartUp*, se ejecuta un código adicional:

```
bool StartUp(const bool init = false)
{
   if(WorkSymbols == "") // one current symbol
   {
      ...
      if(!init) // final initialization after OnInit
      {
         controller[].attached(currenciesCount);
         // split _Symbol into 2 currencies from the currencies array []
         int first, second;
         if(!SplitSymbolToCurrencyIndices(_Symbol, first, second))
         {
            PrintFormat("Can't find currencies (%s %s) for %s",
               (first == -1 ? "base" : ""), (second == -1 ? "profit" : ""), _Symbol);
            return false;
         }
         // create a pool from a single strategy
         pool = new TradingStrategyPool(new UnityMartingale(settings,
            new UnitySignal(controller[], first, second)));
      }
   }
   else // symbol basket
   {
      ...
      if(!init) // final initialization after OnInit
      {
         controller[].attached(currenciesCount);

         const int n = ArraySize(settings);
         pool = new TradingStrategyPool(n);
         for(int i = 0; i < n; i++)
         {
            ...
            // split settings[i].symbol into 2 currencies from currencies[]
            int first, second;
            if(!SplitSymbolToCurrencyIndices(settings[i].symbol, first, second))
            {
               PrintFormat("Can't find currencies (%s %s) for %s",
                  (first == -1 ? "base" : ""), (second == -1 ? "profit" : ""),
                  settings[i].symbol);
            }
            else
            {
               // add a strategy to the pool on the next trading symbol
               pool[].push(new UnityMartingale(settings[i],
                  new UnitySignal(controller[], first, second)));
            }
         }
      }
   }
```

La función de ayuda *SplitSymbolToCurrencyIndices* selecciona la divisa base y la divisa de beneficio del símbolo pasado y encuentra sus índices en el array *currencies*. De este modo, obtenemos los datos de referencia para generar señales en los objetos de *UnitySignal*. Cada uno de ellos tendrá su propio par de índices de divisas.

```
bool SplitSymbolToCurrencyIndices(const string symbol, int &first, int &second)
{
   const string s1 = SymbolInfoString(symbol, SYMBOL_CURRENCY_BASE);
   const string s2 = SymbolInfoString(symbol, SYMBOL_CURRENCY_PROFIT);
   first = second = -1;
   for(int i = 0; i < ArraySize(currencies); ++i)
   {
      if(currencies[i] == s1) first = i;
      else if(currencies[i] == s2) second = i;
   }

   return first != -1 && second != -1;
}
```

En general, el Asesor Experto está listo.

> Puede ver que en los últimos ejemplos de Asesores Expertos tenemos clases de estrategias y clases de señales de trading. Deliberadamente las hicimos descendientes de las interfaces genéricas *TradingStrategy* y *TradingSignal* para poder posteriormente recopilar colecciones de implementaciones compatibles pero diferentes que puedan combinarse en el desarrollo de futuros Asesores Expertos. Por lo general, estas clases concretas unificadas deben separarse en archivos de encabezado independientes. En nuestros ejemplos no lo hemos hecho para simplificar la modificación paso a paso.
>
> No obstante, el enfoque descrito es estándar para la programación orientada a objetos (POO). En particular, como mencionamos en la sección sobre creación de borradores de Asesor Experto, junto con MetaTrader 5 viene un *framework* de archivos de encabezado con clases estándar de operaciones de trading, indicadores de señal, y gestión de dinero, que se utilizan en el Asistente MQL. En el sitio *mql5.com* se publican otras soluciones similares en los artículos y en la sección Code Base.
>
> Puede utilizar las jerarquías de clases ya creadas como base para sus proyectos, siempre que sean adecuadas en términos de capacidades y facilidad de uso.

Para completar el cuadro, hemos querido introducir en el Asesor Experto nuestro propio criterio de optimización basado en R2. Para evitar la contradicción entre la regresión lineal en la fórmula de cálculo de R2 y los lotes variables que se incluyen en nuestra estrategia, calcularemos el coeficiente no para la línea de balance habitual, sino para sus incrementos acumulados normalizados por tamaños de lote en cada operación.

Para ello, en el manejador *OnTester*, seleccionamos transacciones con los tipos DEAL_TYPE_BUY y DEAL_TYPE_SELL y con la dirección OUT. Solicitaremos todas las propiedades de transacción que forman el resultado financiero (beneficio/pérdida), es decir, DEAL_PROFIT, DEAL_SWAP, DEAL_COMMISSION, DEAL_FEE, así como su volumen DEAL_VOLUME.

```
#defineSTAT_PROPS5// number of requested deal properties

doubleOnTester()
{
HistorySelect(0,LONG_MAX);

constENUM_DEAL_PROPERTY_DOUBLEprops[STAT_PROPS] =
{
DEAL_PROFIT,DEAL_SWAP,DEAL_COMMISSION,DEAL_FEE,DEAL_VOLUME
};
doubleexpenses[][STAT_PROPS];
ulongtickets[];// needed because of 'select' method prototype, but useful for debuggi

DealFilterfilter;
filter.let(DEAL_TYPE, (1<<DEAL_TYPE_BUY) | (1<<DEAL_TYPE_SELL),IS::OR_BITWISE)
.let(DEAL_ENTRY, (1<<DEAL_ENTRY_OUT) | (1<<DEAL_ENTRY_INOUT) | (1<<DEAL_ENTRY_OUT_BY)
IS::OR_BITWISE)
.select(props,tickets,expenses);
...
```

A continuación, en el array *balance*, acumulamos los beneficios/pérdidas normalizados por los volúmenes de trading y calculamos el criterio R2 para ello.

```
   const int n = ArraySize(tickets);
   double balance[];
   ArrayResize(balance, n + 1);
   balance[0] = TesterStatistics(STAT_INITIAL_DEPOSIT);

   for(int i = 0; i < n; ++i)
   {
      double result = 0;
      for(int j = 0; j < STAT_PROPS - 1; ++j)
      {
         result += expenses[i][j];
      }
      result /= expenses[i][STAT_PROPS - 1]; // normalize by volume
      balance[i + 1] = result + balance[i];
   }
   const double r2 = RSquaredTest(balance);
   return r2 * 100;
}
```

La primera versión del Asesor Experto está básicamente lista. No hemos incluido la comprobación del modelo de ticks mediante *TickModel.mqh*. Se supone que el Asesor Experto será probado cuando genere ticks en el modo OHLC M1 o mejor. Cuando se detecta el modelo «sólo precios abiertos», el Asesor Experto enviará un frame especial con un estado de error al terminal y se descargará del probador. Por desgracia, esto sólo detendrá este pase, pero la optimización continuará. Por lo tanto, la copia del Asesor Experto que se ejecuta en el terminal emite una «alerta» para que el usuario interrumpa la optimización manualmente.

```
void OnTesterPass()
{
   ulong   pass;
   string  name;
   long    id;
   double  value;
   uchar   data[];
   while(FrameNext(pass, name, id, value, data))
   {
      if(name == "status" && id == 1)
      {
         Alert("Please stop optimization!");
         Alert("Tick model is incorrect: OHLC M1 or better is required");
         // it would be logical if the next call would stop all optimization,
         // but it is not
         ExpertRemove();
      }
   }
}
```

Puede optimizar los parámetros de SYMBOL SETTINGS para cualquier símbolo y repetir la optimización para símbolos diferentes. Al mismo tiempo, los grupos COMMON SETTINGS y UNITY SETTINGS deben contener siempre los mismos ajustes, ya que se aplican a todos los símbolos e instancias de los sistemas de trading. Por ejemplo, *Trailing* debe estar activado o desactivado para todas las optimizaciones. Observe también que las variables de entrada de un solo símbolo (es decir, el grupo SYMBOL SETTINGS) sólo tienen efecto mientras *WorkSymbols* contenga una cadena vacía. Por lo tanto, en la fase de optimización, debe mantenerla vacía.

Por ejemplo, para diversificar los riesgos, puede optimizar constantemente un Asesor Experto en pares completamente independientes: EURUSD, AUDJPY, GBPCHF, NZDCAD, o en otras combinaciones. El código fuente incluye tres archivos con ejemplos de configuraciones privadas.

```
#property tester_set "UnityMartingale-eurusd.set"
#property tester_set "UnityMartingale-gbpchf.set"
#property tester_set "UnityMartingale-audjpy.set"
```

Para operar con tres símbolos a la vez, estos ajustes deben «empaquetarse» en un parámetro común *WorkSymbols*:

```
EURUSD+0.01*1.6^5(200,200)[17,21];GBPCHF+0.01*1.2^8(600,800)[7,20];AUDJPY+0.01*1.2^8(
```

Esta configuración también se incluye en un archivo aparte.

```
#property tester_set "UnityMartingale-combo.set"
```

Uno de los problemas con la versión actual del Asesor Experto es que el informe del probador proporcionará estadísticas generales para todos los símbolos (más precisamente, para todas las estrategias de trading, ya que podemos incluir diferentes clases en el conjunto), mientras que sería interesante para nosotros supervisar y evaluar cada componente del sistema por separado.

Para ello es necesario aprender a calcular de forma independiente los principales indicadores financieros de trading, por analogía con la forma en que el probador lo hace por nosotros. Nos ocuparemos de esto en la segunda etapa del desarrollo del Asesor Experto.

UnityMartingaleDraft2.mq5

El cálculo estadístico puede ser necesario con bastante frecuencia, por lo que lo implementaremos en un archivo de encabezado separado *TradeReport.mqh*, donde organizaremos el código fuente en las clases apropiadas.

Llamemos a la clase principal *TradeReport*. Muchas variables de trading dependen de las curvas de saldo y margen libre (fondos propios). Por lo tanto, la clase contiene variables para el seguimiento del saldo actual y el beneficio, así como un array constantemente actualizado con el historial de saldos. No almacenaremos el historial del capital, porque puede cambiar en cada tick, y es mejor calcularlo sobre la marcha. Veremos un poco más adelante la razón de tener la curva de equilibrio.

```
class TradeReport
{
   double balance;     // current balance
   double floating;    // current floating profit
   double data[];      // full balance curve - prices
   datetime moments[]; // and date/time
   ...
```

La modificación y lectura de los campos de la clase se realiza mediante métodos, incluido el constructor, en el que el saldo se inicializa mediante la propiedad ACCOUNT_BALANCE.

```
   TradeReport()
   {
      balance = AccountInfoDouble(ACCOUNT_BALANCE);
   }

   void resetFloatingPL()
   {
      floating = 0;
   }

   void addFloatingPL(const double pl)
   {
      floating += pl;
   }

   void addBalance(const double pl)
   {
      balance += pl;
   }

   double getCurrent() const
   {
      return balance + floating;
   }
   ...
```

Estos métodos serán necesarios para calcular iterativamente (sobre la marcha) la reducción del capital. El array de saldos *data* será necesario para un cálculo puntual de la reducción del saldo (lo haremos al final de la prueba).

En función de las fluctuaciones de la curva (da igual que sea de saldo o capital), se debe calcular la reducción absoluta y relativa utilizando el mismo algoritmo. Por lo tanto, este algoritmo y las variables internas necesarias para él, que almacenan estados intermedios, se implementan en la estructura anidada *DrawDown*. En el siguiente código se muestran sus principales métodos y propiedades:

```
struct DrawDown
{
   double
   series_start,
   series_min,
   series_dd,
   series_dd_percent,
   series_dd_relative_percent,
   series_dd_relative;
   ...
   void reset();
   void calcDrawdown(const double &data[]);
   void calcDrawdown(const double amount);
   void print() const;
};
```

El primer método *calcDrawdown* calcula las reducciones cuando conocemos todo el array y esto se utilizará para el saldo. El segundo método *calcDrawdown* calcula la reducción de forma iterativa: cada vez que se le llama, se le indica el siguiente valor de la serie, y éste se utilizará para el capital.

Además de la reducción, como sabemos, existe un gran número de estadísticas estándar para los informes, pero para empezar sólo admitiremos algunas de ellas. Para ello, describimos los campos correspondientes en otra estructura anidada, *GenericStats*. Se hereda de *DrawDown* porque seguimos necesitando la reducción en el informe.

```
struct GenericStats: public DrawDown
{
   long deals;
   long trades;
   long buy_trades;
   long wins;
   long buy_wins;
   long sell_wins;

   double profits;
   double losses;
   double net;
   double pf;
   double average_trade;
   double recovery;
   double max_profit;
   double max_loss;
   double sharpe;
   ...
```

Por los nombres de las variables es fácil adivinar a qué métrica estándar corresponden. Algunas métricas son redundantes y, por tanto, se omiten. Por ejemplo, dado el número total de operaciones (*trades*) y el número de operaciones de compra entre ellas (*buy_trades*), podemos encontrar

fácilmente el número de operaciones de venta (*trades* - *sell_trades*). Lo mismo ocurre con las estadísticas complementarias de ganancias y pérdidas. Las rachas ganadoras y perdedoras no se tienen en cuenta. Quienes lo deseen pueden completar nuestro informe con estos indicadores.

Para la unificación con las estadísticas generales del probador, existe el método *fillByTester* que rellena todos los campos a través de la función *TesterStatistics*. Lo utilizaremos más adelante.

```
      void fillByTester()
      {
         deals = (long)TesterStatistics(STAT_DEALS);
         trades = (long)TesterStatistics(STAT_TRADES);
         buy_trades = (long)TesterStatistics(STAT_LONG_TRADES);
         wins = (long)TesterStatistics(STAT_PROFIT_TRADES);
         buy_wins = (long)TesterStatistics(STAT_PROFIT_LONGTRADES);
         sell_wins = (long)TesterStatistics(STAT_PROFIT_SHORTTRADES);

         profits = TesterStatistics(STAT_GROSS_PROFIT);
         losses = TesterStatistics(STAT_GROSS_LOSS);
         net = TesterStatistics(STAT_PROFIT);
         pf = TesterStatistics(STAT_PROFIT_FACTOR);
         average_trade = TesterStatistics(STAT_EXPECTED_PAYOFF);
         recovery = TesterStatistics(STAT_RECOVERY_FACTOR);
         sharpe = TesterStatistics(STAT_SHARPE_RATIO);
         max_profit = TesterStatistics(STAT_MAX_PROFITTRADE);
         max_loss = TesterStatistics(STAT_MAX_LOSSTRADE);

         series_start = TesterStatistics(STAT_INITIAL_DEPOSIT);
         series_min = TesterStatistics(STAT_EQUITYMIN);
         series_dd = TesterStatistics(STAT_EQUITY_DD);
         series_dd_percent = TesterStatistics(STAT_EQUITYDD_PERCENT);
         series_dd_relative_percent = TesterStatistics(STAT_EQUITY_DDREL_PERCENT);
         series_dd_relative = TesterStatistics(STAT_EQUITY_DD_RELATIVE);
      }
   };
```

Por supuesto, tenemos que implementar nuestro propio cálculo para aquellos saldos y fondos propios de los sistemas de trading que el probador no puede calcular. Anteriormente se han presentado prototipos de métodos de *calcDrawdown*. Durante la operación, rellenan el último grupo de campos con el prefijo «serie_dd». Además, la clase *TradeReport* contiene un método para calcular el ratio de Sharpe. Como entrada, toma una serie de números y un tipo de financiación sin riesgo. El código fuente completo se encuentra en el archivo adjunto.

```
   static double calcSharpe(const double &data[], const double riskFreeRate = 0);
```

Como se puede adivinar, al llamar a este método, el array miembro relevante de la clase *TradeReport* con saldos se pasará en el parámetro *data*. El proceso de llenar este array y llamar a los métodos anteriores para indicadores específicos ocurre en el método *calcStatistics* (ver abajo). Se le pasa un objeto filtro de transacciones como entrada (*filter*), depósito inicial (*start*) y tiempo (*origin*). Se supone que el código de llamada configurará el filtro de tal manera que sólo las operaciones del sistema de trading que nos interesa caigan bajo él.

El método devuelve una estructura llena *GenericStats*, y además, llena dos arrays dentro del objeto *TradeReport*, *data* y *moments*, con valores de saldo y referencias temporales de cambios, respectivamente. Lo necesitaremos en la versión final del Asesor Experto.

```
GenericStats calcStatistics(DealFilter &filter,
   const double start = 0, const datetime origin = 0,
   const double riskFreeRate = 0)
{
   GenericStats stats;
   ArrayResize(data, 0);
   ArrayResize(moments, 0);
   ulong tickets[];
   if(!filter.select(tickets)) return stats;

   balance = start;
   PUSH(data, balance);
   PUSH(moments, origin);

   for(int i = 0; i < ArraySize(tickets); ++i)
   {
      DealMonitor m(tickets[i]);
      if(m.get(DEAL_TYPE) == DEAL_TYPE_BALANCE) //deposit/withdrawal
      {
         balance += m.get(DEAL_PROFIT);
         PUSH(data, balance);
         PUSH(moments, (datetime)m.get(DEAL_TIME));
      }
      else if(m.get(DEAL_TYPE) == DEAL_TYPE_BUY
         || m.get(DEAL_TYPE) == DEAL_TYPE_SELL)
      {
         const double profit = m.get(DEAL_PROFIT) + m.get(DEAL_SWAP)
            + m.get(DEAL_COMMISSION) + m.get(DEAL_FEE);
         balance += profit;

         stats.deals++;
         if(m.get(DEAL_ENTRY) == DEAL_ENTRY_OUT
            || m.get(DEAL_ENTRY) == DEAL_ENTRY_INOUT
            || m.get(DEAL_ENTRY) == DEAL_ENTRY_OUT_BY)
         {
            PUSH(data, balance);
            PUSH(moments, (datetime)m.get(DEAL_TIME));
            stats.trades++;        // trades are counted by exit deals
            if(m.get(DEAL_TYPE) == DEAL_TYPE_SELL)
            {
               stats.buy_trades++; // closing with a deal in the opposite directio
            }
            if(profit >= 0)
            {
               stats.wins++;
               if(m.get(DEAL_TYPE) == DEAL_TYPE_BUY)
               {
                  stats.sell_wins++; // closing with a deal in the opposite direct
               }
               else
               {
```

```
                        stats.buy_wins++;
                     }
                  }
               }
               else if(!TU::Equal(profit, 0))
               {
                  PUSH(data, balance); // entry fee (if any)
                  PUSH(moments, (datetime)m.get(DEAL_TIME));
               }

               if(profit >= 0)
               {
                  stats.profits += profit;
                  stats.max_profit = fmax(profit, stats.max_profit);
               }
               else
               {
                  stats.losses += profit;
                  stats.max_loss = fmin(profit, stats.max_loss);
               }
            }
         }

         if(stats.trades > 0)
         {
            stats.net = stats.profits + stats.losses;
            stats.pf = -stats.losses > DBL_EPSILON ?
               stats.profits / -stats.losses : MathExp(10000.0); // NaN(+inf)
            stats.average_trade = stats.net / stats.trades;
            stats.sharpe = calcSharpe(data, riskFreeRate);
            stats.calcDrawdown(data);     // fill in all fields of the DrawDown substruc
            stats.recovery = stats.series_dd > DBL_EPSILON ?
               stats.net / stats.series_dd : MathExp(10000.0);
         }
         return stats;
      }
   };
```

Aquí puede ver cómo llamamos a *calcSharpe* y *calcDrawdown* para obtener los indicadores correspondientes en el array *data*. El resto de indicadores se calculan directamente en el bucle dentro de *calcStatistics*.

La clase *TradeReport* está lista, y podemos ampliar la funcionalidad del Asesor Experto a la versión *UnityMartingaleDraft2.mq5*.

Añadamos nuevos miembros a la clase *UnityMartingale*.

```
class UnityMartingale: public TradingStrategy
{
protected:
   ...
   TradeReport report;
   TradeReport::DrawDown equity;
   const double deposit;
   const datetime epoch;
   ...
```

Necesitamos el objeto *report* para llamar a *calcStatistics*, donde se incluirá la reducción de saldos. El objeto *equity* es necesario para un cálculo independiente de la reducción de capital. El saldo y la fecha iniciales, así como el inicio del cálculo de la reducción del capital, se establecen en el constructor.

```
public:
   UnityMartingale(const Settings &state, TradingSignal *signal):
      symbol(state.symbol), deposit(AccountInfoDouble(ACCOUNT_BALANCE)),
      epoch(TimeCurrent())
   {
      ...
      equity.calcDrawdown(deposit);
      ...
   }
```

La continuación del cálculo de la reducción por capital se realiza sobre la marcha, con cada llamada al método *trade*.

```
   virtual bool trade() override
   {
      ...
      if(MQLInfoInteger(MQL_TESTER))
      {
         if(position[])
         {
            report.resetFloatingPL();
            // after reset, sum all floating profits
            // why we call addFloatingPL for each existing position,
            // but this strategy has a maximum of 1 position at a time
            report.addFloatingPL(position[].get(POSITION_PROFIT)
               + position[].get(POSITION_SWAP));
            // after taking into account all the amounts - update the drawdown
            equity.calcDrawdown(report.getCurrent());
         }
      }
      ...
   }
```

Esto no es todo lo que se necesita para un cálculo correcto. Hay que tener en cuenta la ganancia o pérdida flotante sobre el saldo. La parte de código anterior sólo muestra la llamada a *addFloatingPL*, pero la clase *TradeReport* también tiene un método para modificar el saldo: *addBalance*. No obstante, el saldo sólo cambia cuando se cierra la posición.

Gracias al concepto POO, cerrar una posición en nuestra situación corresponde a borrar el objeto *position* de la clase *PositionState*. Entonces, ¿por qué no podemos interceptarla?

La clase *PositionState* no proporciona ningún medio para ello, pero podemos declarar una clase derivada *PositionStateWithEquity* con un constructor y un destructor especiales.

Al crear un objeto, no sólo se pasa al constructor el identificador de posición, sino también un puntero al objeto de informe al que habrá que enviar información.

```
class PositionStateWithEquity: public PositionState
{
   TradeReport *report;

public:
   PositionStateWithEquity(const long t, TradeReport *r):
      PositionState(t), report(r) { }
   ...
```

En el destructor encontramos todas las operaciones por el ID de la posición cerrada, calculamos el resultado financiero total (junto con las comisiones y otras deducciones), y luego llamamos a *addBalance* para relacionar el objeto *report*.

```
   ~PositionStateWithEquity()
   {
      if(HistorySelectByPosition(get(POSITION_IDENTIFIER)))
      {
         double result = 0;
         DealFilter filter;
         int props[] = {DEAL_PROFIT, DEAL_SWAP, DEAL_COMMISSION, DEAL_FEE};
         Tuple4<double, double, double, double> overheads[];
         if(filter.select(props, overheads))
         {
            for(int i = 0; i < ArraySize(overheads); ++i)
            {
               result += NormalizeDouble(overheads[i]._1, 2)
                  + NormalizeDouble(overheads[i]._2, 2)
                  + NormalizeDouble(overheads[i]._3, 2)
                  + NormalizeDouble(overheads[i]._4, 2);
            }
         }
         if(CheckPointer(report) != POINTER_INVALID) report.addBalance(result);
      }
   }
};
```

Queda por aclarar un punto: cómo crear objetos de clase *PositionStateWithEquity* para las posiciones en lugar de *PositionState*. Para ello, basta con cambiar el operador *new* en un par de lugares en los que se llama en la clase *TradingStrategy*.

```
position=MQLInfoInteger(MQL_TESTER) ?
newPositionStateWithEquity(tickets[0], &report) :newPositionState(tickets[0]);
```

Así, hemos puesto en marcha la recogida de datos. Ahora necesitamos generar directamente un informe, es decir, llamar a *calcStatistics*. Aquí tenemos que ampliar nuestra interfaz *TradingStrategy*: le añadimos el método *statement*.

```
interface TradingStrategy
{
   virtual bool trade(void);
   virtual bool statement();
};
```

A continuación, en esta implementación actual, pensada para nuestra estrategia, podremos llevar el trabajo a su conclusión lógica.

```
class UnityMartingale: public TradingStrategy
{
   ...
   virtual bool statement() override
   {
      if(MQLInfoInteger(MQL_TESTER))
      {
         Print("Separate trade report for ", settings.symbol);
         // equity drawdown should already be calculated on the fly
         Print("Equity DD:");
         equity.print();

         // balance drawdown is calculated in the resulting report
         Print("Trade Statistics (with Balance DD):");
         // configure the filter for a specific strategy
         DealFilter filter;
         filter.let(DEAL_SYMBOL, settings.symbol)
            .let(DEAL_MAGIC, settings.magic, IS::EQUAL_OR_ZERO);
           // zero "magic" number is needed for the last exit deal
           // - it is done by the tester itself
         HistorySelect(0, LONG_MAX);
         TradeReport::GenericStats stats =
            report.calcStatistics(filter, deposit, epoch);
         stats.print();
      }
      return false;
   }
   ...
```

El nuevo método simplemente imprimirá todos los indicadores calculados en el registro. Reenviando el mismo método a través del conjunto de sistemas de trading *TradingStrategyPool*, vamos a solicitar informes separados para todos los símbolos al manejador *OnTester*.

```
double OnTester()
{
   ...
   if(pool[] != NULL)
   {
      pool[].statement(); // ask all trading systems to display their results
   }
   ...
}
```

Comprobemos si nuestro informe es correcto. Para ello, vamos a ejecutar el Asesor Experto en el probador, un símbolo cada vez, y a comparar el informe estándar con nuestros cálculos. Por ejemplo, para configurar *UnityMartingale-eurusd.set*, operando en EURUSD H1 obtendremos tales indicadores para 2021.

Bars	6232	Ticks	1474964	Symbols	1
Initial Deposit	10 000.00				
Total Net Profit	18.35	Balance Drawdow...	1.49	Equity Drawdown ...	1.80
Gross Profit	57.97	Balance Drawdow...	5.73 (0.06%)	Equity Drawdown ...	6.23 (0.06%)
Gross Loss	-39.62	Balance Drawdow...	0.06% (5.73)	Equity Drawdown ...	0.06% (6.23)
Profit Factor	1.46	Expected Payoff	0.19	Margin Level	81232.33%
Recovery Factor	2.95	Sharpe Ratio	0.15	Z-Score	1.22 (77.75%)
AHPR	1.0000 (0.00...	LR Correlation	0.95	OnTester result	80.38985394...
GHPR	1.0000 (0.00...	LR Standard Error	2.13		
Total Trades	97	Short Trades (won ...	54 (42.59%)	Long Trades (won ...	43 (44.19%)
Total Deals	194	Profit Trades (% of...	42 (43.30%)	Loss Trades (% of t...	55 (56.70%)
	Largest	profit trade	2.00	loss trade	-2.01
	Average	profit trade	1.38	loss trade	-0.72
	Maximum	consecutive wins ($)	5 (4.37)	consecutive losses ...	7 (-4.77)
	Maximal	consecutive profit ...	6.00 (3)	consecutive loss (c...	-4.77 (7)
	Average	consecutive wins	2	consecutive losses	2

Informe del probador para 2021, EURUSD H1

En el registro, nuestra versión se muestra como dos estructuras: *DrawDown* con reducción del capital y *GenericStats* con indicadores de reducción del saldo y otras estadísticas.

```
Informe comercial separado para EURUSD
Equity DD:
[maxpeak] [minpeak] [series_start] [series_min] [series_dd] [series_dd_percent] »
[0] 10022.48 10017.03 10000.00 9998.20 6.23 0.06 »
» [series_dd_relative_percent] [series_dd_relative]
» 0.06 6.23

Trade Statistics (with Balance DD):
[maxpeak] [minpeak] [series_start] [series_min] [series_dd] [series_dd_percent] »
[0] 10022.40 10017.63 10000.00 9998.51 5.73 0.06 »
» [series_dd_relative_percent] [series_dd_relative] »
» 0.06 5.73 »
» [deals] [trades] [buy_trades] [wins] [buy_wins] [sell_wins] [profits] [losses] [net
» 194 97 43 42 19 23 57.97 -39.62 18.35 1.46 »
» [average_trade] [recovery] [max_profit] [max_loss] [sharpe]
» 0.19 3.20 2.00 -2.01 0.15
```

Es fácil comprobar que estos números coinciden con el informe del probador.

Ahora vamos a empezar a operar en el mismo periodo para tres símbolos a la vez (configurando *UnityMartingale-combo.set*).

Además de las entradas EURUSD, en el diario aparecerán estructuras para GBPCHF y AUDJPY.

```
Informe comercial separado para GBPCHF
Equity DD:
[maxpeak] [minpeak] [series_start] [series_min] [series_dd] [series_dd_percent] »
[0] 10029.50 10000.19 10000.00 9963.65 62.90 0.63 »
» [series_dd_relative_percent] [series_dd_relative]
» 0.63 62.90
Trade Statistics (with Balance DD):
[maxpeak] [minpeak] [series_start] [series_min] [series_dd] [series_dd_percent] »
[0] 10023.68 9964.28 10000.00 9964.28 59.40 0.59 »
» [series_dd_relative_percent] [series_dd_relative] »
» 0.59 59.40 »
» [deals] [trades] [buy_trades] [wins] [buy_wins] [sell_wins] [profits] [losses] [net
» 600 300 154 141 63 78 394.53 -389.33 5.20 1.01 »
» [average_trade] [recovery] [max_profit] [max_loss] [sharpe]
» 0.02 0.09 9.10 -6.73 0.01

Informe comercial separado para AUDJPY
Equity DD:
[maxpeak] [minpeak] [series_start] [series_min] [series_dd] [series_dd_percent] »
[0] 10047.14 10041.53 10000.00 9961.62 48.20 0.48 »
» [series_dd_relative_percent] [series_dd_relative]
» 0.48 48.20
Trade Statistics (with Balance DD):
[maxpeak] [minpeak] [series_start] [series_min] [series_dd] [series_dd_percent] »
[0] 10045.21 10042.75 10000.00 9963.62 44.21 0.44 »
» [series_dd_relative_percent] [series_dd_relative] »
» 0.44 44.21 »
» [deals] [trades] [buy_trades] [wins] [buy_wins] [sell_wins] [profits] [losses] [net
» 332 166 91 89 54 35 214.79 -170.20 44.59 1.26 »
» [average_trade] [recovery] [max_profit] [max_loss] [sharpe]
» 0.27 1.01 7.58 -5.17 0.09
```

En este caso, el informe del probador contendrá datos generalizados, por lo que, gracias a nuestras clases, hemos recibido detalles antes inaccesibles.

Sin embargo, consultar un pseudoinforme en un registro no resulta muy cómodo. Además, me gustaría ver una representación gráfica de la línea de balance como mínimo, ya que su aspecto suele decir más sobre la idoneidad del sistema que las meras estadísticas.

Vamos a mejorar el Asesor Experto dándole la capacidad de generar informes visuales en formato HTML: al fin y al cabo, los informes del probador también pueden exportarse a HTML, guardarse y compararse a lo largo del tiempo. Además, en el futuro, estos informes podrán transmitirse en frames al terminal justo durante la optimización, y el usuario podrá empezar a estudiar los informes de pasadas concretas incluso antes de que finalice todo el proceso.

Esta será la penúltima versión del ejemplo *UnityMartingaleDraft3.mq5*.

UnityMartingaleDraft3.mq5

La visualización del informe de trading incluye una línea de balance y una tabla con indicadores estadísticos. No generaremos un informe completo similar al del probador, sino que nos limitaremos a los valores seleccionados más importantes. Nuestro propósito es implantar un mecanismo de trabajo que luego pueda personalizarse en función de las necesidades personales.

Dispondremos la base del algoritmo en forma de la clase *TradeReportWriter* (*TradeReportWriter.mqh*). La clase podrá almacenar un número arbitrario de informes de diferentes sistemas de trading: cada uno en un objeto separado *DataHolder*, que incluye arrays de valores de saldo y marcas de tiempo (*data* y *when*, respectivamente), la estructura *stats* con estadísticas, así como el título, el color y la anchura de la línea que se desea mostrar.

```
class TradeReportWriter
{
protected:
   class DataHolder
   {
   public:
      double data[];                    // balance changes
      datetime when[];                  // balance timestamps
      string name;                      // description
      color clr;                        // color
      int width;                        // line width
      TradeReport::GenericStats stats; // trading indicators
   };
   ...
```

Disponemos de un array de punteros automáticos *curves* asignados a los objetos de la clase *DataHolder*. Además, necesitaremos límites comunes en cuanto a importes y plazos para que coincidan con las líneas de todos los sistemas de trading del cuadro. Esto lo proporcionarán las variables *lower*, *upper*, *start* y *stop*.

```
   AutoPtr<DataHolder> curves[];
   double lower, upper;
   datetime start, stop;

public:
   TradeReportWriter(): lower(DBL_MAX), upper(-DBL_MAX), start(0), stop(0) { }
   ...
```

El método *addCurve* añade una línea de balance.

```
virtual bool addCurve(double &data[], datetime &when[], const string name,
   const color clr = clrNONE, const int width = 1)
{
   if(ArraySize(data) == 0 || ArraySize(when) == 0) return false;
   if(ArraySize(data) != ArraySize(when)) return false;
   DataHolder *c = new DataHolder();
   if(!ArraySwap(data, c.data) || !ArraySwap(when, c.when))
   {
      delete c;
      return false;
   }

   const double max = c.data[ArrayMaximum(c.data)];
   const double min = c.data[ArrayMinimum(c.data)];

   lower = fmin(min, lower);
   upper = fmax(max, upper);
   if(start == 0) start = c.when[0];
   else if(c.when[0] != 0) start = fmin(c.when[0], start);
   stop = fmax(c.when[ArraySize(c.when) - 1], stop);

   c.name = name;
   c.clr = clr;
   c.width = width;
   ZeroMemory(c.stats); // no statistics by default
   PUSH(curves, c);
   return true;
}
```

La segunda versión del método *addCurve* añade no sólo una línea de balance, sino también un conjunto de variables financieras en la estructura *GenericStats*.

```
virtual bool addCurve(TradeReport::GenericStats &stats,
   double &data[], datetime &when[], const string name,
   const color clr = clrNONE, const int width = 1)
{
   if(addCurve(data, when, name, clr, width))
   {
      curves[ArraySize(curves) - 1].stats = stats;
      return true;
   }
   return false;
}
```

El método más importante de la clase que visualiza el informe se hace abstracto.

```
virtual void render() = 0;
```

Esto permite implementar muchas formas de visualizar los informes, por ejemplo, tanto registrando en archivos de distintos formatos, como dibujando directamente sobre el gráfico. A continuación nos limitaremos a la formación de archivos HTML, ya que es el método tecnológicamente más avanzado y extendido.

La nueva clase *HTMLReportWriter* tiene un constructor, cuyos parámetros especifican el nombre del archivo, así como el tamaño de la imagen con curvas de balance. Generaremos la imagen propiamente dicha en el conocido formato de gráficos vectoriales SVG: es ideal en este caso porque es un subconjunto del lenguaje XML, que es el propio HTML.

```
class HTMLReportWriter: public TradeReportWriter
{
   int handle;
   int width, height;

public:
   HTMLReportWriter(const string name, const int w = 600, const int h = 400):
      width(w), height(h)
   {
      handle = FileOpen(name,
         FILE_WRITE | FILE_TXT | FILE_ANSI | FILE_REWRITE);
   }

   ~HTMLReportWriter()
   {
      if(handle != 0) FileClose(handle);
   }

   void close()
   {
      if(handle != 0) FileClose(handle);
      handle = 0;
   }
   ...
```

Antes de pasar al principal método público *render*, es necesario presentar al lector una tecnología que se describirá en detalle en la Parte 7 y última del libro. Estamos hablando de recursos: archivos y arrays de datos arbitrarios conectados a un programa MQL para trabajar con multimedia (sonido e imágenes), incrustar indicadores compilados, o simplemente como repositorio de información de la aplicación. Es esta última opción la que utilizaremos ahora.

La cuestión es que es mejor generar una página HTML no completamente a partir de código MQL, sino basándose en una plantilla (plantilla de página), en la que el código MQL sólo insertará los valores de algunas variables. Se trata de una técnica muy conocida en programación que permite separar el algoritmo y la representación externa del programa (o el resultado de su trabajo). Gracias a ello, podemos experimentar por separado con la plantilla HTML y el código MQL, trabajando con cada uno de los componentes en un entorno familiar. En concreto, MetaEditor todavía no es muy adecuado para editar páginas web y visualizarlas, al igual que un navegador estándar no sabe nada de MQL5 (aunque esto se puede arreglar).

Almacenaremos las plantillas de informes HTML en archivos de texto conectados al código fuente MQL5 como recursos. La conexión se realiza mediante una directiva especial *#resource*. Por ejemplo, en el archivo *TradeReportWriter.mqh* aparece la siguiente línea:

```
#resource "TradeReportPage.htm" as string ReportPageTemplate
```

Significa que junto al código fuente debe estar el archivo *TradeReportPage.htm*, que estará disponible en el código MQL como cadena *ReportPageTemplate*. Por extensión, se puede entender que el archivo

es una página web. He aquí el contenido de este archivo con abreviaturas (no tenemos la tarea de formar al lector en desarrollo web, aunque, al parecer, tener conocimientos al respecto puede ser útil también para un operador de trading). Las sangrías se añaden para representar visualmente la jerarquía de anidamiento de las etiquetas HTML; no hay sangrías en el archivo.

```
<!DOCTYPE html>
<html>
   <head>
      <title>Trade Report</title>
      <style>
         *{font: 9pt "Segoe UI";}
         .center{width:fit-content;margin:0 auto;}
         ...
      </style>
   </head>
   <body>
      <div class="center">
         <h1>Trade Report</h1>
         ~
      </div>
   </body>
   <script>
   ...
   </script>
</html>
```

Los fundamentos de las plantillas son elegidos por el desarrollador. Existe un gran número de sistemas de plantillas HTML ya preparadas, pero ofrecen muchas funciones redundantes y, por tanto, son demasiado complejas para nuestro ejemplo. Desarrollaremos nuestro propio concepto.

Para empezar, observemos que la mayoría de las páginas web tienen una parte inicial (encabezado), una parte final (pie de página) y entre ambas se sitúa la información útil. El proyecto de informe mencionado no es una excepción en este sentido. Utiliza el carácter de tilde '~' para indicar contenido útil. En lugar de ello, el código MQL tendrá que insertar una imagen de saldo y una tabla con indicadores. Pero la presencia de '~' no es necesaria, ya que la página puede ser un todo único, es decir, la parte central muy útil: después de todo, el código MQL puede, si es necesario, insertar el resultado del procesamiento de una plantilla en otra.

> Para terminar la digresión sobre las plantillas HTML, vamos a prestar atención a una cosa más. En teoría, una página web se compone de etiquetas que realizan funciones esencialmente diferentes. Las etiquetas HTML estándar indican al navegador qué debe mostrar. Además de ellos, existen estilos en cascada (CSS), que describen cómo mostrarlo. Por último, la página puede tener un componente dinámico en forma de scripts de JavaScript que controlen interactivamente tanto la primera como la segunda.
>
> Normalmente, estos tres componentes se planifican de forma independiente, es decir, por ejemplo, una plantilla HTML, en sentido estricto, debe contener sólo HTML, pero no CSS ni JavaScript. Esto permite ***«desvincular» el contenido, la apariencia y el comportamiento*** de la página web, lo que facilita el desarrollo (se recomienda encarecidamente seguir el mismo enfoque en MQL5).
>
> Sin embargo, en nuestro ejemplo, hemos incluido todos los componentes en la plantilla. En particular, en la plantilla anterior, vemos la etiqueta <style> con estilos CSS y la etiqueta <script>

con algunas funciones JavaScript, que se omiten. Esto se hace para simplificar el ejemplo, con hincapié en las características MQL5 en lugar de desarrollo web.

Teniendo una plantilla de página web en la variable *ReportPageTemplate* conectada como recurso, podemos escribir el método *render*.

```
virtual void render() override
{
   string headerAndFooter[2];
   StringSplit(ReportPageTemplate, '~', headerAndFooter);
   FileWriteString(handle, headerAndFooter[0]);
   renderContent();
   FileWriteString(handle, headerAndFooter[1]);
}
...
```

En realidad, divide la página en mitad superior e inferior mediante el carácter '~', las muestra tal cual y llama a un método de ayuda *renderContent* entre ellas.

Ya hemos descrito que el informe consistirá en un cuadro general con curvas de balance y tablas con indicadores de sistemas de trading, por lo que la implementación *renderContent* es natural.

```
private:
   void renderContent()
   {
      renderSVG();
      renderTables();
   }
```

La generación de imágenes dentro de *renderSVG* se basa en otro archivo de plantilla *TradeReportSVG.htm*, que se vincula a una variable de cadena *SVGBoxTemplate*:

```
#resource "TradeReportSVG.htm" as string SVGBoxTemplate
```

El contenido de esta plantilla es el último que enumeramos aquí. Quienes lo deseen pueden consultar por sí mismos los códigos fuente del resto de plantillas.

```
<span id="params" style="display:block;width:%WIDTH%px;text-align:center;"></span>
<a id="main" style="display:block;text-align:center;">
   <svg width="%WIDTH%" height="%HEIGHT%" xmlns="http://www.w3.org/2000/svg">
      <style>.legend {font: bold 11px Consolas;}</style>
      <rect x="0" y="0" width="%WIDTH%" height="%HEIGHT%"
         style="fill:none; stroke-width:1; stroke: black;"/>
      ~
   </svg>
</a>
```

En el código del método *renderSVG*, veremos el conocido truco de dividir el contenido en dos bloques «antes» y «después» de la tilde, pero aquí hay algo nuevo:

```
   void renderSVG()
   {
      string headerAndFooter[2];
      if(StringSplit(SVGBoxTemplate, '~', headerAndFooter) != 2) return;
      StringReplace(headerAndFooter[0], "%WIDTH%", (string)width);
      StringReplace(headerAndFooter[0], "%HEIGHT%", (string)height);
      FileWriteString(handle, headerAndFooter[0]);

      for(int i = 0; i < ArraySize(curves); ++i)
      {
         renderCurve(i, curves[i][].data, curves[i][].when,
            curves[i][].name, curves[i][].clr, curves[i][].width);
      }

      FileWriteString(handle, headerAndFooter[1]);
   }
```

En la parte superior de la página, en la cadena *headerAndFooter[0]*, buscamos subcadenas de la forma especial «%WIDTH%» y «%HEIGHT%», y las sustituimos por la anchura y la altura requeridas de la imagen. Este es el principio por el que funciona la sustitución de valores en nuestras plantillas. Por ejemplo, en esta plantilla, estas subcadenas aparecen en la etiqueta *rect*:

```
<rect x="0" y="0" width="%WIDTH%" height="%HEIGHT%" style="fill:none; stroke-width:1;
```

Así, si el informe se pide con un tamaño de 600 por 400, la línea se convertirá en la siguiente:

```
<rect x="0" y="0" width="600" height="400" style="fill:none; stroke-width:1; stroke:
```

Esto mostrará un borde negro de 1 píxel de grosor de las dimensiones especificadas en el navegador.

La generación de etiquetas para dibujar líneas de balance específicas se gestiona mediante el método *renderCurve*, al que pasamos todos los arrays necesarios y otros ajustes (nombre, color y grosor). Dejaremos este método y otros muy especializados (*renderTables*, *renderTable*) para un estudio independiente.

Volvamos al módulo principal del Asesor Experto *UnityMartingaleDraft3.mq5*. Ajuste el tamaño de la imagen de los gráficos de balance y conecte *TradeReportWriter.mqh*.

```
#define MINIWIDTH  400
#define MINIHEIGHT 200

#include <MQL5Book/TradeReportWriter.mqh>
```

Para «conectar» las estrategias con el generador de informes, tendrá que modificar el método *statement* en la interfaz *TradingStrategy*: pase un puntero al objeto *TradeReportWriter*, que el código de llamada puede crear y configurar.

```
interface TradingStrategy
{
   virtual bool trade(void);
   virtual bool statement(TradeReportWriter *writer = NULL);
};
```

Ahora vamos a añadir algunas líneas en la implementación específica de este método en nuestra clase de estrategia *UnityMartingale*.

```
class UnityMartingale: public TradingStrategy
{
   ...
   TradeReport report;
   ...
   virtual bool statement(TradeReportWriter *writer = NULL) override
   {
      if(MQLInfoInteger(MQL_TESTER))
      {
         ...
         // it's already been done
         DealFilter filter;
         filter.let(DEAL_SYMBOL, settings.symbol)
            .let(DEAL_MAGIC, settings.magic, IS::EQUAL_OR_ZERO);
         HistorySelect(0, LONG_MAX);
         TradeReport::GenericStats stats =
            report.calcStatistics(filter, deposit, epoch);
         ...
         // adding this
         if(CheckPointer(writer) != POINTER_INVALID)
         {
            double data[];               // balance values
            datetime time[];             // balance points time to synchronize curves
            report.getCurve(data, time); // fill in the arrays and transfer to write
            return writer.addCurve(stats, data, time, settings.symbol);
         }
         return true;
      }
      return false;
   }
```

Todo se reduce a obtener un array de balance y una estructura con indicadores del objeto *report* (clase *TradeReport*) y pasarlo al objeto *TradeReportWriter*, llamando a *addCurve*.

Por supuesto, el conjunto de estrategias de trading garantiza la transferencia del mismo objeto *TradeReportWriter* a todas las estrategias para generar un informe combinado.

```
class TradingStrategyPool: public TradingStrategy
{
   ...
   virtual bool statement(TradeReportWriter *writer = NULL) override
   {
      bool result = false;
      for(int i = 0; i < ArraySize(pool); i++)
      {
         result = pool[i][].statement(writer) || result;
      }
      return result;
   }
```

Por último, el manejador *OnTester* ha sufrido la mayor modificación. Las siguientes líneas bastarían para generar un informe HTML de las estrategias de trading.

```
double OnTester()
{
   ...
   const static string tempfile = "temp.html";
   HTMLReportWriter writer(tempfile, MINIWIDTH, MINIHEIGHT);
   if(pool[] != NULL)
   {
      pool[].statement(&writer); // ask strategies to report their results
   }
   writer.render(); // write the received data to a file
   writer.close();
}
```

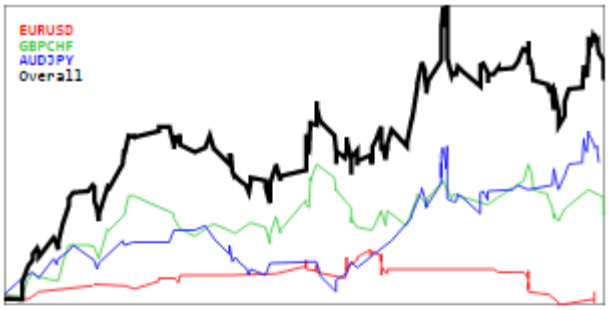
Sin embargo, para mayor claridad y comodidad del usuario, sería estupendo añadir al informe una curva de balance general, así como un cuadro con indicadores generales. Tiene sentido emitirlos sólo cuando se especifican varios símbolos en la configuración del Asesor Experto porque, de lo contrario, el informe de una estrategia coincide con el general del archivo.

Esto requería un poco más de código.

```
double OnTester()
{
   ...
   // had it before
   DealFilter filter;
   // set up the filter and fill in the array of deals based on it tickets
   ...
   const int n = ArraySize(tickets);

   // add this
   const bool singleSymbol = WorkSymbols == "";
   double curve[];    // total balance curve
   datetime stamps[]; // date and time of total balance points

   if(!singleSymbol) // the total balance is displayed only if there are several symb
   {
      ArrayResize(curve, n + 1);
      ArrayResize(stamps, n + 1);
      curve[0] = TesterStatistics(STAT_INITIAL_DEPOSIT);

      // MQL5 does not allow to know the test start time,
      // this could be found out from the first transaction,
      // but it is outside the filter conditions of a specific system,
      // so let's just agree to skip time 0 in calculations
      stamps[0] = 0;
   }

   for(int i = 0; i < n; ++i) // deal cycle
   {
      double result = 0;
      for(int j = 0; j < STAT_PROPS - 1; ++j)
      {
         result += expenses[i][j];
      }
      if(!singleSymbol)
      {
         curve[i + 1] = result + curve[i];
         stamps[i + 1] = (datetime)HistoryDealGetInteger(tickets[i], DEAL_TIME);
      }
      ...
   }
   if(!singleSymbol) // send the tester's statistics and the overall curve to the rep
   {
      TradeReport::GenericStats stats;
      stats.fillByTester();
      writer.addCurve(stats, curve, stamps, "Overall", clrBlack, 3);
   }
   ...
}
```

Veamos qué tenemos. Si ejecutamos el Asesor Experto con la configuración *UnityMartingale-combo.set*, tendremos el archivo *temp.html* en la carpeta *MQL5/Files* de uno de los agentes. Este es el aspecto que tiene en el registro:

Trade Report



EURUSD

Profit>	45.09	1.03	<Profit Factor	Trades	Wins	Losses	Total
Loss>	-43.99	0.01	<Sharpe	Buys	16	18	34
Net>	1.10	0.07	<Recovery	Sells	17	21	38
Deposit>	10000.00	144	<Deals	Total	33	39	72

DD	Absolute	Relative	Deposit	Max. Profit	Average	Max. Loss
Money	**16.83**	16.83	-0.00			
Percent	0.17%	**0.17%**	-0.00%	2.00	0.02	-2.00

GBPCHF

Profit>	131.26	1.25	<Profit Factor	Trades	Wins	Losses	Total
Loss>	-105.30	0.09	<Sharpe	Buys	17	18	35
Net>	25.96	1.32	<Recovery	Sells	18	22	40
Deposit>	10000.00	150	<Deals	Total	35	40	75

DD	Absolute	Relative	Deposit	Max. Profit	Average	Max. Loss
Money	**19.74**	19.74	-0.00			
Percent	0.20%	**0.20%**	-0.00%	8.78	0.35	-5.55

AUDJPY

Profit>	145.96	1.38	<Profit Factor	Trades	Wins	Losses	Total
Loss>	-106.06	0.13	<Sharpe	Buys	30	19	49
Net>	39.90	1.98	<Recovery	Sells	21	22	43
Deposit>	10000.00	184	<Deals	Total	51	41	92

DD	Absolute	Relative	Deposit	Max. Profit	Average	Max. Loss
Money	**20.19**	20.19	-0.00			
Percent	0.20%	**0.20%**	-0.00%	7.03	0.43	-4.90

Overall

Profit>	322.31	1.26	<Profit Factor	Trades	Wins	Losses	Total
Loss>	-255.35	0.09	<Sharpe	Buys	63	55	118
Net>	66.96	1.65	<Recovery	Sells	56	65	121
Deposit>	10000.00	478	<Deals	Total	119	120	239

DD	Absolute	Relative	Deposit	Max. Profit	Average	Max. Loss
Money	**40.68**	40.68	-0.00			
Percent	0.40%	**0.40%**	-0.00%	8.78	0.28	-5.55

Informe HTML para Asesor Experto con múltiples estrategias/símbolos de trading

Ahora que sabemos cómo generar informes en una pasada de prueba, podemos enviarlos al terminal durante la optimización, seleccionar los mejores sobre la marcha y presentarlos al usuario antes de que finalice todo el proceso. Todos los informes se colocarán en una carpeta separada dentro de *MQL5/Files*

del terminal. La carpeta recibirá un nombre que contendrá el símbolo y el marco temporal de la configuración del probador, así como el nombre del Asesor Experto.

UnityMartingale.mq5

Como sabemos, para enviar un archivo al terminal, basta con llamar a la función *FrameAdd*. Ya hemos generado el archivo en el marco de la versión anterior.

```
double OnTester()
{
   ...
   if(MQLInfoInteger(MQL_OPTIMIZATION))
   {
      FrameAdd(tempfile, 0, r2 * 100, tempfile);
   }
}
```

En la instancia receptora del Asesor Experto, realizaremos la preparación necesaria. Describamos la estructura *Pass* con los principales parámetros de cada pase de optimización.

```
struct Pass
{
   ulong id;          // pass number
   double value;      // optimization criterion value
   string parameters; // optimized parameters as list 'name=value'
   string preset;     // text to generate set-file (with all parameters)
};
```

En las cadenas *parameters*, los pares «name=value» se conectan con el símbolo '&'. Esto será útil para la interacción de páginas web de informes en el futuro (el símbolo '&' es el estándar para combinar parámetros en direcciones web). No hemos descrito el formato de los archivos de conjuntos, pero el siguiente código fuente que forma la cadena *preset* permite estudiar esta cuestión en la práctica.

A medida que lleguen frames, escribiremos mejoras según el criterio de optimización en el array *TopPasses*. La mejor pasada actual será siempre la última pasada del array y también está disponible en la variable *BestPass*.

```
Pass TopPasses[];     // stack of constantly improving passes (last one is best)
Pass BestPass;        // current best pass
string ReportPath;    // dedicated folder for all html files of this optimization
```

En el manejador *OnTesterInit* vamos a crear un nombre de carpeta.

```
void OnTesterInit()
{
   BestPass.value = -DBL_MAX;
   ReportPath = _Symbol + "-" + PeriodToString(_Period) + "-"
      + MQLInfoString(MQL_PROGRAM_NAME) + "/";
}
```

En el manejador *OnTesterPass* seleccionaremos secuencialmente sólo aquellos frames en los que el indicador haya mejorado, encontraremos para ellos los valores de los parámetros optimizados y otros, y añadiremos toda esta información al array de estructuras *Pass*.

```
void OnTesterPass()
{
   ulong   pass;
   string  name;
   long    id;
   double  value;
   uchar   data[];

   // input parameters for the pass corresponding to the current frame
   string  params[];
   uint    count;

   while(FrameNext(pass, name, id, value, data))
   {
      // collect passes with improved stats
      if(value > BestPass.value && FrameInputs(pass, params, count))
      {
         BestPass.preset = "";
         BestPass.parameters = "";
         // get optimized and other parameters for generating a set-file
         for(uint i = 0; i < count; i++)
         {
            string name2value[];
            int n = StringSplit(params[i], '=', name2value);
            if(n == 2)
            {
               long pvalue, pstart, pstep, pstop;
               bool enabled = false;
               if(ParameterGetRange(name2value[0], enabled, pvalue, pstart, pstep, ps
               {
                  if(enabled)
                  {
                     if(StringLen(BestPass.parameters)) BestPass.parameters += "&";
                     BestPass.parameters += params[i];
                  }

                  BestPass.preset += params[i] + "||" + (string)pstart + "||"
                    + (string)pstep + "||" + (string)pstop + "||"
                    + (enabled ? "Y" : "N") + "<br>\n";
               }
               else
               {
                  BestPass.preset += params[i] + "<br>\n";
               }
            }
         }

         BestPass.value = value;
         BestPass.id = pass;
         PUSH(TopPasses, BestPass);
         // write the frame with the report to the HTML file
```

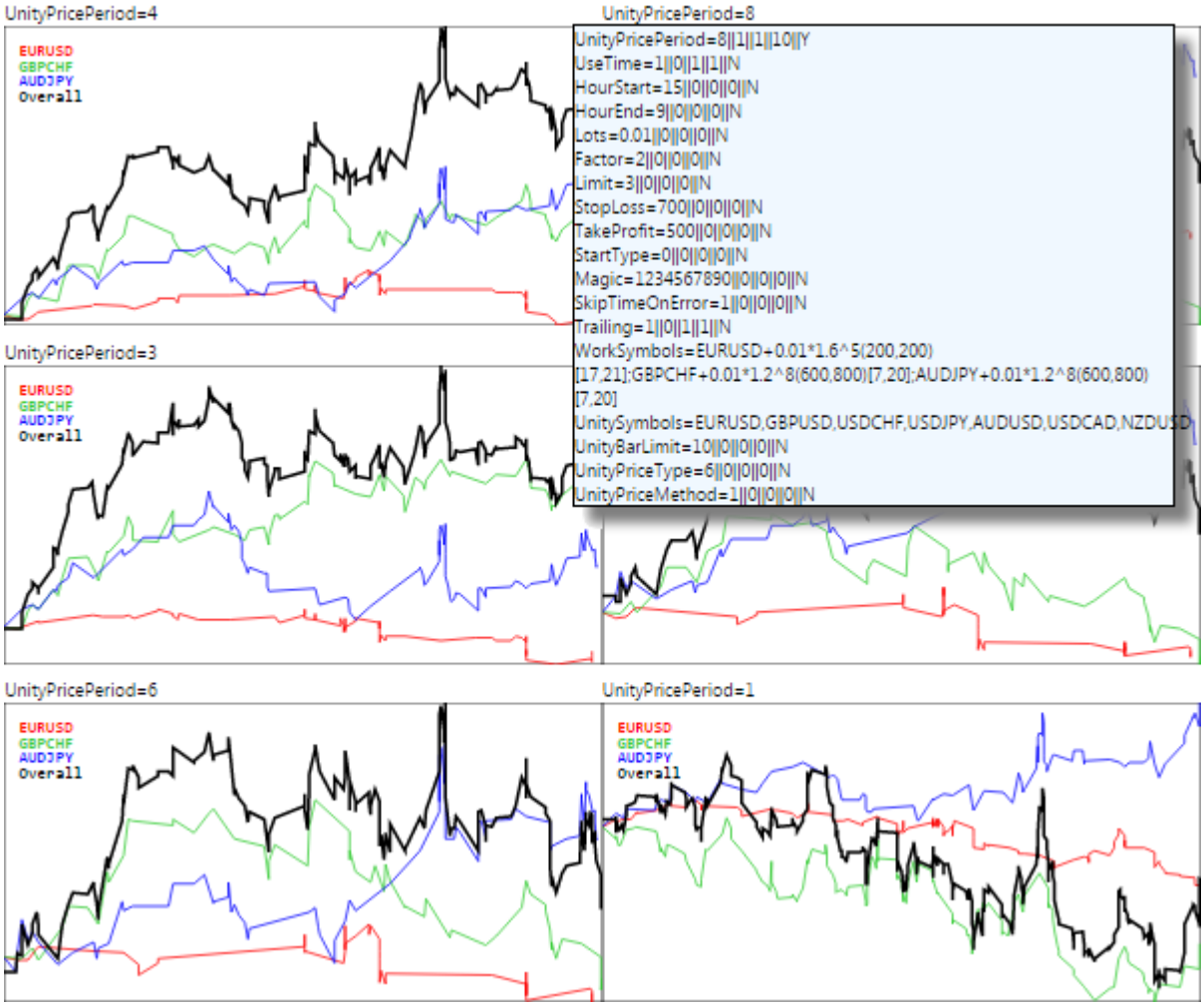
```
            const string text = CharArrayToString(data);
            int handle = FileOpen(StringFormat(ReportPath + "%06.3f-%lld.htm", value, pa
               FILE_WRITE | FILE_TXT | FILE_ANSI);
            FileWriteString(handle, text);
            FileClose(handle);
         }
      }
   }
```

Los informes resultantes con las mejoras se guardan en archivos con nombres que incluyen el valor del criterio de optimización y el número de pasada.

Ahora viene lo más interesante: en el manejador *OnTesterDeinit*, podemos formar un archivo HTML común (*overall.htm*), que permite ver todos los informes a la vez (o, digamos, los 100 primeros). Utiliza el mismo esquema con plantillas que hemos visto antes.

```
#resource "OptReportPage.htm" as string OptReportPageTemplate
#resource "OptReportElement.htm" as string OptReportElementTemplate

void OnTesterDeinit()
{
   int handle = FileOpen(ReportPath + "overall.htm",
      FILE_WRITE | FILE_TXT | FILE_ANSI, 0, CP_UTF8);
   string headerAndFooter[2];
   StringSplit(OptReportPageTemplate, '~', headerAndFooter);
   StringReplace(headerAndFooter[0], "%MINIWIDTH%", (string)MINIWIDTH);
   StringReplace(headerAndFooter[0], "%MINIHEIGHT%", (string)MINIHEIGHT);
   FileWriteString(handle, headerAndFooter[0]);
   // read no more than 100 best records from TopPasses
   for(int i = ArraySize(TopPasses) - 1, k = 0; i >= 0 && k < 100; --i, ++k)
   {
      string p = TopPasses[i].parameters;
      StringReplace(p, "&", " ");
      const string filename = StringFormat("%06.3f-%lld.htm",
         TopPasses[i].value, TopPasses[i].id);
      string element = OptReportElementTemplate;
      StringReplace(element, "%FILENAME%", filename);
      StringReplace(element, "%PARAMETERS%", TopPasses[i].parameters);
      StringReplace(element, "%PARAMETERS_SPACED%", p);
      StringReplace(element, "%PASS%", IntegerToString(TopPasses[i].id));
      StringReplace(element, "%PRESET%", TopPasses[i].preset);
      StringReplace(element, "%MINIWIDTH%", (string)MINIWIDTH);
      StringReplace(element, "%MINIHEIGHT%", (string)MINIHEIGHT);
      FileWriteString(handle, element);
   }
   FileWriteString(handle, headerAndFooter[1]);
   FileClose(handle);
}
```

En la siguiente imagen se muestra el aspecto de la página web de resumen después de optimizar *UnityMartingale.mq5* mediante el parámetro *UnityPricePeriod* en modo multidivisa.



Página web general con informes de trading de los mejores pases de optimización

Para cada informe, mostramos sólo la parte superior, donde cae el gráfico del balance. Esta parte es la más conveniente para obtener una estimación con sólo mirarla.

Encima de cada gráfico aparecen listas de parámetros optimizados («name=value&name=value...»). Al pulsar sobre una línea se abre un bloque con el texto del archivo de configuración de todas las opciones de configuración de este pase. Si hace clic dentro de un bloque, su contenido se copiará en el portapapeles. Se puede guardar en un editor de texto y obtener así un archivo de conjunto listo.

Si hace clic en el gráfico, accederá a la página específica del informe, junto con los scorecards (indicados anteriormente).

Al final de la sección abordamos una cuestión más. Antes prometimos demostrar el efecto de la función *TesterHideIndicators*. El Asesor Experto *UnityMartingale.mq5* utiliza actualmente el indicador *UnityPercentEvent.mq5*. Después de cualquier prueba, el indicador se muestra en el gráfico de apertura. Supongamos que queremos ocultar al usuario el mecanismo de trabajo del Asesor Experto y de dónde toma las señales. A continuación, puede llamar a la función *TesterHideIndicators* (con el parámetro *true*) en el manejador *OnInit*, antes de crear el objeto *UnityController*, en el que se recibe el descriptor a través de *iCustom*.

```
int OnInit()
{
   ...
   TesterHideIndicators(true);
   ...
   controller = new UnityController(UnitySymbols, barwise,
      UnityBarLimit, UnityPriceType, UnityPriceMethod, UnityPricePeriod);
   return INIT_SUCCEEDED;
}
```

Esta versión del Asesor Experto ya no mostrará el indicador en el gráfico. Sin embargo, no está muy bien escondido. Si miramos en el registro del probador, veremos líneas sobre programas cargados entre un montón de información útil: primero, un mensaje sobre la carga del propio Asesor Experto, y un poco más tarde, sobre la carga del indicador.

```
...
expert file added: Experts\MQL5Book\p6\UnityMartingale.ex5.
...
program file added: \Indicators\MQL5Book\p6\UnityPercentEvent.ex5.
...
```

Así, un usuario meticuloso puede averiguar el nombre del indicador. Esta posibilidad puede eliminarse mediante el mecanismo de recursos, que ya hemos mencionado de pasada en el contexto de los espacios en blanco de las páginas web. Resulta que el indicador compilado también se puede incrustar en un programa MQL (en un Asesor Experto u otro indicador) como un recurso. Y estos programas de recursos ya no se mencionan en el registro del probador. Estudiaremos los recursos en detalle en la 7ª Parte del libro, y ahora mostraremos las líneas asociadas a ellos en la versión final de nuestro Asesor Experto.

En primer lugar, vamos a describir el recurso con la directiva de indicador *#resource*. De hecho, contiene simplemente la ruta al archivo del indicador compilado (obviamente, ya debe estar compilado de antemano), y aquí es obligatorio utilizar barras invertidas dobles como delimitadores, ya que no se admiten barras diagonales simples en las rutas de recursos.

```
#resource "\\Indicators\\MQL5Book\\p6\\UnityPercentEvent.ex5"
```

A continuación, en las líneas con la llamada *iCustom*, sustituimos el operador anterior:

```
   UnityController(const string symbolList, const int offset, const int limit,
      const ENUM_APPLIED_PRICE type, const ENUM_MA_METHOD method, const int period):
      bar(offset), tickwise(!offset)
   {
      handle = iCustom(_Symbol, _Period,
         "MQL5Book/p6/UnityPercentEvent",                    // <---
         symbolList, limit, type, method, period);
      ...
```

Exactamente igual, pero con un enlace al recurso (nótese la sintaxis con un par de dos puntos '::' al principio, necesaria para distinguir entre las rutas normales en el sistema de archivos y las rutas dentro de los recursos).

```
UnityController(const string symbolList, const int offset, const int limit,
   const ENUM_APPLIED_PRICE type, const ENUM_MA_METHOD method, const int period):
   bar(offset), tickwise(!offset)
{
   handle = iCustom(_Symbol, _Period,
      "::Indicators\\MQL5Book\\p6\\UnityPercentEvent.ex5",  // <---
      symbolList, limit, type, method, period);
   ...
```

Ahora la versión compilada del Asesor Experto puede entregarse a los usuarios por sí sola, sin un indicador separado, ya que está oculta dentro del Asesor Experto. Esto no afecta en absoluto a su rendimiento, pero teniendo en cuenta el reto *TesterHideIndicators*, el dispositivo interno queda oculto. Hay que tener en cuenta que si el indicador se actualiza, el Asesor Experto también tendrá que volver a compilarse.

6.5.17 Cálculos matemáticos

El probador del terminal MetaTrader 5 puede utilizarse no sólo para probar estrategias de trading, sino también para realizar cálculos matemáticos. Para ello, seleccione el modo adecuado en los ajustes del probador, en la lista desplegable Simulación. Esta es la misma lista en la que seleccionamos el método de generación de ticks, pero en este caso, el probador no generará ticks ni cotizaciones, ni siquiera conectará el entorno de trading (cuenta de trading y símbolos).

La elección entre la enumeración completa de parámetros y un algoritmo genético depende del tamaño del espacio de búsqueda. Para el criterio de optimización, seleccione «Máximo personalizado». Otros campos de entrada en la configuración del probador (como el rango de fechas o los retrasos) no son importantes y, por tanto, se desactivan automáticamente.

En el modo «Cálculos matemáticos», cada ejecución del agente de pruebas se realiza con una llamada a sólo tres funciones: *OnInit*, *OnTester*, *OnDeinit*.

Un problema matemático típico para resolver en el probador de MetaTrader 5 es encontrar un extremo para una función de muchas variables. Para resolverlo es necesario declarar los parámetros de la función en forma de variables de entrada y colocar el bloque para calcular sus valores en *OnTester*.

El valor de la función para un conjunto específico de variables de entrada se devuelve como valor de salida de *OnTester*. No utilice ninguna función integrada que no sean funciones matemáticas en los cálculos.

Hay que recordar que, al optimizar, siempre se busca el valor máximo de la función *OnTester*. Por lo tanto, si necesita encontrar el mínimo, debe devolver los valores inversos o los valores multiplicados por -1.

Para entender cómo funciona, tomemos como ejemplo una función relativamente sencilla de dos variables con un máximo. Vamos a describirlo en el algoritmo de Asesor Experto *MathCalc.mq5*.

Se suele suponer que no conocemos la representación de la función de forma analítica, ya que, de lo contrario, sería posible calcular sus extremos. Pero ahora tomemos una fórmula bien conocida para asegurarnos de que la respuesta es correcta.

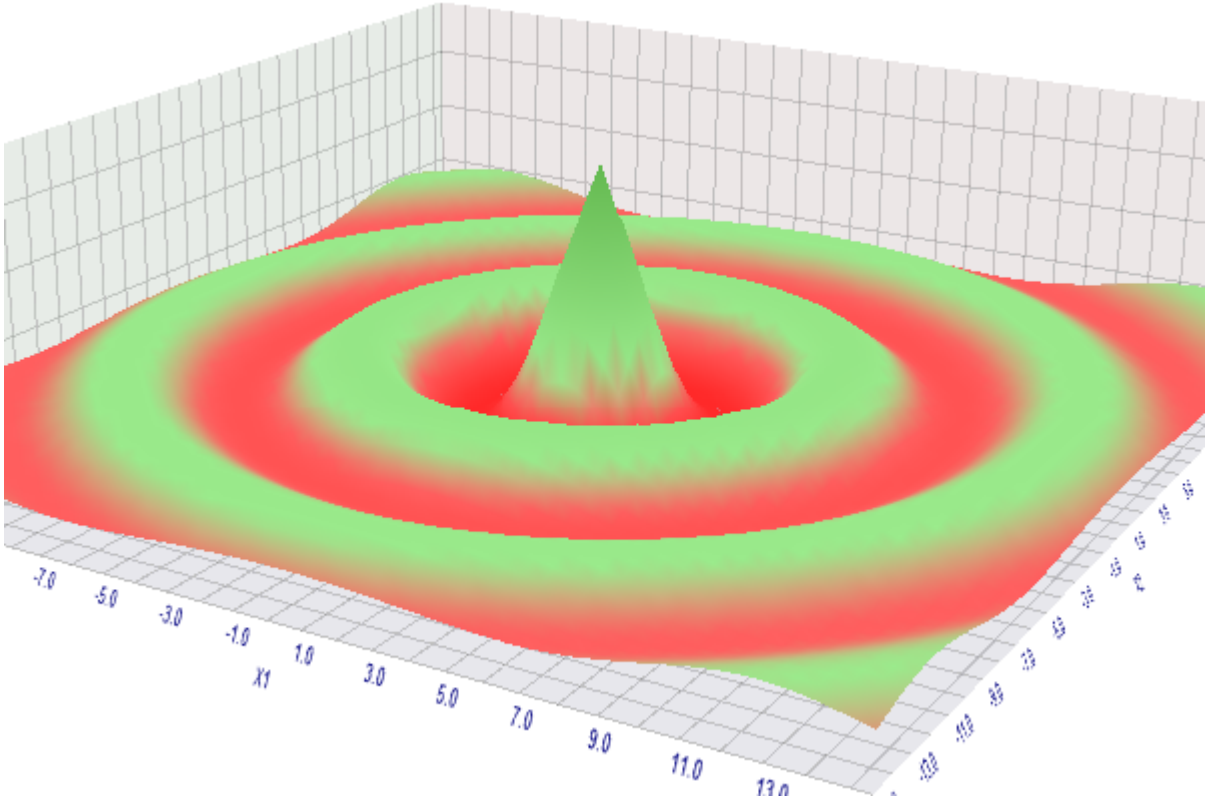
```
input double X1;
input double X2;

double OnTester()
{
   const double r = 1 + sqrt(X1 * X1 + X2 * X2);
   return sin(r) / r;
}
```

El Asesor Experto va acompañado del archivo *MathCalc.set* con los parámetros para la optimización: los argumentos X1 y X2 se iteran en los rangos [-15, +15] con un paso de 0.5.

Vamos a ejecutar la optimización y ver la solución en la tabla de optimización. La mejor pasada da el resultado correcto:

```
  X1=0.0
  X2=0.0
OnTester result 0.8414709848078965
```

En el gráfico de optimización, puede activar el modo 3D y acceder visualmente a la forma de la superficie.



Resultado de la optimización (maximización) de una función en el modo de cálculo matemático.

Al mismo tiempo, el uso del probador en el modo de cálculos matemáticos no se limita a la investigación puramente científica. Sobre su base, en particular, es posible organizar la optimización de los sistemas de trading utilizando métodos de optimización alternativos bien conocidos, como el método del «enjambre de partículas» o el «recocido simulado». Por supuesto, para ello, tendrá que cargar el historial de cotizaciones o ticks en archivos y conectarlos al Asesor Experto probado, así como emular la ejecución de operaciones, contabilizando posiciones y fondos. Este trabajo rutinario puede resultar atractivo por el hecho de que se puede personalizar libremente el proceso de optimización (a diferencia

de la «caja negra» integrada con un algoritmo genético) y controlar los recursos (principalmente la RAM).

6.5.18 Depuración y creación de perfiles

El probador de MetaTrader 5 es útil no sólo para probar la rentabilidad de las estrategias de trading, sino también para depurar programas MQL. La detección de errores se asocia principalmente a la capacidad de reproducir la situación problemática. Si sólo pudiéramos ejecutar programas MQL en línea, la depuración y el análisis de la ejecución del código fuente requerirían una cantidad de esfuerzo poco realista. Sin embargo, el probador le permite «ejecutar» programas en secciones arbitrarias del historial, cambiar la configuración de la cuenta y los símbolos de trading.

Recordemos que en el MetaEditor hay 2 comandos en el menú *Depurar*:

- *Empezar/Continuar con datos reales* (F5)
- *Empezar/Continuar con datos históricos* (Ctrl-F5)

En ambos casos, el programa se recompila rápidamente de forma especial con información de depuración adicional en el archivo ex5 y, a continuación, se ejecuta directamente en el terminal (primera opción) o en el probador (segunda opción).

Al depurar en el probador, puede utilizar tanto el modo rápido (en segundo plano) como el modo visual. Este ajuste se proporciona en el cuadro de diálogo *Ajuste* en la pestaña *Depurar/Perfil*: activar o desactivar la bandera *Usar modo visual para depurar en históricos*. El entorno y los ajustes del programa que se está depurando se pueden tomar directamente del probador (tal y como se configuraron por última vez para este programa) o en el mismo cuadro de diálogo en los campos de entrada bajo la bandera *Usar ajustes especificados* (para que funcionen, la bandera debe estar activada).

Puede preestablecer puntos de interrupción (F9) en los operadores de la parte en la que supuestamente algo empieza a funcionar mal. El probador detendrá el proceso cuando alcance la ubicación especificada en el código fuente.

Tenga en cuenta que, en el probador, el número de barras del historial que se cargan al iniciarse depende de distintos factores (como el marco temporal, el número de días en un año, etc.) y puede variar significativamente. Si es necesario, retrase la hora de inicio de la prueba.

Además de los fallos obvios que hacen que el programa se detenga o funcione explícitamente mal, hay una clase de fallos sutiles que afectan negativamente al rendimiento. Por regla general, no son tan evidentes, pero se convierten en problemas a medida que aumenta la cantidad de datos procesados, por ejemplo, en cuentas de operaciones con un historial muy largo, o en gráficos con un gran número de objetos de marcado.

Para encontrar «cuellos de botella» en términos de rendimiento, el depurador proporciona un mecanismo de perfilado del código fuente. También se puede realizar en línea o en el probador, y esto último es especialmente valioso, ya que permite comprimir significativamente el tiempo. Los comandos correspondientes también están disponibles en el menú de depuración.

- *Comenzar el perfilado con datos reales*
- *Comenzar el perfilado con datos históricos*

Para la creación de perfiles, el programa también está precompilado con ajustes especiales, así que no olvide compilar el programa de nuevo en modo normal después de que la depuración o la creación de perfiles se haya completado (especialmente si planea enviarlo a un cliente o subirlo al Mercado MQL5).

Como resultado de la creación de perfiles en MetaEditor, recibirá estadísticas de tiempo de ejecución de su código, desglosadas por líneas y funciones (métodos). Como resultado, quedará claro qué es exactamente lo que ralentiza el programa. La siguiente etapa del desarrollo suele ser el código fuente *refactoring*, es decir, su reescritura utilizando algoritmos mejorados, estructuras de datos u otros principios de la organización constructiva de módulos (componentes). Por desgracia, una parte importante del tiempo de programación se dedica a reescribir el código existente, encontrar y corregir errores.

El propio programa puede, si es necesario, averiguar su modo de funcionamiento y adaptar su comportamiento al entorno (por ejemplo, cuando se ejecuta en el probador, no intentará descargar datos de Internet, ya que esta función está desactivada, sino que los leerá de un determinado archivo).

En la fase de compilación, las versiones de depuración y de producción del programa pueden formarse de forma diferente debido a las macros del preprocesador _DEBUG y _RELEASE.

En la fase de ejecución del programa, sus modos pueden distinguirse utilizando las opciones de la función *MQLInfoInteger*.

En la tabla siguiente se resumen todas las combinaciones disponibles que afectan a las características específicas del tiempo de ejecución.

Tiempo de ejecución \ banderas	MQL_DEBUG	MQL_PROFILER	Normal (liberación)
En línea	+	+	+
Probador MQL_TESTER	+	+	+
Probador (MQL_TESTER+MQL_VISUAL_MODE)	+	-	+

La creación de perfiles en el comprobador sólo es posible sin el modo visual, por lo que deberá medir las operaciones con gráficos y objetos en línea.

No se permite la depuración durante el proceso de optimización, incluidos los manejadores especiales *OnTesterInit*, *OnTesterDeinit* y *OnTesterPass*. Si necesita comprobar su rendimiento, considere la posibilidad de llamar a su código en otras condiciones.

6.5.19 Limitaciones de las funciones del probador

Al utilizar el probador, debe tener en cuenta algunas restricciones impuestas a las funciones integradas. Algunas de las funciones de la API de MQL5 nunca se ejecutan en el probador de estrategias y otras sólo funcionan en pasadas únicas, pero no durante la optimización.

Por lo tanto, para aumentar el rendimiento al optimizar los Asesores Expertos, las funciones *Comment*, *Print* y *PrintFormat* no se ejecutan.

La excepción es el uso de estas funciones dentro del manejador *OnInit* que se hace para facilitar la búsqueda de posibles causas de errores de inicialización.

Las funciones que proporcionan interacción con el «mundo» no se ejecutan en el probador de estrategias. Entre ellas figuran *MessageBox*, *PlaySound*, *SendFTP*, *SendMail*, *SendNotification*, *WebRequest* y las funciones para trabajar con sockets.

Además, muchas funciones para trabajar con gráficos y objetos no tienen ningún efecto. En concreto, no podrá cambiar el símbolo o el período del gráfico actual llamando a *ChartSetSymbolPeriod*, enumerar todos los indicadores (incluidos los subordinados) con *ChartIndicatorGet*, trabajar con plantillas *ChartSaveTemplate*, etc.

En el probador, incluso en el modo visual, los eventos de ratón, teclado, objeto y gráfico interactivo no se generan para el manejador *OnChartEvent*.

Parte 7. Herramientas MQL5 avanzadas

En esta parte del libro, descubriremos características adicionales de la API de MQL5 en varias áreas que pueden ser necesarias a la hora de desarrollar programas para el entorno MetaTrader 5. Algunas de ellas tienen carácter de trading aplicado, como por ejemplo, los instrumentos financieros o el calendario económico integrado. Otras representan tecnologías universales que pueden ser útiles en todas partes: funciones de red, bases de datos, criptografía, etc.

Además, consideraremos la posibilidad de ampliar los programas MQL utilizando recursos que son archivos de tipo arbitrario que pueden incrustarse en el código y contienen multimedia, configuraciones «pesadas» de programas externos (por ejemplo, configuraciones de redes neuronales o modelos de aprendizaje automático ya hechos) u otros programas MQL (indicadores) de forma compilada.

Se dedicarán un par de capítulos al desarrollo modular de programas MQL. En este contexto, consideraremos un tipo especial de programa: las bibliotecas, que pueden conectarse a otros programas MQL para proporcionar conjuntos de API específicas en forma cerrada, pero que no pueden utilizarse de forma independiente. También exploraremos las posibilidades de organizar el proceso de desarrollo de complejos de software y de combinar programas lógicamente interrelacionados en proyectos.

Por último, presentaremos la integración con otros entornos de software, en particular, con Python.

El libro no aborda algunos temas altamente especializados que pueden ser de interés para usuarios avanzados, como las capacidades de hardware para computación paralela utilizando OpenCL, así como gráficos 2D y 3D basados en DirectX. Le sugerimos que se familiarice con estas tecnologías utilizando la documentación y los artículos del sitio web mql5.com.

Programación en MQL5 para operadores de trading: códigos fuente del libro. Parte 7

Los ejemplos del libro también están disponibles en el proyecto público \MQL5\Shared Projects\MQL5Book

7.1 Recursos

El funcionamiento de los programas MQL puede requerir muchos recursos auxiliares, que son arrays de datos de aplicación o archivos de varios tipos, incluyendo imágenes, sonidos y fuentes. El entorno de desarrollo MQL permite incluir todos estos recursos en el archivo ejecutable en la fase de compilación. Esto elimina la necesidad de su transferencia e instalación paralelas junto con el programa principal y lo convierte en un producto completo autosuficiente y cómodo para el usuario final.

En este capítulo aprenderemos a describir diferentes tipos de recursos y funciones integradas para operaciones posteriores con recursos conectados.

Las imágenes rasterizadas, representadas como arrays de puntos (píxeles) en el ampliamente reconocido formato BMP, ocupan una posición única entre los recursos. La API de MQL5 permite la creación, manipulación y visualización dinámica de estos recursos gráficos en los gráficos.

Hemos hablado ya con anterioridad de los objetos gráficos y, en concreto, de los objetos de tipo OBJ_BITMAP y OBJ_BITMAP_LABEL que resultan útiles para diseñar interfaces de usuario. Para estos objetos, existe la propiedad OBJPROP_BMPFILE que especifica la imagen como archivo o recurso. Anteriormente hemos considerado únicamente los ejemplos con archivos; ahora aprenderemos a trabajar con imágenes de recursos.

7.1.1 Descripción de recursos mediante la directiva #resource

Para incluir un archivo de recursos en la versión compilada del programa, utilice la directiva *#resource* en el código fuente. La directiva tiene diferentes formas según el tipo de archivo. En cualquier caso, la directiva contiene la palabra clave *#resource* seguida de una cadena constante.

```
#resource "path_file_name"
```

El comando *#resource* indica al compilador que incluya (en formato binario *ex5*) un archivo con el nombre y, opcionalmente, la ubicación especificados (en el momento de la compilación) en el programa ejecutable que se está generando. La ruta es opcional: si la cadena solo contiene el nombre del archivo, se busca en el directorio junto al código fuente compilado. Si hay una ruta en la cadena, se aplican las reglas descritas a continuación.

El compilador busca el recurso en la ruta especificada en la siguiente secuencia:

- Si la ruta está precedida por una barra invertida '\\' (debe ser doble, ya que una sola barra invertida es un carácter de control; en concreto, '\' se utiliza para nuevas líneas '\r', '\n' y tabulaciones '\t'), entonces el recurso se busca a partir de la carpeta MQL5 dentro del directorio de datos del terminal.
- Si no hay barra invertida, el recurso se busca en relación con la ubicación del archivo fuente en el que está registrado este recurso.

Tenga en cuenta que en las cadenas constantes con rutas de recursos, debe utilizar barras invertidas dobles como separadores. Aquí no se admiten las barras diagonales simples, a diferencia de las rutas en el sistema de archivos.

Por ejemplo:

```
#resource "\\Images\\euro.bmp" // euro.bmp is in /MQL5/Images/
#resource "picture.bmp"        // picture.bmp is in the same directory,
                               // where the source file is (mq5 or mqh)
#resource "Resource\\map.bmp"  // map.bmp is in the Resource subfolder of the directo
                               // where the source file is (mq5 or mqh)
```

Si el recurso se declara con una ruta relativa en el archivo de encabezado mqh, la ruta se considera relativa a este archivo mqh y no al archivo mq5 del programa que se está compilando.

Las subcadenas «..\\» y «:\» no están permitidas en la ruta del recurso.

Utilizando unas pocas directivas, puede, por ejemplo, poner todas las imágenes y sonidos necesarios directamente en el archivo ex5. Entonces, para ejecutar un programa de este tipo en otro terminal, no es necesario transferirlos por separado. En las siguientes secciones estudiaremos las formas programáticas de acceder a los recursos desde MQL5.

La longitud de la cadena constante «path_file_name» no debe superar los 63 caracteres. El tamaño del archivo de recursos no puede ser superior a 128 Mb. Los archivos de recursos se comprimen automáticamente antes de incluirlos en el ejecutable.

Una vez declarado el recurso mediante la directiva *#resource*, puede utilizarse en cualquier parte del programa. El nombre del recurso se convierte en la cadena constante especificada en la directiva sin una barra al principio (si existe), y debe añadirse un signo especial del recurso (dos dos puntos, «::») antes del contenido de la cadena.

A continuación presentamos ejemplos de recursos, con sus nombres en los comentarios.

```
#resource "\\Images\\euro.bmp"          // resource name - ::Images\\euro.bmp
#resource "picture.bmp"                 // resource name - ::picture.bmp
#resource "Resource\\map.bmp"           // resource name - ::Resource\\map.bmp
#resource "\\Files\\Pictures\\good.bmp" // resource name - ::Files\\Pictures\\good.bm
#resource "\\Files\\demo.wav";          // resource name - ::Files\\demo.wav"
#resource "\\Sounds\\thrill.wav";       // resource name - ::Sounds\\thrill.wav"
```

Más adelante en el código MQL se puede hacer referencia a estos recursos de la siguiente manera (aquí, solo conocemos ya las funciones *ObjectSetString* y *PlaySound*, pero hay otras opciones como *RecursoReadImage*, que se describirán en las secciones siguientes).

```
ObjectSetString(0, bitmap_name, OBJPROP_BMPFILE, 0, "::Images\\euro.bmp");
...
ObjectSetString(0, my_bitmap, OBJPROP_BMPFILE, 0, "::picture.bmp");
...
ObjectSetString(0, bitmap_label, OBJPROP_BMPFILE, 0, "::Resource\\map.bmp");
ObjectSetString(0, bitmap_label, OBJPROP_BMPFILE, 1, "::Files\\Pictures\\good.bmp");
...
PlaySound("::Files\\demo.wav");
...
PlaySound("::Sounds\\thrill.wav");
```

Debe tenerse en cuenta que al establecer una imagen de un recurso en los objetos OBJ_BITMAP y OBJ_BITMAP_LABEL, el valor de la propiedad OBJPROP_BMPFILE no puede modificarse manualmente (en el cuadro de diálogo de propiedades del objeto).

Tenga en cuenta que los archivos wav se establecen por defecto para la función *PlaySound* en relación con la carpeta *Sounds* (o sus subcarpetas) situada en el directorio de datos del terminal. Al mismo tiempo, los recursos (incluidos los de sonido), si se describen con una barra inicial en la ruta, se buscan dentro del directorio MQL5. Por lo tanto, en el ejemplo anterior, la cadena «\\Sounds\\thrill.wav» se refiere al archivo *MQL5/Sounds/thrill.wav* y no a *Sounds/thrill.wav* relativo al directorio de datos (sí existe el directorio *Sounds* con sonidos estándar del terminal).

La sencilla sintaxis de la directiva *#resource* comentada anteriormente solo permite describir recursos de imagen (formato BMP) y recursos de sonido (formato WAV). Si se intenta describir un archivo de otro tipo como recurso se producirá un error de «tipo de recurso desconocido».

Como resultado del procesamiento de la directiva *#resource*, los archivos se incrustan en el programa binario ejecutable y son accesibles mediante el nombre del recurso. Además, debe prestar atención a una propiedad especial de tales recursos, que es su disponibilidad pública desde otros programas (más al respecto en la siguiente sección).

MQL5 también admite otra forma de incrustar un archivo en un programa: en forma de una variable de recursos. Este método utiliza la sintaxis ampliada de la directiva *#resource* y permite conectar no solo archivos BMP o WAV, sino también otros, por ejemplo, texto o un array de estructuras.

Analizaremos un ejemplo práctico de conexión de recursos en un par de secciones.

7.1.2 Uso compartido de recursos de distintos programas MQL

El nombre del recurso es único en todo el terminal. Más adelante aprenderemos a crear recursos no en la fase de compilación (mediante la directiva *#resource*), sino de forma dinámica, utilizando la función *ResourceCreate*. En cualquier caso, el recurso se declara en el contexto del programa que lo crea, de

modo que la unicidad del nombre completo se proporciona automáticamente mediante la vinculación al sistema de archivos (ruta y nombre de un archivo específico *ex5*).

Además de contener y utilizar recursos, un programa MQL también puede acceder a los recursos de otro programa compilado (archivo ex5). Esto es posible siempre que el programa que utiliza el recurso conozca la ruta de ubicación y el nombre de otro programa que contenga el recurso necesario, así como el nombre de este recurso.

Así, el terminal proporciona una importante propiedad de los recursos que es su uso compartido: los recursos de un archivo ex5 pueden utilizarse en muchos otros programas.

Para utilizar un recurso de un archivo ex5 de terceros, debe especificarse con la forma «ruta_nombre_archivo.ex5::nombre_recurso». Por ejemplo, supongamos que el script *DrawingScript.mq5* hace referencia a un recurso de imagen especificado en el archivo *triangle.bmp*:

```
#resource "\\Files\\triangle.bmp"
```

Entonces su nombre para el uso en el script real se verá como «::Files\\triangle.bmp».

Para utilizar el mismo recurso desde otro programa, por ejemplo, un Asesor Experto, el nombre del recurso debe ir precedido de la ruta del archivo de script ex5 relativa a la carpeta MQL5 en el directorio de datos del terminal, así como el nombre del propio script (en la forma compilada, *DrawingScript.ex5*). Dejemos que el script esté en la carpeta estándar *MQL5/Scripts/*. En este caso, se debe acceder a la imagen utilizando la cadena «\\Scripts\DrawingScript.ex5::Files\\triangle.bmp». La extensión «.ex5» es opcional.

Si, al acceder al recurso de otro archivo ex5, no se especifica la ruta a este archivo, se busca dicho archivo en la misma carpeta en la que se encuentra el programa que solicita el recurso. Por ejemplo, si suponemos que el mismo Asesor Experto se encuentra en la carpeta estándar *MQL5/Experts/* y consulta un recurso sin especificar la ruta (por ejemplo, «DrawingScript.ex5::Files\\triangle.bmp»), entonces *DrawingScript.ex5* se buscará en la carpeta *MQL5/Experts/*.

Debido al uso compartido de los recursos, su creación y actualización dinámicas pueden utilizarse para intercambiar datos entre programas MQL. Esto ocurre directamente en la memoria y, por lo tanto, es una buena alternativa a los archivos o variables globales.

Tenga en cuenta que, para cargar un recurso desde un programa MQL, no es necesario ejecutarlo: para leer recursos basta con tener un archivo ex5 con recursos.

Una excepción importante en la que no es posible compartir informes es cuando un recurso se describe en forma de variable de recursos.

7.1.3 Variables de recursos

La directiva *#resource* tiene una forma especial con la que se pueden declarar archivos externos como variables de recursos y acceder a ellos dentro del programa como variables normales del tipo correspondiente. El formato de la declaración es:

```
#resource "path_file_name" as resource_variable_type resource_variable_name
```

He aquí algunos ejemplos de declaraciones:

```
#resource "data.bin" as int Data[]            //array of int type with data from the f
#resource "rates.dat" as MqlRates Rates[]     // array of MqlRates structures from the
#resource "data.txt" as string Message        // line with the contents of the file da
#resource "image.bmp" as bitmap Bitmap1[]     // one-dimensional array with image pixe
                                              // from file image.bmp
#resource "image.bmp" as bitmap Bitmap2[][]   // two-dimensional array with the same i
```

Demos algunas explicaciones. Las variables de recursos son constantes (no se pueden modificar en el código MQL5). Por ejemplo, para editar imágenes antes de mostrarlas en pantalla, debe crear copias de variables de array de recursos.

En el caso de los archivos de texto (recursos de tipo *string*), la codificación se determina automáticamente por la presencia de un encabezado de BOM. Si no hay BOM, la codificación viene determinada por el contenido del archivo. Se admiten las codificaciones ANSI, UTF-8 y UTF-16. Al leer datos de archivos, todas las cadenas se convierten a Unicode.

El uso de variables de cadena de recursos puede facilitar enormemente la escritura de programas basados no solo en MQL5 puro, sino también en tecnologías adicionales. Por ejemplo, puede escribir código OpenCL (que está admitido en MQL5 como una extensión) en un archivo separado y, a continuación, incluirlo como una cadena en los recursos de un programa MQL. En el ejemplo de Asesor Experto hemos utilizado ya cadenas de recursos para incluir plantillas HTML.

Para las imágenes se ha introducido un tipo especial *bitmap*; este tipo tiene varias características.

El tipo *bitmap* describe un único punto o píxel en una imagen y se representa mediante un entero sin signo de 4 bytes (*uint*). El píxel contiene 4 bytes que corresponden a los componentes de color en formato ARGB o XRGB (una letra = un byte), donde R es rojo, G es verde, B es azul, A es transparencia (canal alfa), X es un byte ignorado (sin transparencia). La transparencia puede utilizarse para obtener diversos efectos al superponer imágenes en un gráfico y unas sobre otras.

Estudiaremos la definición de los formatos ARGB y XRGB en la sección dedicada a la creación dinámica de recursos gráficos (véase *ResourceCreate*). Por ejemplo, para ARGB, el número hexadecimal 0xFFFF0000 especifica un píxel totalmente opaco (el byte más alto es 0xFF) de color rojo (el byte siguiente también es 0xFF) y los bytes siguientes para los componentes verde y azul son cero.

Es importante señalar que la codificación del color del píxel es diferente de la representación en bytes del tipo *color*. Recordemos que el valor del tipo *color* puede escribirse en forma hexadecimal de la siguiente manera: 0x00BBGGRR, donde BB, GG, RR son los componentes azul, verde y rojo, respectivamente (en cada byte, el valor 255 da la intensidad máxima del componente). Con un registro similar de un píxel, hay un orden inverso de bytes: 0xAARRGGBB. La transparencia total se obtiene cuando el byte alto (aquí denotado AA) es 0 y el valor 255 es un color sólido. La función *ColorToARGB* se puede utilizar para convertir *color* a ARGB.

Los archivos BMP pueden tener varios métodos de codificación (si los crea o edita en cualquier editor, compruebe este tema en la documentación de este programa). Los recursos MQL5 no admiten todos los métodos de codificación existentes. Puede comprobar si un archivo concreto es compatible utilizando la función *ResourceCreate*. Si se especifica en la directiva un archivo de formato BMP no compatible, se producirá un error de compilación.

Al cargar un archivo con codificación de color de 24 bits, todos los píxeles del componente del canal alfa se establecen en 255 (opaco). Cuando se carga un archivo con una codificación de color de 32 bits sin canal alfa, tampoco implica transparencia, es decir, para todos los píxeles de la imagen, el componente del canal alfa se establece en 255. Cuando se carga un archivo codificado en color de 32 bits con un canal alfa, no se manipula ningún píxel.

Las imágenes pueden describirse mediante arrays unidimensionales y bidimensionales. Esto solo afecta al método de direccionamiento, mientras que la cantidad de memoria ocupada será la misma. En ambos casos, los tamaños de los arrays se establecen automáticamente en función de los datos del archivo BMP. El tamaño de un array unidimensional será igual al producto de la altura y la anchura de la imagen *(height * width)*, y un array bidimensional obtendrá dimensiones separadas *[height][width]*: el primer índice es el número de línea; el segundo, un punto de la línea.

> ¡Atención! Al declarar un recurso vinculado a una variable de recurso, la única forma de acceder al recurso es a través de esa variable, y la forma estándar de leer a través del nombre «::nombre_del_recurso» (o más generalmente «nombre_del_archivo_de_la_ruta.ex5::nombre_del_recurso») ya no funciona. Esto también significa que dichos recursos no pueden utilizarse como recursos compartidos de otros programas.

Vamos a considerar dos indicadores como ejemplo; ambos carecen de búfer. Este tipo de programa MQL fue elegido solo por razones de conveniencia, ya que se puede aplicar al gráfico sin conflicto además de otros indicadores, mientras que un Asesor Experto requeriría un gráfico sin otro Asesor Experto. Además, permanecen en el gráfico y están disponibles para posteriores cambios de configuración, a diferencia de los scripts.

El indicador *BmpOwner.mq5* contiene una descripción de tres recursos:

- Una imagen «search1.bmp» con una simple directiva *#resource* accesible desde otros programas.
- Una imagen «search2.bmp» como variable de array de recursos de tipo *bitmap*, inaccesible desde el exterior
- Un archivo de texto «message.txt» como cadena de recursos para mostrar una advertencia al usuario.

Ambas imágenes no se utilizan en modo alguno en este indicador. La línea de advertencia es necesaria en la función *OnInit* para llamar a *Alert*, ya que el indicador no está pensado para un uso independiente, sino que solo actúa como proveedor de un recurso de imagen.

> Si la variable de recurso no se utiliza en el código fuente, es posible que el compilador no incluya el recurso en absoluto en el código binario del programa, pero esto no se aplica a las imágenes.

```
#resource "search1.bmp"
#resource "search2.bmp" as bitmap image[]
#resource "message.txt" as string Message
```

Los tres archivos se encuentran en el mismo directorio en el que se encuentra el código fuente del indicador: *MQL5/Indicators/MQL5Book/p7/*.

Si el usuario intenta ejecutar el indicador, éste muestra una advertencia e inmediatamente deja de funcionar. La advertencia está contenida en la variable de cadena del recurso Mensaje.

```
int OnInit()
{
   Alert(Message); // equivalent to the following line of the code
   // Alert("This indicator is not intended to run, it holds a bitmap resource");

   // remove the indicator explicitly, because otherwise it remains "hanging" on the
   ChartIndicatorDelete(0, 0, MQLInfoString(MQL_PROGRAM_NAME));
   return INIT_FAILED;
}
```

En el segundo indicador *BmpUser.mq5*, intentaremos utilizar los recursos externos especificados en las variables de entrada *ResourceOff* y *ResourceOn*, para mostrarlos en el objeto OBJ_BITMAP_LABEL.

```
input string ResourceOff = "BmpOwner.ex5::search1.bmp";
input string ResourceOn = "BmpOwner.ex5::search2.bmp";
```

De manera predeterminada, el estado del objeto es desactivado/liberado («Off»), y la imagen para él se toma del indicador anterior «BmpOwner.ex5::search1.bmp». Esta ruta y el nombre del recurso son similares a la notación completa «\\Indicators\MQL5Book\p7\BmpOwner.ex5::search1.bmp». La forma abreviada es aceptable en este caso, dado que los indicadores están situados uno al lado del otro. Si posteriormente abre el cuadro de diálogo de propiedades del objeto, verá la notación completa en los campos *Bitmap file (On/Off)*.

Para el estado pulsado, en *ResourceOn* debemos leer el recurso «BmpOwner.ex5::search2.bmp» (veamos qué ocurre).

En otras variables de entrada, puede seleccionar la esquina del gráfico, respecto a la cual se establece el posicionamiento de la imagen, y las sangrías horizontal y vertical.

```
input int X = 25;
input int Y = 25;
input ENUM_BASE_CORNER Corner = CORNER_RIGHT_LOWER;
```

La creación del objeto OBJ_BITMAP_LABEL y la configuración de sus propiedades, incluido el nombre del recurso como imagen para OBJPROP_BMPFILE, se realizan en *OnInit*.

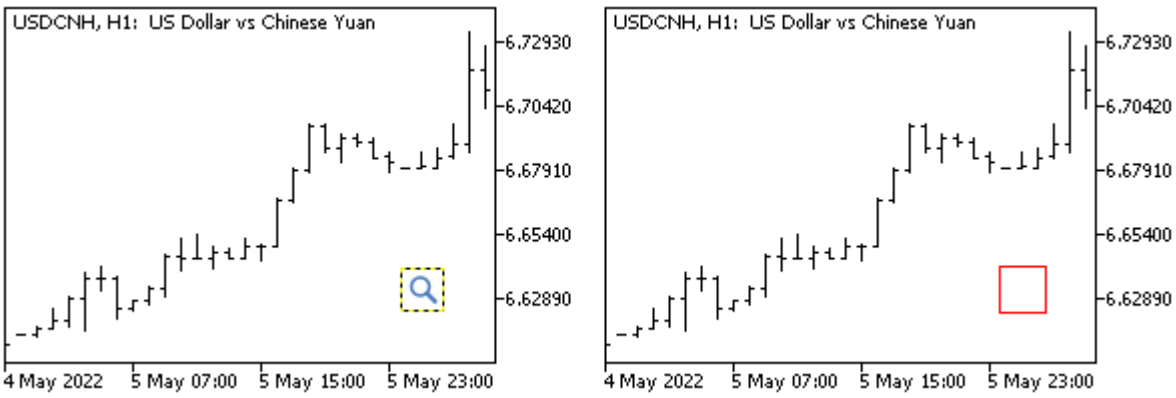
```
const string Prefix = "BMP_";
const ENUM_ANCHOR_POINT Anchors[] =
{
   ANCHOR_LEFT_UPPER,
   ANCHOR_LEFT_LOWER,
   ANCHOR_RIGHT_LOWER,
   ANCHOR_RIGHT_UPPER
};

void OnInit()
{
   const string name = Prefix + "search";
   ObjectCreate(0, name, OBJ_BITMAP_LABEL, 0, 0, 0);

   ObjectSetString(0, name, OBJPROP_BMPFILE, 0, ResourceOn);
   ObjectSetString(0, name, OBJPROP_BMPFILE, 1, ResourceOff);
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, X);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, Y);
   ObjectSetInteger(0, name, OBJPROP_CORNER, Corner);
   ObjectSetInteger(0, name, OBJPROP_ANCHOR, Anchors[(int)Corner]);
}
```

Recuerde que al especificar imágenes en OBJPROP_BMPFILE, el estado pulsado se indica mediante el modificador 0, y el estado liberado (no pulsado) (predeterminado) se indica mediante el modificador 1, lo cual es hasta cierto punto inesperado.

El manejador *OnDeinit* borra el objeto al descargar el indicador.

```
void OnDeinit(const int)
{
   ObjectsDeleteAll(0, Prefix);
}
```

Vamos a compilar ambos indicadores y ejecutar *BmpUser.ex5* con la configuración por defecto. La imagen del archivo gráfico *search1.bmp* debería aparecer en el gráfico (véase a la izquierda).



Visualización normal (izquierda) e incorrecta (derecha) de los recursos gráficos de un objeto en un gráfico

Si hace clic en la imagen, es decir, la cambia al estado pulsado, el programa intentará acceder al recurso «BmpOwner.ex5::search2.bmp» (que no está disponible debido al array de recursos *bitmap* adjunto). Como resultado, veremos un cuadrado rojo, que indica un objeto vacío sin imagen (véase arriba, a la derecha). Siempre se producirá una situación similar si el parámetro de entrada especifica

una ruta o un nombre con un recurso a sabiendas inexistente o no compartido. Puede crear su propio programa, describir en él un recurso que haga referencia a algún archivo bmp existente y, a continuación, especificar en el indicador los parámetros de entrada *BmpUser*. En este caso, el indicador podrá mostrar la imagen en el gráfico.

7.1.4 Conectar indicadores personalizados como recursos

Para su funcionamiento, los programas MQL pueden requerir uno o varios indicadores personalizados. Todos ellos pueden incluirse como recursos en el ejecutable ex5, lo que facilita su distribución e instalación.

La directiva *#resource* con la descripción del indicador anidado tiene el siguiente formato:

```
#resource "path_indicator_name.ex5"
```

Las reglas de configuración y búsqueda del archivo especificado son las mismas que para todos los recursos en general.

Ya hemos utilizado esta función en el ejemplo de Asesor Experto, en la versión final de *UnityMartingale.mq5*.

```
#resource "\\Indicators\\MQL5Book\\p6\\UnityPercentEvent.ex5"
```

En ese Asesor Experto, en lugar del nombre del indicador, este recurso se pasó a la función *iCustom*: «::Indicators\\MQL5Book\\p6\\UnityPercentEvent.ex5».

El caso en que un indicador personalizado en la función *OnInit* crea una o más instancias de sí mismo requiere una consideración aparte (si esta solución técnica en sí parece extraña, daremos un ejemplo práctico después de los ejemplos introductorios).

Como sabemos, para utilizar un recurso desde un programa MQL, debe especificarse de la siguiente forma: ruta_nombre_archivo.ex5::nombre_del_recurso. Por ejemplo, si el indicador *EmbeddedIndicator.ex5* se incluye como recurso en otro indicador *MainIndicator.mq5* (más concretamente, en su imagen binaria *MainIndicator.ex5*), entonces el nombre especificado al llamarse a sí mismo a través de *iCustom* ya no puede ser corto, sin una ruta, y la ruta debe incluir la ubicación del indicador «padre» dentro de la carpeta MQL5. De lo contrario, el sistema no podrá encontrar el indicador anidado.

De hecho, en circunstancias normales, un indicador puede llamarse a sí mismo utilizando, por ejemplo, el operador *iCustom(_Symbol, _Period, myself,...)*, donde *myself* es una cadena igual a *MQLInfoString(MQL_PROGRAM_NAME)* o al nombre que se asignó previamente a la propiedad INDICATOR_SHORTNAME en el código. Pero cuando el indicador se encuentra dentro de otro programa MQL como recurso, el nombre ya no hace referencia al archivo correspondiente porque el archivo que servía de prototipo para el recurso se quedó en el ordenador donde se realizó la compilación, y en el ordenador del usuario sólo existe el archivo *MainIndicator.ex5*. Esto requerirá un cierto análisis del entorno del programa al iniciar el programa.

Veámoslo en la práctica.

Para empezar, vamos a crear un indicador *NonEmbeddedIndicator.mq5*. Es importante señalar que se encuentra en la carpeta *MQL5/Indicators/MQL5Book/p7/SubFolder/*, es decir, en una *SubFolder* relativa a la carpeta *p7* asignada para todos los indicadores de esta parte del libro. Esto se hace intencionadamente para emular una situación en la que el archivo compilado no está presente en el ordenador del usuario. Ahora veremos cómo funciona (o mejor dicho, demuestra el problema).

El indicador tiene un único parámetro de entrada *Reference*. Su propósito es contar el número de copias de sí mismo: cuando se crea por primera vez, el parámetro es igual a 0, y el indicador creará su propia copia con el valor de parámetro de 1. La segunda copia, después de «ver» el valor 1, ya no creará otra copia (de lo contrario nos quedaríamos rápidamente sin recursos sin la condición límite para detener la reproducción).

```
input int Reference = 0;
```

La variable *handle* está reservada para el manejador del indicador de copia.

```
int handle = 0;
```

En el manejador *OnInit*, para mayor claridad, primero mostramos el nombre y la ruta del programa MQL.

```
int OnInit()
{
   const string name = MQLInfoString(MQL_PROGRAM_NAME);
   const string path = MQLInfoString(MQL_PROGRAM_PATH);
   Print(Reference);
   Print("Name: " + name);
   Print("Full path: " + path);
   ...
```

A continuación viene el código adecuado para el auto-lanzamiento de un indicador independiente (existente en forma del conocido archivo *NonEmbeddedIndicator.ex5*).

```
   if(Reference == 0)
   {
      handle = iCustom(_Symbol, _Period, name, 1);
      if(handle == INVALID_HANDLE)
      {
         return INIT_FAILED;
      }
   }
   Print("Success");
   return INIT_SUCCEEDED;
}
```

Podríamos colocar con éxito un indicador de este tipo en el gráfico y recibir entradas del tipo siguiente en el registro (usted tendrá sus propias rutas del sistema de archivos):

```
0
Name: NonEmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\SubFolder\NonEmbedded
Success
1
Name: NonEmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\SubFolder\NonEmbedded
Success
```

La copia se ha iniciado correctamente con sólo utilizar el nombre «NonEmbeddedIndicator».

Dejemos este indicador por ahora y creemos un segundo, *FaultyIndicator.mq5*, en el que incluiremos el primer indicador como recurso (preste atención a la especificación de *subfolder* en la ruta relativa del

recurso; esto es necesario porque el indicador *FaultyIndicator.mq5* se encuentra en la carpeta de un nivel superior: *MQL5/Indicators/MQL5Book/p7/*).

```
// FaultyIndicator.mq5
#resource "SubFolder\\NonEmbeddedIndicator.ex5"

int handle;

int OnInit()
{
   handle = iCustom(_Symbol, _Period, "::SubFolder\\NonEmbeddedIndicator.ex5");
   if(handle == INVALID_HANDLE)
   {
      return INIT_FAILED;
   }
   return INIT_SUCCEEDED;
}
```

Si intenta ejecutar el archivo compilado *FaultyIndicator.ex5* se producirá un error:

```
0
Name: NonEmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\FaultyIndicator.ex5 »
» ::SubFolder\NonEmbeddedIndicator.ex5
cannot load custom indicator 'NonEmbeddedIndicator' [4802]
```

Cuando se lanza una copia de un indicador anidado, se busca en la carpeta del indicador principal, en la que se describe el recurso. Pero no hay ningún archivo *NonEmbeddedIndicator.ex5* porque el recurso requerido está dentro de *FaultyIndicator.ex5*.

Para resolver el problema, modificamos *NonEmbeddedIndicator.mq5*. En primer lugar, démosle otro nombre más apropiado, *EmbeddedIndicator.mq5*. En el código fuente tenemos que añadir una función de ayuda *GetMQL5Path*, que puede aislar la parte relativa dentro de la carpeta MQL5 de la ruta general del programa MQL lanzado (esta parte también contendrá el nombre del recurso si el indicador se lanza desde un recurso).

```
// EmbeddedIndicator.mq5
string GetMQL5Path()
{
   static const string MQL5 = "\\MQL5\\";
   static const int length = StringLen(MQL5) - 1;
   static const string path = MQLInfoString(MQL_PROGRAM_PATH);
   const int start = StringFind(path, MQL5);
   if(start != -1)
   {
      return StringSubstr(path, start + length);
   }
   return path;
}
```

Teniendo en cuenta la nueva función, cambiaremos la llamada a *iCustom* en el manejador *OnInit*.

```
int OnInit()
{
   ...
   const string location = GetMQL5Path();
   Print("Location in MQL5:" + location);
   if(Reference == 0)
   {
      handle = iCustom(_Symbol, _Period, location, 1);
      if(handle == INVALID_HANDLE)
      {
         return INIT_FAILED;
      }
   }
   return INIT_SUCCEEDED;
}
```

Asegurémonos de que esta edición no rompe el lanzamiento del indicador. La superposición en un gráfico hace que aparezcan las líneas esperadas en el registro:

```
0
Name: EmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\SubFolder\EmbeddedInd
Location in MQL5:\Indicators\MQL5Book\p7\SubFolder\EmbeddedIndicator.ex5
Success
1
Name: EmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\SubFolder\EmbeddedInd
Location in MQL5:\Indicators\MQL5Book\p7\SubFolder\EmbeddedIndicator.ex5
Success
```

Aquí añadimos la salida de depuración de la ruta relativa que recibió la función *GetMQL5Path*. Esta línea se utiliza ahora en *iCustom*, y funciona de este modo: se ha creado una copia.

Ahora vamos a incrustar este indicador como un recurso en otro indicador en la carpeta *MQL5Book/p7* con el nombre *MainIndicator.mq5*. *MainIndicator.mq5* es completamente idéntico a *FaultyIndicator.mq5* excepto por el recurso conectado.

```
                                                              // MainIndicator.mq5
#resource "SubFolder\\EmbeddedIndicator.ex5"
...
int OnInit()
{
   handle = iCustom(_Symbol, _Period, "::SubFolder\\EmbeddedIndicator.ex5");
   ...
}
```

Vamos a compilarlo y ejecutarlo. Las entradas aparecen en el registro con una nueva ruta relativa que incluye el recurso anidado.

```
0
Name: EmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\MainIndicator.ex5 »
» ::SubFolder\EmbeddedIndicator.ex5
Location in MQL5:\Indicators\MQL5Book\p7\MainIndicator.ex5::SubFolder\EmbeddedIndicat
Success
1
Name: EmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\MainIndicator.ex5 »
» ::SubFolder\EmbeddedIndicator.ex5
Location in MQL5:\Indicators\MQL5Book\p7\MainIndicator.ex5::SubFolder\EmbeddedIndicat
Success
```

Como podemos ver, esta vez el indicador anidado ha creado correctamente una copia de sí mismo, ya que ha utilizado un nombre cualificado con una ruta relativa y un nombre de recurso «\Indicators\MQL5Book\p7\MainIndicator.ex5::SubFolder\EmbeddedIndicator.ex5».

> Durante múltiples experimentos con el lanzamiento de este indicador, tenga en cuenta que las copias anidadas no se descargan inmediatamente del gráfico después de que se elimine el indicador principal. Por lo tanto, los reinicios deben realizarse sólo después de haber esperado a que se produzca la descarga: de lo contrario, se reutilizarán las copias que aún se estén ejecutando, y las líneas de inicialización anteriores no aparecerán en el registro. Para controlar la descarga, se ha añadido una impresión del valor *Reference* al manejador *OnDeinit*.

Prometimos demostrar que crear una copia del indicador no es algo extraordinario. Como demostración aplicada de esta técnica, utilizamos el indicador *DeltaPrice.mq5*, que calcula la diferencia en incrementos de precio de una orden determinada. Orden 0 significa sin diferenciación (solo para comprobar la serie temporal original), 1 significa diferenciación simple, 2 significa diferenciación doble, y así sucesivamente.

El orden se especifica en el parámetro de entrada *Differentiating*.

```
input int Differencing = 1;
```

Las series de diferencias se mostrarán en un único búfer en la subventana.

```
#property indicator_separate_window
#property indicator_buffers 1
#property indicator_plots   1

#property indicator_type1 DRAW_LINE
#property indicator_color1 clrDodgerBlue
#property indicator_width1 2
#property indicator_style1 STYLE_SOLID

double Buffer[];
```

En el manejador *OnInit*, configuramos el búfer y creamos el mismo indicador, pasando el valor reducido en 1 en el parámetro de entrada.

```
#include <MQL5Book/AppliedTo.mqh> // APPLIED_TO_STR macro

int handle = 0;

int OnInit()
{
   const string label = "DeltaPrice (" + (string)Differencing + "/"
      + APPLIED_TO_STR() + ")";
   IndicatorSetString(INDICATOR_SHORTNAME, label);
   PlotIndexSetString(0, PLOT_LABEL, label);

   SetIndexBuffer(0, Buffer);
   if(Differencing > 1)
   {
      handle = iCustom(_Symbol, _Period, GetMQL5Path(), Differencing - 1);
      if(handle == INVALID_HANDLE)
      {
         return INIT_FAILED;
      }
   }
   return INIT_SUCCEEDED;
}
```

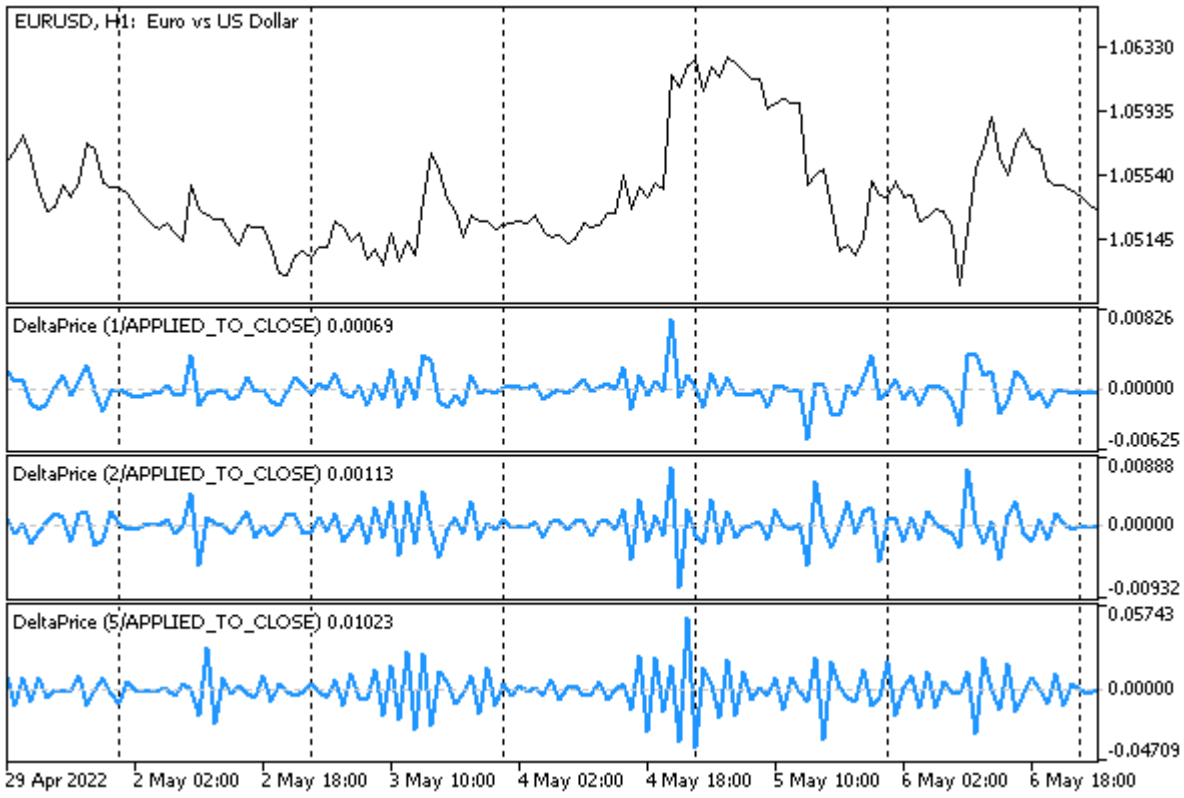
Para evitar posibles problemas con la incrustación del indicador como recurso, utilizamos la función ya probada *GetMQL5Path*.

En la función *OnCalculate* realizamos la operación de restar valores vecinos de la serie temporal. Cuando *Differentiating* es igual a 1, los operandos son elementos del array *price*. Con un valor mayor de *Differentiating*, leemos el búfer de la copia del indicador creada para la orden anterior.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
   for(int i = fmax(prev_calculated - 1, 1); i < rates_total; ++i)
   {
      if(Differencing > 1)
      {
         static double value[2];
         CopyBuffer(handle, 0, rates_total - i - 1, 2, value);
         Buffer[i] = value[1] - value[0];
      }
      else if(Differencing == 1)
      {
         Buffer[i] = price[i] - price[i - 1];
      }
      else
      {
         Buffer[i] = price[i];
      }
   }
   return rates_total;
}
```

El tipo inicial de precio diferenciado se establece en el cuadro de diálogo de configuración del indicador, en la lista desplegable *Apply to*. De manera predeterminada, se trata del precio *Close*.

Así es como se ven varias copias del indicador en el gráfico con diferentes órdenes de diferenciación.



Diferencia en los precios de cierre de distintas órdenes

7.1.5 Creación de recursos dinámicos: ResourceCreate

Las directivas *#resource* incrustan recursos en el programa en la fase de compilación, por lo que pueden denominarse estáticas. Sin embargo, a menudo es necesario generar recursos (crear unos completamente nuevos o modificar los existentes) en la fase de ejecución del programa. Para estos propósitos, MQL5 ofrece la función *ResourceCreate*. Los recursos creados con ayuda de esta función se denominarán dinámicos.

La función tiene dos formas: la primera permite cargar imágenes y sonidos desde archivos, y la segunda está diseñada para crear imágenes de mapa de bits a partir de un array de píxeles preparado en memoria.

bool ResourceCreate(const string resource, const string filepath)

La función carga el recurso denominado *resource* desde un archivo ubicado en *filepath*. Si la ruta comienza con una barra invertida '\' (en cadenas constantes debe duplicarse: «\\path\\name.ext»), entonces el archivo se busca en esta ruta relativa a la carpeta MQL5 en el directorio de datos del terminal (por ejemplo, «\Files \CustomSounds\\Hello.wav» se refiere a *MQL5/Files/CustomSounds/Hello.wav*). Si no hay barra invertida, el recurso se busca a partir de la carpeta donde se encuentra el archivo ejecutable desde el que llamamos a la función.

La ruta puede apuntar a un recurso estático integrado en un programa MQL de terceros o actual. Por ejemplo, un determinado script es capaz de crear un recurso basado en una imagen del indicador *BmpOwner.mq5* comentado en la sección sobre variables de recursos.

```
ResourceCreate("::MyImage", "\\Indicators\\MQL5Book\\p7\\BmpOwner.ex5::search1.bmp");
```

El nombre del recurso en el parámetro *resource* puede contener un doble dos puntos inicial (aunque esto no es necesario, ya que, si no está presente, el prefijo «::» se añadirá al nombre automáticamente). Esto garantiza la unificación del uso de una línea para declarar un recurso en la llamada a *ResourceCreate*, así como para el acceso posterior al mismo (por ejemplo, al establecer la propiedad OBJPROP_BMPFILE).

Por supuesto, la declaración anterior para crear un recurso dinámico es redundante si solo queremos cargar un recurso de imagen de terceros en nuestro objeto en el gráfico, ya que es suficiente con asignar directamente la cadena «\Indicators\MQL5Book\p7\BmpOwner.ex5:» a la propiedad OBJPROP_BMPFILE: search1.bmp. Sin embargo, si necesita editar una imagen, un recurso dinámico es indispensable. A continuación, mostraremos un ejemplo en la sección Leer y modificar datos de recursos.

Los recursos dinámicos están disponibles públicamente desde otros programas MQL por su nombre completo, que incluye la ruta y el nombre del programa que creó el recurso. Por ejemplo, si la anterior llamada a *ResourceCreate* fue producida por el script *MQL5/Scripts/MyExample.ex5*, entonces otro programa MQL puede acceder al mismo recurso utilizando el enlace completo «\\Scripts\ \MyExample.ex5::MyImage», y cualquier otro script en la misma carpeta puede acceder a la abreviatura «MyExample.ex5::MyImage» (aquí la ruta relativa es simplemente degenerada). Las reglas para escribir rutas completas (desde la carpeta raíz de MQL5) y relativas se dieron anteriormente.

La función *ResourceCreate* devuelve un indicador booleano de éxito (*true*) o error (*false*) como resultado de la ejecución. El código de error, como de costumbre, se puede encontrar en la variable *_LastError*. En concreto, es probable que reciba los siguientes errores:

- ERR_RESOURCE_NAME_DUPLICATED (4015) - coincidencia de los nombres de los recursos dinámicos y estáticos
- ERR_RESOURCE_NOT_FOUND (4016) - no se encuentra el recurso/archivo indicado en el parámetro *filepath*
- ERR_RESOURCE_UNSUPPOTED_TYPE (4017) - tipo de recurso no admitido o tamaño superior a 2 GB
- ERR_RESOURCE_NAME_IS_TOO_LONG (4018) - el nombre del recurso supera los 63 caracteres

Todo esto se aplica no sólo a la primera forma de la función, sino también a la segunda.

bool ResourceCreate(const string resource, const uint &data[], uint img_width, uint img_height, uint data_xoffset, uint data_yoffset, uint data_width, ENUM_COLOR_FORMAT color_format)

El parámetro *resource* sigue significando el nombre del nuevo recurso, y el contenido de la imagen viene dado por el resto de parámetros.

El array *data* puede ser unidimensional (*data[]*) o bidimensional (*data[][]*): pasa los puntos (píxeles) de la trama. Los parámetros *img_width* y *img_height* establecen las dimensiones de la imagen visualizada (en píxeles). Estos tamaños pueden ser inferiores al tamaño físico de la imagen en el array *data*, debido a lo cual se consigue el efecto de encuadre cuando solo se emite una parte de la imagen original. Los parámetros *data_xoffset* y *data_yoffset* determinan la coordenada de la esquina superior izquierda del «marco».

El parámetro *data_width* indica la anchura total de la imagen original (en el array *data*). Un valor de 0 implica que esta anchura es la misma que *img_width*. El parámetro *data_width* solo tiene sentido cuando se especifica un array unidimensional en el parámetro *data*, ya que para un array bidimensional se conocen sus dimensiones en ambas dimensiones (en este caso, el parámetro *data_width* se ignora y se asume igual a la segunda dimensión del array *data[][]*).

En el caso más común, cuando desee mostrar la imagen completa («tal cual»), utilice la siguiente sintaxis:

```
ResourceCreate(name, data, width, height, 0, 0, 0, ...);
```

Por ejemplo, si el programa tiene un recurso estático descrito como un array bidimensional *bitmap*:

```
#resource "static.bmp" as bitmap data[][]
```

A continuación, la creación de un recurso dinámico basado en él puede realizarse de la siguiente manera:

```
ResourceCreate("dynamic", data, ArrayRange(data, 1), ArrayRange(data, 0), 0, 0, 0, ..
```

La creación de un recurso dinámico basado en uno estático es muy solicitada no solo cuando se requiere una edición directa, sino también para controlar cómo se procesan los colores al mostrar un recurso. Este modo se selecciona utilizando el último parámetro de la función: *color_format*. Utiliza la enumeración ENUM_COLOR_FORMAT.

Identificador	Descripción
COLOR_FORMAT_XRGB_NOALPHA	Se ignora el componente del canal alfa (transparencia)
COLOR_FORMAT_ARGB_RAW	El terminal no procesa los componentes de color
COLOR_FORMAT_ARGB_NORMALIZE	Los componentes de color son procesados por el terminal (véase más abajo)

En el modo COLOR_FORMAT_XRGB_NOALPHA, la imagen se muestra sin efectos: cada punto se muestra en un color sólido (es la forma más rápida de dibujar). Los otros dos modos muestran los píxeles teniendo en cuenta la transparencia en el byte alto de cada píxel, pero tienen efectos diferentes. En el caso de COLOR_FORMAT_ARGB_NORMALIZE, el terminal realiza las siguientes transformaciones de los componentes de color de cada punto al preparar el ráster en el momento de la llamada a *ResourceCreate*:

```
R = R * A / 255
G = G * A / 255
B = B * A / 255
A = A
```

Los recursos de imágenes estáticas en directivas *#resource* están conectadas con la ayuda de COLOR_FORMAT_ARGB_NORMALIZE.

En un recurso dinámico, el tamaño del array está limitado por el valor de INT_MAX bytes (2147483647, 2 Gb), que supera considerablemente el límite impuesto por el compilador al procesar la directiva estática *#resource*: el tamaño del archivo no puede superar los 128 Mb.

Si se llama a la segunda versión de la función para crear un recurso con el mismo nombre, pero cambiando otros parámetros (el contenido del array de píxeles, la anchura, la altura o el desplazamiento), entonces no se vuelve a crear el nuevo recurso, sino que simplemente se actualiza el existente. Solo el programa propietario del recurso (el programa que lo creó en primer lugar) puede modificar un recurso de esta forma.

> Si, al crear recursos dinámicos a partir de distintas copias del programa que se ejecutan en distintos gráficos, necesita su propio recurso en cada copia, deberá añadir *ChartID* al nombre del recurso.

Para demostrar la creación dinámica de imágenes en varios esquemas de color, proponemos desmontar el script *ARGBbitmap.mq5*.

Se adjunta estáticamente la imagen «argb.bmp».

```
#resource "argb.bmp" as bitmap Data[][]
```

El usuario selecciona el método de formateo del color mediante el parámetro *ColorFormat*.

```
input ENUM_COLOR_FORMAT ColorFormat = COLOR_FORMAT_XRGB_NOALPHA;
```

El nombre del objeto en el que se mostrará la imagen y el nombre del recurso dinámico se describen mediante las variables *BitmapObject* y *ResName*.

```
const string BitmapObject = "BitmapObject";
const string ResName = "::image";
```

A continuación se muestra la función principal del script.

```
void OnStart()
{
   ResourceCreate(ResName, Data, ArrayRange(Data, 1), ArrayRange(Data, 0),
      0, 0, 0, ColorFormat);

   ObjectCreate(0, BitmapObject, OBJ_BITMAP_LABEL, 0, 0, 0);
   ObjectSetInteger(0, BitmapObject, OBJPROP_XDISTANCE, 50);
   ObjectSetInteger(0, BitmapObject, OBJPROP_YDISTANCE, 50);
   ObjectSetString(0, BitmapObject, OBJPROP_BMPFILE, ResName);

   Comment("Press ESC to stop the demo");
   const ulong start = TerminalInfoInteger(TERMINAL_KEYSTATE_ESCAPE);
   while(!IsStopped()  // waiting for the user's command to end the demo
   && TerminalInfoInteger(TERMINAL_KEYSTATE_ESCAPE) == start)
   {
      Sleep(1000);
   }

   Comment("");
   ObjectDelete(0, BitmapObject);
   ResourceFree(ResName);
}
```

El script crea un nuevo recurso en el modo de color especificado y lo asigna a la propiedad OBJPROP_BMPFILE de un objeto de tipo OBJ_BITMAP_LABEL. A continuación, el script espera a que el usuario detenga explícitamente el script o pulse *Esc* y luego borra el objeto (llamando a *ObjectDelete*) y el recurso utilizando la función *ResourceFree*. Tenga en cuenta que borrar un objeto no elimina automáticamente el recurso. Por eso necesitamos la función *ResourceFree*, de la que hablaremos en la sección siguiente.

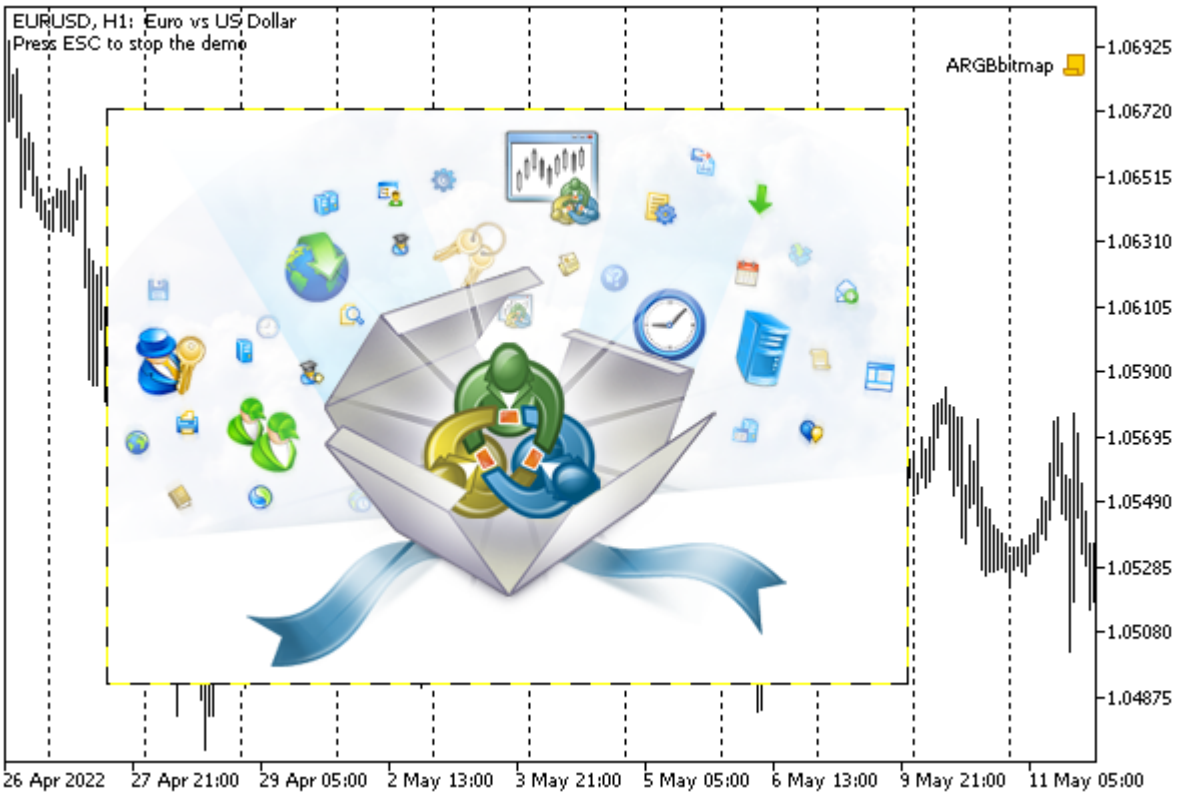
Si no llamamos a *ResourceFree*, los recursos dinámicos permanecen en la memoria del terminal incluso después de que finalice el programa MQL, hasta que se cierre el terminal. Esto permite utilizarlos como repositorios o como medio de intercambio de información entre programas MQL.

Un recurso dinámico creado mediante la segunda forma de *ResourceCreate* no tiene por qué llevar ninguna imagen. El array *data* puede contener datos arbitrarios si no la utilizamos para el renderizado. En este caso, es importante establecer el esquema COLOR_FORMAT_XRGB_NOALPHA. En algún momento mostraremos un ejemplo de este tipo.

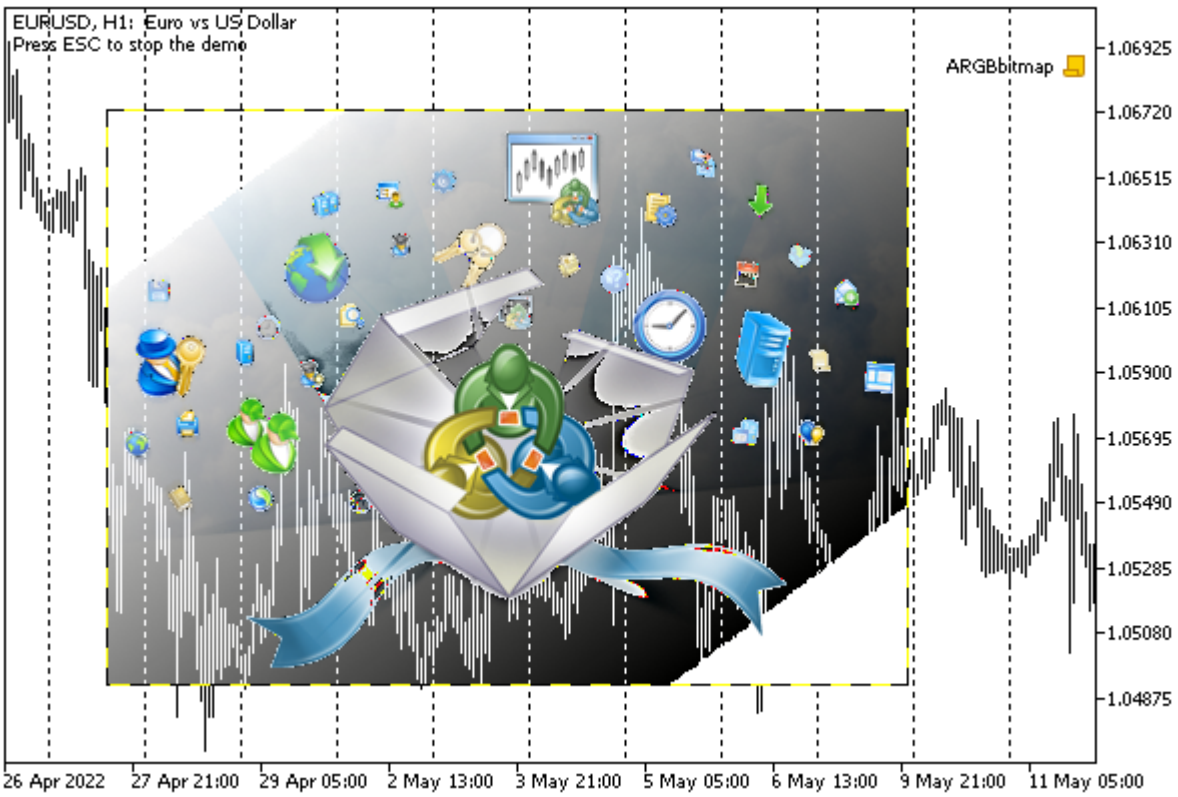
Mientras tanto, comprobemos cómo funciona el script *ARGBbitmap.mq5*.

La imagen anterior «argb.bmp» contiene información sobre la transparencia: la esquina superior izquierda tiene un fondo completamente transparente, y la transparencia se desvanece en diagonal hacia la esquina inferior derecha.

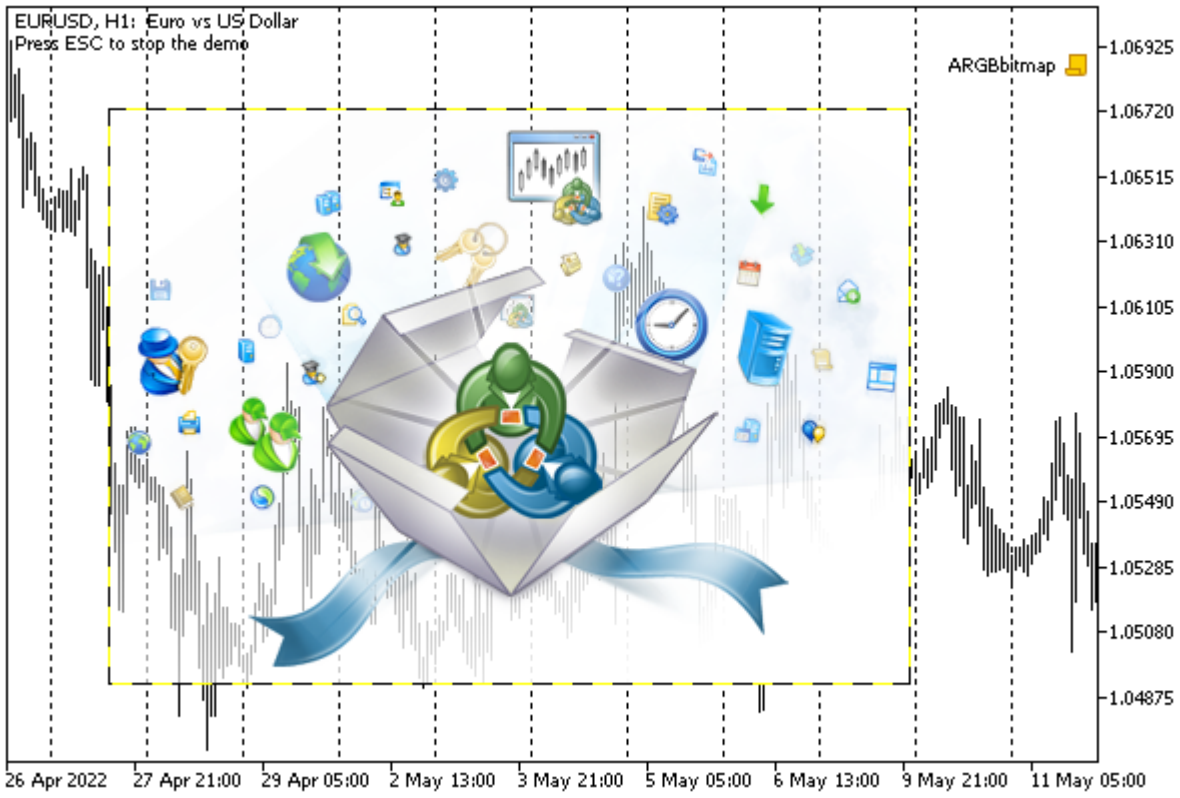
Las siguientes imágenes muestran los resultados de ejecutar el script en tres modos diferentes.



Salida de imagen en formato de color COLOR_FORMAT_XRGB_NOALPHA



Salida de imagen en formato de color COLOR_FORMAT_ARGB_RAW



Salida de imagen en formato de color COLOR_FORMAT_ARGB_NORMALIZE

7.1.6 Eliminar recursos dinámicos: ResourceFree

La función *ResourceFree* elimina el recurso dinámico creado anteriormente y libera la memoria que ocupa. Si no llama a *ResourceFree*, el recurso dinámico permanecerá en memoria hasta el final de la sesión del terminal actual. Puede utilizarse como una forma cómoda de almacenar datos, pero para el trabajo habitual con imágenes, se recomienda liberarlas cuando desaparezca la necesidad de utilizarlas.

Los objetos gráficos adjuntos al recurso que se elimina se mostrarán correctamente incluso después de su eliminación. No obstante, los objetos gráficos recién creados (OBJ_BITMAP y OBJ_BITMAP_LABEL) ya no podrán utilizar el recurso eliminado.

bool ResourceFree(const string resource)

El nombre del recurso se establece en el parámetro *resource* y debe empezar por «::».

La función devuelve un indicador de éxito (*true*) o de error (*false*).

La función borra sólo los recursos dinámicos creados por el programa MQL dado, pero no los de «terceros».

En la sección anterior, vimos un ejemplo del script *ARGBbitmap.mq5*, que llamaba a *ResourceFree* al finalizar su operación.

7.1.7 Leer y modificar datos de recursos: RecursoReadImage

La función *ResourceReadImage* permite leer los datos del recurso creado por la función *ResourceCreate* o incrustado en el ejecutable en tiempo de compilación según la directiva *#resource*. A pesar del sufijo

«Image» del nombre, la función opera con cualquier array de datos, incluidos los personalizados (véase el ejemplo de *Reservoir.mq5* más abajo).

bool ResourceReadImage(const string resource, uint &data[], uint &width, uint &height)

El nombre del recurso se especifica en el parámetro *resource*. Para acceder a sus propios recursos, basta con la forma abreviada «::nombre_del_recurso». Para leer un recurso de otro archivo compilado, necesita el nombre completo seguido de la ruta según las reglas de resolución de rutas descritas en la sección sobre recursos. En concreto, una ruta que empiece por una barra invertida significa la ruta desde la carpeta raíz del MQL5 (de esta forma se busca «\\path\\filename.ex5::nombre_del_recurso» en el archivo */MQL5/path/filename.ex5* bajo el nombre «nombre_del_recurso»), y la ruta sin este carácter inicial significa la ruta relativa a la carpeta donde se encuentra el programa ejecutado.

La información interna del recurso se escribirá en el array receptor *data*, y los parámetros *width* y *height* recibirán, respectivamente, la anchura y la altura, es decir, el tamaño del array (*width*height*) de forma indirecta. Por separado, *width* y *height* solo son relevantes si la imagen está almacenada en el recurso. El array debe ser dinámico o fijo, pero de tamaño suficiente. De lo contrario, obtendremos un error SMALL_ARRAY (5052).

Si en el futuro desea crear un recurso gráfico basado en el array *data*, entonces el recurso fuente deberá utilizar el formato de color COLOR_FORMAT_ARGB_NORMALIZE o COLOR_FORMAT_XRGB_NOALPHA. Si el array *data* contiene datos arbitrarios de la aplicación, utilice COLOR_FORMAT_XRGB_NOALPHA.

Como primer ejemplo, consideremos el script *ResourceReadImage.mq5*, que sirve para hacer una demostración de varios aspectos del trabajo con recursos gráficos:

- Creación de un recurso de imagen a partir de un archivo externo
- Lectura y modificación de los datos de esta imagen en otro recurso creado dinámicamente
- Conservación de los recursos creados en la memoria del terminal entre lanzamientos de scripts
- Utilización de recursos en objetos del gráfico
- Eliminación de un objeto y sus recursos

La modificación de la imagen en este caso concreto significa la inversión de todos los colores (como lo más visual).

Todos los métodos de trabajo anteriores se realizan en tres etapas: cada etapa se realiza en una ejecución del script. El script determina la etapa actual mediante el análisis de los recursos disponibles y el objeto:

1. En ausencia de los recursos gráficos necesarios, el script los creará (una imagen original y otra invertida).
2. Si hay recursos pero no hay objeto gráfico, el script creará un objeto con dos imágenes del primer paso para los estados activado/desactivado (se pueden cambiar haciendo clic con el ratón).
3. Si hay un objeto, el script borrará el objeto y los recursos.

La función principal del script comienza definiendo los nombres de los recursos y del objeto en el gráfico.

```
void OnStart()
{
   const static string resource = "::Images\\pseudo.bmp";
   const static string inverted = resource + "_inv";
   const static string object = "object";
   ...
```

Tenga en cuenta que hemos elegido un nombre para el recurso original que se parece a la ubicación del archivo *bmp* en la carpeta estándar *Images*, pero no existe tal archivo. Esto pone de relieve la naturaleza virtual de los recursos y le permite hacer sustituciones para cumplir requisitos técnicos o dificultar la ingeniería inversa de sus programas.

La siguiente llamada a *ResourceReadImage* se utiliza para comprobar si el recurso ya existe. En el estado inicial (en la primera ejecución), obtendremos un resultado negativo (*false*) y comenzaremos el primer paso: creamos el recurso original a partir del archivo «\Images\dollar.bmp» y luego lo invertimos en un nuevo recurso con el sufijo «_inv».

```
   uint data[], width, height;
   // check for resource existence
   if(!PRTF(ResourceReadImage(resource, data, width, height)))
   {
      Print("Initial state: Creating 2 bitmaps");
      PRTF(ResourceCreate(resource, "\\Images\\dollar.bmp")); // try "argb.bmp"
      ResourceCreateInverted(resource, inverted);
   }
   ...
```

A continuación se presenta el código fuente de la función de ayuda *ResourceCreateInverted*.

Si se encuentra el recurso (segunda ejecución), el script comprueba la existencia del objeto y, si es necesario, lo crea, incluyendo la configuración de propiedades con recursos de imagen en la función *ShowBitmap* (véase más adelante).

```
   else
   {
      Print("Resources (bitmaps) are detected");
      if(PRTF(ObjectFind(0, object) < 0))
      {
         Print("Active state: Creating object to draw 2 bitmaps");
         ShowBitmap(object, resource, inverted);
      }
      ...
```

Si tanto los recursos como el objeto ya están en el gráfico, entonces estamos en la etapa final y debemos eliminar todos los recursos.

```
      else
      {
         Print("Cleanup state: Removing object and resources");
         PRTF(ObjectDelete(0, object));
         PRTF(ResourceFree(resource));
         PRTF(ResourceFree(inverted));
      }
   }
}
```

La función *ResourceCreateInverted* utiliza la llamada *ResourceReadImage* para obtener un array de píxeles y luego invierte el color en ellos utilizando el operador '^' (XOR) y un operando con todos los bits singulares en los componentes de color.

```
bool ResourceCreateInverted(const string resource, const string inverted)
{
   uint data[], width, height;
   PRTF(ResourceReadImage(resource, data, width, height));
   for(int i = 0; i < ArraySize(data); ++i)
   {
      data[i] = data[i] ^ 0x00FFFFFF;
   }
   return PRTF(ResourceCreate(inverted, data, width, height, 0, 0, 0,
      COLOR_FORMAT_ARGB_NORMALIZE));
}
```

El nuevo array *data* se transfiere a *ResourceCreate* para crear la segunda imagen.

La función *ShowBitmap* crea un objeto gráfico de la forma habitual (en la esquina inferior derecha del gráfico) y establece sus propiedades para los estados activado y desactivado a las imágenes original e invertida, respectivamente.

```
void ShowBitmap(const string name, const string resourceOn, const string resourceOff
{
   ObjectCreate(0, name, OBJ_BITMAP_LABEL, 0, 0, 0);

   ObjectSetString(0, name, OBJPROP_BMPFILE, 0, resourceOn);
   if(resourceOff != NULL) ObjectSetString(0, name, OBJPROP_BMPFILE, 1, resourceOff);
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, 50);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, 50);
   ObjectSetInteger(0, name, OBJPROP_CORNER, CORNER_RIGHT_LOWER);
   ObjectSetInteger(0, name, OBJPROP_ANCHOR, ANCHOR_RIGHT_LOWER);
}
```

Como el objeto recién creado está desactivado por defecto, primero veremos la imagen invertida y podremos cambiarla por la original al hacer clic con el ratón. Pero recordemos que nuestro script realiza acciones paso a paso, y por lo tanto, antes de que la imagen aparezca en el gráfico, el script debe ejecutarse dos veces. En todas las etapas se registran el estado actual y las acciones realizadas (junto con una indicación de éxito o error).

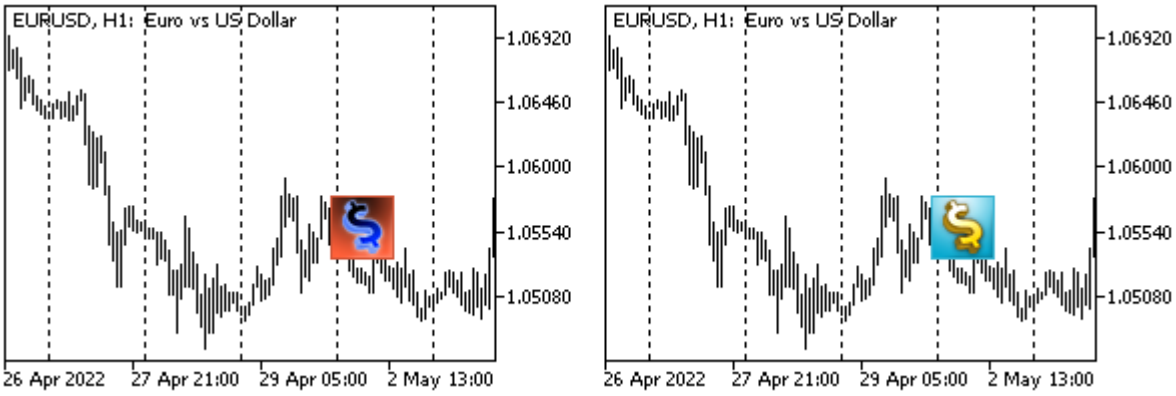
Tras el primer lanzamiento, aparecerán las siguientes entradas en el registro:

```
ResourceReadImage(resource,data,width,height)=false / RESOURCE_NOT_FOUND(4016)
Initial state: Creating 2 bitmaps
ResourceCreate(resource,\Images\dollar.bmp)=true / ok
ResourceReadImage(resource,data,width,height)=true / ok
ResourceCreate(inverted,data,width,height,0,0,0,COLOR_FORMAT_XRGB_NOALPHA)=true / ok
```

Los registros indican que no se han encontrado los recursos y por eso el script los ha creado. Tras la segunda ejecución, el registro dirá que se han encontrado recursos (que quedaron en memoria de la ejecución anterior del script) pero que el objeto aún no está ahí, y el script lo creará basándose en los recursos.

```
ResourceReadImage(resource,data,width,height)=true / ok
Resources (bitmaps) are detected
ObjectFind(0,object)<0=true / OBJECT_NOT_FOUND(4202)
Active state: Creating object to draw 2 bitmaps
```

Veremos un objeto y una imagen en el gráfico. Se puede cambiar de estado haciendo clic con el ratón (loseventos sobre cambios de estado no se tratan aquí).



Imágenes invertida y original en un objeto de un gráfico

Por último, durante la tercera ejecución, el script detectará el objeto y borrará todos sus desarrollos.

```
ResourceReadImage(resource,data,width,height)=true / ok
Resources (bitmaps) are detected
ObjectFind(0,object)<0=false / ok
Cleanup state: Removing object and resources
ObjectDelete(0,object)=true / ok
ResourceFree(resource)=true / ok
ResourceFree(inverted)=true / ok
```

Después puede repetir el ciclo.

El segundo ejemplo de la sección considerará el uso de recursos para almacenar datos arbitrarios de la aplicación, es decir, una especie de portapapeles dentro del terminal (en teoría, puede haber cualquier número de tales búferes, ya que cada uno de ellos es un recurso separado con nombre). Debido a la universalidad del problema, crearemos la clase *Reservoir* con la funcionalidad principal (en el archivo *Reservoir.mqh*), y sobre su base escribiremos un script de demostración (*Reservoir.mq5*).

Antes de «sumergirnos» directamente en Reservoir, vamos a introducir una unión auxiliar *ByteOverlay* que se necesitará con frecuencia. Una unión permitirá convertir cualquier tipo simple integrado (incluidas las estructuras simples) en un array de bytes y viceversa. Por «simple» entendemos todos los tipos numéricos integrados, fecha y hora, enumeraciones, color y banderas booleanas. Sin embargo, los objetos y los arrays dinámicos ya no son sencillos y no serán compatibles con nuestro nuevo

almacenamiento (debido a limitaciones técnicas de la plataforma). Las cadenas tampoco se consideran simples, pero para ellas haremos una excepción y las procesaremos de forma especial.

```
template<typename T>
union ByteOverlay
{
   uchar buffer[sizeof(T)];
   T value;

   ByteOverlay(const T &v)
   {
      value = v;
   }

   ByteOverlay(const uchar &bytes[], const int offset = 0)
   {
      ArrayCopy(buffer, bytes, 0, offset, sizeof(T));
   }
};
```

Como sabemos, los recursos se construyen sobre la base de arrays del tipo *uint*, por lo que describimos un array de este tipo (*storage*) en la clase *Reservoir*. Allí añadiremos todos los datos que se escribirán posteriormente en el recurso. La posición actual en el array desde la que se escriben o leen los datos se almacena en el campo *offset*.

```
class Reservoir
{
   uint storage[];
   int offset;
public:
   Reservoir(): offset(0) { }
   ...
```

Para colocar un array de datos de tipo arbitrario en *storage*, puede utilizar el método de plantilla *packArray*. En la primera mitad, convertimos el array pasado en un array de bytes utilizando *ByteOverlay*.

```
   template<typename T>
   int packArray(const T &data[])
   {
      const int bytesize = ArraySize(data) * sizeof(T); // TODO: check for overflow
      uchar buffer[];
      ArrayResize(buffer, bytesize);
      for(int i = 0; i < ArraySize(data); ++i)
      {
         ByteOverlay<T> overlay(data[i]);
         ArrayCopy(buffer, overlay.buffer, i * sizeof(T));
      }
      ...
```

En la segunda mitad, convertimos el array de bytes en una secuencia de valores *uint*, que se escriben en *storage* con un *offset*. El número de elementos necesarios *uint* se determina teniendo en cuenta si

queda un resto después de dividir el tamaño de los datos en bytes por el tamaño de *uint*: opcionalmente añadimos un elemento adicional.

```
      const int size = bytesize / sizeof(uint) + (bool)(bytesize % sizeof(uint));
      ArrayResize(storage, offset + size + 1);
      storage[offset] = bytesize;       // write the size of the data before the data
      for(int i = 0; i < size; ++i)
      {
         ByteOverlay<uint> word(buffer, i * sizeof(uint));
         storage[offset + i + 1] = word.value;
      }

      offset = ArraySize(storage);

      return offset;
   }
```

Antes de los datos propiamente dichos, escribimos el tamaño de los datos en bytes: se trata del protocolo más pequeño posible para la comprobación de errores a la hora de recuperar datos. En el futuro sería posible también escribir los datos de *typename(T)* en *storage*.

El método devuelve la posición actual en el almacenamiento después de la escritura.

Basándonos en *packArray* es fácil implementar un método para guardar cadenas:

```
int packString(const string text)
{
   uchar data[];
   StringToCharArray(text, data, 0, -1, CP_UTF8);
   return packArray(data);
}
```

También existe la opción de almacenar un número independiente:

```
template<typename T>
int packNumber(const T number)
{
   T array[1] = {number};
   return packArray(array);
}
```

Un método para restaurar un array de tipo arbitrario T desde el almacenamiento de tipo *uint* «pierde» todas las operaciones en sentido contrario. Si se encuentran incoherencias en el tipo legible y la cantidad de datos con el almacenamiento, el método devuelve 0 (un signo de error). En modo normal, se devuelve la posición actual en el array *storage* (siempre es mayor que 0 si se ha leído algo con éxito).

```
template<typename T>
int unpackArray(T &output[])
{
   if(offset >= ArraySize(storage)) return 0; // out of array bounds
   const int bytesize = (int)storage[offset];
   if(bytesize % sizeof(T) != 0) return 0;    // wrong data type
   if(bytesize > (ArraySize(storage) - offset) * sizeof(uint)) return 0;

   uchar buffer[];
   ArrayResize(buffer, bytesize);
   for(int i = 0, k = 0; i < ArraySize(storage) - 1 - offset
      && k < bytesize; ++i, k += sizeof(uint))
   {
      ByteOverlay<uint> word(storage[i + 1 + offset]);
      ArrayCopy(buffer, word.buffer, k);
   }

   int n = bytesize / sizeof(T);
   n = ArrayResize(output, n);
   for(int i = 0; i < n; ++i)
   {
      ByteOverlay<T> overlay(buffer, i * sizeof(T));
      output[i] = overlay.value;
   }

   offset += 1 + bytesize / sizeof(uint) + (bool)(bytesize % sizeof(uint));

   return offset;
}
```

El desempaquetado de cadenas y números se realiza llamando a *unpackArray*.

```
int unpackString(string &output)
{
   uchar bytes[];
   const int p = unpackArray(bytes);
   if(p == offset)
   {
      output = CharArrayToString(bytes, 0, -1, CP_UTF8);
   }
   return p;
}

template<typename T>
int unpackNumber(T &number)
{
   T array[1] = {};
   const int p = unpackArray(array);
   number = array[0];
   return p;
}
```

Unos sencillos métodos de ayuda permiten averiguar el tamaño del almacén y la posición actual en él, así como borrarlo.

```
int size() const
{
   return ArraySize(storage);
}

int cursor() const
{
   return offset;
}

void clear()
{
   ArrayFree(storage);
   offset = 0;
}
```

Ahora llegamos a lo más interesante: la interacción con los recursos.

Una vez rellenado el array *storage* con datos de la aplicación, es fácil «moverlo» a un recurso proporcionado.

```
bool submit(const string resource)
{
   return ResourceCreate(resource, storage, ArraySize(storage), 1,
      0, 0, 0, COLOR_FORMAT_XRGB_NOALPHA);
}
```

Además, podemos leer datos de un recurso en un array *storage* interno.

```
bool acquire(const string resource)
{
   uint width, height;
   if(ResourceReadImage(resource, storage, width, height))
   {
      return true;
   }
   return false;
}
```

Mostraremos en el script *Reservoir.mq5* cómo utilizarlo.

En la primera mitad de *OnStart* describimos el nombre para el recurso de almacenamiento y el objeto de clase *Reservoir*, y luego «empaquetamos» secuencialmente en este objeto una cadena, la estructura *MqlTick* y el número *double*. La estructura se «envuelve» en un array de un elemento para demostrar explícitamente el método *packArray*. Además, luego tendremos que comparar los datos restaurados con los originales, y MQL5 no proporciona el operador '==' para estructuras. Por lo tanto, será más conveniente utilizar la función *ArrayCompare*.

```
#include <MQL5Book/Reservoir.mqh>
#include <MQL5Book/PRTF.mqh>

void OnStart()
{
   const string resource = "::reservoir";

   Reservoir res1;
   string message = "message1";     // string to write to the resource
   PRTF(res1.packString(message));

   MqlTick tick1[1];                // add a simple structure
   SymbolInfoTick(_Symbol, tick1[0]);
   PRTF(res1.packArray(tick1));
   PRTF(res1.packNumber(DBL_MAX));  // real number
   ...
```

Cuando todos los datos necesarios estén «empaquetados» en el objeto, escríbalos en el recurso y borre el objeto.

```
res1.submit(resource);           // create a resource with storage data
res1.clear();                    // clear the object, but not the resource
```

En la segunda mitad de *OnStart* vamos a realizar las operaciones inversas de lectura de datos del recurso.

```
   string reply;                    // new variable for message
   MqlTick tick2[1];                // new structure for tick
   double result;                   // new variable for number

   PRTF(res1.acquire(resource));    // connect the object to the given resource
   PRTF(res1.unpackString(reply));  // read line
   PRTF(res1.unpackArray(tick2));   // read simple structure
   PRTF(res1.unpackNumber(result)); // read number

   // output and compare data element by element
   PRTF(reply);
   PRTF(ArrayCompare(tick1, tick2));
   ArrayPrint(tick2);
   PRTF(result == DBL_MAX);

   // make sure the storage is read completely
   PRTF(res1.size());
   PRTF(res1.cursor());
   ...
```

Al final, limpiamos el recurso, ya que se trata de una prueba. En tareas prácticas, lo más probable es que un programa MQL deje el recurso creado en memoria para que pueda ser leído por otros programas. En la jerarquía de nombres, los recursos se declaran anidados en el programa que los creó. Por lo tanto, para acceder desde otros programas, debe especificar el nombre del recurso junto con el nombre del programa y, opcionalmente, la ruta (si el creador del programa y el lector del programa están en carpetas diferentes). Por ejemplo, para leer un recurso recién creado desde el exterior, la ruta completa «\Scripts\MQL5Book\p7\Reservoir.ex5::reservoir» se encargará de hacerlo.

```
   PrintFormat("Cleaning up local storage '%s'", resource);
   ResourceFree(resource);
}
```

Dado que todas las llamadas a métodos principales están controladas por la macro PRTF, cuando ejecutemos el script veremos un «informe» de progreso detallado en el registro.

```
res1.packString(message)=4 / ok
res1.packArray(tick1)=20 / ok
res1.packNumber(DBL_MAX)=23 / ok
res1.acquire(resource)=true / ok
res1.unpackString(reply)=4 / ok
res1.unpackArray(tick2)=20 / ok
res1.unpackNumber(result)=23 / ok
reply=message1 / ok
ArrayCompare(tick1,tick2)=0 / ok
[time] [bid] [ask] [last] [volume] [time_msc] [flags] [volume_real]
[0] 2022.05.19 23:09:32 1.05867 1.05873 0.0000 0 1653001772050 6 0.00000
result==DBL_MAX=true / ok
res1.size()=23 / ok
res1.cursor()=23 / ok
Cleaning up local storage '::reservoir'
```

Los datos se han copiado correctamente en el recurso y luego se han restaurado desde allí.

Los programas pueden utilizar este enfoque para intercambiar datos voluminosos que no caben en mensajes personalizados (eventosCHARTEVENT_CUSTOM+). Basta con enviar en un parámetro de cadena *sparam* el nombre del recurso que se va a leer. Para devolver datos, cree su propio recurso con ellos y envíe un mensaje de respuesta.

7.1.8 Guardar imágenes en un archivo: ResourceSave

La API de MQL5 permite escribir un recurso en un archivo BMP mediante la función *ResourceSave*. Actualmente, el marco sólo admite recursos de imagen.

bool ResourceSave(const string resource, const string filename)

Los parámetros *resource* y *filename* especifican el nombre del recurso y del archivo, respectivamente. El nombre del recurso debe empezar por «::». El nombre del archivo puede contener una ruta relativa a la carpeta *MQL5/Files*. Si es necesario, la función creará todos los subdirectorios intermedios. Si el archivo especificado existe, se sobrescribirá.

La función devuelve *true* en caso de éxito.

Para comprobar el funcionamiento de esta función conviene crear una imagen original. Tenemos exactamente la imagen adecuada para ello.

Como parte del estudio de la POO, en el capítulo Clases e interfaces iniciamos una serie de ejemplos sobre formas gráficas: desde la primera versión *Shapes1.mq5* en la sección sobre Definición de clases a la última versión *Shapes6.mq5* en la sección sobre Tipos anidados. Entonces, dibujar no estaba a nuestro alcance, y no fue sino en el capítulo sobre objetos gráficos que pudimos implementar la visualización en el script *ObjectShapesDraw.mq5*. Ahora, tras estudiar los recursos gráficos, ha llegado el momento de otra «mejora».

En la nueva versión del script *ResourceShapesDraw.mq5* dibujaremos las formas. Para facilitar el análisis de los cambios con respecto a la versión anterior, mantendremos el mismo conjunto de formas: rectángulo, cuadrado, óvalo, círculo y triángulo. Esto se hace para dar un ejemplo, y no porque algo nos limite a la hora de dibujar; al contrario: existe un potencial para ampliar el conjunto de formas, efectos visuales y etiquetado. Veremos las características en algunos ejemplos, empezando por el actual. No obstante, tenga en cuenta que no es posible demostrar toda la gama de aplicaciones en el ámbito de este libro.

Una vez generadas y dibujadas las formas, guardamos el recurso resultante en un archivo.

La base de la jerarquía de clases de forma es la clase *Shape* que tenía un método *draw*.

```
class Shape
{
public:
   ...
   virtual void draw() = 0;
   ...
}
```

En las clases derivadas, se implementó sobre la base de objetos gráficos, con llamadas a *ObjectCreate* y la posterior configuración de los objetos mediante las funciones de *ObjectSet*. El lienzo compartido de tal dibujo era el propio gráfico.

Ahora tenemos que pintar píxeles en algún recurso compartido de acuerdo con la forma en concreto. Es deseable asignar un recurso común y métodos para modificar píxeles en él en una clase separada o, mejor, una interfaz.

Una entidad abstracta nos permitirá no establecer vínculos con el método de creación y configuración del recurso. En concreto, nuestra próxima implementación colocará el recurso en un objeto OBJ_BITMAP_LABEL (como ya hemos hecho en este capítulo), y para algunos, puede ser suficiente para generar imágenes en la memoria y guardar en el disco sin trazado (ya que a muchos operadores les gusta capturar periódicamente gráficos de los estados).

Llamemos a la interfaz *Drawing*.

```
interface Drawing
{
   void point(const float x1, const float y1, const uint pixel);
   void line(const int x1, const int y1, const int x2, const int y2, const color clr)
   void rect(const int x1, const int y1, const int x2, const int y2, const color clr)
};
```

He aquí sólo tres de los métodos más básicos para dibujar, que son suficientes para este caso.

El método *point* es público (lo que permite poner un punto aparte), pero en cierto sentido, es de bajo nivel, ya que todos los demás se implementarán a través de él. Por ello, las coordenadas que contiene son reales, y el contenido del píxel es un valor ya preparado del tipo *uint*. Esto permitirá, si es necesario, aplicar diversos algoritmos de suavizado para que las formas no se vean escalonadas debido al pixelado. Aquí no abordaremos esta cuestión.

Teniendo en cuenta una interfaz, el método *Shape::draw* se convierte en el siguiente:

```
virtual void draw(Drawing *drawing) = 0;
```

A continuación, en la clase *Rectangle*, es muy fácil delegar el dibujo del rectángulo a una nueva interfaz.

```
class Rectangle : public Shape
{
protected:
   int dx, dy; // size (width, height)
   ...
public:
   void draw(Drawing *drawing) override
   {
 // x, y - anchor point (center) in Shape
      drawing.rect(x - dx / 2, y - dy / 2, x + dx / 2, y + dy / 2, backgroundColor);
   }
};
```

Hay que esforzarse más para dibujar una elipse.

```
class Ellipse : public Shape
{
protected:
   int dx, dy; // large and small radii
   ...
public:
   void draw(Drawing *drawing) override
   {
      // (x, y) - center
      const int hh = dy * dy;
      const int ww = dx * dx;
      const int hhww = hh * ww;
      int x0 = dx;
      int step = 0;

      // main horizontal diameter
      drawing.line(x - dx, y, x + dx, y, backgroundColor);

      // horizontal lines in the upper and lower half, symmetrically decreasing in le
      for(int j = 1; j <= dy; j++)
      {
         for(int x1 = x0 - (step - 1); x1 > 0; --x1)
         {
            if(x1 * x1 * hh + j * j * ww <= hhww)
            {
               step = x0 - x1;
               break;
            }
         }
         x0 -= step;
         drawing.line(x - x0, y - j, x + x0, y - j, backgroundColor);
         drawing.line(x - x0, y + j, x + x0, y + j, backgroundColor);
      }
   }
};
```

Por último, para el triángulo, el renderizado se implementa como sigue:

```
class Triangle: public Shape
{
protected:
   int dx;  // one size, because triangles are equilateral
   ...
public:
   virtual void draw(Drawing *drawing) override
   {
      // (x, y) - center
      // R = a * sqrt(3) / 3
      // p0: x, y + R
      // p1: x - R * cos(30), y - R * sin(30)
      // p2: x + R * cos(30), y - R * sin(30)
      // Pythagorean height: dx * dx = dx * dx / 4 + h * h
      // sqrt(dx * dx * 3/4) = h
      const double R = dx * sqrt(3) / 3;
      const double H = sqrt(dx * dx * 3 / 4);
      const double angle = H / (dx / 2);

      // main vertical line (triangle height)
      const int base = y + (int)(R - H);
      drawing.line(x, y + (int)R, x, base, backgroundColor);

      // smaller vertical lines left and right, symmetrical
      for(int j = 1; j <= dx / 2; ++j)
      {
         drawing.line(x - j, y + (int)(R - angle * j), x - j, base, backgroundColor);
         drawing.line(x + j, y + (int)(R - angle * j), x + j, base, backgroundColor);
      }
   }
};
```

Pasemos ahora a la clase *MyDrawing*, derivada de la interfaz *Drawing*. Se trata de *MyDrawing*, que debe, guiado por llamadas a métodos de interfaz en formas, garantizar que un determinado recurso se muestre en un mapa de bits. Por lo tanto, la clase describe variables para los nombres del objeto gráfico (*object*) y del recurso (*sheet*), así como el array *data* de tipo *uint* para almacenar la imagen. Además, hemos movido el array *shapes* de formas, declarado anteriormente en el manejador *OnStart*. Dado que *MyDrawing* es responsable de dibujar todas las formas, es mejor gestionar su conjunto aquí.

```
class MyDrawing: public Drawing
{
   const string object;     // object with bitmap
   const string sheet;      // resource
   uint data[];             // pixels
   int width, height;       // dimensions
   AutoPtr<Shape> shapes[]; // figures/shapes
   const uint bg;           // background color
   ...
```

En el constructor creamos un objeto gráfico para el tamaño de todo el gráfico y asignamos memoria para el array *data*. El lienzo se rellena con ceros (lo que significa «transparencia negra») o con cualquier valor que se pase en el parámetro *background*, tras lo cual se crea un recurso basado en él.

De manera predeterminada, el nombre del recurso empieza por la letra «D» e incluye el ID del gráfico actual, pero puede especificar otra cosa.

```
public:
   MyDrawing(const uint background = 0, const string s = NULL) :
      object((s == NULL ? "Drawing" : s)),
      sheet("::" + (s == NULL ? "D" + (string)ChartID() : s)), bg(background)
   {
      width = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
      height = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS);
      ArrayResize(data, width * height);
      ArrayInitialize(data, background);

      ResourceCreate(sheet, data, width, height, 0, 0, width, COLOR_FORMAT_ARGB_NORMA

      ObjectCreate(0, object, OBJ_BITMAP_LABEL, 0, 0, 0);
      ObjectSetInteger(0, object, OBJPROP_XDISTANCE, 0);
      ObjectSetInteger(0, object, OBJPROP_YDISTANCE, 0);
      ObjectSetInteger(0, object, OBJPROP_XSIZE, width);
      ObjectSetInteger(0, object, OBJPROP_YSIZE, height);
      ObjectSetString(0, object, OBJPROP_BMPFILE, sheet);
   }
```

El código de llamada puede averiguar el nombre del recurso utilizando el método *resource*.

```
   string resource() const
   {
      return sheet;
   }
```

El recurso y el objeto se eliminan en el destructor.

```
   ~MyDrawing()
   {
      ResourceFree(sheet);
      ObjectDelete(0, object);
   }
```

El método *push* rellena el array de formas.

```
   Shape *push(Shape *shape)
   {
      shapes[EXPAND(shapes)] = shape;
      return shape;
   }
```

El método *draw* dibuja las formas. Simplemente llama al método *draw* de cada forma en el bucle y luego actualiza el recurso y el gráfico.

```
void draw()
{
   for(int i = 0; i < ArraySize(shapes); ++i)
   {
      shapes[i][].draw(&this);
   }
   ResourceCreate(sheet, data, width, height, 0, 0, width, COLOR_FORMAT_ARGB_NORMA
   ChartRedraw();
}
```

A continuación se muestran los métodos más importantes que son los métodos de la interfaz *Drawing* y que realmente implementan el dibujo.

Empecemos por el método *point*, que por ahora presentamos de forma simplificada (más adelante nos ocuparemos de las mejoras).

```
virtual void point(const float x1, const float y1, const uint pixel) override
{
   const int x_main = (int)MathRound(x1);
   const int y_main = (int)MathRound(y1);
   const int index = y_main * width + x_main;
   if(index >= 0 && index < ArraySize(data))
   {
      data[index] = pixel;
   }
}
```

Basándose en *point*, es fácil implementar el dibujo de líneas. Cuando las coordenadas de los puntos inicial y final coinciden en una de las dimensiones, utilizamos el método *rect* para dibujar, ya que una línea recta es un caso degenerado de un rectángulo de espesor unitario.

```
virtual void line(const int x1, const int y1, const int x2, const int y2, const co
{
   if(x1 == x2) rect(x1, y1, x1, y2, clr);
   else if(y1 == y2) rect(x1, y1, x2, y1, clr);
   else
   {
      const uint pixel = ColorToARGB(clr);
      double angle = 1.0 * (y2 - y1) / (x2 - x1);
      if(fabs(angle) < 1) // step along the axis with the largest distance, x
      {
         const int sign = x2 > x1 ? +1 : -1;
         for(int i = 0; i <= fabs(x2 - x1); ++i)
         {
            const float p = (float)(y1 + sign * i * angle);
            point(x1 + sign * i, p, pixel);
         }
      }
      else // or y-step
      {
         const int sign = y2 > y1 ? +1 : -1;
         for(int i = 0; i <= fabs(y2 - y1); ++i)
         {
            const float p = (float)(x1 + sign * i / angle);
            point(p, y1 + sign * i, pixel);
         }
      }
   }
}
```

Y he aquí el método *rect*:

```
virtual void rect(const int x1, const int y1, const int x2, const int y2, const co
{
   const uint pixel = ColorToARGB(clr);
   for(int i = fmin(x1, x2); i <= fmax(x1, x2); ++i)
   {
      for(int j = fmin(y1, y2); j <= fmax(y1, y2); ++j)
      {
         point(i, j, pixel);
      }
   }
}
```

Ahora necesitamos modificar el manejador *OnStart*, y el script estará listo.

En primer lugar, configuramos el gráfico (ocultamos todos los elementos). En teoría, esto no es necesario: se deja para que coincida con el script prototipo.

```
void OnStart()
{
   ChartSetInteger(0, CHART_SHOW, false);
   ...
```

A continuación, describimos el objeto de la clase *MyDrawing*, generamos un número predefinido de formas aleatorias (aquí todo permanece inalterado, incluido el generador *addRandomShape* y la macro FIGURES igual a 21), las dibujamos en el recurso y las mostramos en el objeto del gráfico.

```
   MyDrawing raster;

   for(int i = 0; i < FIGURES; ++i)
   {
      raster.push(addRandomShape());
   }

   raster.draw(); // display the initial state
   ...
```

En el ejemplo *ObjectShapesDraw.mq5*, iniciamos un bucle sin fin en el que movemos las piezas aleatoriamente. Repitamos este truco aquí. Aquí tendremos que añadir la clase *MyDrawing* ya que el array de formas se almacena dentro de ella. Escribamos un método sencillo *shake*.

```
class MyDrawing: public Drawing
{
public:
   ...
   void shake()
   {
      ArrayInitialize(data, bg);
      for(int i = 0; i < ArraySize(shapes); ++i)
      {
         shapes[i][].move(random(20) - 10, random(20) - 10);
      }
   }
   ...
};
```

A continuación, en *OnStart*, podemos utilizar el nuevo método en un bucle hasta que el usuario detenga la animación.

```
void OnStart()
{
   ...
   while(!IsStopped())
   {
      Sleep(250);
      raster.shake();
      raster.draw();
   }
   ...
}
```

En este punto, prácticamente se repite la funcionalidad del ejemplo anterior. Pero necesitamos añadir el guardado de imágenes en un archivo, así que vamos a añadir un parámetro de entrada *SaveImage*.

```
input bool SaveImage = false;
```

Cuando se establece en true, compruebe el rendimiento de la función *ResourceSave*.

```
void OnStart()
{
   ...
   if(SaveImage)
   {
      const string filename = "temp.bmp";
      if(ResourceSave(raster.resource(), filename))
      {
         Print("Bitmap image saved: ", filename);
      }
      else
      {
         Print("Can't save image ", filename, ", ", E2S(_LastError));
      }
   }
}
```

Además, ya que estamos hablando de variables de entrada, deje que el usuario seleccione un fondo y pase el valor resultante al constructor *MyDrawing*.

```
input color BackgroundColor = clrNONE;
void OnStart()
{
   ...
   MyDrawing raster(BackgroundColor != clrNONE ? ColorToARGB(BackgroundColor) : 0);
   ...
}
```

Así pues, todo está listo para la primera prueba.

Si ejecuta el script *ResourceShapesDraw.mq5*, el gráfico formará una imagen como la siguiente:

Mapa de bits de un recurso con un conjunto de formas aleatorias

Al comparar esta imagen con lo que vimos en el ejemplo *ObjectShapesDraw.mq5* resulta que nuestra nueva forma de renderizar es algo diferente a cómo el terminal muestra los objetos. Aunque las formas y los colores son correctos, los lugares donde se superponen las formas están indicados de forma diferente.

Nuestro script pinta las formas con el color especificado, superponiéndolas unas sobre otras en el orden en que aparecen en el array. Las formas posteriores se superponen a las anteriores. El terminal, por su parte, aplica algún tipo de mezcla de colores (inversión) en los lugares de solapamiento.

Ambos métodos tienen derecho a existir, aquí no hay errores. Sin embargo, ¿es posible conseguir un efecto similar al dibujar?

Tenemos control total sobre el proceso de dibujo, por lo que se le puede aplicar cualquier efecto, no sólo el del terminal.

Además de la forma original y sencilla de dibujar, implementemos algunos modos más. Todos ellos se resumen en la enumeración COLOR_EFFECT.

```
enum COLOR_EFFECT
{
   PLAIN,         // simple drawing with overlap (default)
   COMPLEMENT,    // draw with a complementary color (like in the terminal)
   BLENDING_XOR,  // mixing colors with XOR '^'
   DIMMING_SUM,   // "darken" colors with '+'
   LIGHTEN_OR,    // "lighten" colors with '|'
};
```

Añadamos una variable de entrada para seleccionar el modo.

```
input COLOR_EFFECT ColorEffect = PLAIN;
```

Apoyemos los modos en la clase *MyDrawing*. En primer lugar, describamos el campo y el método correspondientes.

```
class MyDrawing: public Drawing
{
   ...
   COLOR_EFFECT xormode;
   ...
public:
   void setColorEffect(const COLOR_EFFECT x)
   {
      xormode = x;
   }
   ...
```

A continuación, mejoramos el método *point*.

```
   virtual void point(const float x1, const float y1, const uint pixel) override
   {
      ...
      if(index >= 0 && index < ArraySize(data))
      {
         switch(xormode)
         {
         case COMPLEMENT:
            data[index] = (pixel ^ (1 - data[index])); // blending with complementary
            break;
         case BLENDING_XOR:
            data[index] = (pixel & 0xFF000000) | (pixel ^ data[index]); // direct mix
            break;
         case DIMMING_SUM:
            data[index] =  (pixel + data[index]); // "darkening" (SUM)
            break;
         case LIGHTEN_OR:
            data[index] =  (pixel & 0xFF000000) | (pixel | data[index]); // "lighteni
            break;
         case PLAIN:
         default:
            data[index] = pixel;
         }
      }
   }
```

Puede probar a ejecutar el script en diferentes modos y comparar los resultados. No olvide la posibilidad de personalizar el fondo. He aquí un ejemplo de cómo queda el aclaramiento.

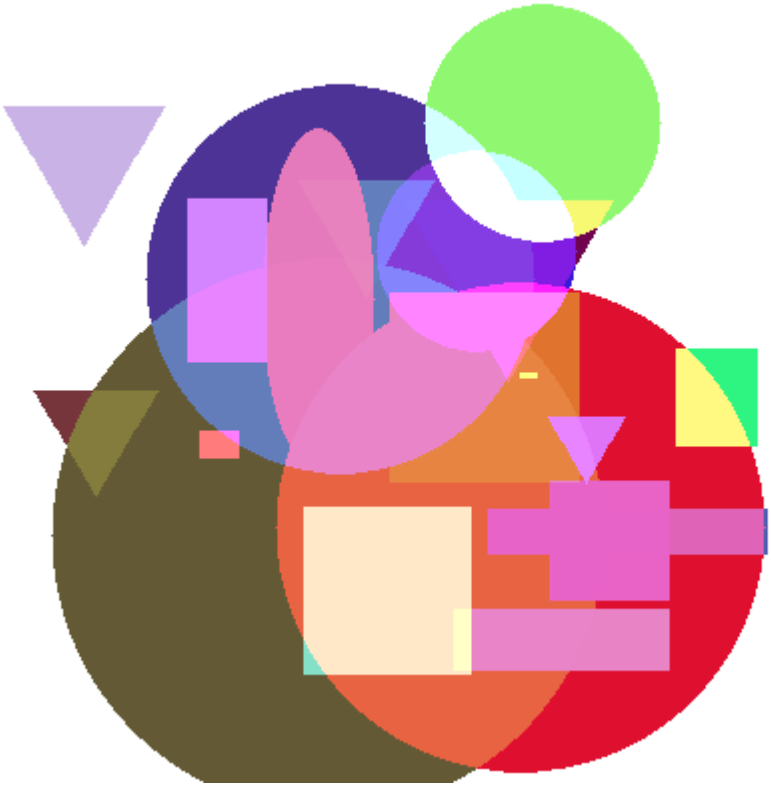


Imagen de formas con mezcla de colores aclarados

Para apreciar visualmente la diferencia de efectos, puede desactivar la aleatorización de colores y el movimiento de formas. La forma estándar de superponer objetos corresponde a la constante COMPLEMENT.

Como experimento final, active la opción *SaveImage*. En el manejador *OnStart*, al generar el nombre del archivo con la imagen, usamos ahora el nombre del modo actual. Necesitamos obtener una copia de la imagen del gráfico en el archivo.

```
...
if(SaveImage)
{
   const string filename = EnumToString(ColorEffect) + ".bmp";
   if(ResourceSave(raster.resource(), filename))
   ...
```

Para construcciones gráficas más sofisticadas de nuestra interfaz, *Drawing* puede no ser suficiente. Por lo tanto, puede utilizar clases de dibujo ya preparadas suministradas con MetaTrader 5 o disponibles en la base de código mql5.com. En concreto, eche un vistazo al archivo *MQL5/Include/Canvas/Canvas.mqh*.

7.1.9 Fuentes y salida de texto a recursos gráficos

Además de renderizar píxeles individuales en un array de un recurso gráfico, podemos utilizar funciones integradas para mostrar texto. Las funciones permiten cambiar la fuente actual y sus características (*TextSetFont*), obtener las dimensiones del rectángulo en el que se puede inscribir la cadena dada (*TextGetSize*), así como insertar directamente el pie de foto en la imagen generada (*TextOut*).

bool TextSetFont(const string name, int size, uint flags, int orientation = 0)

La función establece la fuente y sus características para el posterior dibujo de texto en el búfer de imagen mediante la función *TextOut* (véase más adelante). El parámetro *name* puede contener el nombre de una fuente integrada de Windows o un archivo de fuentes ttf (TrueType Font) conectado por la directiva de recursos (si el nombre empieza por «::»).

El tamaño (*size*) puede especificarse en puntos (unidad de medida tipográfica) o en píxeles (puntos de pantalla). Los valores positivos significan que la unidad de medida es un píxel, y los valores negativos se miden en décimas de punto. La altura en píxeles tendrá un aspecto diferente para los usuarios en función de las capacidades técnicas y los ajustes de sus monitores. La altura en puntos será aproximadamente («a ojo») la misma para todos.

> Un punto tipográfico es una unidad física de longitud, tradicionalmente igual a 1/72 de pulgada. Por tanto, 1 punto equivale a 0.352778 milímetros. Un píxel en la pantalla es una medida virtual de longitud. Su tamaño físico depende de la resolución de hardware de la pantalla. Por ejemplo, con una densidad de pantalla de 96 PPP (puntos por pulgada), 1 píxel ocupará 0.264583 milímetros o 0.75 puntos. Sin embargo, la mayoría de las pantallas modernas tienen valores de PPP mucho más altos y, por tanto, píxeles más pequeños. Por ello, los sistemas operativos, incluido Windows, disponen desde hace tiempo de ajustes para aumentar la escala visible de los elementos de la interfaz. Así, si especifica un tamaño en puntos (valores negativos), el tamaño del texto en píxeles dependerá de la configuración de visualización y escala del sistema operativo (por ejemplo, «estándar» 100 %, «medio» 125 % o «grande» 150 %).
>
> El zoom hace que los píxeles visualizados sean ampliados artificialmente por el sistema. Esto equivale a reducir el tamaño de la pantalla en píxeles, y el sistema aplica los PPP efectivos para conseguir el mismo tamaño físico. Si el escalado está habilitado, entonces son los PPP efectivos los que se notifican a los programas, incluyendo el terminal y después los programas MQL. Si es necesario, puede averiguar los PPP de la pantalla a partir de la propiedad TERMINAL_SCREEN_DPI (véase Especificaciones de la pantalla). Sin embargo, en realidad, al establecer el tamaño de la fuente en puntos, nos libramos de la necesidad de recalcular su tamaño en función de los PPP, ya que el sistema lo hará por nosotros.

La fuente por defecto es Arial y el tamaño por defecto es -120 (12 pt). Los controles, en particular los objetos integrados en los gráficos, también funcionan con tamaños de fuente en puntos. Por ejemplo, si en un programa MQL quiere dibujar un texto del mismo tamaño que el texto del objeto OBJ_LABEL, que tiene un tamaño de 10 puntos, debe utilizar el parámetro *size* igual a -100.

El parámetro *flags* establece una combinación de banderas que describen el estilo de la fuente. La combinación está formada por una máscara de bits que utiliza el operador OR ('|'). Las banderas se dividen en dos grupos: banderas de estilos y banderas de negritas.

En la tabla siguiente se enumeran las banderas de estilos. Se pueden mezclar.

Bandera	Descripción
FONT_ITALIC	Cursiva
FONT_UNDERLINE	Subrayado
FONT_STRIKEOUT	Tachado

Las banderas de negritas tienen pesos relativos correspondientes (dados para comparar los efectos esperados).

Bandera	Descripción
FW_DONTCARE	0 (se aplicará el valor predeterminado del sistema)
FW_THIN	100
FW_EXTRALIGHT, FW_ULTRALIGHT	200
FW_LIGHT	300
FW_NORMAL, FW_REGULAR	400
FW_MEDIUM	500
FW_SEMIBOLD, FW_DEMIBOLD	600
FW_BOLD	700
FW_EXTRABOLD, FW_ULTRABOLD	800
FW_HEAVY, FW_BLACK	900

Utilice solamente uno de estos valores en una combinación de banderas.

El parámetro *orientation* especifica el ángulo del texto con respecto a la horizontal, en décimas de grado. Por ejemplo, orientación = 0 significa salida de texto normal, mientras que orientación = 450 dará lugar a una inclinación de 45 grados (en sentido contrario a las agujas del reloj).

Tenga en cuenta que los ajustes realizados en una llamada a *TextSetFont* afectarán a todas las llamadas posteriores a *TextOut* hasta que se modifiquen.

La función devuelve *true* si tiene éxito, o *false* si se producen problemas (por ejemplo, si no se encuentra la fuente).

Consideraremos un ejemplo de utilización de esta función, así como de las otras dos después de describirlas todas.

bool TextGetSize(const string text, uint &width, uint &height)

La función devuelve la anchura y la altura de la línea con la configuración de fuente actual (puede ser la fuente predeterminada o la especificada en la llamada anterior a *TextSetFont*).

El parámetro *text* pasa una cadena en la que se requiere la longitud y la anchura en píxeles. La función escribe los valores de las dimensiones basándose en las referencias de los parámetros *width* y *height*.

Debe tenerse en cuenta que la rotación (inclinación) del texto mostrado especificada por el parámetro *orientation* cuando se llama a *TextSetFont* no afecta al tamaño de ninguna manera. En otras palabras, si el texto debe girar 45 grados, el propio programa MQL debe calcular el cuadrado mínimo en el que cabe el texto. La función *TextGetSize* calcula el tamaño del texto en una posición estándar (horizontal).

bool TextOut(const string text, int x, int y, uint anchor, uint &data[], uint width, uint height, uint color, ENUM_COLOR_FORMAT color_format)

La función dibuja texto en el búfer gráfico en las coordenadas especificadas teniendo en cuenta el color, el formato y los ajustes previos (fuente, estilo y orientación).

El texto se pasa en el parámetro *text* y debe tener la forma de una línea.

Las coordenadas *x* y *y* especificadas en píxeles definen el punto del búfer gráfico donde se muestra el texto. El lugar de la inscripción generada en el punto (x, y) depende del método de vinculación en el parámetro *anchor* (véase más adelante).

El búfer está representado por el array *data*, y aunque el array es unidimensional, almacena un «lienzo» bidimensional con dimensiones de *width* x *height* puntos. Este array se puede obtener de la función *ResourceReadImage*, o asignado por un programa MQL. Una vez finalizadas todas las operaciones de edición, incluida la salida de texto, deberá crear un nuevo recurso basado en este búfer o aplicarlo a un recurso ya existente. En ambos casos, debe llamar a *ResourceCreate*.

El color del texto y la forma en que se maneja el color se establecen mediante los parámetros *color* y *color_format* (véase ENUM_COLOR_FORMAT). Tenga en cuenta que el tipo utilizado para el color es *uint*, es decir, para transmitir el *color*, debe convertirlo utilizando *ColorToARGB*.

El método de anclaje especificado por el parámetro anchor es una combinación de dos banderas de posición de texto: vertical y horizontal.

Las banderas de posición de texto en horizontal son:

- TA_LEFT - punto de anclaje en el lado izquierdo del rectángulo delimitador
- TA_CENTER - punto de anclaje en el centro entre los lados izquierdo y derecho del rectángulo
- TA_RIGHT - punto de anclaje en el lado derecho del rectángulo delimitador

Las banderas de posición vertical del texto son:

- TA_TOP - punto de anclaje en el lado superior del rectángulo delimitador
- TA_VCENTER - punto de anclaje en el centro entre la parte superior e inferior del rectángulo
- TA_BOTTOM - punto de anclaje en el lado inferior del rectángulo delimitador

En total, hay 9 combinaciones válidas de banderas para describir el método de anclaje.

Posición del texto de salida en relación con el punto de anclaje

Aquí, el centro de la imagen contiene un punto grande deliberadamente exagerado en la imagen generada con coordenadas (x, y). En función de las banderas, el texto aparece en relación con este punto en las posiciones especificadas (el contenido del texto corresponde al método de anclaje aplicado).

Para facilitar la consulta, todas las inscripciones se han realizado en la posición horizontal estándar. Sin embargo, nótese que también podría aplicarse un ángulo a cualquiera de ellos (*orientation*), y entonces la inscripción correspondiente giraría alrededor del punto. En esta imagen solo se gira la etiqueta centrada en ambas dimensiones.

Estas banderas no deben confundirse con la alineación del texto. El cuadro delimitador siempre tiene el tamaño adecuado para el texto, y su posición respecto al punto de anclaje es, en cierto sentido, la opuesta a la de los nombres de las banderas.

Veamos algunos ejemplos utilizando tres funciones.

Para empezar, vamos a comprobar las opciones más sencillas de configuración del estilo y la negrita de la fuente. El script *ResourceText.mq5* permite seleccionar el nombre de la fuente y su tamaño, así como los colores del fondo y del texto en las variables de entrada. Las etiquetas se mostrarán en el gráfico durante el número de segundos especificado.

```
input string Font = "Arial";             // Font Name
input int    Size = -240;                // Size
input color  Color = clrBlue;            // Font Color
input color  Background = clrNONE;       // Background Color
input uint   Seconds = 10;               // Demo Time (seconds)
```

El nombre de cada gradación de negrita se mostrará en el texto de la etiqueta, por lo que para simplificar el proceso (mediante *EnumToString*) se declara la enumeración ENUM_FONT_WEIGHTS.

```
enum ENUM_FONT_WEIGHTS
{
   _DONTCARE = FW_DONTCARE,
   _THIN = FW_THIN,
   _EXTRALIGHT = FW_EXTRALIGHT,
   _LIGHT = FW_LIGHT,
   _NORMAL = FW_NORMAL,
   _MEDIUM = FW_MEDIUM,
   _SEMIBOLD = FW_SEMIBOLD,
   _BOLD = FW_BOLD,
   _EXTRABOLD = FW_EXTRABOLD,
   _HEAVY = FW_HEAVY,
};

const int nw = 10; // number of different weights
```

Las banderas de inscripción se recogen en el array *rendering* y de ella se seleccionan combinaciones aleatorias.

```
   const uint rendering[] =
   {
      FONT_ITALIC,
      FONT_UNDERLINE,
      FONT_STRIKEOUT
   };
   const int nr = sizeof(rendering) / sizeof(uint);
```

Para obtener un número aleatorio en un rango existe una función auxiliar *Random*.

```
int Random(const int limit)
{
   return rand() % limit;
}
```

En la función principal del script, encontramos el tamaño del gráfico y creamos un objeto OBJ_BITMAP_LABEL que abarca todo el espacio.

```
void OnStart()
{
   ...
   const string name = "FONT";
   const int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
   const int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS);

   // object for a resource with a picture filling the whole window
   ObjectCreate(0, name, OBJ_BITMAP_LABEL, 0, 0, 0);
   ObjectSetInteger(0, name, OBJPROP_XSIZE, w);
   ObjectSetInteger(0, name, OBJPROP_YSIZE, h);
   ...
```

A continuación, asignamos memoria al búfer de imagen, lo rellenamos con el color de fondo especificado (o lo dejamos transparente, por defecto), creamos un recurso basado en el búfer y lo vinculamos al objeto.

```
   uint data[];
   ArrayResize(data, w * h);
   ArrayInitialize(data, Background == clrNONE ? 0 : ColorToARGB(Background));
   ResourceCreate(name, data, w, h, 0, 0, w, COLOR_FORMAT_ARGB_RAW);
   ObjectSetString(0, name, OBJPROP_BMPFILE, "::" + name);
   ...
```

Por si acaso, tenga en cuenta que podemos establecer la propiedad OBJPROP_BMPFILE sin modificador (0 o 1) en la llamada a *ObjectSetString*, a menos que el objeto deba cambiar entre dos estados.

Todos los pesos de las fuentes se enumeran en el array *weights*.

```
const uint weights[] =
{
   FW_DONTCARE,
   FW_THIN,
   FW_EXTRALIGHT, // FW_ULTRALIGHT,
   FW_LIGHT,
   FW_NORMAL,     // FW_REGULAR,
   FW_MEDIUM,
   FW_SEMIBOLD,   // FW_DEMIBOLD,
   FW_BOLD,
   FW_EXTRABOLD,  // FW_ULTRABOLD,
   FW_HEAVY,      // FW_BLACK
};
const int nw = sizeof(weights) / sizeof(uint);
```

En el bucle, en orden, establecemos la siguiente gradación de negrita para cada línea utilizando *TextSetFont*, preseleccionando un estilo aleatorio. Una descripción de la fuente, incluyendo su nombre y peso, se dibuja en el búfer utilizando *TextOut*.

```
const int step = h / (nw + 2);
int cursor = 0;    // Y coordinate of the current "text line"

for(int weight = 0; weight < nw; ++weight)
{
   // apply random style
   const int r = Random(8);
   uint render = 0;
   for(int j = 0; j < 3; ++j)
   {
      if((bool)(r & (1 << j))) render |= rendering[j];
   }
   TextSetFont(Font, Size, weights[weight] | render);

   // generate font description
   const string text = Font + EnumToString((ENUM_FONT_WEIGHTS)weights[weight]);

   // draw text on a separate "line"
   cursor += step;
   TextOut(text, w / 2, cursor, TA_CENTER | TA_TOP, data, w, h,
      ColorToARGB(Color), COLOR_FORMAT_ARGB_RAW);
}
...
```

Actualice ahora el recurso y el gráfico.

```
ResourceCreate(name, data, w, h, 0, 0, w, COLOR_FORMAT_ARGB_RAW);
ChartRedraw();
...
```

El usuario puede detener la demostración con antelación.

```
   const uint timeout = GetTickCount() + Seconds * 1000;
   while(!IsStopped() && GetTickCount() < timeout)
   {
      Sleep(1000);
   }
```

Por último, el script elimina el recurso y el objeto.

```
   ObjectDelete(0, name);
   ResourceFree("::" + name);
}
```

El resultado del script se muestra en la siguiente imagen.

Dibujar texto en diferentes pesos y estilos

En el segundo ejemplo de *ResourceFont.mq5* haremos la tarea más difícil incluyendo una fuente personalizada como recurso y utilizando la rotación del texto en incrementos de 90 grados.

El archivo de fuentes se encuentra junto al script.

```
#resource "a_LCDNova3DCmObl.ttf"
```

El mensaje puede modificarse en el parámetro de entrada.

```
input string Message = "Hello world!";    // Message
```

Esta vez, OBJ_BITMAP_LABEL no ocupará toda la ventana y, por tanto, estará centrado tanto horizontal como verticalmente.

```
void OnStart()
{
   const string name = "FONT";
   const int w = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS);
   const int h = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS);

   // object for a resource with a picture
   ObjectCreate(0, name, OBJ_BITMAP_LABEL, 0, 0, 0);
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, w / 2);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, h / 2);
   ObjectSetInteger(0, name, OBJPROP_ANCHOR, ANCHOR_CENTER);
   ...
```

Para empezar, se asigna el búfer del tamaño mínimo, justo para completar la creación de recursos. Más adelante lo ampliaremos para ajustarlo a las dimensiones de la inscripción, para lo que existen las variables reservadas *width* y *height*.

```
   uint data[], width, height;
   ArrayResize(data, 1);
   ResourceCreate(name, data, 1, 1, 0, 0, 1, COLOR_FORMAT_ARGB_RAW);
   ObjectSetString(0, name, OBJPROP_BMPFILE, "::" + name);
   ...
```

En un bucle con la cuenta atrás del tiempo de prueba necesitamos cambiar la orientación de la inscripción, para lo cual existe la variable *angle* (en ella se desplazarán los grados). La orientación cambiará una vez por segundo, la cuenta está en la variable *remain*.

```
   const uint timeout = GetTickCount() + Seconds * 1000;
   int angle = 0;
   int remain = 10;
   ...
```

En el bucle, cambiamos constantemente la rotación del texto, y en el propio texto, mostramos un contador de segundos. Para cada nueva inscripción se calcula su tamaño mediante *TextGetSize*, en función del cual se reasigna el búfer.

```
   while(!IsStopped() && GetTickCount() < timeout)
   {
      // apply new angle
      TextSetFont("::a_LCDNova3DCmObl.ttf", -240, 0, angle * 10);

      // form the text
      const string text = Message + " (" + (string)remain-- + ")";

      // get the text size, allocate the array
      TextGetSize(text, width, height);
      ArrayResize(data, width * height);
      ArrayInitialize(data, 0);            // transparency

      // for vertical orientation, swap sizes
      if((bool)(angle / 90 & 1))
      {
         const uint t = width;
         width = height;
         height = t;
      }

      // adjust the size of the object
      ObjectSetInteger(0, name, OBJPROP_XSIZE, width);
      ObjectSetInteger(0, name, OBJPROP_YSIZE, height);

      // draw text
      TextOut(text, width / 2, height / 2, TA_CENTER | TA_VCENTER, data, width, heigh
         ColorToARGB(clrBlue), COLOR_FORMAT_ARGB_RAW);

      // update resource and chart
      ResourceCreate(name, data, width, height, 0, 0, width, COLOR_FORMAT_ARGB_RAW);
      ChartRedraw();

      // change angle
      angle += 90;

      Sleep(100);
   }
   ...
```

Tenga en cuenta que, si el texto es vertical, habrá que intercambiar las dimensiones. En términos más generales, con el texto girado en un ángulo arbitrario, se necesitaban más cálculos matemáticos para conseguir que el tamaño del búfer se ajustara a todo el texto.

Al final eliminamos también el objeto y el recurso.

```
   ObjectDelete(0, name);
   ResourceFree("::" + name);
}
```

Uno de los momentos de la ejecución del script se muestra en la siguiente captura de pantalla:

Inscripción con fuente personalizada

Como ejemplo final, echemos un vistazo al script *ResourceTextAnchOrientation.mq5* mostrando varias rotaciones y puntos de anclaje del texto.

El script genera el número especificado de etiquetas (*ExampleCount*) utilizando la fuente especificada.

```
input string Font = "Arial";              // Font Name
input int    Size = -150;                 // Size
input int    ExampleCount = 11;           // Number of examples
```

Los puntos de anclaje y las rotaciones se eligen aleatoriamente.

Para especificar los nombres de los puntos de anclaje en las etiquetas, existe la enumeración ENUM_TEXT_ANCHOR con todas las opciones válidas declaradas. Así, podemos llamar simplemente *EnumToString* a cualquier elemento seleccionado al azar.

```
enum ENUM_TEXT_ANCHOR
{
   LEFT_TOP = TA_LEFT | TA_TOP,
   LEFT_VCENTER = TA_LEFT | TA_VCENTER,
   LEFT_BOTTOM = TA_LEFT | TA_BOTTOM,
   CENTER_TOP = TA_CENTER | TA_TOP,
   CENTER_VCENTER = TA_CENTER | TA_VCENTER,
   CENTER_BOTTOM = TA_CENTER | TA_BOTTOM,
   RIGHT_TOP = TA_RIGHT | TA_TOP,
   RIGHT_VCENTER = TA_RIGHT | TA_VCENTER,
   RIGHT_BOTTOM = TA_RIGHT | TA_BOTTOM,
};
```

En el manejador *OnStart* se declara un array de estas nuevas constantes.

```
void OnStart()
{
   const ENUM_TEXT_ANCHOR anchors[] =
   {
      LEFT_TOP,
      LEFT_VCENTER,
      LEFT_BOTTOM,
      CENTER_TOP,
      CENTER_VCENTER,
      CENTER_BOTTOM,
      RIGHT_TOP,
      RIGHT_VCENTER,
      RIGHT_BOTTOM,
   };
   const int na = sizeof(anchors) / sizeof(uint);
   ...
```

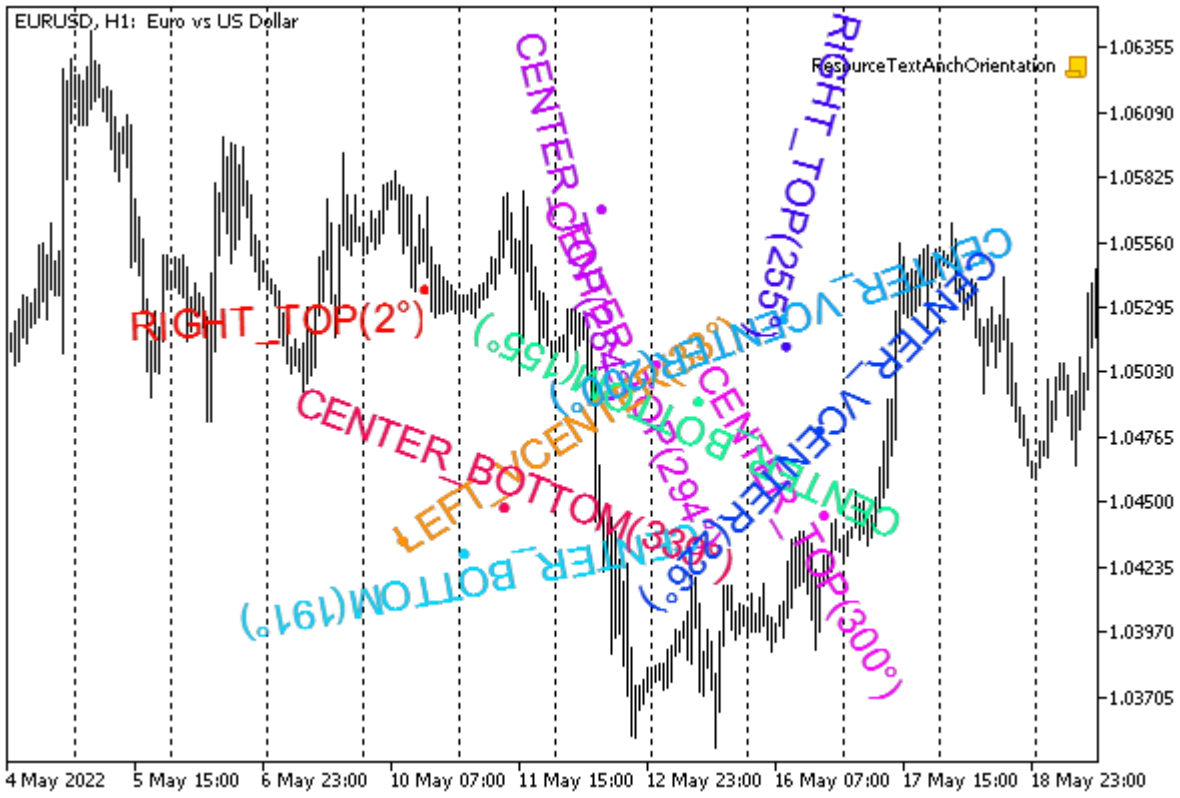
La creación inicial de objetos y recursos es similar a la del ejemplo con *ResourceText.mq5*, así que vamos a omitirlos aquí. Lo más interesante ocurre en el bucle:

```
   for(int i = 0; i < ExampleCount; ++i)
   {
      // apply a random angle
      const int angle = Random(360);
      TextSetFont(Font, Size, 0, angle * 10);

      // take random coordinates and an anchor point
      const ENUM_TEXT_ANCHOR anchor = anchors[Random(na)];
      const int x = Random(w / 2) + w / 4;
      const int y = Random(h / 2) + h / 4;
      const color clr = ColorMix::HSVtoRGB(angle);

     // draw a circle directly in that place of the image,
     // where the anchor point goes
      TextOut(ShortToString(0x2022), x, y, TA_CENTER | TA_VCENTER, data, w, h,
         ColorToARGB(clr), COLOR_FORMAT_ARGB_NORMALIZE);

      // form the text describing the anchor type and angle
      const string text =  EnumToString(anchor) +
         "(" + (string)angle + CharToString(0xB0) + ")";

      // draw text
      TextOut(text, x, y, anchor, data, w, h,
         ColorToARGB(clr), COLOR_FORMAT_ARGB_NORMALIZE);
   }
   ...
```

Solo queda actualizar la imagen y el gráfico, y luego esperar la orden del usuario y liberar recursos.

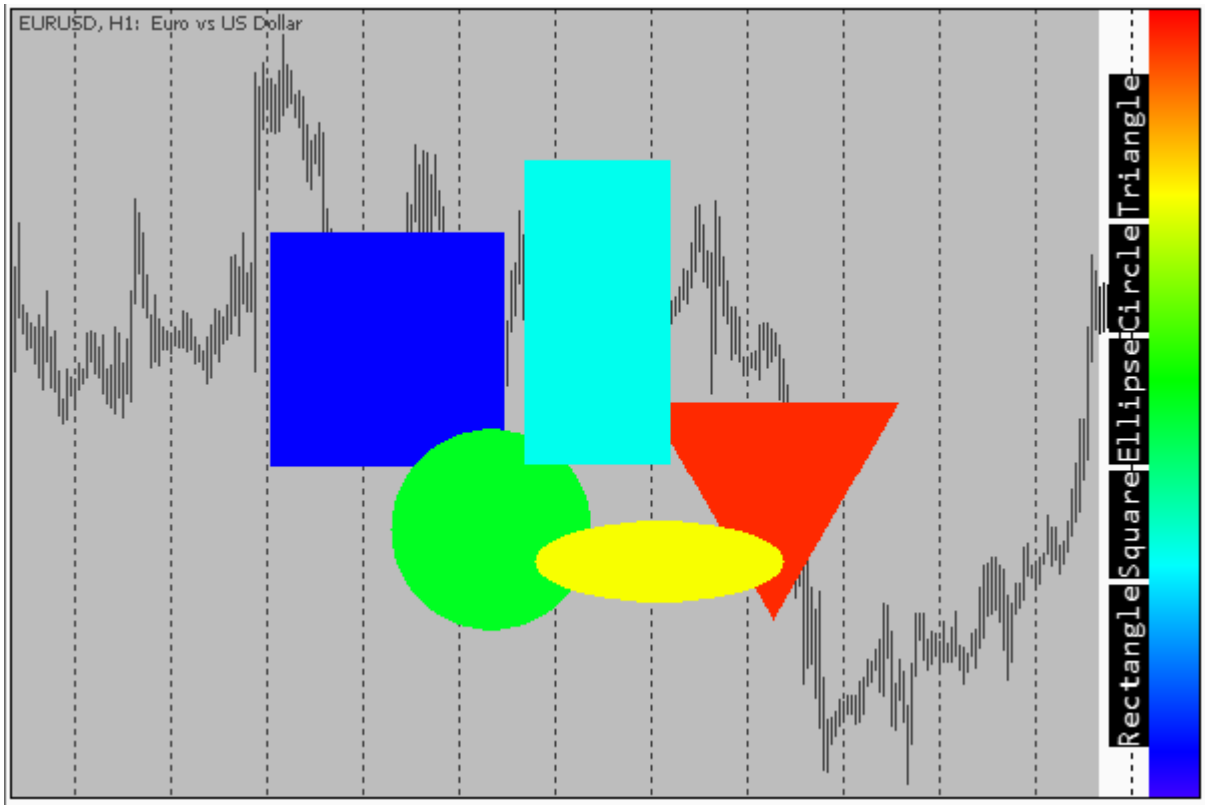
```
      ResourceCreate(name, data, w, h, 0, 0, w, COLOR_FORMAT_ARGB_NORMALIZE);
      ChartRedraw();

      const uint timeout = GetTickCount() + Seconds * 1000;
      while(!IsStopped() && GetTickCount() < timeout)
      {
         Sleep(1000);
      }

      ObjectDelete(0, name);
      ResourceFree("::" + name);
   }
```

Esto es lo que obtenemos como resultado:



Salida de texto con coordenadas, puntos de anclaje y ángulos aleatorios

Además, para un estudio independiente, el libro proporciona un editor de gráficos de juguete *SimpleDrawing.mq5* diseñado como un indicador sin búfer y que utiliza en su trabajo las clases de formas consideradas anteriormente (véase el ejemplo con *ResourceShapesDraw.mq5*). Se colocan en el archivo de encabezado *ShapesDrawing.mqh* casi sin cambios. Anteriormente, las formas eran generadas aleatoriamente por el script. Ahora el usuario puede seleccionarlas y trazarlas en el gráfico. Para ello, se ha implementado una interfaz con una paleta de colores y una barra de botones en función del número de clases de formas registradas. La interfaz está implementada por la clase *SimpleDrawing* (*SimpleDrawing.mqh*).



Editor gráfico sencillo

El panel y la paleta pueden colocarse a lo largo de cualquier borde del gráfico, lo que demuestra la posibilidad de girar las etiquetas.

La selección de la siguiente forma que se va a dibujar se realiza pulsando el botón en el panel: el botón «se pega» en el estado pulsado, y su color de fondo indica el color de dibujo seleccionado. Para cambiar el color, haga clic en cualquier lugar de la paleta.

Cuando se selecciona uno de los tipos de forma en el panel (uno de los botones está «activo»), al hacer clic en el área de dibujo (el resto del gráfico, indicado por el sombreado) se dibuja una forma de tamaño predefinido en ese lugar. En ese momento, el botón se «apaga». En este estado, cuando todos los botones están inactivos, puede mover las formas por el espacio de trabajo utilizando el ratón. Si mantenemos pulsada la tecla *Ctrl*, la forma se redimensiona en lugar de moverse. El «punto caliente» se sitúa en el centro de cada forma (el tamaño del área sensible se establece mediante una macro en el código fuente y probablemente habrá que aumentarlo para pantallas con PPP muy elevados).

Tenga en cuenta que el editor incluye el ID de trazado (*ChartID*) en los nombres de los recursos generados. Esto permite ejecutar el editor en paralelo en varios gráficos.

7.1.10 Aplicación de recursos gráficos en trading

Por supuesto, embellecer no es el objetivo principal de los recursos. Veamos cómo crear una herramienta útil basada en ellos. También eliminaremos una omisión más: hasta ahora solo hemos utilizado recursos dentro de objetos OBJ_BITMAP_LABEL, que se posicionan en coordenadas de pantalla. Sin embargo, los recursos gráficos también pueden incrustarse en objetos OBJ_BITMAP con referencia a coordenadas de cotización: precios y hora.

Anteriormente en el libro hemos visto el indicador *IndDeltaVolume.mq5* que calcula el volumen delta (tick o real) para cada barra. Además de esta representación del volumen delta, existe otra no menos

popular entre los usuarios: el perfil de mercado. Se trata de la distribución de volúmenes en el contexto de los niveles de precios. Dicho histograma puede construirse para toda la ventana, para una profundidad determinada (por ejemplo, dentro de un día) o para una sola barra.

Es la última opción la que aplicamos en forma de nuevo indicador *DeltaVolumeProfile.mq5*. Ya hemos considerado los principales detalles técnicos de la solicitud del historial de ticks en el marco del indicador anterior, por lo que ahora nos centraremos principalmente en el componente gráfico.

La bandera *ShowSplittedDelta* en la variable de entrada controlará cómo se muestran los volúmenes: desglosados por direcciones de compra/venta o colapsados.

```
input bool ShowSplittedDelta = true;
```

No habrá búferes en el indicador. Calculará y mostrará un histograma para una barra específica a petición del usuario, y en concreto, haciendo clic en dicha barra. Por lo tanto, utilizaremos el manejador *OnChartEvent*. En este manejador, obtenemos las coordenadas de la pantalla, las recalculamos en precio y tiempo, y llamamos a alguna función de ayuda *RequestData*, que inicia el cálculo.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   if(id == CHARTEVENT_CLICK)
   {
      datetime time;
      double price;
      int window;
      ChartXYToTimePrice(0, (int)lparam, (int)dparam, window, time, price);
      time += PeriodSeconds() / 2;
      const int b = iBarShift(_Symbol, _Period, time, true);
      if(b != -1 && window == 0)
      {
         RequestData(b, iTime(_Symbol, _Period, b));
      }
   }
   ...
}
```

Para rellenarlo, necesitamos la clase *DeltaVolumeProfile*, que está construida para ser similar a la clase *CalcDeltaVolume* de *IndDeltaVolume.mq5*.

La nueva clase describe variables que tienen en cuenta el método de cálculo del volumen (*tickType*), el tipo de precio sobre el que se construye el gráfico (*barType*), el modo de la variable de entrada *ShowSplittedDelta* (se colocará en una variable miembro *delta*), así como un prefijo para los objetos generados en el gráfico.

```
class DeltaVolumeProfile
{
   const COPY_TICKS tickType;
   const ENUM_SYMBOL_CHART_MODE barType;
   const bool delta;

   static const string prefix;
   ...
public:
   DeltaVolumeProfile(const COPY_TICKS type, const bool d) :
      tickType(type), delta(d),
      barType((ENUM_SYMBOL_CHART_MODE)SymbolInfoInteger(_Symbol, SYMBOL_CHART_MODE))
   {
   }

   ~DeltaVolumeProfile()
   {
      ObjectsDeleteAll(0, prefix, 0); // TODO: delete resources
   }
   ...
};

static const string DeltaVolumeProfile::prefix = "DVP";

DeltaVolumeProfile deltas(TickType, ShowSplittedDelta);
```

La dirección *tick type* puede cambiarse al valor TRADE_TICKS solo para los instrumentos de trading para los que se dispone de volúmenes reales. De manera predeterminada, está activado el modo INFO_TICKS, que funciona en todos los instrumentos.

Los ticks de una barra concreta se solicitan mediante el método *createProfileBar*.

```
int createProfileBar(const int i)
{
   MqlTick ticks[];
   const datetime time = iTime(_Symbol, _Period, i);
   // prev and next - time limits of the bar
   const datetime prev = time;
   const datetime next = prev + PeriodSeconds();
   ResetLastError();
   const int n = CopyTicksRange(_Symbol, ticks, COPY_TICKS_ALL,
      prev * 1000, next * 1000 - 1);
   if(n > -1 && _LastError == 0)
   {
      calcProfile(i, time, ticks);
   }
   else
   {
      return -_LastError;
   }
   return n;
}
```

El análisis directo de los ticks y el cálculo de los volúmenes se realiza en el método protegido *calcProfile*. En él, en primer lugar, averiguamos el rango de precios de la barra y su tamaño en píxeles.

```
void calcProfile(const int b, const datetime time, const MqlTick &ticks[])
{
   const string name = prefix + (string)(ulong)time;
   const double high = iHigh(_Symbol, _Period, b);
   const double low = iLow(_Symbol, _Period, b);
   const double range = high - low;

   ObjectCreate(0, name, OBJ_BITMAP, 0, time, high);

   int x1, y1, x2, y2;
   ChartTimePriceToXY(0, 0, time, high, x1, y1);
   ChartTimePriceToXY(0, 0, time, low, x2, y2);

   const int h = y2 - y1 + 1;
   const int w = (int)(ChartGetInteger(0, CHART_WIDTH_IN_PIXELS)
      / ChartGetInteger(0, CHART_WIDTH_IN_BARS));
   ...
```

Basándonos en esta información, creamos un objeto OBJ_BITMAP, asignamos un array para la imagen y creamos un recurso. El fondo de toda la imagen está vacío (transparente). Cada objeto está anclado por el punto medio superior al precio *High* de su barra y tiene una anchura de una barra.

```
uint data[];
ArrayResize(data, w * h);
ArrayInitialize(data, 0);
ResourceCreate(name + (string)ChartID(), data, w, h, 0, 0, w, COLOR_FORMAT_ARGB

ObjectSetString(0, name, OBJPROP_BMPFILE, "::" + name + (string)ChartID());
ObjectSetInteger(0, name, OBJPROP_XSIZE, w);
ObjectSetInteger(0, name, OBJPROP_YSIZE, h);
ObjectSetInteger(0, name, OBJPROP_ANCHOR, ANCHOR_UPPER);
...
```

A continuación se calculan los volúmenes en ticks del array pasado. El número de niveles de precios es igual a la altura de la barra en píxeles (*h*). Suele ser inferior al rango de precios en puntos, por lo que los píxeles actúan como una especie de cesta para calcular las estadísticas. Si en un marco temporal pequeño, el rango de puntos es menor que el tamaño en píxeles, el histograma será visualmente escaso. Los volúmenes de compras y ventas se acumulan por separado en los arrays *plus* y *minus*.

```
long plus[], minus[], max = 0;
ArrayResize(plus, h);
ArrayResize(minus, h);
ArrayInitialize(plus, 0);
ArrayInitialize(minus, 0);

const int n = ArraySize(ticks);
for(int j = 0; j < n; ++j)
{
   const double p1 = price(ticks[j]); // returns Bid or Last
   const int index = (int)((high - p1) / range * (h - 1));
   if(tickType == TRADE_TICKS)
   {
      // if real volumes are available, we can take them into account
      if((ticks[j].flags & TICK_FLAG_BUY) != 0)
      {
         plus[index] += (long)ticks[j].volume;
      }
      if((ticks[j].flags & TICK_FLAG_SELL) != 0)
      {
         minus[index] += (long)ticks[j].volume;
      }
   }
   else // tickType == INFO_TICKS or tickType == ALL_TICKS
   if(j > 0)
   {
     // if there are no real volumes,
     // price movement up/down is an estimate of the volume type
      if((ticks[j].flags & (TICK_FLAG_ASK | TICK_FLAG_BID)) != 0)
      {
         const double d = (((ticks[j].ask + ticks[j].bid)
                        - (ticks[j - 1].ask + ticks[j - 1].bid)) / _Point);
         if(d > 0) plus[index] += (long)d;
         else minus[index] -= (long)d;
      }
   }
   ...
```

Para normalizar el histograma, buscamos el valor máximo.

```
         if(delta)
         {
            if(plus[index] > max) max = plus[index];
            if(minus[index] > max) max = minus[index];
         }
         else
         {
            if(fabs(plus[index] - minus[index]) > max)
               max = fabs(plus[index] - minus[index]);
         }
      }
      ...
```

Por último, las estadísticas resultantes se envían al búfer gráfico *data* y se envían al recurso. Los volúmenes de compra aparecen en azul y los de venta, en rojo. Si el modo neto está activado, el importe aparece en verde.

```
      for(int i = 0; i < h; i++)
      {
         if(delta)
         {
            const int dp = (int)(plus[i] * w / 2 / max);
            const int dm = (int)(minus[i] * w / 2 / max);
            for(int j = 0; j < dp; j++)
            {
               data[i * w + w / 2 + j] = ColorToARGB(clrBlue);
            }
            for(int j = 0; j < dm; j++)
            {
               data[i * w + w / 2 - j] = ColorToARGB(clrRed);
            }
         }
         else
         {
            const int d = (int)((plus[i] - minus[i]) * w / 2 / max);
            const int sign = d > 0 ? +1 : -1;
            for(int j = 0; j < fabs(d); j++)
            {
               data[i * w + w / 2 + j * sign] = ColorToARGB(clrGreen);
            }
         }
      }
      ResourceCreate(name + (string)ChartID(), data, w, h, 0, 0, w, COLOR_FORMAT_ARGB
   }
```

Ahora podemos volver a la función *RequestData*: su tarea es llamar al método *createProfileBar* y gestionar los errores (si los hay).

```
void RequestData(const int b, const datetime time, const int count = 0)
{
   Comment("Requesting ticks for ", time);
   if(deltas.createProfileBar(b) <= 0)
   {
      Print("No data on bar ", b, ", at ", TimeToString(time),
         ". Sending event for refresh...");
      ChartSetSymbolPeriod(0, _Symbol, _Period); // request to update the chart
      EventChartCustom(0, TRY_AGAIN, b, count + 1, NULL);
   }
   Comment("");
}
```

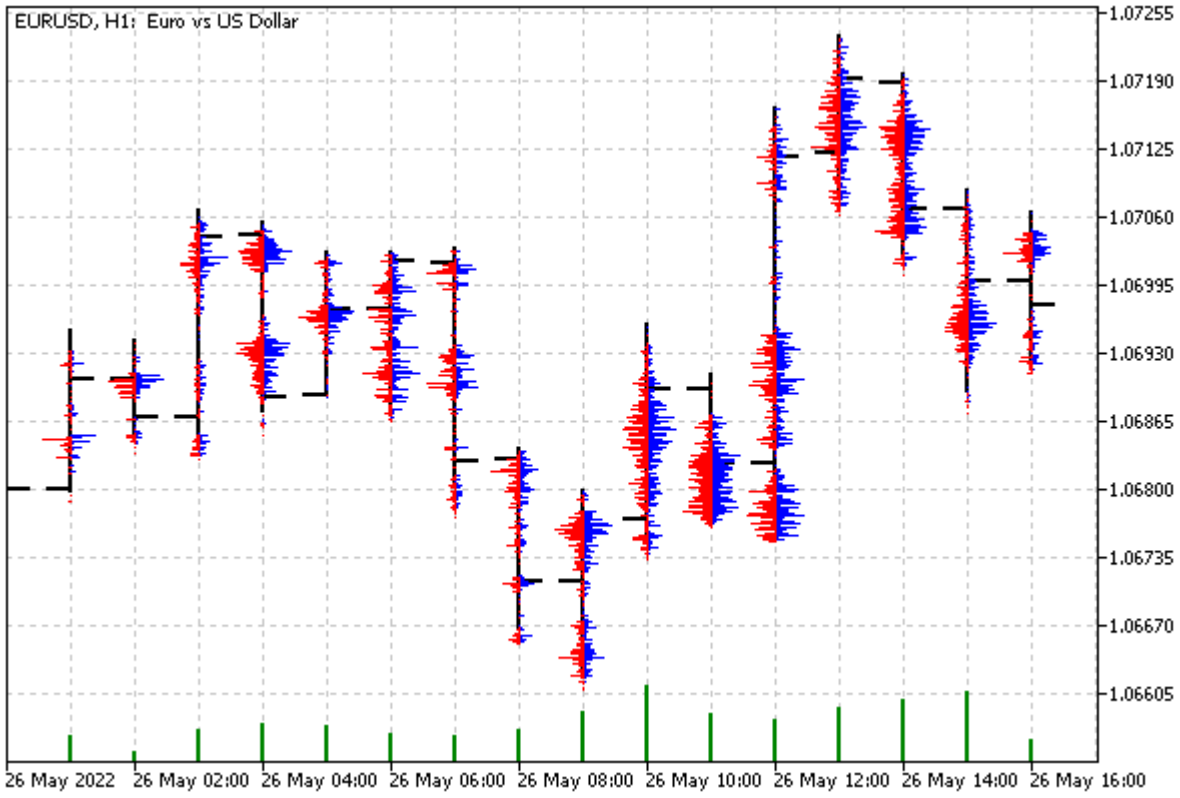
La única estrategia de gestión de errores es intentar solicitar de nuevo los ticks porque puede que no hayan tenido tiempo de cargarse. Para ello, la función envía un mensaje TRY_AGAIN personalizado al gráfico y lo procesa ella misma.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   ...
   else if(id == CHARTEVENT_CUSTOM + TRY_AGAIN)
   {
      Print("Refreshing... ", (int)dparam);
      const int b = (int)lparam;
      if((int)dparam < 5)
      {
         RequestData(b, iTime(_Symbol, _Period, b), (int)dparam);
      }
      else
      {
         Print("Give up. Check tick history manually, please, then click the bar agai
      }
   }
}
```

Repetimos este proceso no más de 5 veces, porque el historial de ticks puede tener una profundidad limitada, y no tiene sentido cargar el ordenador sin motivo.

La clase *DeltaVolumeProfile* también dispone del mecanismo para procesar el mensaje CHARTEVENT_CHART_CHANGE con el fin de redibujar los objetos existentes en caso de cambio de tamaño o escala del gráfico. Los detalles se pueden encontrar en el código fuente.

El resultado de este indicador se muestra en la siguiente imagen:



Visualización de histogramas por barras de volúmenes separados en recursos gráficos

Tenga en cuenta que los histogramas no se muestran inmediatamente después de dibujar el indicador: tiene que hacer clic en la barra para calcular su histograma.

7.2 Símbolos personalizados

Una de las características técnicas interesantes de MetaTrader 5 es el soporte para instrumentos financieros personalizados. Se trata de los símbolos que define, no el bróker en el servidor, sino el operador directamente en el terminal.

Los símbolos personalizados pueden añadirse a la lista *Observación de Mercado* junto con los símbolos estándar. Los gráficos de tales símbolos con ellos se pueden utilizar de una manera habitual.

La forma más sencilla de crear un símbolo personalizado es especificar su fórmula de cálculo en la propiedad correspondiente. Para ello, desde la interfaz del terminal, llame al menú contextual de la ventana *Observación de Mercado*, ejecute el comando *Símbolos*, vaya a la jerarquía de símbolos y a su rama *Custom* y pulse el botón *Crear símbolo*. Como resultado, se abrirá un cuadro de diálogo para establecer las propiedades del nuevo símbolo. En el mismo lugar, puede importar historial de ticks externos (pestaña *Ticks*) o cotizaciones (pestaña *Bars*) en herramientas similares, desde archivos. Esto se analiza en detalle en la documentación de MetaTrader 5.

Sin embargo, la API de MQL5 proporciona el control más completo sobre los símbolos personalizados.

Para los símbolos personalizados, la API proporciona un grupo de funciones que operan con Instrumentos financieros y Observación de Mercado. En concreto, estos símbolos pueden listarse desde el programa utilizando funciones estándar como *SymbolsTotal*, *SymbolName* y *SymbolInfo*. Ya hemos abordado brevemente esta posibilidad y ofrecido un ejemplo en la sección sobre Propiedades de

símbolos personalizados. Una característica distintiva de un símbolo personalizado es la bandera habilitada (propiedad) SYMBOL_CUSTOM.

Utilizando las funciones integradas, puede empalmar Futuros, generar series temporales aleatorias con características especificadas, emular renko, barras de igual rango, equivolumen y otros tipos de gráficos no estándar (por ejemplo, segundos marcos temporales). Además, a diferencia de la importación de archivos estáticos, los símbolos personalizados controlados por software pueden generarse en tiempo real a partir de los datos de servicios web como las bolsas de criptomonedas. La conversación sobre la integración de los programas MQL con la web aún está por llegar, pero no se puede ignorar esta posibilidad.

Un símbolo personalizado puede utilizarse fácilmente para probar estrategias en el probador o como método adicional de análisis técnico. Sin embargo, esta tecnología tiene sus limitaciones.

Dado que los símbolos personalizados se definen en el terminal y no en el servidor, no pueden negociarse en línea. En concreto, si crea un gráfico renko, las estrategias de trading basadas en él deberán adaptarse de un modo u otro para que las señales de trading y las operaciones estén realmente separadas por símbolos diferentes: usuario artificial y bróker real. Veremos un par de soluciones al problema.

Además, dado que la duración de todas las barras de un marco temporal es la misma en la plataforma, cualquier emulación de barras con periodos diferentes (Renko, equivolumen, etc.) suele basarse en el menor de los marcos temporales M1 disponibles y no proporciona una sincronización temporal completa con la realidad. En otras palabras: los ticks pertenecientes a una barra de este tipo son forzados a tener un tiempo artificial dentro de los 60 segundos, incluso si un «ladrillo» de gráfico renko o una barra de un volumen dado en realidad ha requerido mucho más tiempo para formarse. De lo contrario, si ponemos ticks en tiempo real, se formarían las siguientes barras M1, violando las reglas de renko o equivolumen. Además, hay situaciones en las que se debe crear un «ladrillo» de gráfico renko u otra barra artificial con un intervalo de tiempo inferior a 1 minuto desde la barra anterior (por ejemplo, cuando hay un aumento rápido de la volatilidad). En tales casos, será necesario cambiar la hora de las barras históricas en las cotizaciones del instrumento personalizado (desplazarlas a la izquierda «retroactivamente») o poner horas futuras en barras nuevas (lo cual es muy poco deseable). Este problema no puede resolverse de forma general en el marco de la tecnología de símbolos definidos por el usuario.

7.2.1 Crear y eliminar símbolos personalizados

Las dos primeras funciones que necesita para trabajar con símbolos personalizados son *CustomSymbolCreate* y *CustomSymbolDelete*.

bool CustomSymbolCreate(const string name, const string path = "", const string origin = NULL)

La función crea un símbolo personalizado con el nombre especificado (*name*) en el grupo especificado (*path*) y, si es necesario, con las propiedades de un símbolo ejemplar: su nombre puede especificarse en el parámetro *origin*.

El parámetro *name* debe ser un identificador simple, sin jerarquía. Si es necesario, uno o más niveles requeridos de grupos (subcarpetas) deben especificarse en el parámetro *path*, donde el carácter delimitador debe ser una barra invertida '\' (aquí la barra diagonal no se admite, a diferencia del sistema de archivos). La barra invertida debe duplicarse en las cadenas literales («\\»).

De manera predeterminada, si la cadena *path* está vacía («» o NULL), el símbolo se crea directamente en la carpeta *Custom*, que se asigna en la jerarquía general de símbolos para los símbolos de usuario. Si

la ruta está llena, se crea dentro de la carpeta *Custom* hasta el fondo (si aún no hay carpetas correspondientes).

El nombre de un símbolo, así como el de un grupo de cualquier nivel, puede contener letras y números latinos, sin signos de puntuación, espacios ni caracteres especiales. Además, sólo se permiten '.', '_', '&' y '#'.

El nombre debe ser único en toda la jerarquía de símbolos, con independencia del grupo en el que se supone que se crea el símbolo. Si ya existe un símbolo con el mismo nombre, la función devolverá *false* y establecerá el código de error 5300 (ERR_NOT_CUSTOM_SYMBOL) o 5304 (ERR_CUSTOM_SYMBOL_EXIST) en *_LastError*.

Tenga en cuenta que si el último (o incluso el único) elemento de la jerarquía en la cadena *path* coincide exactamente con *name* (distingue mayúsculas de minúsculas), entonces se trata como un nombre de símbolo que forma parte de la ruta y no como una carpeta. Por ejemplo, si el nombre y la ruta contienen las cadenas «Ejemplo» y «MQL5Book\\Example», respectivamente, entonces el símbolo «Ejemplo» se creará en la carpeta «Custom\MQL5Book\». Al mismo tiempo, si cambiamos el nombre a «ejemplo», obtendremos el símbolo «ejemplo» en la carpeta «Custom\MQL5Book\\Example».

Esta característica tiene otra consecuencia. La propiedad SYMBOL_PATH devuelve la ruta junto con el nombre del símbolo al final. Por lo tanto, si transferimos su valor sin cambios de algún símbolo ejemplar a otro recién creado, obtendremos el siguiente efecto: se creará una carpeta con el nombre del símbolo antiguo, dentro de la cual aparecerá un símbolo nuevo. Así, si desea crear un símbolo personalizado en el mismo grupo que el símbolo original, debe eliminar el nombre del símbolo original de la cadena obtenida de la propiedad SYMBOL_PATH.

Demostraremos el efecto secundario de copiar la propiedad SYMBOL_PATH en un ejemplo en la siguiente sección. No obstante, este efecto también puede utilizarse como positivo. En concreto, al crear varios de sus símbolos basándose en un símbolo original, la copia de SYMBOL_PATH garantizará que todos los símbolos nuevos se coloquen en la carpeta con el nombre del original, es decir, agrupará los símbolos según su símbolo prototipo.

La propiedad SYMBOL_PATH para símbolos personalizados comienza siempre por la carpeta «Custom\\» (este prefijo se añade automáticamente).

La longitud del nombre está limitada a 31 caracteres. Cuando se supera el límite, *CustomSymbolCreate* devuelve *false* y establece el código de error 5302 (ERR_CUSTOM_SYMBOL_NAME_LONG).

La longitud máxima de la ruta del parámetro es de 127 caracteres, incluyendo «Custom\\», los separadores de grupo «\\» y el nombre del símbolo, si se especifica al final.

El parámetro *origin* permite especificar opcionalmente el nombre del símbolo del que se copiarán las propiedades del símbolo personalizado creado. Después de crear un símbolo personalizado, puede cambiar cualquiera de sus propiedades al valor deseado utilizando las funciones apropiadas (véanse las funciones *CustomSymbolSet*).

Si se proporciona un símbolo inexistente como parámetro *origin*, el símbolo personalizado se creará «vacío», como si no se hubiera especificado el parámetro *origin*. Esto provocará el error 4301 (ERR_MARKET_UNKNOWN_SYMBOL).

En un nuevo símbolo creado «en blanco», todas las propiedades se establecen en sus valores por defecto. Por ejemplo, el tamaño del contrato es 100000, el número de dígitos del precio es 4, el cálculo del margen se realiza de acuerdo con las reglas de Forex y los gráficos se basan en los precios de *Bid*.

Si especifica *origin*, solo se transfieren las configuraciones de este símbolo al nuevo, pero no las cotizaciones ni los ticks, ya que deben generarse por separado. Esto se tratará en las secciones siguientes.

La creación de un símbolo no lo añade automáticamente a *Observación de Mercado*. Por lo tanto, esto debe hacerse explícitamente (de forma manual o mediante programación). Sin cotizaciones, la ventana del gráfico estará vacía.

bool CustomSymbolDelete(const string name)

La función borra un símbolo personalizado con el nombre especificado. No solo se borran los ajustes, sino también todos los datos del símbolo (cotizaciones y ticks). Cabe señalar que el historial no se borra inmediatamente, sino solo después de un cierto retraso, lo que puede ser una fuente de problemas si tiene la intención de volver a crear un símbolo con el mismo nombre (vamos a tocar este punto en el ejemplo de la sección Añadir, sustituir y suprimir cotizaciones).

Solo se puede borrar un símbolo personalizado. Tampoco se puede eliminar un símbolo seleccionado en *Observación de Mercado* o un símbolo que tenga un gráfico abierto. Tenga en cuenta que también se puede seleccionar un símbolo implícitamente, sin mostrarlo en la lista visible (en estos casos, la propiedad SYMBOL_VISIBLE es *false*, y la propiedad SYMBOL_SELECT es *true*). Un símbolo de este tipo debe «ocultarse» primero llamando a *SymbolSelect("name", false)* antes de intentar borrarlo: de lo contrario, obtendremos un error CUSTOM_SYMBOL_SELECTED (5306).

Si al borrar un símbolo queda una carpeta (o jerarquía de carpetas) vacía, también se borra.

Por ejemplo, vamos a crear un sencillo script *CustomSymbolCreateDelete.mq5*. En los parámetros de entrada, puede especificar un nombre, una ruta y un símbolo de ejemplo.

```
input string CustomSymbol = "Dummy";          // Custom Symbol Name
input string CustomPath = "MQL5Book\\Part7"; // Custom Symbol Folder
input string Origin;
```

En el manejador *OnStart*, vamos a comprobar si ya existe un símbolo con el nombre dado. Si no es así, tras la confirmación del usuario, crearemos dicho símbolo. Si el símbolo ya está ahí y es un símbolo personalizado, lo borraremos con el permiso del usuario (esto facilitará la limpieza una vez finalizado el experimento).

```
void OnStart()
{
   bool custom = false;
   if(!PRTF(SymbolExist(CustomSymbol, custom)))
   {
      if(IDYES == MessageBox("Create new custom symbol?", "Please, confirm", MB_YESNC
      {
         PRTF(CustomSymbolCreate(CustomSymbol, CustomPath, Origin));
      }
   }
   else
   {
      if(custom)
      {
         if(IDYES == MessageBox("Delete existing custom symbol?", "Please, confirm",
         {
            PRTF(CustomSymbolDelete(CustomSymbol));
         }
      }
      else
      {
         Print("Can't delete non-custom symbol");
      }
   }
}
```

Dos ejecuciones consecutivas con las opciones por defecto deberían dar como resultado las siguientes entradas de registro:

```
SymbolExist(CustomSymbol,custom)=false / ok
Create new custom symbol?
CustomSymbolCreate(CustomSymbol,CustomPath,Origin)=true / ok

SymbolExist(CustomSymbol,custom)=true / ok
Delete existing custom symbol?
CustomSymbolDelete(CustomSymbol)=true / ok
```

Entre una ejecución y otra puede abrir el cuadro de diálogo de símbolos en el terminal y comprobar que el símbolo personalizado correspondiente ha aparecido en la jerarquía de símbolos.

7.2.2 Propiedades de símbolos personalizados

Los símbolos personalizados tienen las mismas propiedades que los símbolos proporcionados por el bróker. Las propiedades son leídas por las funciones estándar que abordamos en el capítulo sobre instrumentos financieros.

Las propiedades de los símbolos personalizados pueden establecerse mediante un grupo especial de funciones *CustomSymbolSet*, una función para cada tipo fundamental (número entero, real, cadena).

bool CustomSymbolSetInteger(const string name, ENUM_SYMBOL_INFO_INTEGER property, long value)

bool CustomSymbolSetDouble(const string name, ENUM_SYMBOL_INFO_DOUBLE property, double value)

bool CustomSymbolSetString(const string name, ENUM_SYMBOL_INFO_STRING property, string value)

Las funciones establecidas para un símbolo personalizado denominaron *name* a un valor de *property* a *value*. Todas las propiedades existentes se agrupan en las enumeraciones ENUM_SYMBOL_INFO_INTEGER, ENUM_SYMBOL_INFO_DOUBLE, ENUM_SYMBOL_INFO_STRING, que se consideraron elemento por elemento en las secciones del capítulo mencionado.

Las funciones devuelven una indicación de éxito (*true*) o de error (*false*). Un posible problema para los errores es que no todas las propiedades pueden cambiar. Al intentar establecer una propiedad de sólo lectura, obtenemos el error CUSTOM_SYMBOL_PROPERTY_WRONG (5307). Si intenta escribir un valor no válido en la propiedad, obtendrá un error CUSTOM_SYMBOL_PARAMETER_ERROR (5308).

Tenga en cuenta que el historial de minutos y ticks de un símbolo personalizado se borra por completo si se modifica alguna de las siguientes propiedades en la especificación del símbolo:

- SYMBOL_CHART_MODE - tipo de precio utilizado para construir barras (*Bid* o *Last*)
- SYMBOL_DIGITS - número de decimales en los valores de precios
- SYMBOL_POINT - valor de un punto
- SYMBOL_TRADE_TICK_SIZE - el valor de un tick, el mínimo cambio de precio permitido.
- SYMBOL_TRADE_TICK_VALUE - costo de cambio de precio por tick (véase también SYMBOL_TRADE_TICK_VALUE_PROFIT, SYMBOL_TRADE_TICK_VALUE_LOSS)
- SYMBOL_FORMULA - fórmula para el cálculo del precio

Si un símbolo personalizado se calcula mediante una fórmula, después de borrar su historial el terminal intentará crear automáticamente un nuevo historial utilizando las propiedades actualizadas. No obstante, para los símbolos generados mediante programación, el propio programa MQL debe encargarse del recálculo.

La edición de propiedades individuales está más solicitada para modificar símbolos personalizados creados anteriormente (después de especificar el tercer parámetro *origin* en la función *CustomSymbolCreate*).

En otros casos, cambiar las propiedades en bloque puede causar efectos sutiles. La cuestión es que las propiedades están vinculadas internamente y cambiar una de ellas puede requerir un cierto estado de otras propiedades para que la operación se complete con éxito. Además, la configuración de algunas propiedades provoca cambios automáticos en otras.

En el ejemplo más sencillo, después de establecer la propiedad SYMBOL_DIGITS, verá que la propiedad SYMBOL_POINT también ha cambiado. Este es el caso menos obvio: asignar SYMBOL_CURRENCY_MARGIN o SYMBOL_CURRENCY_PROFIT no tiene ningún efecto sobre los símbolos Forex, ya que el sistema asume que los nombres de las divisas ocupan las 3 primeras y las 3 siguientes letras del nombre («XXXYYY[sufijo]»), respectivamente. Tenga en cuenta que inmediatamente después de la creación de un símbolo «vacío», por defecto se considera un símbolo Forex, y por lo tanto estas propiedades no se pueden establecer para él sin cambiar primero el mercado.

Al copiar o configurar las propiedades de los símbolos, tenga en cuenta que la plataforma implica algunas particularidades. En concreto, la propiedad SYMBOL_TRADE_CALC_MODE tiene un valor por defecto de 0 (inmediatamente después de que se cree el símbolo, pero antes de que se establezca cualquier propiedad), mientras que 0 en la enumeración ENUM_SYMBOL_CALC_MODE corresponde al miembro SYMBOL_CALC_MODE_FOREX. Al mismo tiempo, se aplican normas de denominación especiales para los símbolos de divisas de la forma XXXYYY (donde XXX e YYY son códigos de divisas) más un sufijo opcional. Por lo tanto, si no cambia SYMBOL_TRADE_CALC_MODE a otro modo requerido por adelantado, las subcadenas del nombre del símbolo especificado (el primer y segundo triple de símbolos) caerán automáticamente en las propiedades de la divisa base (SYMBOL_CURRENCY_BASE) y la divisa de los beneficios (SYMBOL_CURRENCY_PROFIT). Por ejemplo, si especifica el nombre «Dummy», se dividirá en 2 pseudodivisas: «Dum» y «my».

Otro matiz es que, antes de establecer el valor de SYMBOL_POINT con una precisión de N decimales, hay que asegurarse de que SYMBOL_DIGITS sea al menos N.

El libro incluye el script *CustomSymbolProperties.mq5*, que permite experimentar con la creación de copias del símbolo del gráfico actual y estudiar en la práctica los efectos resultantes. En concreto, puede elegir el nombre del símbolo, su ruta y la dirección de derivación (ajuste) de todas las propiedades admitidas, directas o inversas en términos de numeración de propiedades en el idioma. El script utiliza una clase especial *CustomSymbolMonitor*, que es un envoltorio para las funciones integradas anteriores: la describiremos *más adelante*.

7.2.3 Fijación de coeficientes de margen

Ya hemos estudiado la función *SymbolInfoMarginRate*, que devuelve los coeficientes de margen por símbolo establecidos por el bróker. En el caso de un símbolo personalizado, podemos fijar libremente estos coeficientes mediante la función *CustomSymbolSetMarginRate*.

bool CustomSymbolSetMarginRate(const string name, ENUM_ORDER_TYPE orderType, double initial, double maintenance)

La función fija los coeficientes de margen en función del tipo y la dirección de la orden (según el valor *orderType* de la enumeración ENUM_ORDER_TYPE). Los coeficientes para calcular el margen inicial y de mantenimiento (garantía por cada lote de una posición abierta o existente) se transmiten, respectivamente, en los parámetros *initial* y *maintenance*.

Los importes finales de los márgenes se determinan en función de varias propiedades de los símbolos (SYMBOL_TRADE_CALC_MODE, SYMBOL_MARGIN_INITIAL, SYMBOL_MARGIN_MAINTENANCE y otras) que se describen en la sección Requisitos de margen, por lo que también deben establecerse en el símbolo personalizado si es necesario.

La función devolverá un indicador de éxito (*true*) o error (*false*).

Con la ayuda de esta función y de las propiedades relacionadas con el cálculo de márgenes, puede emular las condiciones de trading de servidores que no estén disponibles por un motivo u otro, y depurar sus programas MQL en el probador.

7.2.4 Configurar sesiones de trading y cotización

Dos funciones de API permiten establecer sesiones de trading y cotización de un instrumento personalizado. Estos dos conceptos se abordaron en la sección Horarios de sesiones de trading y cotización.

```
bool CustomSymbolSetSessionQuote(const string name, ENUM_DAY_OF_WEEK dayOfWeek,
 uint sessionIndex, datetime from, datetime to)

bool CustomSymbolSetSessionTrade(const string name, ENUM_DAY_OF_WEEK dayOfWeek,
 uint sessionIndex, datetime from, datetime to)
```

CustomSymbolSetSessionQuote establece la hora de inicio y fin de la sesión de cotización especificada por número (*sessionIndex*) para un día concreto de la semana (*dayOfWeek*). *CustomSymbolSetSessionTrade* hace lo mismo para las sesiones de trading.

La numeración de las sesiones empieza por 0.

Las sesiones solo pueden añadirse secuencialmente, es decir, una sesión con índice 1 sólo puede añadirse si ya existe una sesión con índice 0. Si se infringe esta regla, no se creará una nueva sesión y la función devolverá *false*.

Los valores de fecha de los parámetros *from* y *to* se miden en segundos, y *from* debe ser inferior a *to*. El rango está limitado a dos días, de 0 (00 horas 00 minutos 00 segundos) a 172800 (23 horas 59 minutos 59 segundos del día siguiente). El cambio de día era necesario para poder especificar las sesiones que empiezan antes de medianoche y terminan después de medianoche. Esta situación se produce a menudo cuando la bolsa se encuentra al otro lado del mundo en relación con los servidores del bróker (operador).

Si se pasan parámetros de inicio y fin nulos (*from = 0* y *to = 0*) para la sesión *sessionIndex*, ésta se borra y la numeración de las sesiones siguientes (si las hay) se desplaza hacia abajo.

Las sesiones de trading no pueden ir más allá de las de cotización.

Por ejemplo, podemos crear una copia de un instrumento para una zona horaria diferente cambiando la hora de cotización intradía y el horario de la sesión para depurar el robot en diferentes condiciones, como con cualquier bróker exótico.

7.2.5 Añadir, sustituir y suprimir cotizaciones

Un símbolo personalizado se rellena de cotizaciones mediante dos funciones integradas: *CustomRatesUpdate* y *CustomRatesReplace*. En la entrada, además del nombre del símbolo, ambos esperan un array de estructuras *MqlRates* para el marco temporal M1 (los marcos temporales superiores se completan automáticamente a partir de M1). *CustomRatesReplace* tiene un par de parámetros adicionales (*from* y *to*) que definen el intervalo de tiempo al que se limita la edición del historial.

```
int CustomRatesUpdate(const string symbol, const MqlRates &rates[], uint count = WHOLE_ARRAY)

int CustomRatesReplace(const string symbol, datetime from, datetime to, const MqlRates &rates[],
uint count = WHOLE_ARRAY)
```

CustomRatesUpdate añade las barras que faltan al historial y sustituye las barras coincidentes existentes por datos del array.

CustomRatesReplace sustituye completamente el historial en el intervalo de tiempo especificado por los datos del array.

La diferencia entre las funciones se debe a los distintos escenarios de la aplicación prevista. Las diferencias se detallan en el cuadro siguiente:

CustomRatesUpdate	CustomRatesReplace
Aplica los elementos del array MqlRates pasada al historial, independientemente de sus marcas de tiempo.	Aplica solo aquellos elementos del array MqlRates pasada que se encuentren dentro del intervalo especificado.
Deja intactas en el historial aquellas barras M1 que ya estaban ahí antes de la llamada a la función y no coinciden en el tiempo con las barras del array.	Deja intacta todo el historial fuera del intervalo.
Sustituye las barras del historial existentes por las barras del array cuando las marcas de tiempo coinciden.	Borra completamente las barras del historial existentes en el rango especificado.
Inserta elementos del array como barras «nuevas» si no hay coincidencias con las barras antiguas.	Inserta las barras del array que se encuentran dentro del intervalo pertinente en el intervalo histórico especificado.

Los datos del array *rates* deben estar representados por precios OHLC válidos, y las horas de apertura de las barras no deben contener segundos.

Se establece un intervalo dentro de *from* y *to* inclusive: *from* es igual al tiempo de la primera barra que se procesa y *to* es igual al tiempo de la última.

En el siguiente diagrama se ilustran estas reglas con mayor claridad. Cada marca de tiempo única de un compás se designa con su propia letra latina. Las barras disponibles en el historial se muestran en mayúsculas, mientras que las barras del array se muestran en minúsculas. El carácter '-' es un hueco en el historial o en el array para la hora correspondiente.

```
History                       ABC-EFGHIJKLMN-PQRST------    B
Array                         -------hijk--nopqrstuvwxyz    A
Result of CustomRatesUpdate   ABC-EFGhijkLMnopqrstuvwxyz    R
Result of CustomRatesReplace  ABC-E--hijk--nopqrstuvw---    S
                                   ^               ^
                                   |from         to|    TIME
```

El parámetro opcional *count* establece el número de elementos del array *rates* que deben utilizarse (se ignorarán los demás). Esto le permite procesar parcialmente el array pasado. El valor predeterminado WHOLE_ARRAY significa el array completo.

El historial de cotizaciones de un símbolo personalizado puede borrarse total o parcialmente mediante la función *CustomRatesDelete*.

int CustomRatesDelete(const string symbol, datetime from, datetime to)

Aquí, los parámetros *from* y *to* también establecen el intervalo de tiempo de las barras eliminadas. Para cubrir todo el historial, especifique 0 y LONG_MAX.

Las tres funciones devuelven el número de barras procesadas: actualizadas o eliminadas. En caso de error, el resultado es -1.

Cabe señalar que las cotizaciones de un símbolo personalizado pueden formarse no solo añadiendo barras ya hechas, sino también mediante arrays de ticks o incluso una secuencia de ticks individuales.

Las funciones pertinentes se presentarán en la sección siguiente. Al añadir ticks, el terminal calculará automáticamente barras basadas en ellos. La diferencia entre estos métodos es que el historial de ticks personalizado le permite probar los programas MQL en el modo de ticks «reales», mientras que el historial de barras únicamente le obligará a limitarse a los modos OHLC M1 o de precio abierto o a confiar en la emulación de ticks implementada por el probador.

Además, añadir ticks de uno en uno permite simular eventos estándar *OnTick* y *OnCalculate* en el gráfico de un símbolo personalizado, lo que «anima» el gráfico de forma similar a las herramientas disponibles en línea, y lanza las funciones de manejador correspondientes en los programas MQL si se trazan en el gráfico. Pero hablaremos de ello en la próxima sección.

Como ejemplo de uso de nuevas funciones, vamos a considerar el script *CustomSymbolRandomRates.mq5*, diseñado para generar cotizaciones aleatorias según el principio del «paseo aleatorio» o cotizaciones existentes de ruido. Esto último puede utilizarse para evaluar la estabilidad de un Asesor Experto.

Para comprobar la corrección de la formación de las cotizaciones, también se admite el modo en el que se crea una copia completa del instrumento original, en el gráfico del que se lanzó el script.

Todos los modos se recogen en la enumeración RANDOMIZATION.

```
enum RANDOMIZATION
{
   ORIGINAL,
   RANDOM_WALK,
   FUZZY_WEAK,
   FUZZY_STRONG,
};
```

Aplicamos ruido de cotizaciones con dos niveles de intensidad: débil y fuerte.

En los parámetros de entrada, puede elegir, además del modo, una carpeta en la jerarquía de símbolos, un intervalo de fechas y un número para inicializar el generador aleatorio (para poder reproducir los resultados).

```
input string CustomPath = "MQL5Book\\Part7";    // Custom Symbol Folder
input RANDOMIZATION RandomFactor = RANDOM_WALK;
input datetime _From;                           // From (default: 120 days ago)
input datetime _To;                             // To (default: current time)
input uint RandomSeed = 0;
```

De manera predeterminada, cuando no se especifican fechas, el script genera las cotizaciones de los últimos 120 días. El valor 0 en el parámetro *RandomSeed* significa inicialización aleatoria.

El nombre del símbolo se genera en función del símbolo del gráfico actual y de los ajustes seleccionados.

```
const string CustomSymbol = _Symbol + "." + EnumToString(RandomFactor)
   + (RandomSeed ? "_" + (string)RandomSeed : "");
```

Al principio de *OnStart* prepararemos y comprobaremos los datos.

```
datetime From;
datetime To;

void OnStart()
{
   From = _From == 0 ? TimeCurrent() - 60 * 60 * 24 * 120 : _From;
   To = _To == 0 ? TimeCurrent() / 60 * 60 : _To;
   if(From > To)
   {
      Alert("Date range must include From <= To");
      return;
   }

   if(RandomSeed != 0) MathSrand(RandomSeed);
   ...
```

Dado que lo más probable es que el script deba ejecutarse varias veces, ofreceremos la posibilidad de eliminar el símbolo personalizado creado anteriormente con una solicitud de confirmación preliminar por parte del usuario.

```
   bool custom = false;
   if(PRTF(SymbolExist(CustomSymbol, custom)) && custom)
   {
      if(IDYES == MessageBox(StringFormat("Delete custom symbol '%s'?", CustomSymbol)
         "Please, confirm", MB_YESNO))
      {
         if(CloseChartsForSymbol(CustomSymbol))
         {
            Sleep(500); // wait for the changes to take effect (opportunistically)
            PRTF(CustomRatesDelete(CustomSymbol, 0, LONG_MAX));
            PRTF(SymbolSelect(CustomSymbol, false));
            PRTF(CustomSymbolDelete(CustomSymbol));
         }
      }
   }
   ...
```

La función de ayuda *CloseChartsForSymbol* no se muestra aquí (quien lo desee puede consultar el código fuente adjunto): su propósito es ver la lista de gráficos abiertos y cerrar aquellos en los que el símbolo de trabajo es el símbolo personalizado que se está borrando (sin esto, el borrado no funcionará).

Lo más importante es prestar atención a la llamada a *CustomRatesDelete* con un intervalo completo de fechas. Si no se hace así, los datos del símbolo de usuario anterior permanecerán en el disco durante un tiempo en la base de datos del historial (carpeta *bases/Custom/history/<symbol-name>*). En otras palabras: la llamada a *CustomSymbolDelete*, que se muestra en la última línea anterior, no es suficiente para borrar realmente el símbolo personalizado del terminal.

Si el usuario decide volver a crear inmediatamente un símbolo con el mismo nombre (y en el código que aparece a continuación ofrecemos esa posibilidad), las cotizaciones antiguas pueden mezclarse con las nuevas.

Además, tras la confirmación del usuario, se inicia el proceso de generación de cotizaciones. De ello se encarga la función *GenerateQuotes* (véase más adelante).

```
if(IDYES == MessageBox(StringFormat("Create new custom symbol '%s'?", CustomSymbol
   "Please, confirm", MB_YESNO))
{
   if(PRTF(CustomSymbolCreate(CustomSymbol, CustomPath, _Symbol)))
   {
      if(RandomFactor == RANDOM_WALK)
      {
         CustomSymbolSetInteger(CustomSymbol, SYMBOL_DIGITS, 8);
      }

      CustomSymbolSetString(CustomSymbol, SYMBOL_DESCRIPTION, "Randomized quotes")

      const int n = GenerateQuotes();
      Print("Bars M1 generated: ", n);
      if(n > 0)
      {
         SymbolSelect(CustomSymbol, true);
         ChartOpen(CustomSymbol, PERIOD_M1);
      }
   }
}
```

Si tiene éxito, el símbolo recién creado se selecciona en *Observación de Mercado* y se abre un gráfico para él. Por el camino, aquí se demuestra el establecimiento de un par de propiedades: SYMBOL_DIGITS y SYMBOL_DESCRIPTION.

En la función *GenerateQuotes* es necesario solicitar las cotizaciones del símbolo original para todos los modos excepto RANDOM_WALK.

```
int GenerateQuotes()
{
   MqlRates rates[];
   MqlRates zero = {};
   datetime start;     // time of the current bar
   double price;       // last closing price

   if(RandomFactor != RANDOM_WALK)
   {
      if(PRTF(CopyRates(_Symbol, PERIOD_M1, From, To, rates)) <= 0)
      {
         return 0; // error
      }
      if(RandomFactor == ORIGINAL)
      {
         return PRTF(CustomRatesReplace(CustomSymbol, From, To, rates));
      }
      ...
```

Es importante recordar que *CopyRates* se ve afectado por el límite en el número de barras del gráfico, que se establece en la configuración del terminal.

En el caso del modo ORIGINAL, simplemente reenviamos el array resultante *rates* a la función *CustomRatesReplace*. Para los modos de ruido, establecemos las variables *price* y *start* especialmente seleccionadas en los valores iniciales de precio y tiempo de la primera barra.

```
      price = rates[0].open;
      start = rates[0].time;
   }
   ...
```

En el modo de paseo aleatorio, las cotizaciones no son necesarias, por lo que sólo asignamos el array *rates* para futuras barras aleatorias M1.

```
   else
   {
      ArrayResize(rates, (int)((To - From) / 60) + 1);
      price = 1.0;
      start = From;
   }
   ...
```

Más adelante en el bucle a través del array *rates*, se añaden valores aleatorios a los precios ruidosos del símbolo original o «tal cual». En el modo RANDOM_WALK, nosotros mismos nos encargamos de aumentar el tiempo en la variable *start*. En otros modos, el tiempo ya está en las cotizaciones iniciales.

```
      const int size = ArraySize(rates);

      double hlc[3]; // future High Low Close (in unknown order)
      for(int i = 0; i < size; ++i)
      {
         if(RandomFactor == RANDOM_WALK)
         {
            rates[i] = zero;             // zeroing the structure
            rates[i].time = start += 60; // plus a minute to the last bar
            rates[i].open = price;       // start from the last price
            hlc[0] = RandomWalk(price);
            hlc[1] = RandomWalk(price);
            hlc[2] = RandomWalk(price);
         }
         else
         {
            double delta = 0;
            if(i > 0)
            {
               delta = rates[i].open - price; // cumulative correction
            }
            rates[i].open = price;
            hlc[0] = RandomWalk(rates[i].high - delta);
            hlc[1] = RandomWalk(rates[i].low - delta);
            hlc[2] = RandomWalk(rates[i].close - delta);
         }
         ArraySort(hlc);

         rates[i].high = fmax(hlc[2], rates[i].open);
         rates[i].low = fmin(hlc[0], rates[i].open);
         rates[i].close = price = hlc[1];
         rates[i].tick_volume = 4;
      }
      ...
```

A partir del precio de cierre de la última barra, se generan 3 valores aleatorios (utilizando la función *RandomWalk*). El máximo y el mínimo de ellos se convierten, respectivamente, en los precios *High* y *Low* de una nueva barra. La media es el precio *Close*.

Al final del bucle, pasamos el array a *CustomRatesReplace*.

```
      return PRTF(CustomRatesReplace(CustomSymbol, From, To, rates));
   }
```

En la función *RandomWalk* se ha intentado simular una distribución con colas anchas, típica de las cotizaciones reales.

```
double RandomWalk(const double p)
{
   const static double factor[] = {0.0, 0.1, 0.01, 0.05};
   const static double f = factor[RandomFactor] / 100;
   const double r = (rand() - 16383.0) / 16384.0; // [-1,+1]
   const int sign = r >= 0 ? +1 : -1;
   if(r != 0)
   {
      return p + p * sign * f * sqrt(-log(sqrt(fabs(r))));
   }
   return p;
}
```

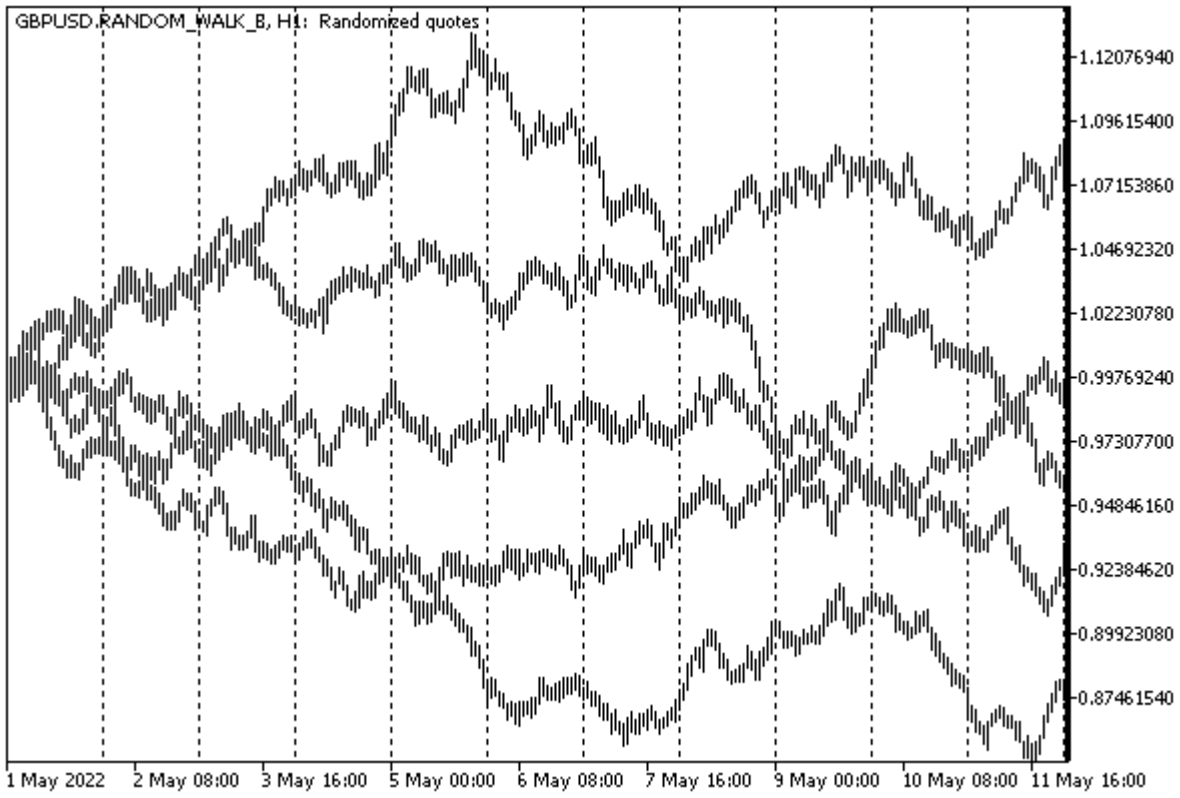
Los coeficientes de dispersión de las variables aleatorias dependen del modo. Por ejemplo, el ruido débil añade (o resta) un máximo de 1 centésima de porcentaje, y el ruido fuerte añade 5 centésimas de porcentaje del precio.

Mientras se ejecuta, el script muestra un registro detallado como éste:

```
Create new custom symbol 'GBPUSD.RANDOM_WALK'?
CustomSymbolCreate(CustomSymbol,CustomPath,_Symbol)=true / ok
CustomRatesReplace(CustomSymbol,From,To,rates)=171416 / ok
Bars M1 generated: 171416
```

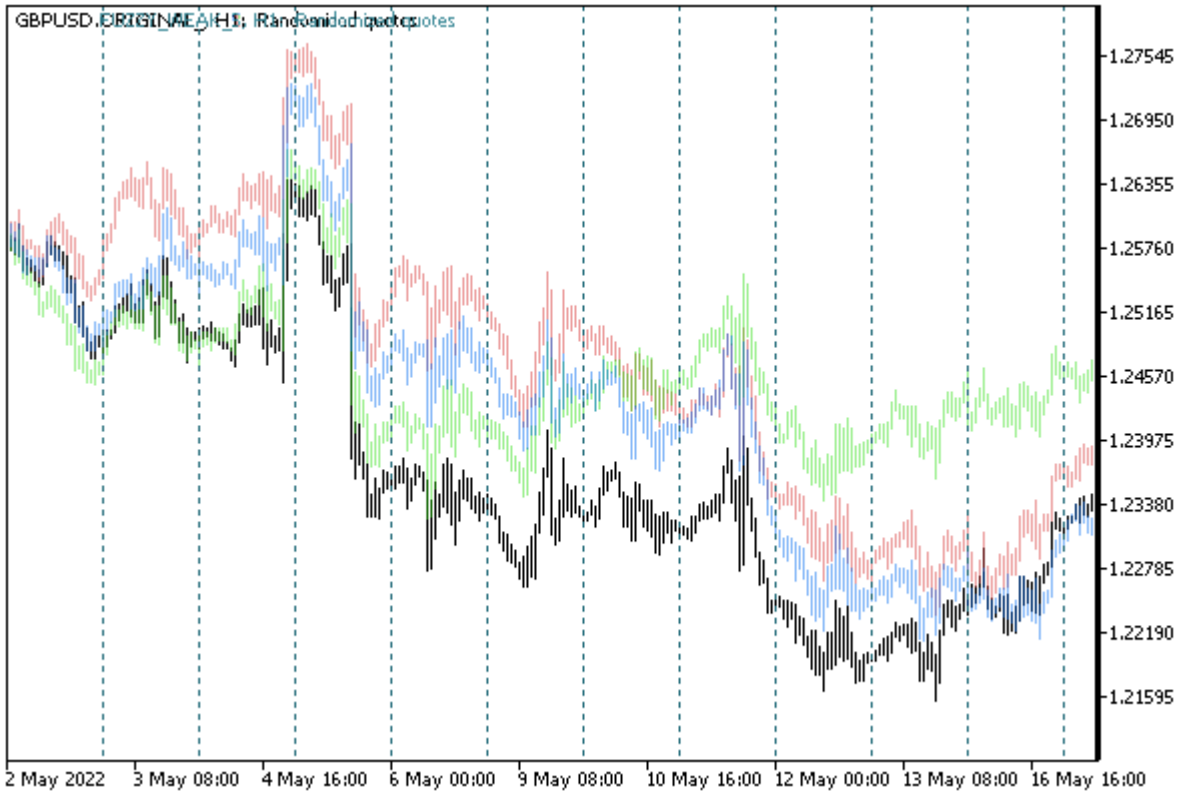
Veamos qué obtenemos como resultado.

En la siguiente imagen se muestran varias implementaciones de un paseo aleatorio (la superposición visual se realiza en un editor gráfico; en realidad, cada símbolo personalizado se abre en una ventana independiente, como es habitual).



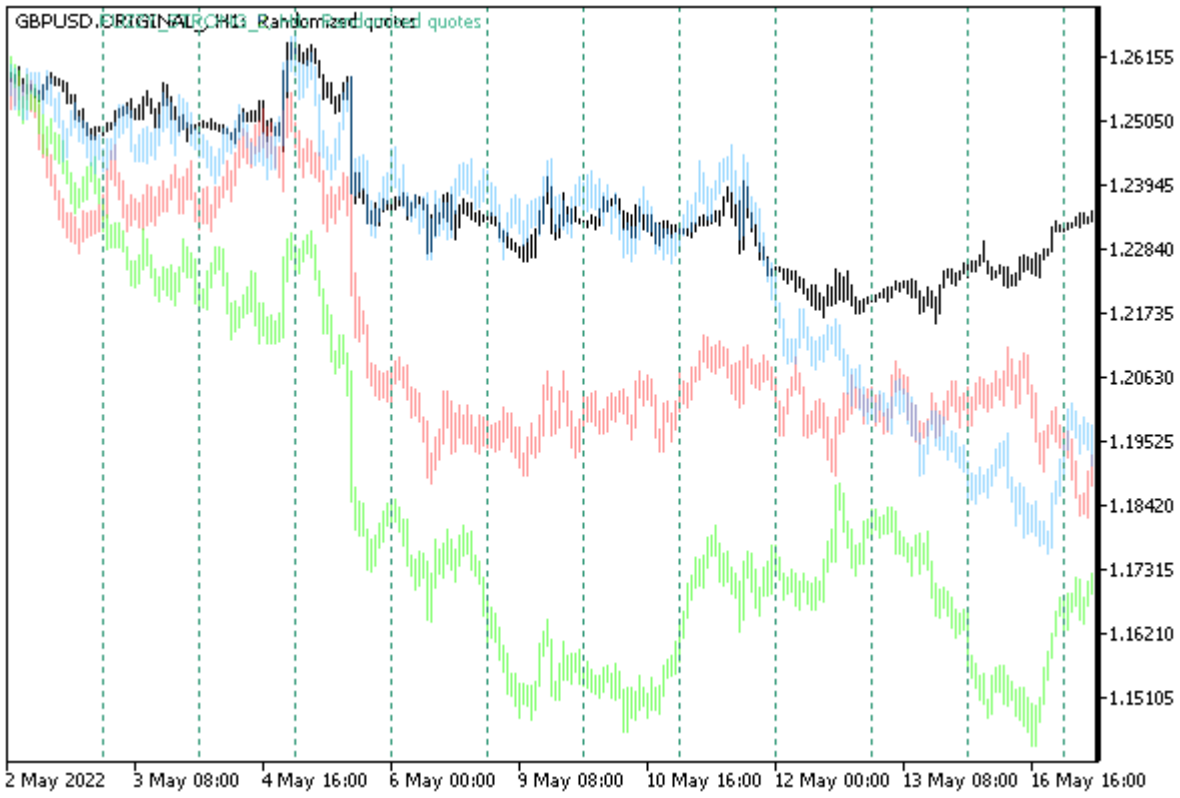
Opciones de cotización para símbolos personalizados con paseo aleatorio

Y así es como se ven las cotizaciones GBPUSD ruidosas (original en negro, color con ruido). Primero, en una versión débil.



Cotizaciones GBPUSD con poco ruido

Y después con ruido intenso.



Cotizaciones GBPUSD con ruido intenso

Las mayores discrepancias son evidentes, aunque con la preservación de las características locales.

7.2.6 Añadir, sustituir y eliminar ticks

La API de MQL5 permite generar el historial de un símbolo personalizado no sólo a nivel de barra, sino también a nivel de tick. De este modo, es posible conseguir un mayor realismo a la hora de probar y optimizar los Asesores Expertos, así como emular la actualización en tiempo real de los gráficos de símbolos personalizados, transmitiéndoles sus ticks. El conjunto de ticks transferidos al sistema se tiene en cuenta automáticamente al formar las barras. En otras palabras: no es necesario llamar a las funciones de la sección anterior que operan sobre las estructuras *MqlRates* si se proporciona información más detallada sobre los cambios de precios para el mismo período en forma de ticks, a saber, los arrays de estructuras *MqlTick*. La única ventaja de las cotizaciones por barra *MqlRates* es el rendimiento y la eficiencia de memoria.

Hay dos funciones para añadir ticks: *CustomTicksAdd* y *CustomTicksReplace*. La primera añade ticks interactivos que llegan a la ventana *Observación de Mercado* (desde donde son transferidos automáticamente por el terminal a la base de datos de ticks) y que generan los correspondientes eventos en los programas MQL. La segunda escribe los ticks directamente en la base de datos de ticks.

int CustomTicksAdd(const string symbol, const MqlTick &ticks[], uint count = WHOLE_ARRAY)

La función *CustomTicksAdd* añade datos del array *ticks* al historial de precios de un símbolo personalizado especificado en *symbol*. De manera predeterminada, si el ajuste *count* es igual a WHOLE_ARRAY, se añade el array completo. Si es necesario, puede especificar un número menor y descargar sólo una parte de los ticks.

Tenga en cuenta que el símbolo personalizado debe estar seleccionado en la ventana *Market Watch* en el momento de la llamada a la función. Para los símbolos no seleccionados en *Observación de Mercado*, debe utilizar la función *CustomTicksReplace* (véase más adelante).

El array de datos de ticks debe estar ordenado por tiempo en orden ascendente, es decir, es necesario que se cumplan las siguientes condiciones: *ticks[i].time_msc <= ticks[j].time_msc* para todos los *i < j*.

La función devuelve el número de bytes añadidos, o -1 en caso de error.

La función *CustomTicksAdd* transmite los ticks al gráfico del mismo modo que si procedieran del servidor del bróker. Normalmente, la función se aplica durante uno o varios ticks. En este caso, se «reproducen» en la ventana *Observación de Mercado*, a partir de la cual se guardan en la base de datos de ticks.

No obstante, cuando se transfiere una gran cantidad de datos en una llamada, la función cambia su comportamiento para ahorrar recursos. Si se transmiten más de 256 ticks, se dividen en dos partes. La primera parte (grande) se escribe inmediatamente en la base de datos de ticks (al igual que *CustomTicksReplace*). La segunda parte, consistente en los últimos 128 ticks (los más recientes), se pasa a la ventana *Observación de Mercado*, y después es guardada por el terminal en la base de datos.

La estructura *MqlTick* tiene dos campos con valores temporales: *time* (tiempo de tick en segundos) y *time_msc* (tiempo de tick en milisegundos). Ambos valores están fechados a partir del 01/01/1970. El campo *time_msc* (no nulo) rellenado tiene prioridad sobre *time*. Observe que *time* se rellena en segundos como resultado del recálculo basado en la fórmula *time_msc / 1000*. Si el campo *time_msc* es cero, se utiliza el valor del campo *time*, y el campo *time_msc* obtiene a su vez el valor en

milisegundos de la fórmula *time * 1000*. Si ambos campos son iguales a cero, la hora actual del servidor (con precisión de milisegundos) se pone en un tick.

De los dos campos que describen el volumen, *volume_real* tiene mayor prioridad que *volume*.

Dependiendo de qué otros campos se rellenen en un elemento de array concreto (estructura *MqlTick*), el sistema establece indicadores para el tick guardado en el campo *flags*:

- ticks[i].bid - TICK_FLAG_BID (el tick ha cambiado el precio de compra (Bid))
- ticks[i].ask - TICK_FLAG_ASK (el tick ha cambiado el precio de venta (Ask))
- ticks[i].last - TICK_FLAG_LAST (el tick ha cambiado el precio de la última operación)
- ticks[i].volume o ticks[i].volume_real - TICK_FLAG_VOLUME (el tick ha cambiado de volumen)

Si el valor de algún campo es menor o igual que cero, la bandera correspondiente no se escribe en el campo *flags*.

> Las banderas TICK_FLAG_BUY y TICK_FLAG_SELL no se añaden al historial de un símbolo personalizado.

La función *CustomTicksReplace* sustituye completamente el historial de precios del símbolo personalizado en el intervalo de tiempo especificado por los datos del array pasado.

```
int CustomTicksReplace(const string symbol, long from_msc, long to_msc,
 const MqlTick &ticks[], uint count = WHOLE_ARRAY)
```

El intervalo se establece mediante los parámetros *from_msc* y *to_msc*, en milisegundos desde el 01/01/1970. Ambos valores se incluyen en el intervalo.

El array *ticks* debe estar ordenado por orden cronológico de llegada de los ticks, lo que corresponde a un tiempo creciente, o mejor dicho, no decreciente, ya que los ticks con la misma hora suelen aparecer seguidos en un flujo con una precisión de milisegundos.

El parámetro *count* puede utilizarse para procesar una parte del array.

Los ticks se sustituyen secuencialmente día a día antes de la hora especificada en *to_msc*, o hasta que se produzca un error en el orden de los ticks. Primero se procesa el primer día del intervalo especificado, luego va el día siguiente, y así sucesivamente. En cuanto se detecta una discrepancia entre la hora del tick y el orden ascendente (no descendente), el proceso de sustitución del tick se detiene en el día actual. En este caso, los ticks de los días anteriores se sustituirán correctamente, mientras que el día actual (en el momento del tick erróneo) y todos los días restantes del intervalo especificado permanecerán sin cambios. La función devolverá -1, y el código de error en *_LastError* será 0 («sin error»).

Si el array *ticks* no tiene datos para algún periodo dentro del intervalo general entre *from_msc* y *to_msc* (ambos inclusive), después de ejecutar la función, el historial del símbolo personalizado tendrá un hueco correspondiente a los datos que faltan.

Si no hay datos en la base de datos de ticks en el intervalo de tiempo especificado, *CustomTicksReplace* le añadirá ticks desde el array *ticks*.

La función *CustomTicksDelete* puede utilizarse para borrar todos los ticks del intervalo de tiempo especificado.

int CustomTicksDelete(const string symbol, long from_msc, long to_msc)

El nombre del símbolo personalizado que se está editando se establece en el parámetro *symbol*, y el intervalo que debe borrarse se establece mediante los parámetros *from_msc* y *to_msc* (ambos inclusive), en milisegundos.

La función devuelve el número de bytes escritos, o -1 en caso de error.

> ¡Atención! La supresión de los ticks con *CustomTicksDelete* conlleva la supresión automática de las barras correspondientes. Sin embargo, llamar a *CustomRatesDelete*, es decir, eliminar barras, ¡no elimina ticks!

Para dominar el material en la práctica, resolveremos varios problemas aplicados utilizando las funciones recién consideradas.

Para empezar, abordemos una tarea tan interesante como la creación de un símbolo personalizado basado en un símbolo real pero con una densidad de ticks reducida. Esto acelerará la simulación y la optimización, además de reducir el consumo de recursos (principalmente RAM) en comparación con el modo basado en ticks reales, manteniendo al mismo tiempo una calidad aceptable, cercana a la ideal, del proceso.

Acelerar la simulación y la optimización

Los operadores a menudo buscan formas de acelerar los procesos de optimización y simulación de Asesores Expertos. Entre las posibles soluciones, las hay obvias, para las que basta con cambiar la configuración (cuando está permitido), y las hay que llevan más tiempo y requieren la adaptación de un Asesor Experto o un entorno de prueba.

Entre el primer tipo de soluciones se encuentran:

· Reducir el espacio de optimización eliminando algunos parámetros o reduciendo su paso;
· Reducir el periodo de optimización;
· Cambiar al modo de simulación de ticks de menor calidad (por ejemplo, de los reales a OHLC M1);
· Permitir el cálculo de beneficios en puntos en lugar de dinero;
· Actualizar el ordenador;
· Utilizar MQL Cloud u ordenadores adicionales de la red local.

Entre el segundo tipo de soluciones relacionadas con el desarrollo se encuentran:

· Crear perfiles de código, sobre la base de los cuales se pueden eliminar los «cuellos de botella» del código;
· Si es posible, utilizar el cálculo de indicadores con eficiencia de recursos, es decir, sin la directiva *#property tester_everytick_calculate* ;
· Transferir los algoritmos de los indicadores (si se utilizan) directamente al código del Asesor Experto: las llamadas a los indicadores imponen ciertos costes generales;
· Eliminar gráficos y objetos;
· Almacenar en caché los cálculos, si es posible;
· Reducir el número de posiciones abiertas simultáneamente y de órdenes colocadas (su cálculo en cada tick puede llegar a ser notable con un número elevado);
· Virtualizar por completo liquidaciones, órdenes, transacciones y posiciones: el mecanismo de contabilidad integrado, debido a su versatilidad, compatibilidad multidivisa y otras características, tiene sus propios gastos generales, que pueden eliminarse realizando acciones similares en el código MQL5 (aunque esta opción es la que más tiempo consume).

> La reducción de la densidad de ticks pertenece a un tipo intermedio de solución: requiere la creación programática de un símbolo personalizado, pero no afecta al código fuente del Asesor Experto.

El script *CustomSymbolFilterTicks.mq5* generará un símbolo personalizado con ticks reducidos. El instrumento inicial será el símbolo de trabajo del gráfico en el que se lanza el script. En los parámetros de entrada, puede especificar la carpeta para el símbolo personalizado y la fecha de inicio del tratamiento del historial. Por defecto, si no se indica ninguna fecha, el cálculo se realiza para los últimos 120 días.

```
input string CustomPath = "MQL5Book\\Part7"; // Custom Symbol Folder
input datetime _Start;                       // Start (default: 120 days back)
```

El nombre del símbolo se forma a partir del nombre del instrumento fuente y el sufijo «.TckFltr». Más adelante añadiremos la designación del método de reducción de ticks.

```
string CustomSymbol = _Symbol + ".TckFltr";
const uint DailySeconds = 60 * 60 * 24;
datetime Start = _Start == 0 ? TimeCurrent() - DailySeconds * 120 : _Start;
```

Por comodidad, en el manejador *OnStart* es posible borrar una copia anterior de un símbolo si ya existe.

```
void OnStart()
{
   bool custom = false;
   if(PRTF(SymbolExist(CustomSymbol, custom)) && custom)
   {
      if(IDYES == MessageBox(StringFormat("Delete existing custom symbol '%s'?", Cust
         "Please, confirm", MB_YESNO))
      {
         SymbolSelect(CustomSymbol, false);
         CustomRatesDelete(CustomSymbol, 0, LONG_MAX);
         CustomTicksDelete(CustomSymbol, 0, LONG_MAX);
         CustomSymbolDelete(CustomSymbol);
      }
      else
      {
         return;
      }
   }
```

A continuación, con el consentimiento del usuario, se crea un símbolo. El historial se rellena con datos de tick en la función auxiliar *GenerateTickData*. Si tiene éxito, el script añade un nuevo símbolo a *Observación de Mercado* y abre el gráfico.

```
      if(IDYES == MessageBox(StringFormat("Create new custom symbol '%s'?", CustomSymbol
         "Please, confirm", MB_YESNO))
      {
         if(PRTF(CustomSymbolCreate(CustomSymbol, CustomPath, _Symbol)))
         {
            CustomSymbolSetString(CustomSymbol, SYMBOL_DESCRIPTION, "Prunned ticks by "
            if(GenerateTickData())
            {
               SymbolSelect(CustomSymbol, true);
               ChartOpen(CustomSymbol, PERIOD_H1);
            }
         }
      }
   }
```

La función *GenerateTickData* procesa los ticks en un bucle en porciones, por día. Los ticks por día se solicitan llamando a *CopyTicksRange*. A continuación, necesitan ser reducidos de una manera u otra, lo cual es implementado por la clase *TickFilter*, que mostraremos a continuación. Por último, el array de ticks se añade al historial de símbolos personalizado mediante *CustomTicksReplace*.

```
bool GenerateTickData()
{
   bool result = true;
   datetime from = Start / DailySeconds * DailySeconds; // round up to the beginning
   ulong read = 0, written = 0;
   uint day = 0;
   const uint total = (uint)((TimeCurrent() - from) / DailySeconds + 1);
   MqlTick array[];

   while(!IsStopped() && from < TimeCurrent())
   {
      Comment(TimeToString(from, TIME_DATE), " ", day++, "/", total);

      const int r = CopyTicksRange(_Symbol, array, COPY_TICKS_ALL,
         from * 1000L, (from + DailySeconds) * 1000L - 1);
      if(r < 0)
      {
         Alert("Error reading ticks at ", TimeToString(from, TIME_DATE));
         result = false;
         break;
      }
      read += r;

      if(r > 0)
      {
         const int t = TickFilter::filter(Mode, array);
         const int w = CustomTicksReplace(CustomSymbol,
            from * 1000L, (from + DailySeconds) * 1000L - 1, array);
         if(w <= 0)
         {
            Alert("Error writing custom ticks at ", TimeToString(from, TIME_DATE));
            result = false;
            break;
         }
         written += w;
      }
      from += DailySeconds;
   }

   if(read > 0)
   {
      PrintFormat("Done ticks - read: %lld, written: %lld, ratio: %.1f%%",
         read, written, written * 100.0 / read);
   }
   Comment("");
   return result;
}
```

El control de errores y el recuento de ticks procesados se aplican en todas las fases. Al final, enviamos al registro el número de ticks iniciales y restantes, así como el factor de «compresión».

Ahora pasemos directamente a la técnica de reducción de ticks. Obviamente, puede haber muchos enfoques, y cada uno de ellos se adapta mejor o peor a una estrategia de trading concreta. Ofreceremos 3 versiones básicas combinadas en la clase *TickFilter* (*TickFilter.mqh*). Además, para completar el cuadro, también se admite el modo de copia de ticks sin reducción.

Así, en la clase se implementan los siguientes modos:

- Sin reducción
- Omitiendo secuencias de ticks con un cambio de precio monótono sin inversión (al modo «zig-zag»).
- Omitiendo fluctuaciones de precios dentro del diferencial
- Registrando sólo los ticks con una configuración fractal cuando el precio *Bid* o *Ask* representa un extremo entre dos ticks adyacentes.

Estos modos se describen como elementos de la enumeración FILTER_MODE.

```
class TickFilter
{
public:
   enum FILTER_MODE
   {
      NONE,
      SEQUENCE,
      FLUTTER,
      FRACTALS,
   };
   ...
```

Cada uno de los modos se implementa mediante un método estático independiente que acepta como entrada un array de ticks que necesita ser diluido. La edición de un array se realiza in situ (sin asignar un nuevo array de salida).

```
   static int filterBySequences(MqlTick &data[]);
   static int filterBySpreadFlutter(MqlTick &data[]);
   static int filterByFractals(MqlTick &data[]);
```

Todos los métodos devuelven el número de ticks restantes (tamaño de array reducido).

Para unificar la ejecución del procedimiento en diferentes modos, se proporciona el método *filter*. Para el modo NONE, el array *data* permanece igual.

```
   static int filter(FILTER_MODE mode, MqlTick &data[])
   {
      switch(mode)
      {
      case SEQUENCE: return filterBySequences(data);
      case FLUTTER: return filterBySpreadFlutter(data);
      case FRACTALS: return filterByFractals(data);
      }
      return ArraySize(data);
   }
```

Por ejemplo, así es como se implementa el filtrado por secuencias monótonas de ticks en el método *filterBySequences*.

```
static int filterBySequences(MqlTick &data[])
{
   const int size = ArraySize(data);
   if(size < 3) return size;

   int index = 2;
   bool dirUp = data[1].bid - data[0].bid + data[1].ask - data[0].ask > 0;

   for(int i = 2; i < size; i++)
   {
      if(dirUp)
      {
         if(data[i].bid - data[i - 1].bid + data[i].ask - data[i - 1].ask < 0)
         {
            dirUp = false;
            data[index++] = data[i];
         }
      }
      else
      {
         if(data[i].bid - data[i - 1].bid + data[i].ask - data[i - 1].ask > 0)
         {
            dirUp = true;
            data[index++] = data[i];
         }
      }
   }
   return ArrayResize(data, index);
}
```

Y así es como se ve el adelgazamiento fractal.

```
static int filterByFractals(MqlTick &data[])
{
   int index = 1;
   const int size = ArraySize(data);
   if(size < 3) return size;

   for(int i = 1; i < size - 2; i++)
   {
      if((data[i].bid < data[i - 1].bid && data[i].bid < data[i + 1].bid)
      || (data[i].ask > data[i - 1].ask && data[i].ask > data[i + 1].ask))
      {
         data[index++] = data[i];
      }
   }

   return ArrayResize(data, index);
}
```

Vamos a crear secuencialmente un símbolo personalizado para EURUSD en varios modos de reducción de densidad de ticks, y a comparar su rendimiento, es decir, el grado de «compresión», la rapidez de la simulación y cómo cambiará el rendimiento de trading del Asesor Experto.

Por ejemplo, al diluir las secuencias de ticks se obtienen los siguientes resultados (para un historial de un año y medio en MQ Demo).

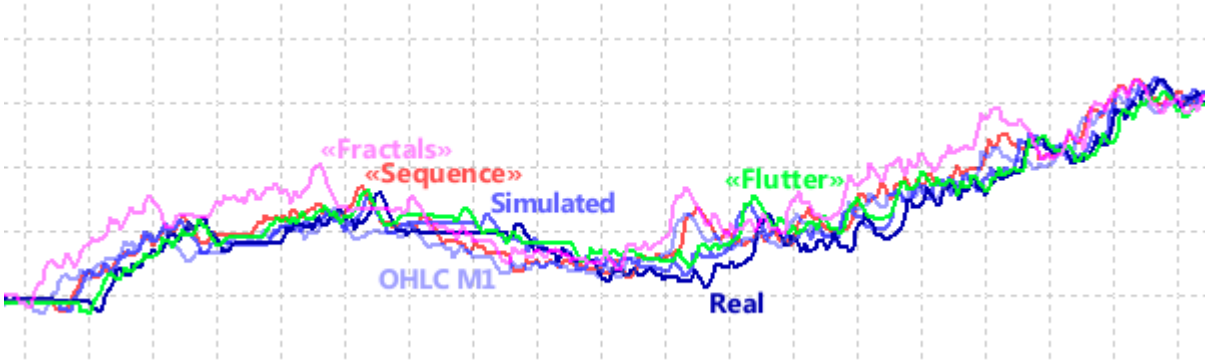
```
Create new custom symbol 'EURUSD.TckFltr-SE'?
Fixing SYMBOL_TRADE_TICK_VALUE: 0.0 <<< 1.0
true  SYMBOL_TRADE_TICK_VALUE 1.0 -> SUCCESS (0)
Fixing SYMBOL_TRADE_TICK_SIZE: 0.0 <<< 1e-05
true  SYMBOL_TRADE_TICK_SIZE 1e-05 -> SUCCESS (0)
Number of found discrepancies: 2
Fixed
Done ticks - read: 31553509, written: 16927376, ratio: 53.6%
```

Para los modos de suavizar las fluctuaciones y para los fractales, los indicadores son diferentes:

```
EURUSD.TckFltr-FL will be updated
Done ticks - read: 31568782, written: 22205879, ratio: 70.3%
...
Create new custom symbol 'EURUSD.TckFltr-FR'?
...
Done ticks - read: 31569519, written: 12732777, ratio: 40.3%
```

Para realizar experimentos prácticos de trading basados en ticks comprimidos necesitamos un Asesor Experto. Tomemos la versión adaptada de *BandOsMATicks.mq5*, en la que, frente a la original, se habilita el trading en cada tick (en el método *SimpleStrategy::trade*, la línea*if(lastBar == iTime(_Symbol, _Period, 0)) return false;* está deshabilitada), y los valores de los indicadores de señal se toman de las barras 0 y 1 (anteriormente solo se completaban las barras 1 y 2).

Vamos a ejecutar el Asesor Experto utilizando el rango de fechas desde principios de 2021 hasta el 1 de junio de 2022. Los ajustes se adjuntan en el archivo *MQL5/Presets/MQL5Book/BandOsMAticks.set*. El comportamiento general de la curva de balance en todos los modos es bastante similar.



Gráficos combinados de balances de prueba en diferentes modos por ticks

El desplazamiento horizontal de los extremos equivalentes de las diferentes curvas se debe al hecho de que el gráfico de informes estándar no utiliza el tiempo, sino el número de operaciones para la coordenada horizontal, que, por supuesto, difiere debido a la precisión de la activación de las señales de trading para diferentes bases de ticks.

Las diferencias en las métricas de rendimiento se muestran en la siguiente tabla (N: número de operaciones; $: beneficio; PF: factor de beneficio; RF: factor de recuperación; DD: reducción):

Modo	Ticks	Hora mm:ss.msec	Memoria	N	$	PF	RF	DD
Real	31002919	02:45.251	835 Mb	962	166.24	1.32	2.88	54.99
Emulación	25808139	01:58.131	687 Mb	928	171.94	1.34	3.44	47.64
OHLC M1	2084820	00:11.094	224 Mb	856	193.52	1.39	3.97	46.55
Secuencia	16310236	01:24.784	559 Mb	860	168.95	1.34	2.92	55.16
Flutter	21362616	01:52.172	623 Mb	920	179.75	1.37	3.60	47.28
Fractal	12270854	01:04.756	430 Mb	866	142.19	1.27	2.47	54.80

Consideraremos que la prueba basada en ticks reales es la más fiable y evaluaremos el resto por lo cerca que está de esta prueba. Evidentemente, la modalidad OHLC M1 mostró la mayor velocidad y un menor coste de recursos debido a una pérdida significativa de precisión (no se tuvo en cuenta la modalidad a precios de apertura). Presenta unos resultados financieros excesivamente optimistas.

Entre los tres modos con ticks comprimidos artificialmente, «Secuencia» es el más parecido al real en cuanto a conjunto de indicadores. Es 2 veces más rápido que el real en términos de tiempo y es 1,5 veces más eficiente en términos de consumo de memoria. El modo «Flutter» parece conservar mejor el número original de operaciones. El modo fractal, más rápido y menos exigente en memoria, por supuesto, requiere más tiempo y recursos que OHLC M1, pero no sobreestima las puntuaciones de trading.

Tenga en cuenta que los algoritmos de reducción de ticks pueden funcionar de forma diferente o, por el contrario, dar malos resultados con diferentes estrategias de trading, instrumentos financieros e incluso el historial de ticks de un determinado bróker. Investigue con sus Asesores Expertos y en su entorno de trabajo.

Como parte del segundo ejemplo de trabajo con símbolos personalizados, consideremos una característica interesante proporcionada por la traducción de ticks mediante *CustomTicksAdd*.

Muchos operadores utilizan paneles de trading, es decir, programas con controles interactivos para realizar manualmente acciones de trading arbitrarias. Tiene que practicar a trabajar con ellos sobre todo en línea, ya que el probador impone algunas restricciones. En primer lugar, el probador no admite eventos y objetos en el gráfico. Esto hace que los controles dejen de funcionar. Además, en el probador no se pueden aplicar objetos arbitrarios para el marcado de gráficos.

Intentemos resolver estos problemas.

Podemos generar un símbolo personalizado basado en ticks históricos a cámara lenta. Entonces, el gráfico de dicho símbolo se convertirá en un análogo de un probador visual.

Este planteamiento tiene varias ventajas:

- Comportamiento estándar de todos los eventos del gráfico
- Aplicación interactiva y establecimiento de indicadores
- Aplicación interactiva y ajuste de objetos
- Conmutación de marcos temporales sobre la marcha
- Pruebe el historial hasta el momento actual, incluido el día de hoy (el probador estándar no permite la simulación del día de hoy).

En cuanto al último punto, observamos que los desarrolladores de MetaTrader 5 prohibieron deliberadamente la comprobación de trading en el último día (actual), aunque a veces es necesario para encontrar rápidamente errores (en el código o en la estrategia de trading).

También es potencialmente interesante modificar los precios sobre la marcha (aumentando el diferencial, por ejemplo).

Basándonos en el gráfico de dicho símbolo personalizado, más adelante podemos implementar un emulador de trading manual sobre datos históricos.

El generador de símbolos será el Asesor Experto no de trading *CustomTester.mq5*. En sus parámetros de entrada, proporcionaremos una indicación de la colocación de un nuevo símbolo personalizado en la jerarquía de símbolos, la fecha de inicio en el pasado para la traducción de ticks (y la construcción de cotizaciones de símbolos personalizados), así como un marco temporal para el gráfico, que se abrirá automáticamente para la simulación visual.

```
input string CustomPath = "MQL5Book\\Part7"; // Custom Symbol Folder
input datetime _Start;                       // Start (120-day indent by default)
input ENUM_TIMEFRAMES Timeframe = PERIOD_H1;
```

El nombre del nuevo símbolo se construye a partir del nombre del símbolo del gráfico actual y el sufijo «.Tester».

```
string CustomSymbol = _Symbol + ".Tester";
```

Si no se especifica la fecha de inicio en los parámetros, el Asesor Experto retrocederá 120 días desde la fecha actual.

```
const uint DailySeconds = 60 * 60 * 24;
datetime Start = _Start == 0 ? TimeCurrent() - DailySeconds * 120 : _Start;
```

Los ticks se leerán del historial de ticks reales del símbolo de trabajo por lotes para todo el día al mismo tiempo. El puntero al día que se está leyendo se almacena en la variable *Cursor*.

```
bool FirstCopy = true;
// additionally 1 day ago, because otherwise, the chart will not update immediately
datetime Cursor = (Start / DailySeconds - 1) * DailySeconds; // round off at the border c
```

Los ticks de un día que se van a reproducir se solicitarán en el array *Ticks*, desde donde se trasladarán en pequeños lotes de tamaño *step* al gráfico de un símbolo personalizado.

```
MqlTick Ticks[];       // ticks for the "current" day in the past
int Index = 0;         // position in ticks within a day
int Step = 32;         // fast forward 32 ticks at a time (default)
int StepRestore = 0;   // remember the speed for the duration of the pause
long Chart = 0;        // created custom symbol chart
bool InitDone = false; // sign of completed initialization
```

Para reproducir los ticks a un ritmo constante, iniciemos el temporizador en *OnInit*.

```
void OnInit()
{
   EventSetMillisecondTimer(100);
}

void OnTimer()
{
   if(!GenerateData())
   {
      EventKillTimer();
   }
}
```

Los ticks serán generados por la función *GenerateData*. Inmediatamente después del lanzamiento, cuando se reinicia la bandera *InitDone*, intentaremos crear un nuevo símbolo o borrar las cotizaciones y ticks antiguos si el símbolo personalizado ya existe.

```
bool GenerateData()
{
   if(!InitDone)
   {
      bool custom = false;
      if(PRTF(SymbolExist(CustomSymbol, custom)) && custom)
      {
         if(IDYES == MessageBox(StringFormat("Clean up existing custom symbol '%s'?",
            CustomSymbol), "Please, confirm", MB_YESNO))
         {
            PRTF(CustomRatesDelete(CustomSymbol, 0, LONG_MAX));
            PRTF(CustomTicksDelete(CustomSymbol, 0, LONG_MAX));
            Sleep(1000);
            MqlRates rates[1];
            MqlTick tcks[];
            if(PRTF(CopyRates(CustomSymbol, PERIOD_M1, 0, 1, rates)) == 1
            || PRTF(CopyTicks(CustomSymbol, tcks) > 0))
            {
               Alert("Can't delete rates and Ticks, internal error");
               ExpertRemove();
            }
         }
         else
         {
            return false;
         }
      }
      else
      if(!PRTF(CustomSymbolCreate(CustomSymbol, CustomPath, _Symbol)))
      {
         return false;
      }
      ... // (A)
```

En este punto, omitiremos algo en (A) y volveremos a este punto más adelante.

Una vez creado el símbolo, lo seleccionamos en *Observación de Mercado* y abrimos un gráfico para él.

```
   SymbolSelect(CustomSymbol, true);
      Chart = ChartOpen(CustomSymbol, Timeframe);
      ... // (B)
      ChartSetString(Chart, CHART_COMMENT, "Custom Tester");
      ChartSetInteger(Chart, CHART_SHOW_OBJECT_DESCR, true);
      ChartRedraw(Chart);
      InitDone = true;
   }
   ...
```

Aquí también faltan un par de líneas (B); están relacionadas con futuras mejoras, pero aún no son necesarias.

Si el símbolo ya ha sido creado, empezamos a emitir ticks en lotes de *Step* ticks, pero no más de 256. Esta limitación está relacionada con las particularidades de la función *CustomTicksAdd*.

```
   else
   {
      for(int i = 0; i <= (Step - 1) / 256; ++i)
      if(Step > 0 && !GenerateTicks())
      {
         return false;
      }
   }
   return true;
}
```

La función de ayuda *GenerateTicks* emite ticks en lotes de *Step* ticks (pero no más de 256), leyéndolos del array diario *Ticks* por *Index* de desplazamiento. Cuando el array está vacío o lo hemos leído hasta el final, solicitamos los ticks del día siguiente llamando a *FillTickBuffer*.

```
bool GenerateTicks()
{
   if(Index >= ArraySize(Ticks)) // daily array is empty or read to the end
   {
      if(!FillTickBuffer()) return false; // fill the array with ticks per day
   }

   const int m = ArraySize(Ticks);
   MqlTick array[];
   const int n = ArrayCopy(array, Ticks, 0, Index, fmin(fmin(Step, 256), m));
   if(n <= 0) return false;

   ResetLastError();
   if(CustomTicksAdd(CustomSymbol, array) != ArraySize(array) || _LastError != 0)
   {
      Print(_LastError); // in case of ERR_CUSTOM_TICKS_WRONG_ORDER (5310)
      ExpertRemove();
   }
   Comment("Speed: ", (string)Step, " / ", STR_TIME_MSC(array[n - 1].time_msc));
   Index += Step; // move forward by 'Step' ticks
   return true;
}
```

La función *FillTickBuffer* utiliza *CopyTicksRange* para el funcionamiento.

```
bool FillTickBuffer()
{
   int r;
   ArrayResize(Ticks, 0);
   do
   {
      r = PRTF(CopyTicksRange(_Symbol, Ticks, COPY_TICKS_ALL, Cursor * 1000L,
         (Cursor + DailySeconds) * 1000L - 1));
      if(r > 0 && FirstCopy)
      {
         // NB: this pre-call is only needed to display the chart
        // from "Waiting for update" state
         PRTF(CustomTicksReplace(CustomSymbol, Cursor * 1000L,
            (Cursor + DailySeconds) * 1000L - 1, Ticks));
         FirstCopy = false;
         r = 0;
      }
      Cursor += DailySeconds;
   }
   while(r == 0 && Cursor < TimeCurrent()); // skip non-trading days
   Index = 0;
   return r > 0;
}
```

Cuando el Asesor Experto se detenga, cerraremos también el gráfico dependiente (para que no se duplique en el siguiente inicio).

```
void OnDeinit(const int)
{
   if(Chart != 0)
   {
      ChartClose(Chart);
   }
   Comment("");
}
```

En este punto, el Asesor Experto podría considerarse completo, pero hay un problema. El caso es que, por una razón u otra, las propiedades de un símbolo personalizado no se copian «tal cual» del símbolo de trabajo original, al menos en la implementación actual de la API de MQL5. Esto se aplica incluso a propiedades muy importantes, como SYMBOL_TRADE_TICK_VALUE, SYMBOL_TRADE_TICK_SIZE. Si imprimimos los valores de estas propiedades inmediatamente después de llamar a *CustomSymbolCreate(CustomSymbol, CustomPath, _Symbol)*, veremos ceros allí.

Para organizar la comprobación de las propiedades, su comparación y, en caso necesario, su corrección, hemos escrito una clase especial *CustomSymbolMonitor* (*CustomSymbolMonitor.mqh*) derivada de *SymbolMonitor*. Puede estudiar su estructura interna por su cuenta, mientras que aquí sólo presentaremos la interfaz pública.

Los constructores permiten crear un monitor de símbolos personalizado, especificando un símbolo de trabajo ejemplar (por nombre en una cadena, o a partir del objeto *SymbolMonitor*) que sirve como fuente de ajustes.

```
class CustomSymbolMonitor: public SymbolMonitor
{
public:
   CustomSymbolMonitor(); // sample - _Symbol
   CustomSymbolMonitor(const string s, const SymbolMonitor *m = NULL);
   CustomSymbolMonitor(const string s, const string other);

   //set/replace sample symbol
   void inherit(const SymbolMonitor &m);

   // copy all properties from the sample symbol in forward or reverse order
   bool setAll(const bool reverseOrder = true, const int limit = UCHAR_MAX);

   // check all properties against the sample, return the number of corrections
   int verifyAll(const int limit = UCHAR_MAX);

   // check the specified properties with the sample, return the number of correction
   int verify(const int &properties[]);

   // copy the given properties from the sample, return true if they all applied
   bool set(const int &properties[]);

   // copy the specific property from the sample, return true if applied
   template<typename E>
   bool set(const E e);

   bool set(const ENUM_SYMBOL_INFO_INTEGER property, const long value) const
   {
      return CustomSymbolSetInteger(name, property, value);
   }

   bool set(const ENUM_SYMBOL_INFO_DOUBLE property, const double value) const
   {
      return CustomSymbolSetDouble(name, property, value);
   }

   bool set(const ENUM_SYMBOL_INFO_STRING property, const string value) const
   {
      return CustomSymbolSetString(name, property, value);
   }
};
```

Dado que los símbolos personalizados, a diferencia de los símbolos estándar, permiten establecer propiedades propias, se ha añadido a la clase un triple de métodos *set*. En concreto, se utilizan para transferir por lotes las propiedades de una muestra y comprobar el éxito de estas acciones en otros métodos de la clase.

Ahora podemos volver al generador de símbolos personalizado y a su fragmento de código fuente, tal y como indicaba antes el comentario (A).

```
   // (A) check important properties and set them in "manual" mode
   SymbolMonitor sm; // _Symbol
   CustomSymbolMonitor csm(CustomSymbol, &sm);
   int props[] = {SYMBOL_TRADE_TICK_VALUE, SYMBOL_TRADE_TICK_SIZE};
   const int d1 = csm.verify(props); // check and try to fix
   if(d1)
   {
      Print("Number of found discrepancies: ", d1); // number of edits
      if(csm.verify(props)) // check again
      {
         Alert("Custom symbol can not be created, internal error!");
         return false; // symbol cannot be used without successful edits
      }
      Print("Fixed");
   }
```

Ahora puede ejecutar el Asesor Experto *CustomTester.mq5* y observar cómo se forman dinámicamente las cotizaciones en el gráfico que se abre automáticamente, y también cómo se reenvían los ticks desde el historial en la ventana *Observación de Mercado*.

No obstante, esto se hace a un ritmo constante de 32 ticks por 0.1 segundo. Es conveniente cambiar la velocidad de reproducción sobre la marcha a petición del usuario, tanto hacia arriba como hacia abajo. Este control puede organizarse, por ejemplo, desde el teclado.

Por lo tanto, es necesario añadir el manejador *OnChartEvent*. Como sabemos, para el evento CHARTEVENT_KEYDOWN, el programa recibe el código de la tecla pulsada en el parámetro *lparam*, y lo pasamos a la función *CheckKeys* (ver más abajo). Un fragmento (C), estrechamente relacionado con (B), ha tenido que ser pospuesto por el momento y volveremos a él en breve.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   ... // (C)
   if(id == CHARTEVENT_KEYDOWN) // these events only arrive while the chart is active
   {
      CheckKeys(lparam);
   }
}
```

En la función *CheckKeys* estamos procesando las teclas «flecha arriba» y «flecha abajo» para aumentar y disminuir la velocidad de reproducción. Además, la tecla «pausa» permite suspender completamente el proceso de «simulación» (transmisión de ticks). Al pulsar de nuevo «pausa» se reanuda el trabajo a la misma velocidad.

```
void CheckKeys(const long key)
{
   if(key == VK_DOWN)
   {
      Step /= 2;
      if(Step > 0)
      {
         Print("Slow down: ", Step);
         ChartSetString(Chart, CHART_COMMENT, "Speed: " + (string)Step);
      }
      else
      {
         Print("Paused");
         ChartSetString(Chart, CHART_COMMENT, "Paused");
         ChartRedraw(Chart);
      }
   }
   else if(key == VK_UP)
   {
      if(Step == 0)
      {
         Step = 1;
         Print("Resumed");
         ChartSetString(Chart, CHART_COMMENT, "Resumed");
      }
      else
      {
         Step *= 2;
         Print("Speed up: ", Step);
         ChartSetString(Chart, CHART_COMMENT, "Speed: " + (string)Step);
      }
   }
   else if(key == VK_PAUSE)
   {
      if(Step > 0)
      {
         StepRestore = Step;
         Step = 0;
         Print("Paused");
         ChartSetString(Chart, CHART_COMMENT, "Paused");
         ChartRedraw(Chart);
      }
      else
      {
         Step = StepRestore;
         Print("Resumed");
         ChartSetString(Chart, CHART_COMMENT, "Speed: " + (string)Step);
      }
   }
}
```

El nuevo código puede probarse en acción después de asegurarse primero de que el gráfico en el que trabaja el Asesor Experto está activo. Recuerde que los eventos de teclado sólo van a la ventana activa. Este es otro problema de nuestro probador.

Dado que el usuario debe realizar acciones de trading en el gráfico de símbolos personalizado, la ventana del generador estará casi siempre en segundo plano. Pasar a la ventana del generador para detener temporalmente el flujo de ticks y reanudarlo después no resulta práctico. Por lo tanto, se requiere de alguna manera para organizar el control interactivo desde el teclado directamente desde la ventana de símbolos personalizados.

Para ello, es adecuado un indicador especial, que podemos añadir automáticamente a la ventana de símbolos personalizados que se abre. El indicador interceptará los eventos de teclado en su propia ventana (ventana con un símbolo personalizado) y los enviará a la ventana del generador.

El código fuente del indicador se adjunta en el archivo *KeyboardSpy.mq5*. Por supuesto, el indicador no tiene gráficos. Un par de parámetros de entrada están dedicados a obtener el ID del gráfico *HostID*, donde deben enviarse los mensajes y el código de evento personalizado *EventID*, en el que se empaquetarán los eventos interactivos.

```
#property indicator_chart_window
#property indicator_plots 0

input long HostID;
input ushort EventID;
```

El trabajo principal se realiza en el manejador *OnChartEvent*.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   if(id == CHARTEVENT_KEYDOWN)
   {
      EventChartCustom(HostID, EventID, lparam,
         // this is always 0 when inside iCustom
         (double)(ushort)TerminalInfoInteger(TERMINAL_KEYSTATE_CONTROL),
         sparam);
   }
}
```

Tenga en cuenta que todas las «teclas de acceso rápido» que hemos elegido son sencillas, es decir, no utilizan atajos con teclas de estado del teclado, como *Ctrl* o *Shift*. Esto se hizo a la fuerza porque dentro de los indicadores creados programáticamente (en particular, a través de *iCustom*) no se lee el estado del teclado. En otras palabras: llamar a *TerminalInfoInteger(TERMINAL_KEYSTATE_XYZ)* siempre devuelve 0. En el manejador anterior, lo hemos añadido sólo a efectos de demostración, para que pueda comprobar esta limitación si lo desea, mostrando los parámetros entrantes en el «lado receptor».

Sin embargo, los clics de flecha y pausa se transferirán al gráfico principal normalmente, y eso es suficiente para nosotros. Lo único que queda por hacer es integrar el indicador con el Asesor Experto.

En el fragmento omitido anteriormente (B), durante la inicialización del generador crearemos un indicador y lo añadiremos al gráfico de símbolos personalizado.

```
#define EVENT_KEY 0xDED // custom event
      ...
      // (B)
      const int handle = iCustom(CustomSymbol, Timeframe, "MQL5Book/p7/KeyboardSpy",
         ChartID(), EVENT_KEY);
      ChartIndicatorAdd(Chart, 0, handle);
```

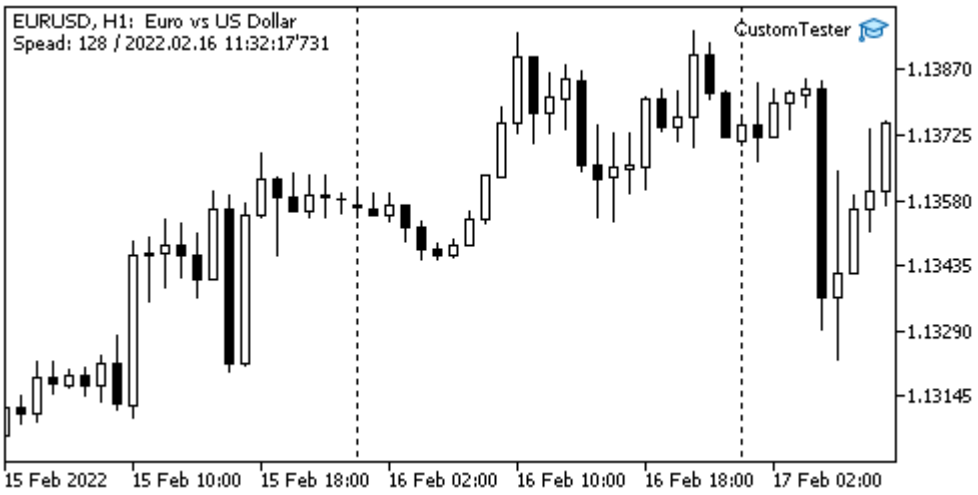
Más adelante, en el fragmento (C), garantizaremos la recepción de mensajes de usuario del indicador y su transferencia a la función ya conocida *CheckKeys*.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   // (C)
   if(id == CHARTEVENT_CUSTOM + EVENT_KEY) // notifications from the dependent chart
   {
      CheckKeys(lparam); // "remote" processing of key presses
   }
   else if(id == CHARTEVENT_KEYDOWN) // these events are only fired while the chart i
   {
      CheckKeys(lparam); // standard processing
   }
}
```

Así, ahora se puede controlar la velocidad de reproducción tanto en el gráfico con el Asesor Experto como en el gráfico del símbolo personalizado generado por él.

Con el nuevo conjunto de herramientas, puede intentar trabajar de forma interactiva con un gráfico que «vive en el pasado». En el gráfico aparece un comentario con la velocidad de reproducción actual o una marca de pausa.

En el gráfico con el Asesor Experto, el tiempo de los ticks de emisión «actual» se muestra en el comentario.



Asesor Experto que reproduce el historial de ticks (y cotizaciones) de un símbolo real

Básicamente, el usuario no puede hacer nada en esta ventana (si solo se elimina el Asesor Experto y se detiene la generación de símbolos personalizados). Aquí no es visible el proceso de traducción de ticks en sí. Además, como el Asesor Experto abre automáticamente un gráfico de símbolos personalizado

(donde se actualizan las cotizaciones históricas), es éste el que se activa. Para obtener la captura de pantalla anterior tuvimos que cambiar brevemente al gráfico original.

Por lo tanto, volvamos al gráfico del símbolo personalizado. La forma en que se actualiza de manera suave y progresiva en el pasado ya es estupenda, pero no se pueden realizar experimentos de trading con ella. Por ejemplo, si ejecuta en él su panel de trading habitual, sus controles, aunque formalmente funcionarán, no ejecutarán las transacciones, ya que el símbolo personalizado no existe en el servidor, y por tanto obtendrá errores. Esta característica se observa en cualquier programa que no esté especialmente adaptado para símbolos personalizados. Veamos un ejemplo de cómo puede virtualizarse el trading con un símbolo personalizado.

En lugar de un panel de trading (para simplificar el ejemplo, pero sin pérdida de generalidad), tomaremos como base el Asesor Experto más sencillo, *CustomOrderSend.mq5*, que puede realizar varias acciones de trading a golpe de tecla:

- 'B' - compra en el mercado
- 'S' - venta en el mercado
- 'U' - colocación de una orden de compra limitada
- 'L' - colocación de una orden de venta limitada
- 'C' - cerrar todas las posiciones
- 'D' - borrar todas las órdenes
- 'R' - enviar un informe de trading al diario

En los parámetros de entrada del Asesor Experto estableceremos el volumen de una operación (por defecto, el lote mínimo) y la distancia a los niveles de Stop Loss y Take Profit en puntos.

```
input double Volume;           // Volume (0 = minimal lot)
input int Distance2SLTP = 0;   // Distance to SL/TP in points (0 = no)

const double Lot = Volume == 0 ? SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN) : Volum
```

Si *Distance2SLTP* se deja igual a cero, no se colocan niveles de protección en las órdenes de mercado y no se forman órdenes pendientes. Cuando *Distance2SLTP* tiene un valor distinto de cero, se utiliza como la distancia desde el precio actual al colocar una orden pendiente (ya sea al alza o a la baja, dependiendo del comando).

Teniendo en cuenta las clases presentadas anteriormente de *MqlTradeSync.mqh*, la lógica anterior se convierte en el siguiente código fuente.

```
#include <MQL5Book/MqlTradeSync.mqh>

#define KEY_B 66
#define KEY_C 67
#define KEY_D 68
#define KEY_L 76
#define KEY_R 82
#define KEY_S 83
#define KEY_U 85

void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   if(id == CHARTEVENT_KEYDOWN)
   {
      MqlTradeRequestSync request;
      const double ask = SymbolInfoDouble(_Symbol, SYMBOL_ASK);
      const double bid = SymbolInfoDouble(_Symbol, SYMBOL_BID);
      const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);

      switch((int)lparam)
      {
      case KEY_B:
         request.buy(Lot, 0,
            Distance2SLTP ? ask - point * Distance2SLTP : Distance2SLTP,
            Distance2SLTP ? ask + point * Distance2SLTP : Distance2SLTP);
         break;
      case KEY_S:
         request.sell(Lot, 0,
            Distance2SLTP ? bid + point * Distance2SLTP : Distance2SLTP,
            Distance2SLTP ? bid - point * Distance2SLTP : Distance2SLTP);
         break;
      case KEY_U:
         if(Distance2SLTP)
         {
            request.buyLimit(Lot, ask - point * Distance2SLTP);
         }
         break;
      case KEY_L:
         if(Distance2SLTP)
         {
            request.sellLimit(Lot, bid + point * Distance2SLTP);
         }
         break;
      case KEY_C:
         for(int i = PositionsTotal() - 1; i >= 0; i--)
         {
            request.close(PositionGetTicket(i));
         }
         break;
      case KEY_D:
         for(int i = OrdersTotal() - 1; i >= 0; i--)
```

```
            {
               request.remove(OrderGetTicket(i));
            }
            break;
         case KEY_R:
   // there should be something here...
            break;
         }
      }
   }
```

Como podemos ver, aquí se utilizan tanto funciones estándar de la API de trading como métodos de *MqlTradeRequestSync*. Este último, indirectamente, también acaba llamando a un montón de funciones integradas. Necesitamos hacer que este Asesor Experto opere con un símbolo personalizado.

La idea más sencilla, aunque requiera tiempo, es sustituir todas las funciones estándar por sus propios análogos que cuenten órdenes, transacciones, posiciones y estadísticas financieras en algunas estructuras. Por supuesto, esto es posible sólo en los casos en que tenemos el código fuente del Asesor Experto, que debe adaptarse.

En el archivo adjunto *CustomTrade.mqh* se muestra una aplicación experimental de este enfoque. Puede familiarizarse con el código completo por su cuenta, ya que en el marco del libro sólo enumeraremos los puntos principales.

En primer lugar, observamos que muchos cálculos se hacen de forma simplificada, muchos modos no son compatibles y no se realiza una comprobación completa de la corrección de los datos. Utilice el código fuente como punto de partida para sus propios desarrollos.

Todo el código está envuelto en el espacio de nombres *CustomTrade* para evitar conflictos.

Las entidades de orden, transacción y posición se formalizan como las clases correspondientes *CustomOrder*, *CustomDeal* y *CustomPosition*. Todos ellos son herederos de la clase *MonitorInterface<I,D,S>::TradeState*. Recordemos que esta clase ya admite automáticamente la formación de arrays de propiedades de enteros, reales y cadenas para cada tipo de objeto y sus triples específicos de enumeraciones. Por ejemplo, *CustomOrder* tiene este aspecto:

```
class CustomOrder: public MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,
   ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PROPERTY_STRING>::TradeState
{
   static long ticket; // order counter and ticket provider
   static int done;    // counter of executed (historical) orders
public:
   CustomOrder(const ENUM_ORDER_TYPE type, const double volume, const string symbol)
   {
      _set(ORDER_TYPE, type);
      _set(ORDER_TICKET, ++ticket);
      _set(ORDER_TIME_SETUP, SymbolInfoInteger(symbol, SYMBOL_TIME));
      _set(ORDER_TIME_SETUP_MSC, SymbolInfoInteger(symbol, SYMBOL_TIME_MSC));
      if(type <= ORDER_TYPE_SELL)
      {
         // TODO: no deferred execution yet
         setDone(ORDER_STATE_FILLED);
      }
      else
      {
         _set(ORDER_STATE, ORDER_STATE_PLACED);
      }

      _set(ORDER_VOLUME_INITIAL, volume);
      _set(ORDER_VOLUME_CURRENT, volume);

      _set(ORDER_SYMBOL, symbol);
   }

   void setDone(const ENUM_ORDER_STATE state)
   {
      const string symbol = _get<string>(ORDER_SYMBOL);
      _set(ORDER_TIME_DONE, SymbolInfoInteger(symbol, SYMBOL_TIME));
      _set(ORDER_TIME_DONE_MSC, SymbolInfoInteger(symbol, SYMBOL_TIME_MSC));
      _set(ORDER_STATE, state);
      ++done;
   }

   bool isActive() const
   {
      return _get<long>(ORDER_TIME_DONE) == 0;
   }

   static int getDoneCount()
   {
      return done;
   }
};
```

Tenga en cuenta que en el entorno virtual de la antigua hora «actual» no puede utilizar la función *TimeCurrent* y en su lugar se toma la última hora conocida del símbolo personalizado *SymbolInfoInteger(symbol, SYMBOL_TIME)*.

Durante el trading virtual, los objetos actuales y su historial se acumulan en arrays de las clases correspondientes.

```
AutoPtr<CustomOrder> orders[];
CustomOrder *selectedOrders[];
CustomOrder *selectedOrder = NULL;
AutoPtr<CustomDeal> deals[];
CustomDeal *selectedDeals[];
CustomDeal *selectedDeal = NULL;
AutoPtr<CustomPosition> positions[];
CustomPosition *selectedPosition = NULL;
```

La metáfora para seleccionar órdenes, transacciones y posiciones era necesaria para simular un enfoque similar en las funciones integradas. Para ellas existen duplicados en el espacio de nombres *CustomTrade* que sustituyen a los originales mediante directivas de sustitución de macros.

```
#define HistorySelect CustomTrade::MT5HistorySelect
#define HistorySelectByPosition CustomTrade::MT5HistorySelectByPosition
#define PositionGetInteger CustomTrade::MT5PositionGetInteger
#define PositionGetDouble CustomTrade::MT5PositionGetDouble
#define PositionGetString CustomTrade::MT5PositionGetString
#define PositionSelect CustomTrade::MT5PositionSelect
#define PositionSelectByTicket CustomTrade::MT5PositionSelectByTicket
#define PositionsTotal CustomTrade::MT5PositionsTotal
#define OrdersTotal CustomTrade::MT5OrdersTotal
#define PositionGetSymbol CustomTrade::MT5PositionGetSymbol
#define PositionGetTicket CustomTrade::MT5PositionGetTicket
#define HistoryDealsTotal CustomTrade::MT5HistoryDealsTotal
#define HistoryOrdersTotal CustomTrade::MT5HistoryOrdersTotal
#define HistoryDealGetTicket CustomTrade::MT5HistoryDealGetTicket
#define HistoryOrderGetTicket CustomTrade::MT5HistoryOrderGetTicket
#define HistoryDealGetInteger CustomTrade::MT5HistoryDealGetInteger
#define HistoryDealGetDouble CustomTrade::MT5HistoryDealGetDouble
#define HistoryDealGetString CustomTrade::MT5HistoryDealGetString
#define HistoryOrderGetDouble CustomTrade::MT5HistoryOrderGetDouble
#define HistoryOrderGetInteger CustomTrade::MT5HistoryOrderGetInteger
#define HistoryOrderGetString CustomTrade::MT5HistoryOrderGetString
#define OrderSend CustomTrade::MT5OrderSend
#define OrderSelect CustomTrade::MT5OrderSelect
#define HistoryOrderSelect CustomTrade::MT5HistoryOrderSelect
#define HistoryDealSelect CustomTrade::MT5HistoryDealSelect
```

Por ejemplo, así es como se implementa la función *MT5HistorySelectByPosition*:

```
bool MT5HistorySelectByPosition(long id)
{
   ArrayResize(selectedOrders, 0);
   ArrayResize(selectedDeals, 0);

   for(int i = 0; i < ArraySize(orders); i++)
   {
      CustomOrder *ptr = orders[i][];
      if(!ptr.isActive())
      {
         if(ptr._get<long>(ORDER_POSITION_ID) == id)
         {
            PUSH(selectedOrders, ptr);
         }
      }
   }

   for(int i = 0; i < ArraySize(deals); i++)
   {
      CustomDeal *ptr = deals[i][];
      if(ptr._get<long>(DEAL_POSITION_ID) == id)
      {
         PUSH(selectedDeals, ptr);
      }
   }
   return true;
}
```

Como puede ver, todas las funciones de este grupo tienen el prefijo MT5, por lo que su doble propósito queda claro de inmediato y es fácil distinguirlas de las funciones del segundo grupo.

El segundo grupo de funciones del espacio de nombres *CustomTrade* realiza acciones utilitarias: comprueba y actualiza los estados de las órdenes, transacciones y posiciones, crea nuevos objetos y elimina los antiguos en función de la situación. En concreto, incluyen las funciones *CheckPositions* y *CheckOrders*, que pueden llamarse con un temporizador o en respuesta a acciones del usuario. Pero no puede hacer esto si utiliza un par de otras funciones diseñadas para mostrar el estado actual e histórico de la cuenta de trading virtual:

- *string ReportTradeState()* devuelve un texto multilínea con una lista de posiciones abiertas y órdenes colocadas.
- *void PrintTradeHistory()* muestra el historial de órdenes y operaciones en el registro.

Estas funciones llaman de forma independiente a *CheckPositions* y *CheckOrders* para proporcionarle información actualizada.

Además, existe una función para visualizar posiciones y órdenes activas en el gráfico en forma de objetos: *DisplayTrades*.

El archivo de encabezado *CustomTrade.mqh* debe incluirse en el Asesor Experto antes que otros encabezados para que la sustitución de macros tenga efecto en todas las líneas posteriores de los códigos fuente.

```
#include <MQL5Book/CustomTrade.mqh>
```

```
#include <MQL5Book/MqlTradeSync.mqh>
```

Ahora, el algoritmo anterior *CustomOrderSend.mq5* puede empezar a «operar» en el entorno virtual basado en el símbolo personalizado actual (que no requiere un servidor o un probador estándar) sin ningún cambio adicional.

Para mostrar rápidamente el estado, iniciaremos un segundo temporizador y cambiaremos periódicamente el comentario, además de mostrar objetos gráficos.

```
int OnInit()
{
   EventSetTimer(1);
   return INIT_SUCCEEDED;
}

void OnTimer()
{
   Comment(CustomTrade::ReportTradeState());
   CustomTrade::DisplayTrades();
}
```

Para construir un informe pulsando 'R', añadimos el manejador *OnChartEvent*.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   if(id == CHARTEVENT_KEYDOWN)
   {
      switch((int)lparam)
      {
      ...
      case KEY_R:
         CustomTrade::PrintTradeHistory();
         break;
      }
   }
}
```

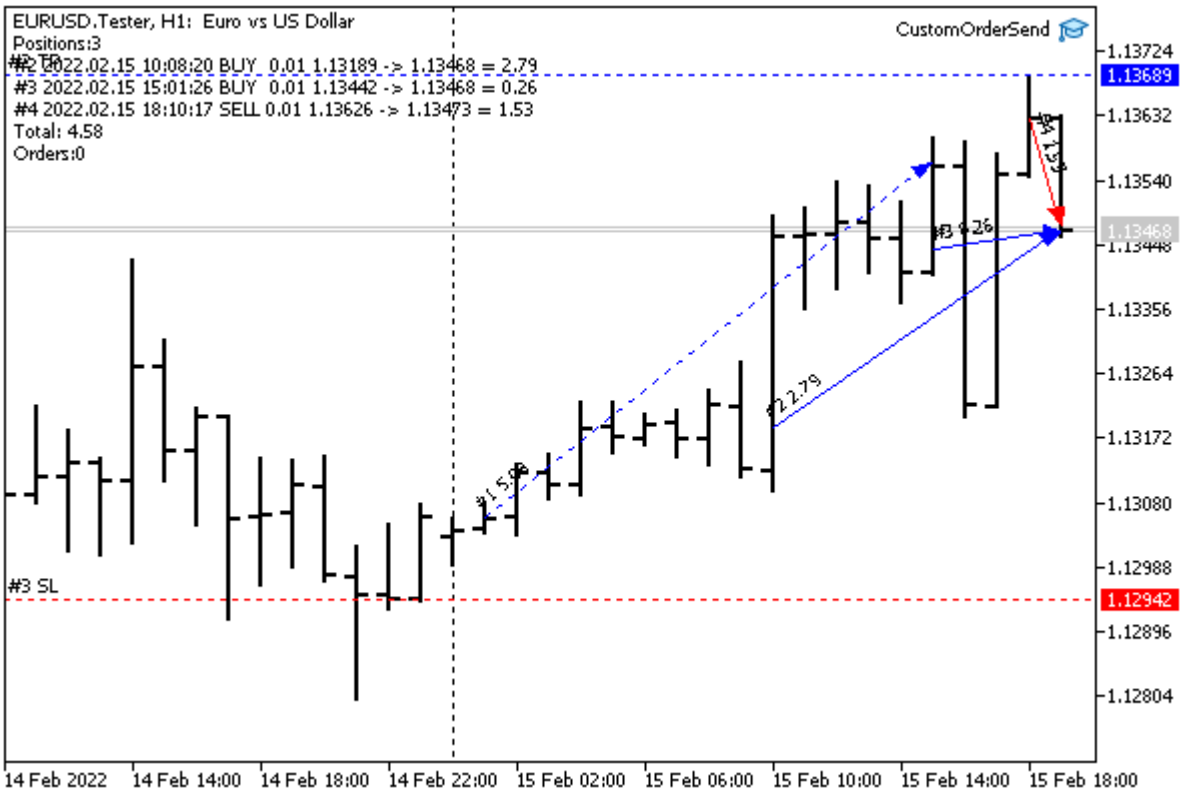
Por último, todo está listo para probar el nuevo paquete de software en acción.

Ejecute el generador de símbolos personalizados *CustomTester.mq5* en EURUSD. En el gráfico «EURUSD.Tester» que se abre, ejecute *CustomOrderSend.mq5* y comience a operar. A continuación se muestra una imagen del proceso de simulación:

Trading virtual en un gráfico de símbolos personalizado

Aquí puede ver dos posiciones largas abiertas (con niveles de protección) y una orden pendiente límite de venta.

Transcurrido un tiempo, una de las posiciones (indicada a continuación por una línea azul discontinua con una flecha) se cierra y se activa una orden de venta pendiente (línea roja con una flecha), lo que da como resultado la siguiente imagen:



Trading virtual en un gráfico de símbolos personalizado

Tras cerrar todas las posiciones (algunas por Take Profit y el resto por orden del usuario), se ordenó un informe pulsando 'R'.

```
History Orders:
(1) #1 ORDER_TYPE_BUY 2022.02.15 01:20:50 -> 2022.02.15 01:20:50 L=0.01 @ 1.1306
(4) #2 ORDER_TYPE_SELL_LIMIT 2022.02.15 02:34:29 -> 2022.02.15 18:10:17 L=0.01 @ 1.13
(2) #3 ORDER_TYPE_BUY 2022.02.15 10:08:20 -> 2022.02.15 10:08:20 L=0.01 @ 1.13189
(3) #4 ORDER_TYPE_BUY 2022.02.15 15:01:26 -> 2022.02.15 15:01:26 L=0.01 @ 1.13442
(1) #5 ORDER_TYPE_SELL 2022.02.15 15:35:43 -> 2022.02.15 15:35:43 L=0.01 @ 1.13568
(2) #6 ORDER_TYPE_SELL 2022.02.16 09:39:17 -> 2022.02.16 09:39:17 L=0.01 @ 1.13724
(4) #7 ORDER_TYPE_BUY 2022.02.16 23:31:15 -> 2022.02.16 23:31:15 L=0.01 @ 1.13748
(3) #8 ORDER_TYPE_SELL 2022.02.16 23:31:15 -> 2022.02.16 23:31:15 L=0.01 @ 1.13742
Deals:
(1) #1 [#1] DEAL_TYPE_BUY DEAL_ENTRY_IN 2022.02.15 01:20:50 L=0.01 @ 1.1306 = 0.00
(2) #2 [#3] DEAL_TYPE_BUY DEAL_ENTRY_IN 2022.02.15 10:08:20 L=0.01 @ 1.13189 = 0.00
(3) #3 [#4] DEAL_TYPE_BUY DEAL_ENTRY_IN 2022.02.15 15:01:26 L=0.01 @ 1.13442 = 0.00
(1) #4 [#5] DEAL_TYPE_SELL DEAL_ENTRY_OUT 2022.02.15 15:35:43 L=0.01 @ 1.13568 = 5.08
(4) #5 [#2] DEAL_TYPE_SELL DEAL_ENTRY_IN 2022.02.15 18:10:17 L=0.01 @ 1.13626 = 0.00
(2) #6 [#6] DEAL_TYPE_SELL DEAL_ENTRY_OUT 2022.02.16 09:39:17 L=0.01 @ 1.13724 = 5.35
(4) #7 [#7] DEAL_TYPE_BUY DEAL_ENTRY_OUT 2022.02.16 23:31:15 L=0.01 @ 1.13748 = -1.22
(3) #8 [#8] DEAL_TYPE_SELL DEAL_ENTRY_OUT 2022.02.16 23:31:15 L=0.01 @ 1.13742 = 3.00
Total: 12.21, Trades: 4
```

Los paréntesis indican los identificadores de posición y los corchetes, los tickets de las órdenes para las transacciones correspondientes (los tickets de ambos tipos van precedidos de una «almohadilla», '#').

Aquí no se tienen en cuenta swaps ni comisiones. Su cálculo puede sumarse.

Consideraremos otro ejemplo de trabajo con ticks de símbolos personalizados en la sección sobre particularidades del trading con símbolos personalizados. Hablaremos de la creación de gráficos de equivolumen.

7.2.7 Conversión de los cambios en el libro de órdenes

Si es necesario, un programa MQL puede generar un libro de órdenes para un símbolo personalizado utilizando la función *CustomBookAdd*. Esto, en concreto, puede ser útil para instrumentos procedentes de bolsas externas, como las criptomonedas.

int CustomBookAdd(const string symbol, const MqlBookInfo &books[], uint count = WHOLE_ARRAY)

La función transmite el estado del libro de órdenes a los programas MQL firmados para el *symbol* utilizando los datos del array *books*. El array describe el estado completo del libro de órdenes, es decir, todas las órdenes de compra y venta. El estado traducido sustituye completamente al anterior y pasa a estar disponible a través de la función *MarketBookGet*.

Utilizando el parámetro *count*, puede especificar el número de elementos del array *books* que se pasarán a la función. Por defecto se utiliza todo el array.

La función devuelve un indicador de éxito (*true*) o de error (*false*).

Para obtener los libros de órdenes generados por la función *CustomBookAdd*, un programa MQL que los requiera debe, como es habitual, suscribirse a los eventos utilizando *MarketBookAdd*.

La actualización de un libro de órdenes no actualiza los precios *Bid* y *Ask* del instrumento. Para actualizar los precios requeridos, añada ticks utilizando *CustomTicksAdd*.

Se comprueba que los datos transmitidos son correctos: los precios y volúmenes deben ser mayores que cero, y para cada elemento se debe especificar su tipo, precio y volumen (campos *volume* y/o *volume_real*). Si al menos un elemento del libro de órdenes se describe incorrectamente, la función devolverá un error.

También se comprueba el parámetro Book Depth (SYMBOL_TICKS_BOOKDEPTH) del instrumento personalizado. Si el número de niveles de venta o compra en el libro de órdenes traducido supera este valor, se descartan los niveles sobrantes.

El volumen con mayor precisión *volume_real* tiene prioridad sobre el *volume* normal. Si se especifican ambos valores para el elemento del libro de órdenes, se utilizará *volume_real*.

> ¡Atención! En la implementación actual, *CustomBookAdd* bloquea automáticamente el símbolo personalizado como si estuviera suscrito a él por *MarketBookAdd*, pero al mismo tiempo, los eventos de *OnBookEvent* no llegan (en teoría, el programa que genera los libros de órdenes puede suscribirse a ellos llamando explícitamente a *MarketBookAdd* y controlando lo que reciben otros programas). Puede eliminar este bloqueo llamando a *MarketBookRelease*.
>
> Esto puede ser necesario debido al hecho de que los símbolos para los que existen suscripciones al libro de órdenes no pueden ocultarse de *Observación de Mercado* por ningún medio (hasta que se cancelen todas las suscripciones explícitas o implícitas de los programas, y se cierre la ventana del libro de órdenes). Por consiguiente, estos símbolos no pueden eliminarse.

A modo de ejemplo, vamos a crear un Asesor Experto no de trading *PseudoMarketBook.mq5*, que generará un pseudoestado del libro de órdenes a partir del historial de ticks más cercano. Esto puede ser útil para los símbolos para los que el libro de órdenes no está traducido, en particular para Forex. Si

lo desea, puede utilizar estos símbolos personalizados para depurar formalmente sus propios algoritmos de trading utilizando el libro de órdenes.

Entre los parámetros de entrada, indicamos la profundidad máxima del libro de órdenes.

```
input uint CustomBookDepth = 20;
```

El nombre del símbolo personalizado se formará añadiendo el sufijo «.Pseudo» al nombre del símbolo del gráfico actual.

```
string CustomSymbol = _Symbol + ".Pseudo";
```

En el manejador *OnInit* creamos un símbolo personalizado y establecemos su fórmula con el nombre del símbolo original. Así, obtendremos una copia del símbolo original actualizada automáticamente por el terminal, y no tendremos que preocuparnos de copiar cotizaciones o ticks.

```
int OnInit()
{
   bool custom = false;
   if(!PRTF(SymbolExist(CustomSymbol, custom)))
   {
      if(PRTF(CustomSymbolCreate(CustomSymbol, CustomPath, _Symbol)))
      {
         CustomSymbolSetString(CustomSymbol, SYMBOL_DESCRIPTION, "Pseudo book generat
         CustomSymbolSetString(CustomSymbol, SYMBOL_FORMULA, "\"" + _Symbol + "\"");
      }
   }
   ...
```

Si el símbolo personalizado ya existe, el Asesor Experto puede ofrecer al usuario borrarlo y completar el trabajo allí (el usuario debe cerrar primero todos los gráficos con este símbolo).

```
   else
   {
      if(IDYES == MessageBox(StringFormat("Delete existing custom symbol '%s'?",
         CustomSymbol), "Please, confirm", MB_YESNO))
      {
         PRTF(MarketBookRelease(CustomSymbol));
         PRTF(SymbolSelect(CustomSymbol, false));
         PRTF(CustomRatesDelete(CustomSymbol, 0, LONG_MAX));
         PRTF(CustomTicksDelete(CustomSymbol, 0, LONG_MAX));
         if(!PRTF(CustomSymbolDelete(CustomSymbol)))
         {
            Alert("Can't delete ", CustomSymbol, ", please, check up and delete manua
         }
         return INIT_PARAMETERS_INCORRECT;
      }
   }
   ...
```

Una característica especial de este símbolo es el establecimiento de la propiedad SYMBOL_TICKS_BOOKDEPTH, así como la lectura del tamaño del contrato SYMBOL_TRADE_CONTRACT_SIZE, que será necesaria cuando se generen volúmenes.

```
   if(SymbolInfoInteger(_Symbol, SYMBOL_TICKS_BOOKDEPTH) != CustomBookDepth
   && SymbolInfoInteger(CustomSymbol, SYMBOL_TICKS_BOOKDEPTH) != CustomBookDepth)
   {
      Print("Adjusting custom market book depth");
      CustomSymbolSetInteger(CustomSymbol, SYMBOL_TICKS_BOOKDEPTH, CustomBookDepth);
   }

   depth = (int)PRTF(SymbolInfoInteger(CustomSymbol, SYMBOL_TICKS_BOOKDEPTH));
   contract = PRTF(SymbolInfoDouble(CustomSymbol, SYMBOL_TRADE_CONTRACT_SIZE));

   return INIT_SUCCEEDED;
}
```

El algoritmo se lanza en el manejador *OnTick*. Aquí llamamos a la función *GenerateMarketBook* que aún está por escribir. Llenará el array de estructuras *MqlBookInfo* pasadas por referencia, y la enviaremos a un símbolo personalizado usando *CustomBookAdd*.

```
void OnTick()
{
   MqlBookInfo book[];
   if(GenerateMarketBook(2000, book))
   {
      ResetLastError();
      if(!CustomBookAdd(CustomSymbol, book))
      {
         Print("Can't add market books, ", E2S(_LastError));
         ExpertRemove();
      }
   }
}
```

La función *GenerateMarketBook* analiza los últimos ticks de *count* y, a partir de ellos, emula el posible estado del libro de órdenes, guiándose por las siguientes hipótesis:

- Lo que se ha comprado es probable que se venda;
- Lo que se ha vendido es probable que se compre.

La división de ticks en aquellos que corresponden a compras y aquellos que corresponden a ventas, en el caso general (en ausencia de banderas de cambio) puede estimarse por el movimiento del precio en sí:

- El movimiento del precio *Ask* al alza se trata como una compra;
- El movimiento del precio *Bid* a la baja se trata como una venta.

Como resultado, obtenemos el siguiente algoritmo:

```
bool GenerateMarketBook(const int count, MqlBookInfo &book[])
{
   MqlTick tick; // order book centre
   if(!SymbolInfoTick(_Symbol, tick)) return false;

   double buys[];  // buy volumes by price levels
   double sells[]; // sell volumes by price levels

   MqlTick ticks[];
   CopyTicks(_Symbol, ticks, COPY_TICKS_ALL, 0, count); // request tick history
   for(int i = 1; i < ArraySize(ticks); ++i)
   {
      // we believe that ask was pushed up by buys
      int k = (int)MathRound((tick.ask - ticks[i].ask) / _Point);
      if(ticks[i].ask > ticks[i - 1].ask)
      {
         // already bought, probably will take profit by selling
         if(k <= 0)
         {
            Place(sells, -k, contract / sqrt(sqrt(ArraySize(ticks) - i)));
         }
      }

      // believe that the bid was pushed down by sells
      k = (int)MathRound((tick.bid - ticks[i].bid) / _Point);
      if(ticks[i].bid < ticks[i - 1].bid)
      {
         // already sold, probably will take profit by buying
         if(k >= 0)
         {
            Place(buys, k, contract / sqrt(sqrt(ArraySize(ticks) - i)));
         }
      }
   }
   ...
```

La función de ayuda *Place* rellena los arrays *buys* y *sells*, acumulando volúmenes en ellos por niveles de precios. Lo mostraremos más abajo. Los índices de los arrays se definen como la distancia en puntos desde los mejores precios actuales (*Bid* o *Ask*). El tamaño del volumen es inversamente proporcional a la antigüedad del tick, es decir, los ticks más lejanos en el tiempo tienen menos efecto.

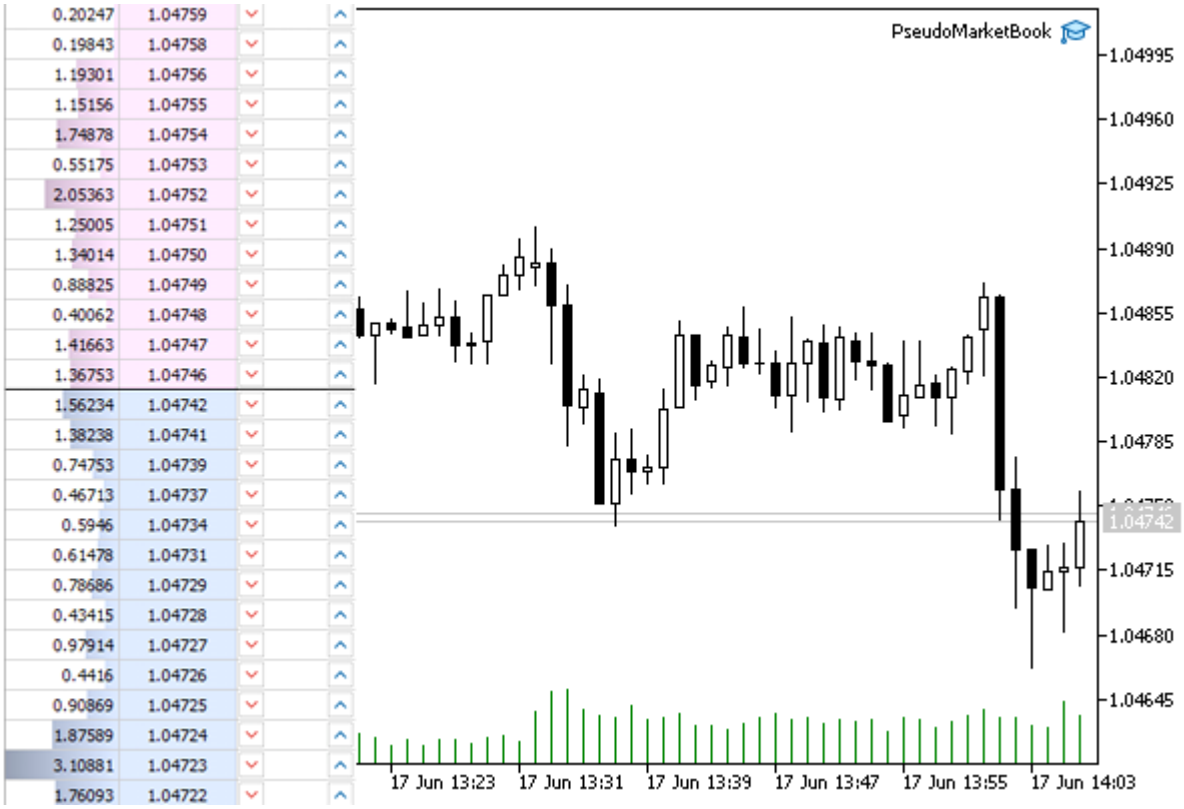
Una vez rellenados los arrays, se forma a partir de ellos un array de estructuras *MqlBookInfo*.

```
      for(int i = 0, k = 0; i < ArraySize(sells) && k < depth; ++i) // top half of the c
      {
         if(sells[i] > 0)
         {
            MqlBookInfo info = {};
            info.type = BOOK_TYPE_SELL;
            info.price = tick.ask + i * _Point;
            info.volume = (long)sells[i];
            info.volume_real = (double)(long)sells[i];
            PUSH(book, info);
            ++k;
         }
      }

      for(int i = 0, k = 0; i < ArraySize(buys) && k < depth; ++i) // bottom half of the
      {
         if(buys[i] > 0)
         {
            MqlBookInfo info = {};
            info.type = BOOK_TYPE_BUY;
            info.price = tick.bid - i * _Point;
            info.volume = (long)buys[i];
            info.volume_real = (double)(long)buys[i];
            PUSH(book, info);
            ++k;
         }
      }

      return ArraySize(book) > 0;
   }
```

La función *Place* es sencilla.

```
   void Place(double &array[], const int index, const double value = 1)
   {
      const int size = ArraySize(array);
      if(index >= size)
      {
         ArrayResize(array, index + 1);
         for(int i = size; i <= index; ++i)
         {
            array[i] = 0;
         }
      }
      array[index] += value;
   }
```

En la siguiente captura de pantalla se muestra un gráfico EURUSD con el Asesor Experto *PseudoMarketBook.mq5* ejecutándose en él, y la versión resultante del libro de órdenes.



Libro de órdenes sintético de un símbolo personalizado basado en EURUSD

7.2.8 Particularidades del trading con símbolos personalizados

El símbolo personalizado solo lo conoce el terminal cliente y no está disponible en el servidor de trading. Por lo tanto, si un símbolo personalizado se construye sobre la base de algún símbolo real, entonces cualquier Asesor Experto colocado en el gráfico de dicho símbolo personalizado debe generar órdenes de trading para el símbolo original.

Como solución más sencilla a este problema, puede colocar un Asesor Experto en el gráfico del símbolo original pero recibir señales (por ejemplo, de indicadores) del símbolo personalizado. Otro enfoque obvio es sustituir los nombres de los símbolos al realizar operaciones de trading. Para probar ambos enfoques, necesitamos un símbolo personalizado y un Asesor Experto.

Como ejemplo práctico interesante de símbolos personalizados, tomemos varios gráficos de equivolumen diferentes.

Un gráfico de equivolumen es un gráfico de barras construido sobre el principio de igualdad del volumen contenido en ellas. En un gráfico normal, cada nueva barra se forma con una frecuencia determinada, que coincide con el tamaño del marco temporal. En un gráfico de equivolumen, cada barra se considera formada cuando la suma de ticks o volúmenes reales alcanza un valor preestablecido. En este momento, el programa empieza a calcular el importe de la barra siguiente. Por supuesto, en el proceso de cálculo de los volúmenes se controlan los movimientos de los precios, y obtenemos los conjuntos habituales de precios en el gráfico: *Open*, *High*, *Low* y *Close*.

Las barras de igual rango se construyen de forma similar: en ellas se abre una nueva barra cuando el precio supera un número determinado de puntos en cualquier dirección.

Así, el Asesor Experto *EqualVolumeBars.mq5* soportará tres modos, es decir, tres tipos de gráficos:

- EqualTickVolumes - barras de equivolumen por ticks
- EqualRealVolumes - barras de equivolumen por volúmenes reales (si se emiten)
- RangeBars - barras de rango iguales

Se seleccionan mediante el parámetro de entrada *WorkMode*.

El tamaño de la barra y la profundidad del historial para el cálculo se especifican en los parámetros *TicksInBar* y *StartDate*.

```
input int TicksInBar = 1000;
input datetime StartDate = 0;
```

Dependiendo del modo, el símbolo personalizado recibirá el sufijo «_Eqv», «_Qrv» o «_Rng», respectivamente, con la adición del tamaño de la barra.

Aunque el eje horizontal de un gráfico Equivolume/Equal-Range sigue representando la cronología, las marcas de tiempo de cada barra son arbitrarias y dependen de la volatilidad (número o tamaño de las operaciones) en cada marco temporal. A este respecto, el marco temporal del gráfico de símbolos personalizado debe elegirse igual al M1 mínimo.

> La limitación de la plataforma es que todas las barras tienen la misma duración nominal, pero en el caso de nuestros gráficos «artificiales» hay que recordar que la duración real de cada barra es diferente y puede superar de forma considerable 1 minuto o, por el contrario, ser inferior. Así, con un volumen dado suficientemente pequeño para una barra, puede darse la situación de que se formen nuevas barras con mucha más frecuencia que una vez por minuto, y entonces el tiempo virtual de las barras de símbolos personalizados se adelantará al tiempo real, hacia el futuro. Para evitar que esto ocurra, debe aumentar el volumen de la barra (el parámetro *TicksInBar*) o mover las barras antiguas hacia la izquierda.

La inicialización y otras tareas auxiliares para la gestión de símbolos personalizados (en particular, la puesta a cero de un historial existente y la apertura de un gráfico con un nuevo símbolo) se realizan de forma similar a otros ejemplos, por lo que las omitiremos. Pasemos a los aspectos específicos de carácter aplicado.

Leeremos el historial de ticks reales utilizando las funciones incorporadas *CopyTicks*/*CopyTicksRange*: la primera es para intercambiar el historial en lotes de 10 000 ticks, y la segunda es para solicitar nuevos ticks desde el procesamiento anterior. Toda esta funcionalidad está empaquetada en la clase *TicksBuffer* (se adjunta el código fuente completo).

```
class TicksBuffer
{
private:
   MqlTick array[]; // internal array of ticks
   int tick;        // incremental index of the next tick for reading
public:
   bool fill(ulong &cursor, const bool history = false);
   bool read(MqlTick &t);
};
```

El método público *fill* está diseñado para llenar el array interno con la siguiente porción de ticks, comenzando desde el tiempo *cursor* (en milisegundos). Al mismo tiempo, el tiempo en *cursor* en cada llamada avanza en función del tiempo del último tick leído en el búfer (nótese que el parámetro se pasa por referencia).

El parámetro *history* determina si se utiliza *CopyTicks* o *CopyTicksRange*. Por regla general, en línea leeremos uno o más nuevos ticks del manejador *OnTick*.

El método *read* devuelve un tick del array interno y desplaza el puntero interno (*tick*) al siguiente tick. Si se alcanza el final del array durante la lectura, el método devolverá *false*, lo que significa que es el momento de llamar al método *fill*.

Utilizando estos métodos, el algoritmo de desvío del historial de ticks se implementa de la siguiente manera (este código se llama indirectamente desde *OnInit* a través del temporizador).

```
ulong cursor = StartDate * 1000;
TicksBuffer tb;

while(tb.fill(cursor, true) && !IsStopped())
{
   MqlTick t;
   while(tb.read(t))
   {
      HandleTick(t, true);
   }
}
```

En la función *HandleTick* se requiere tener en cuenta las propiedades del tick *t* en algunas variables globales que controlan el número de ticks, el volumen total de trading (real, si lo hay), así como la distancia de movimiento del precio. Según el modo de funcionamiento, estas variables deben analizarse de forma diferente para la condición de la formación de una nueva barra. Así que si en el modo de equivolumen, el número de ticks excede *TicksInBar*, deberíamos empezar una nueva barra reiniciando el contador a 1. En este caso, la hora de una nueva barra se toma como la hora del tick redondeada al minuto más cercano.

Este grupo de variables globales permite almacenar la hora virtual de la última barra («actual») de un símbolo personalizado (*now_time*), sus precios OHLC y volúmenes.

```
datetime now_time;
double now_close, now_open, now_low, now_high;
long now_volume, now_real;
```

Las variables se actualizan constantemente, tanto durante la lectura del historial como después, cuando el Asesor Experto empieza a procesar los ticks en línea en tiempo real (volveremos sobre esto un poco más adelante).

De forma algo simplificada, el algoritmo de *HandleTick* es el siguiente:

```
void HandleTick(const MqlTick &t, const bool history = false)
{
   now_volume++;               // count the number of ticks
   now_real += (long)t.volume; // sum up all real volumes

   if(!IsNewBar()) // continue the current bar
   {
      if(t.bid < now_low) now_low = t.bid;   // monitor price fluctuations downward
      if(t.bid > now_high) now_high = t.bid; // and upwards
      now_close = t.bid;                     // update the closing price

      if(!history)
      {
         // update the current bar if we are not in the history
         WriteToChart(now_time, now_open, now_low, now_high, now_close,
            now_volume - !history, now_real);
      }
   }
   else // new bar
   {
      do
      {
         // save the closed bar with all attributes
         WriteToChart(now_time, now_open, now_low, now_high, now_close,
            WorkMode == EqualTickVolumes ? TicksInBar : now_volume,
            WorkMode == EqualRealVolumes ? TicksInBar : now_real);

         // round up the time to the minute for the new bar
         datetime time = t.time / 60 * 60;

         // prevent bars with old or same time
         // if gone to the "future", we should just take the next count M1
         if(time <= now_time) time = now_time + 60;

         // start a new bar from the current price
         now_time = time;
         now_open = t.bid;
         now_low = t.bid;
         now_high = t.bid;
         now_close = t.bid;
         now_volume = 1;             // first tick in the new bar
         if(WorkMode == EqualRealVolumes) now_real -= TicksInBar;
         now_real += (long)t.volume; // initial real volume in the new bar

         // save new bar 0
         WriteToChart(now_time, now_open, now_low, now_high, now_close,
            now_volume - !history, now_real);
      }
      while(IsNewBar() && WorkMode == EqualRealVolumes);
   }
}
```

El parámetro *history* determina si el cálculo se basa en el historial o ya en tiempo real (en los ticks en línea entrantes). Si se basa en la historia, es suficiente para formar cada barra una vez, mientras que en línea, la barra actual se actualiza con cada tick. Esto permite acelerar el procesamiento del historial.

La función de ayuda *IsNewBar* devuelve *true* cuando se cumple la condición para cerrar la siguiente barra según el modo.

```
bool IsNewBar()
{
   if(WorkMode == EqualTickVolumes)
   {
      if(now_volume > TicksInBar) return true;
   }
   else if(WorkMode == EqualRealVolumes)
   {
      if(now_real > TicksInBar) return true;
   }
   else if(WorkMode == RangeBars)
   {
      if((now_high - now_low) / _Point > TicksInBar) return true;
   }

   return false;
}
```

La función *WriteToChart* crea una barra con las características dadas llamando a *CustomRatesUpdate*.

```
void WriteToChart(datetime t, double o, double l, double h, double c, long v, long m
{
   MqlRates r[1];

   r[0].time = t;
   r[0].open = o;
   r[0].low = l;
   r[0].high = h;
   r[0].close = c;
   r[0].tick_volume = v;
   r[0].spread = 0;
   r[0].real_volume = m;

   if(CustomRatesUpdate(SymbolName, r) < 1)
   {
      Print("CustomRatesUpdate failed: ", _LastError);
   }
}
```

El mencionado bucle de lectura y procesamiento de ticks se realiza durante el acceso inicial al historial, tras la creación o el recálculo completo de un símbolo de usuario ya existente. Cuando se trata de nuevos ticks, la función *OnTick* utiliza un código similar pero sin las banderas de «historicidad».

```
void OnTick()
{
   static ulong cursor = 0;
   MqlTick t;

   if(cursor == 0)
   {
      if(SymbolInfoTick(_Symbol, t))
      {
         HandleTick(t);
         cursor = t.time_msc + 1;
      }
   }
   else
   {
      TicksBuffer tb;
      while(tb.fill(cursor))
      {
         while(tb.read(t))
         {
            HandleTick(t);
         }
      }
   }

   RefreshWindow(now_time);
}
```

La función *RefreshWindow* añade un tick de símbolo personalizado en *Observación de Mercado*.

Tenga en cuenta que el reenvío de ticks incrementa en 1 el contador de ticks de la barra, y por tanto, al escribir el contador de ticks en la barra 0, previamente hemos restado uno (véase la expresión *now_volume - !history* al llamar a *WriteToChart*).

La generación de ticks es importante porque activa el evento *OnTick* en los gráficos de instrumentos personalizados, lo que potencialmente permite operar a los Asesores Expertos colocados en dichos gráficos. Sin embargo, esta tecnología requiere algunos trucos adicionales, que estudiaremos más adelante.

```
void RefreshWindow(const datetime t)
{
   MqlTick ta[1];
   SymbolInfoTick(_Symbol, ta[0]);
   ta[0].time = t;
   ta[0].time_msc = t * 1000;
   if(CustomTicksAdd(SymbolName, ta) == -1)
   {
      Print("CustomTicksAdd failed:", _LastError, " ", (long) ta[0].time);
      ArrayPrint(ta);
   }
}
```

Hacemos hincapié en que la hora del tick personalizado generado se establece siempre igual a la etiqueta de la barra actual, ya que no podemos dejar la hora del tick real: si se ha adelantado más de 1 minuto y enviamos dicho tick a *Observación de Mercado*, el terminal creará la siguiente barra M1, lo que violará nuestra estructura de «equivolumen», ya que nuestras barras no se forman por tiempo, sino por llenado de volumen (y nosotros mismos controlamos este proceso).

En teoría, podríamos añadir un milisegundo a cada tic, pero no tenemos ninguna garantía de que la barra no vaya a necesitar almacenar más de 60 000 tics (por ejemplo, si el usuario pide un gráfico con un determinado rango de precios que es impredecible en términos de cuántos tics serán necesarios para ese movimiento).

En los modos por volumen, es teóricamente posible interpolar los componentes de segundo y milisegundo de la hora del tick mediante fórmulas lineales:

- EqualTickVolumes – (now_volume - 1) * 60000 / TicksInBar;
- EqualRealVolumes – (now_real - 1) * 60000 / TicksInBar;

Sin embargo, esto no es más que un medio para identificar los ticks, y no un intento de hacer que la hora de los ticks «artificiales» se aproxime a la de los reales. No se trata sólo de la pérdida de irregularidad del flujo real de ticks, que de por sí ya provocará diferencias de precio entre el símbolo original y el símbolo personalizado generado sobre su base.

El principal problema es la necesidad de redondear el tiempo de los ticks a lo largo del borde de la barra M1 y «empaquetarlos» dentro de un minuto (véase la barra lateral sobre tipos especiales de gráficos). Por ejemplo, el siguiente tick con tiempo real 12:37:05'123 se convierte en el tick 1001 y debería formar una nueva barra de equivolumen. Sin embargo, la barra M1 sólo puede marcarse hasta el minuto, es decir, 12:37. Como resultado, el precio real del instrumento a las 12:37 no coincidirá con el precio del tick que proporcionó el precio *Open* para la barra de equivolumen 12:37. Además, si los siguientes 1000 ticks se extienden durante varios minutos, aún nos veremos obligados a «comprimir» su tiempo para no llegar a la marca de 12:38.

El problema es de naturaleza sistémica debido a la cuantificación del tiempo cuando los gráficos especiales son emulados por un gráfico estándar de marco temporal M1. Este problema no puede resolverse completamente en este tipo de gráficos, pero cuando se generan símbolos personalizados con ticks en tiempo continuo (por ejemplo, con cotizaciones sintéticas o basadas en datos de streaming de servicios externos), este problema no se produce.

> Es importante tener en cuenta que el reenvío de ticks se realiza en línea solamente en esta versión del generador, mientras que los ticks personalizados no se generan en el historial. Esto se hace para acelerar la creación de presupuestos. Si necesita generar un historial de ticks a pesar de la lentitud del proceso, deberá adaptar el Asesor Experto *EqualVolumeBars.mq5*: excluya la función

WriteToChart y realice toda la generación utilizando *CustomTicksReplace*/*CustomTicksAdd*. Al mismo tiempo, hay que recordar que el tiempo original de ticks debe ser sustituido por otro, dentro de una barra de minutos, a fin de no perturbar la estructura del gráfico de equivolumen formado.

Veamos cómo funciona *EqualVolumeBars.mq5*. He aquí el gráfico de trabajo de EURUSD M15 con el Asesor Experto ejecutándose en él. Tiene el gráfico de equivolumen, en el que se asignan 1000 ticks a cada barra.

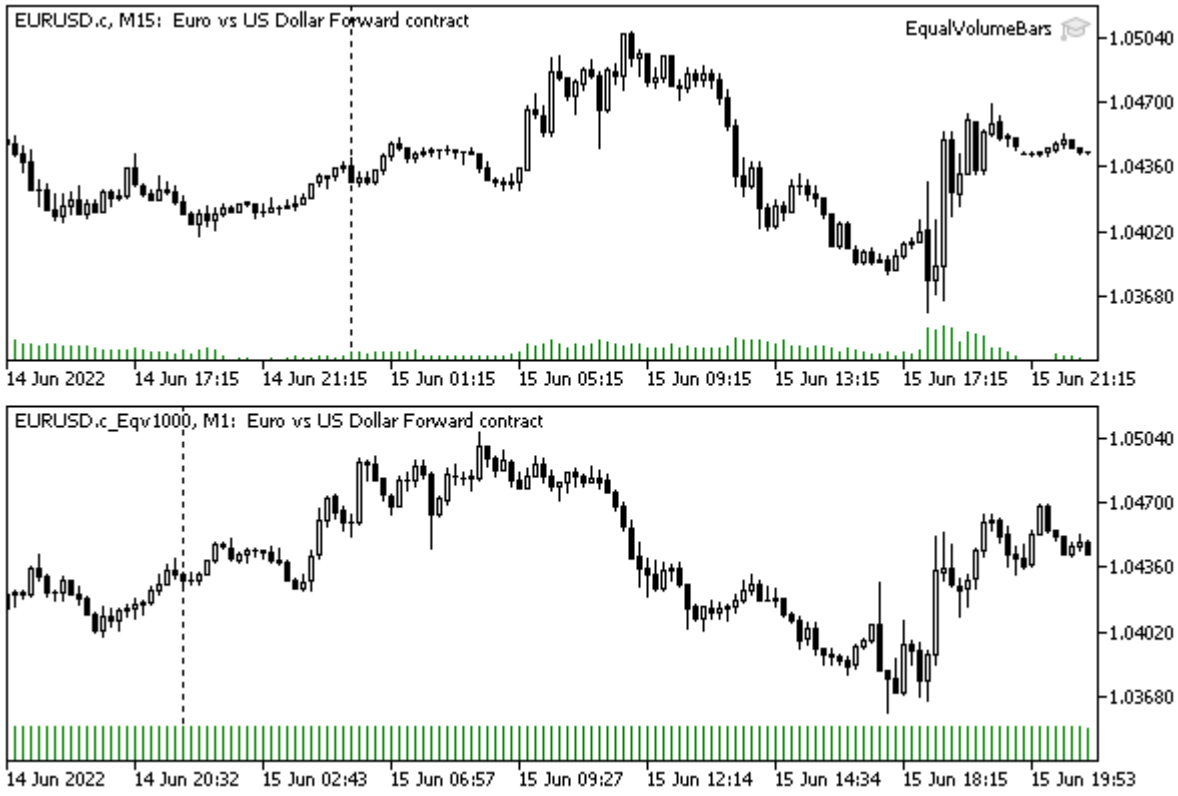


Gráfico EURUSD de equivolumen con 1000 ticks por barra generado por el Asesor Experto EqualVolumeBars

Observe que los volúmenes de ticks en todas las barras son iguales, excepto en la última, que aún se está formando (el recuento de ticks continúa).

Las estadísticas se muestran en el registro.

```
Creating "EURUSD.c_Eqv1000"
Processing tick history...
End of CopyTicks at 2022.06.15 12:47:51
Bar 0: 2022.06.15 12:40:00 866 0
2119 bars written in 10 sec
Open "EURUSD.c_Eqv1000" chart to view results
```

Comprobemos otro modo de funcionamiento: igual rango. A continuación se muestra un gráfico en el que el rango de cada barra es de 250 puntos.

Gráfico de igual rango EURUSD range con barras de 250 pips generadas por EqualVolumeBars

Para los instrumentos bursátiles, el Asesor Experto permite utilizar el modo de volumen real, por ejemplo, de la siguiente manera:

Gráfico de equivolumen y de Ethereum con volumen real de 10000 por barra

El marco temporal del símbolo de trabajo al colocar el generador del Asesor Experto no es importante, ya que el historial de ticks siempre se utiliza para los cálculos.

Al mismo tiempo, el marco temporal del gráfico de símbolos personalizado debe ser igual a M1 (el más pequeño disponible en el terminal). Así, el tiempo de las barras, por regla general, se corresponde de la forma más cercana posible (en la medida de lo posible) con los momentos de su formación. No obstante, durante los movimientos fuertes del mercado, cuando el número de ticks o el tamaño de los volúmenes forman varias barras por minuto, la hora de las barras se adelantará a la real. Cuando el mercado se calme, la situación con las marcas temporales de las barras de equivolumen se normalizará. Esto no afecta al flujo de precios en línea, por lo que probablemente no sea especialmente crítico, ya que el objetivo de utilizar barras de igual volumen o igual rango es desvincularse del tiempo absoluto.

Lamentablemente, el nombre del símbolo original y el símbolo personalizado creado a partir de él no pueden vincularse de ninguna manera a través de la propia plataforma. Sería conveniente disponer de un campo de cadena «origen» (fuente) entre las propiedades del símbolo personalizado, en el que pudiéramos escribir el nombre de la herramienta de trabajo real. Por defecto, estaría vacío, pero si se rellenara, la plataforma podría sustituir el símbolo en todas las órdenes de operaciones y solicitudes de historial, y hacerlo de forma automática y transparente para el usuario. En teoría, entre las propiedades de los símbolos definidos por el usuario, hay un campo SYMBOL_BASIS que es adecuado en cuanto a su significado, pero como no podemos garantizar que los generadores arbitrarios de símbolos definidos por el usuario (cualquier programa MQL) lo rellenen correctamente o lo utilicen exactamente para este propósito, no podemos confiar en su uso.

Como este mecanismo no está en la plataforma, tendremos que implementarlo nosotros mismos. Tendrá que establecer la correspondencia entre los nombres de los símbolos de origen y de usuario mediante parámetros.

Para resolver el problema, hemos desarrollado la clase *CustomOrder* (véase el archivo adjunto *CustomOrder.mqh*). Contiene métodos de envoltorio para todas las funciones de la API de MQL relacionadas con el envío de órdenes de trading y la solicitud de historial, que tienen un parámetro de cadena con el nombre del símbolo. En estos métodos, el símbolo personalizado se sustituye por el de trabajo actual o viceversa. Otras funciones de la API no requieren «unión». A continuación se ofrece un fragmento:

```
class CustomOrder
{
private:
   static string workSymbol;

   static void replaceRequest(MqlTradeRequest &request)
   {
      if(request.symbol == _Symbol && workSymbol != NULL)
      {
         request.symbol = workSymbol;
         if(MQLInfoInteger(MQL_TESTER)
            && (request.type == ORDER_TYPE_BUY
            || request.type == ORDER_TYPE_SELL))
         {
            if(TU::Equal(request.price, SymbolInfoDouble(_Symbol, SYMBOL_ASK)))
               request.price = SymbolInfoDouble(workSymbol, SYMBOL_ASK);
            if(TU::Equal(request.price, SymbolInfoDouble(_Symbol, SYMBOL_BID)))
               request.price = SymbolInfoDouble(workSymbol, SYMBOL_BID);
         }
      }
   }

public:
   static void setReplacementSymbol(const string replacementSymbol)
   {
      workSymbol = replacementSymbol;
   }

   static bool OrderSend(MqlTradeRequest &request, MqlTradeResult &result)
   {
      replaceRequest(request);
      return ::OrderSend(request, result);
   }
   ...
```

Tenga en cuenta que el método de trabajo principal *replaceRequest* sustituye no sólo el símbolo, sino también los precios actuales *Ask* y *Bid*. Esto se debe al hecho de que muchas herramientas personalizadas, como nuestro gráfico de equivolumen, tienen una hora virtual que es diferente de la hora del símbolo del prototipo real. Por lo tanto, los precios del instrumento personalizado emulado por el probador no coinciden con los precios correspondientes del instrumento real.

Este artefacto sólo se produce en el probador. Al operar en línea, el gráfico de símbolos personalizado se actualizará (a precios) de forma sincronizada con el real, aunque las etiquetas de las barras diferirán (una barra M1 «artificial» tiene una duración real de más o menos un minuto, y su tiempo de cuenta atrás no es múltiplo de un minuto). Así, esta conversión de precios es más bien una precaución para evitar recotizaciones en el probador. Sin embargo, en el probador normalmente no necesitamos hacer la sustitución de símbolos, ya que este puede operar con un símbolo personalizado (a diferencia del servidor del bróker). Además, sólo por interés, compararemos los resultados de las pruebas realizadas con y sin sustitución de caracteres.

Para minimizar las modificaciones en el código fuente del cliente, se proporcionan funciones globales y macros de la siguiente forma (para todos los métodos de *CustomOrder*):

```
bool CustomOrderSend(const MqlTradeRequest &request, MqlTradeResult &result)
{
  return CustomOrder::OrderSend((MqlTradeRequest)request, result);
}

#define OrderSend CustomOrderSend
```

Permiten redirigir automáticamente todas las llamadas a funciones estándar de la API a los métodos de la clase *CustomOrder*. Para ello, basta con incluir *CustomOrder.mqh* en el Asesor Experto y establecer el símbolo de trabajo, por ejemplo, en el parámetro *WorkSymbol*:

```
#include <CustomOrder.mqh>
#include <Expert/Expert.mqh>
...
input string WorkSymbol = "";

int OnInit()
{
  if(WorkSymbol != "")
  {
    CustomOrder::setReplacementSymbol(WorkSymbol);

    // initiate the opening of the chart tab of the working symbol (in the visual m
    MqlRates rates[1];
    CopyRates(WorkSymbol, PERIOD_CURRENT, 0, 1, rates);
  }
  ...
}
```

Es importante que la directiva *#include<CustomOrder.mqh>* fuera la primera, antes que las demás. Por lo tanto, afecta a todos los códigos fuente, incluidas las bibliotecas estándar de la distribución de MetaTrader 5. Si no se especifica ningún símbolo de sustitución, el *CustomOrder.mqh* conectado no tiene ningún efecto sobre el Asesor Experto y transfiere «transparentemente» el control a las funciones estándar de la API.

Ahora tenemos todo listo para probar la idea de operar con un símbolo personalizado, incluido el propio símbolo personalizado.

Aplicando la técnica mostrada anteriormente modificamos el ya conocido Asesor Experto *BandOsMaPro*, renombrándolo a *BandOsMaCustom.mq5*. Probémoslo en el gráfico de equivolume EURUSD con un tamaño de barra de 1000 ticks obtenido mediante *EqualVolumeBars.mq5*.

El modo de optimización o simulación se establece en los precios OHLC M1 (métodos más precisos no tienen sentido porque no generamos ticks y también porque esta versión negocia a los precios de las barras formadas). El intervalo de fechas es todo el año 2021 y el primer semestre de 2022. Se adjunta el archivo con la configuración *BandOsMACustom.set*.

En los ajustes del probador, no debe olvidar seleccionar el símbolo personalizado EURUSD_Eqv1000 y el marco temporal M1, ya que es en él donde se emulan las barras de equivolumen.

Cuando el parámetro *WorkSymbol* está vacío, el Asesor Experto negocia un símbolo personalizado. He aquí los resultados:

Bars	32533	Ticks	130132	Symbols	1
Initial Deposit	10 000.00				
Total Net Profit	80.29	Balance Drawdow...	12.07	Equity Drawdown ...	13.69
Gross Profit	603.48	Balance Drawdow...	42.84 (0.43%)	Equity Drawdown ...	44.54 (0.44%)
Gross Loss	-523.19	Balance Drawdow...	0.43% (42.84)	Equity Drawdown ...	0.44% (44.54)
Profit Factor	1.15	Expected Payoff	0.11	Margin Level	81067.80%
Recovery Factor	1.80	Sharpe Ratio	7.95	Z-Score	-1.49 (86.38%)
AHPR	1.0000 (0.00...	LR Correlation	0.92	OnTester result	81.44286473...
GHPR	1.0000 (0.00...	LR Standard Error	11.14		
Total Trades	720	Short Trades (won ...	362 (54.42%)	Long Trades (won ...	358 (49.72%)
Total Deals	1440	Profit Trades (% of...	375 (52.08%)	Loss Trades (% of t...	345 (47.92%)
	Largest	profit trade	8.29	loss trade	-5.00
	Average	profit trade	1.61	loss trade	-1.52
	Maximum	consecutive wins ($)	8 (12.73)	consecutive losses ...	9 (-13.11)
	Maximal	consecutive profit ...	18.56 (6)	consecutive loss (c...	-13.11 (9)
	Average	consecutive wins	2	consecutive losses	2

Informe del probador al operar en el gráfico de equivolumen EURUSD_Eqv1000

Si el parámetro *WorkSymbol* es igual a EURUSD, el Asesor Experto negocia el par EURUSD, a pesar de que trabaja en el gráfico EURUSD_Eqv1000. Los resultados difieren, pero no mucho.

Bars	32533	Ticks	130132	Symbols	1
Initial Deposit	10 000.00				
Total Net Profit	83.21	Balance Drawdow...	11.80	Equity Drawdown ...	13.29
Gross Profit	598.65	Balance Drawdow...	46.34 (0.46%)	Equity Drawdown ...	47.38 (0.47%)
Gross Loss	-515.44	Balance Drawdow...	0.46% (46.34)	Equity Drawdown ...	0.47% (47.38)
Profit Factor	1.16	Expected Payoff	0.12	Margin Level	81073.24%
Recovery Factor	1.76	Sharpe Ratio	8.36	Z-Score	-1.62 (89.48%)
AHPR	1.0000 (0.00...	LR Correlation	0.92	OnTester result	81.11629731...
GHPR	1.0000 (0.00...	LR Standard Error	11.27		
Total Trades	720	Short Trades (won ...	362 (55.80%)	Long Trades (won ...	358 (48.88%)
Total Deals	1440	Profit Trades (% of...	377 (52.36%)	Loss Trades (% of t...	343 (47.64%)
	Largest	profit trade	8.38	loss trade	-6.21
	Average	profit trade	1.59	loss trade	-1.50
	Maximum	consecutive wins ($)	9 (13.78)	consecutive losses ...	9 (-13.00)
	Maximal	consecutive profit ...	18.46 (6)	consecutive loss (c...	-13.00 (9)
	Average	consecutive wins	2	consecutive losses	2

Informe del probador al operar EURUSD desde el gráfico de equivolumen EURUSD_Eqv1000

Sin embargo, como ya se mencionó al principio de la sección, hay una manera más fácil para que los Asesores Expertos que operan con señales de indicadores admitan símbolos personalizados. Para ello, basta con crear indicadores en un símbolo personalizado y colocar el Asesor Experto en el gráfico de un símbolo de trabajo.

Podemos aplicar fácilmente esta opción. Llamémosla *BandOsMACustomSignal.mq5*.

El archivo de encabezado *CustomOrder.mqh* ya no es necesario. En lugar del parámetro de entrada *WorkSymbol*, añadimos dos nuevos:

```
input string SignalSymbol = "";
input ENUM_TIMEFRAMES SignalTimeframe = PERIOD_M1;
```

Deben pasarse al constructor de la clase *BandOsMaSignal* que gestiona los indicadores. Antes, *_Symbol* y *_Period* se utilizaban en todas partes.

```
interface TradingSignal
{
   virtual int signal(void);
   virtual string symbol();
   virtual ENUM_TIMEFRAMES timeframe();
};

class BandOsMaSignal: public TradingSignal
{
   int hOsMA, hBands, hMA;
   int direction;
   const string _symbol;
   const ENUM_TIMEFRAMES _timeframe;
public:
   BandOsMaSignal(const string s, const ENUM_TIMEFRAMES tf,
      const int fast, const int slow, const int signal, const ENUM_APPLIED_PRICE pric
      const int bands, const int shift, const double deviation,
      const int period, const int x, ENUM_MA_METHOD method): _symbol(s), _timeframe(t
   {
      hOsMA = iOsMA(s, tf, fast, slow, signal, price);
      hBands = iBands(s, tf, bands, shift, deviation, hOsMA);
      hMA = iMA(s, tf, period, x, method, hOsMA);
      direction = 0;
   }
   ...
   virtual string symbol() override
   {
      return _symbol;
   }

   virtual ENUM_TIMEFRAMES timeframe() override
   {
      return _timeframe;
   }
}
```

Dado que ahora el símbolo y el marco temporal de las señales pueden diferir del símbolo y el período del gráfico, hemos ampliado la interfaz *TradingSignal* añadiendo métodos de lectura. Los valores reales se pasan al constructor en *OnInit*.

```
int OnInit()
{
   ...
   strategy = new SimpleStrategy(
      new BandOsMaSignal(SignalSymbol != "" ? SignalSymbol : _Symbol,
         SignalSymbol != "" ? SignalTimeframe : _Period,
         p.fast, p.slow, SignalOsMA, PriceOsMA,
         BandsMA, BandsShift, BandsDeviation,
         PeriodMA, ShiftMA, MethodMA),
         Magic, StopLoss, Lots);
   return INIT_SUCCEEDED;
}
```

En la clase *SimpleStrategy*, el método *trade* comprueba ahora la aparición de una nueva barra no según el gráfico actual, sino según las propiedades de la señal.

```
   virtual bool trade() override
   {
      // looking for a signal once at the opening of the bar of the desired symbol an
      if(lastBar == iTime(command[].symbol(), command[].timeframe(), 0)) return false

      int s = command[].signal(); // get signal
      ...
   }
```

Para un experimento comparativo con la misma configuración, el Asesor Experto *BandOsMACustomSignal.mq5* debe lanzarse en EURUSD (puede utilizar M1 u otro marco de tiempo), y EURUSD_Eqv1000 debe especificarse en el parámetro *SignalSymbol*. *SignalTimeframe* debe dejarse igual a PERIOD_M1 de manera predeterminada. Como resultado, obtendremos un informe similar.

Bars	545621	Ticks	2159010	Symbols	1
Initial Deposit	10 000.00				
Total Net Profit	77.36	Balance Drawdow...	12.97	Equity Drawdown ...	14.60
Gross Profit	601.63	Balance Drawdow...	45.33 (0.45%)	Equity Drawdown ...	47.19 (0.47%)
Gross Loss	-524.27	Balance Drawdow...	0.45% (45.33)	Equity Drawdown ...	0.47% (47.19)
Profit Factor	1.15	Expected Payoff	0.11	Margin Level	81068.86%
Recovery Factor	1.64	Sharpe Ratio	1.85	Z-Score	-1.40 (83.85%)
AHPR	1.0000 (0.00...	LR Correlation	0.90	OnTester result	77.49928252...
GHPR	1.0000 (0.00...	LR Standard Error	12.35		
Total Trades	726	Short Trades (won ...	364 (53.57%)	Long Trades (won ...	362 (50.55%)
Total Deals	1452	Profit Trades (% of...	378 (52.07%)	Loss Trades (% of t...	348 (47.93%)
	Largest	profit trade	9.14	loss trade	-5.00
	Average	profit trade	1.59	loss trade	-1.51
	Maximum	consecutive wins ($)	8 (12.73)	consecutive losses ...	6 (-12.48)
	Maximal	consecutive profit ...	19.25 (6)	consecutive loss (c...	-12.48 (6)
	Average	consecutive wins	2	consecutive losses	2

Informe del probador al operar en el gráfico EURUSD basado en las señales del símbolo de equivolumen EURUSD_Eqv1000

El número de barras y ticks es diferente aquí porque EURUSD se eligió como instrumento probado y no el EURUSD_Eqv1000 personalizado.

Los resultados de las tres pruebas son ligeramente diferentes. Esto se debe al «empaquetamiento» de las cotizaciones en barras de minutos y a una ligera desincronización de los movimientos de precio de los instrumentos originales y personalizados. ¿Cuál de los resultados es más exacto? Lo más probable es que esto dependa del sistema de trading concreto y de las características de su aplicación. En el caso de nuestro Asesor Experto *BandOsMa* con control sobre la apertura de barras, la versión con trading directo en EURUSD_Eqv1000 debería tener los resultados más realistas. En teoría, casi siempre se cumple la regla empírica según la cual, de varias comprobaciones alternativas, la más fiable es la menos rentable.

Así, hemos analizado un par de técnicas para adaptar los Asesores Expertos para operar con símbolos personalizados que tienen un prototipo entre los símbolos de trabajo del bróker. Sin embargo, esta situación no es obligatoria. En muchos casos, los símbolos personalizados se generan a partir de datos de sistemas externos, como las bolsas de criptomonedas. El trading en ellos debe hacerse utilizando su API pública con funciones de red de MQL5.

Emulación de tipos especiales de gráficos con símbolos personalizados

Muchos operadores utilizan tipos especiales de gráficos, en los que el tiempo real continuo se excluye de la consideración. Esto incluye no sólo barras de igual rango y equivolumen, sino también Renko, Point-And-Figure (PAF), Kagi, y otros. Los símbolos personalizados permiten emular este tipo de gráficos en MetaTrader 5 utilizando gráficos de marco temporal M1, pero deben tratarse con precaución cuando se trata de simular sistemas de trading en lugar de análisis técnicos.

En tipos especiales de gráficos, la hora real de apertura de la barra (con precisión de milisegundos) casi siempre no coincide exactamente con el minuto con el que se marcará la barra M1. Así, el precio de apertura de una barra personalizada difiere del precio de apertura de la barra M1 de un símbolo estándar.

Además, otros precios OHLC también diferirán porque la duración real de la formación de la barra M1 en un gráfico especial no es igual a un minuto. Por ejemplo, 1000 ticks para un gráfico de equivolumen pueden acumularse durante más de 5 minutos.

El precio de cierre de una barra personalizada tampoco se corresponde con la hora de cierre real porque una barra personalizada es, técnicamente, una barra M1, es decir, tiene una duración nominal de 1 minuto.

Se debe tener especial cuidado cuando se trabaja con este tipo de gráficos como el clásico Renko o PAF. El hecho es que sus barras de inversión tienen un precio de apertura con un gap desde el cierre de la barra anterior. Así, el precio de apertura se convierte en un indicador del movimiento futuro de los precios.

Se supone que el análisis de tales gráficos se realiza en función de las barras formadas, es decir, su precio característico es el precio de cierre, pero cuando se trabaja por barras, el probador proporciona sólo el precio de apertura de la barra actual (última) (no existe el modo por precios de cierre). Aunque tomemos las señales de los indicadores de las barras cerradas (normalmente de la 1ª), de todas formas las transacciones se realizan al precio actual de la barra 0. E incluso si pasamos a los modos de ticks, el probador siempre genera ticks según las reglas habituales, guiándose por puntos de referencia basados en la configuración de cada barra. El probador no tiene

en cuenta la estructura y el comportamiento de los gráficos especiales, que intentamos emular visualmente con las barras M1.

Operar en el probador utilizando tales símbolos en cualquier modo (por precios de apertura, M1 OHLC, o por ticks) afecta a la precisión de los resultados: son demasiado optimistas y pueden servir como fuente de expectativas demasiado altas. A este respecto, es esencial comprobar el sistema de trading no en un gráfico Renko o PAF separado, sino junto con la ejecución de órdenes en un símbolo real.

Los símbolos personalizados también pueden utilizarse para segundos marcos temporales o gráficos de ticks. En este caso se genera también tiempo virtual para barras y ticks, desacoplado del tiempo real. Por lo tanto, estos gráficos son muy adecuados para el análisis operativo, pero requieren una atención adicional a la hora de desarrollar y simular estrategias de trading, especialmente las multisímbolo.

Una alternativa para cualquier símbolo personalizado es el cálculo independiente de arrays de barras y ticks dentro de un Asesor Experto o indicador. No obstante, la depuración y visualización de tales estructuras requiere un esfuerzo adicional.

7.3 Calendario económico

A la hora de elaborar estrategias de trading conviene tener en cuenta los factores fundamentales que afectan al mercado. MetaTrader 5 tiene un calendario económico integrado que está disponible en la interfaz del programa como una pestaña separada en la barra de herramientas, así como etiquetas, que se muestran de forma opcional directamente en el gráfico. El calendario puede activarse mediante un indicador independiente en la pestaña Comunidad del cuadro de diálogo de configuración del terminal (no es necesario iniciar sesión en la comunidad).

Dado que MetaTrader 5 admite el trading algorítmico, también se puede acceder a los eventos del calendario económico de forma programática desde la API de MQL5. En este capítulo presentaremos las funciones y estructuras de datos que permiten leer, filtrar y controlar los cambios en los eventos económicos.

El calendario económico contiene la descripción, el calendario de publicación y los valores históricos de los indicadores macroeconómicos de muchos países. Para cada evento, se conocen el momento exacto de la publicación prevista, el grado de importancia, el impacto en divisas específicas, los valores previstos y otros atributos. Los valores reales de los indicadores macroeconómicos llegan a MetaTrader 5 inmediatamente en el momento de su publicación.

La disponibilidad del calendario permite analizar automáticamente los eventos entrantes y reaccionar a ellos en los Asesores Expertos de diversas maneras; por ejemplo, operando como parte de una estrategia de ruptura o fluctuaciones de volatilidad dentro del corredor. Por otro lado, conocer las próximas fluctuaciones del mercado le permite encontrar horas tranquilas en la programación y desactivar temporalmente aquellos robots para los que los fuertes movimientos de los precios son peligrosos debido a las posibles pérdidas.

Los valores de tipo *datetime* utilizados por todas las funciones y estructuras que trabajan con el calendario económico son iguales a la hora del servidor de trading (*TimeTradeServer*), incluida la zona horaria y el horario de verano. En otras palabras: para probar correctamente los Asesores Expertos que operan con noticias, su desarrollador debe cambiar de forma independiente las horas de las noticias históricas en aquellos periodos (aproximadamente medio año dentro de cada año) en los que el modo DST difiere del actual.

Las funciones de calendario no pueden utilizarse en el probador: al intentar llamar a cualquiera de ellas, obtenemos el error FUNCTION_NOT_ALLOWED (4014). En este sentido, la simulación de las estrategias basadas en el calendario implica primero guardar las entradas del calendario en almacenamientos externos (por ejemplo, en archivos) al ejecutar el programa MQL en el gráfico en línea, y luego cargarlas y leerlas desde el programa MQL que se ejecuta en el probador.

7.3.1 Conceptos básicos del calendario

Al trabajar con el calendario, operaremos con varios conceptos, para cuya descripción formal MQL5 define tipos especiales de estructuras.

En primer lugar, los eventos están relacionados con países concretos, y cada país se describe utilizando la estructura *MqlCalendarCountry*.

```
struct MqlCalendarCountry
{
   ulong  id;              //country identifier according to ISO 3166-1
   string name;            // text name of the country (in the current terminal encod
   string code;            // two-letter country designation according to ISO 3166-1
   string currency;        // international country currency code
   string currency_symbol; // symbol/sign of the country's currency
   string url_name;        // country name used in the URL on the mql5.com website
};
```

En la siguiente sección descubriremos cómo obtener una lista de países disponibles en el calendario y sus atributos como un array de estructuras *MqlCalendarCountry*.

Por ahora, solo prestaremos atención al campo *id*. Es importante porque esta es la clave para determinar si los eventos del calendario pertenecen a un país concreto. En cada país (o en una asociación registrada de países, como la Unión Europea) existe una lista específica, conocida internacionalmente, de tipos de indicadores económicos y eventos informativos que afectan al mercado y que, por tanto, se incluyen en el calendario.

Cada tipo de evento se define mediante la estructura *MqlCalendarEvent*, en la que el campo *country_id* vincula de forma exclusiva el evento con el país. A continuación examinaremos los tipos de enumeraciones utilizados.

```
struct MqlCalendarEvent
{
   ulong                          id;          // event ID
   ENUM_CALENDAR_EVENT_TYPE       type;        // event type
   ENUM_CALENDAR_EVENT_SECTOR     sector;      // sector to which the event belongs
   ENUM_CALENDAR_EVENT_FREQUENCY  frequency;   // frequency (periodicity) of the event
   ENUM_CALENDAR_EVENT_TIMEMODE   time_mode;   // event time mode
   ulong                          country_id;  // country identifier
   ENUM_CALENDAR_EVENT_UNIT       unit;        // indicator unit
   ENUM_CALENDAR_EVENT_IMPORTANCE importance;  // importance of the event
   ENUM_CALENDAR_EVENT_MULTIPLIER multiplier;  // indicator multiplier
   uint                           digits;      // number of decimal places
   string                         source_url;  // URL of the event publication source
   string                         event_code;  // event code
   string                         name;        // text name of the event in the termin
};
```

Es importante comprender que la estructura *MqlCalendarEvent* describe exactamente el tipo de evento (por ejemplo, la publicación del Índice de Precios al Consumo, IPC), pero no un evento concreto que puede producirse una vez al trimestre, una vez al mes o según otro calendario. Contiene las características generales del evento, como importancia, frecuencia, relación con el sector de la economía, unidades de medida, nombre y fuente de información. En cuanto a los indicadores reales y previstos, se proporcionarán en las entradas del calendario para cada evento específico de este tipo: estas entradas se almacenan como estructuras *MqlCalendarValue*, que se tratarán más adelante. Las funciones para consultar los tipos de eventos admitidos se presentarán en secciones posteriores.

El tipo de evento del campo *type* se especifica como uno de los valores de la enumeración ENUM_CALENDAR_EVENT_TYPE.

Identificador	Descripción
CALENDAR_TYPE_EVENT	Evento (reunión, discurso, etc.)
CALENDAR_TYPE_INDICATOR	Indicador económico
CALENDAR_TYPE_HOLIDAY	Vacaciones (fin de semana)

El sector de la economía al que pertenece el evento se selecciona de la enumeración ENUM_CALENDAR_EVENT_SECTOR.

Identificador	Descripción
CALENDAR_SECTOR_NONE	Sector no establecido
CALENDAR_SECTOR_MARKET	Mercado, intercambio
CALENDAR_SECTOR_GDP	Producto Interior Bruto (PIB)
CALENDAR_SECTOR_JOBS	Mercado laboral
CALENDAR_SECTOR_PRICES	Precios

Identificador	Descripción
CALENDAR_SECTOR_MONEY	Dinero
CALENDAR_SECTOR_TRADE	Comercio
CALENDAR_SECTOR_GOVERNMENT	Gobierno
CALENDAR_SECTOR_BUSINESS	Empresas
CALENDAR_SECTOR_CONSUMER	Consumo
CALENDAR_SECTOR_HOUSING	Vivienda
CALENDAR_SECTOR_TAXES	Impuestos
CALENDAR_SECTOR_HOLIDAYS	Vacaciones

La frecuencia del evento se indica en el campo *frequency* utilizando la enumeración ENUM_CALENDAR_EVENT_FREQUENCY.

Identificador	Descripción
CALENDAR_FREQUENCY_NONE	La frecuencia de publicación no está establecida.
CALENDAR_FREQUENCY_WEEK	Semanal
CALENDAR_FREQUENCY_MONTH	Mensual
CALENDAR_FREQUENCY_QUARTER	Trimestral
CALENDAR_FREQUENCY_YEAR	Anual
CALENDAR_FREQUENCY_DAY	Diario

La duración del evento (*time_mode*) puede describirse mediante uno de los elementos de la enumeración ENUM_CALENDAR_EVENT_TIMEMODE.

Identificador	Descripción
CALENDAR_TIMEMODE_DATETIME	Se conoce la hora exacta del acontecimiento
CALENDAR_TIMEMODE_DATE	El evento dura todo el día
CALENDAR_TIMEMODE_NOTIME	La hora no se publica
CALENDAR_TIMEMODE_TENTATIVE	Solo se conoce de antemano el día, pero no la hora exacta del evento (la hora se especifica a posteriori).

La importancia del evento se especifica en el campo *importance* utilizando la enumeración ENUM_CALENDAR_EVENT_IMPORTANCE.

Identificador	Descripción
CALENDAR_IMPORTANCE_NONE	No establecida
CALENDAR_IMPORTANCE_LOW	Baja
CALENDAR_IMPORTANCE_MODERATE	Moderada
CALENDAR_IMPORTANCE_HIGH	Alta

Las unidades de medida en las que se dan los valores de los eventos se definen en el campo *unit* como miembro de la enumeración ENUM_CALENDAR_EVENT_UNIT.

Identificador	Descripción
CALENDAR_UNIT_NONE	La unidad no está establecida.
CALENDAR_UNIT_PERCENT	Interés (%)
CALENDAR_UNIT_CURRENCY	Divisa nacional
CALENDAR_UNIT_HOUR	Número de horas
CALENDAR_UNIT_JOB	Número de centros de trabajo
CALENDAR_UNIT_RIG	Plataformas de perforación
CALENDAR_UNIT_USD	Dólares estadounidenses
CALENDAR_UNIT_PEOPLE	Número de personas
CALENDAR_UNIT_MORTGAGE	Número de préstamos hipotecarios
CALENDAR_UNIT_VOTE	Número de votos
CALENDAR_UNIT_BARREL	Cantidad en barriles
CALENDAR_UNIT_CUBICFEET	Volumen en pies cúbicos
CALENDAR_UNIT_POSITION	Volumen neto de posiciones especulativas en contratos
CALENDAR_UNIT_BUILDING	Número de edificios

En algunos casos, los valores de un indicador económico requieren un *multiplier* según uno de los elementos de la enumeración ENUM_CALENDAR_EVENT_MULTIPLIER.

Identificador	Descripción
CALENDAR_MULTIPLIER_NONE	El multiplicador no está establecido.
CALENDAR_MULTIPLIER_THOUSANDS	Miles
CALENDAR_MULTIPLIER_MILLIONS	Millones
CALENDAR_MULTIPLIER_BILLIONS	Miles de millones
CALENDAR_MULTIPLIER_TRILLIONS	Billones

Así pues, hemos considerado todos los tipos de datos especiales utilizados para describir los tipos de eventos en la estructura *MqlCalendarEvent*.

Una entrada de calendario independiente se forma como una estructura *MqlCalendarValue*. Su descripción detallada figura a continuación, pero por ahora es importante prestar atención al siguiente matiz: *MqlCalendarValue* tiene el campo *event_id* que apunta al identificador del tipo de evento, es decir, contiene una de las estructuras *id* existentes en *MqlCalendarEvent*.

Como vimos anteriormente, la estructura *MqlCalendarEvent* está a su vez relacionada con *MqlCalendarCountry* a través del campo *country_id*. Así, una vez introducida la información sobre un país o un tipo de evento concreto en la base de datos del calendario, es posible registrar un número arbitrario de eventos similares para ellos. Por supuesto, el responsable de rellenar la base de datos es el proveedor de información, no los desarrolladores.

Resumamos el subtotal; el sistema almacena tres tablas internas por separado:

- La tabla de estructura *MqlCalendarCountry* para describir los países;
- La tabla de estructura *MqlCalendarEvent* con descripciones de los tipos de eventos;
- La tabla de estructura *MqlCalendarValue* con indicadores de eventos específicos de varios tipos.

Al hacer referencia a los identificadores de tipo de evento, se elimina la duplicación de información de los registros de eventos específicos. Por ejemplo, las publicaciones mensuales de los valores del IPC sólo se refieren a la misma estructura *MqlCalendarEvent* con las características generales de este tipo de evento. Si no fuera por las diferentes tablas, sería necesario repetir las mismas propiedades en cada entrada del calendario del IPC. Este enfoque para establecer relaciones entre tablas con datos que utilizan campos identificadores se denomina *relational*, y volveremos a él en el capítulo sobre SQLite. Esto se ilustra en el siguiente diagrama:

Diagrama de enlaces entre estructuras por campos con identificadores

Todas las tablas se almacenan en la base de datos interna del calendario, que se mantiene constantemente actualizada mientras el terminal está conectado al servidor.

Las entradas del calendario (eventos específicos) son estructuras de *MqlCalendarValue*. También se identifican por su propio número único en el campo *id* (cada una de las tres tablas tiene su propio campo *id*).

```
struct MqlCalendarValue
{
   ulong      id;                // entry ID
   ulong      event_id;          // event type ID
   datetime   time;              // time and date of the event
   datetime   period;            // reporting period of the event
   int        revision;          // revision of the published indicator in relation
   long       actual_value;      // actual value in ppm or LONG_MIN
   long       prev_value;        // previous value in ppm or LONG_MIN
   long       revised_prev_value; // revised previous value in ppm or LONG_MIN
   long       forecast_value;    // forecast value in ppm or LONG_MIN
   ENUM_CALENDAR_EVENT_IMPACT impact_type;  // potential impact on the exchange rate

 // functions for checking values
   bool HasActualValue(void) const;     // true if the actual_value field is filled
   bool HasPreviousValue(void) const;   // true if the prev_value field is filled
   bool HasRevisedValue(void) const;    // true if the revised_prev_value field is fi
   bool HasForecastValue(void) const;   // true if the forecast_value field is filled

   // functions for getting values
   double GetActualValue(void) const;   // actual_value or nan if value is not set
   double GetPreviousValue(void) const; // prev_value or nan if value is not set
   double GetRevisedValue(void) const;  // revised_prev_value or nan if value is not
   double GetForecastValue(void) const; // forecast_value or nan if value is not set
};
```

Para cada evento, además de la hora de su publicación (*time*), también se almacenan los cuatro valores siguientes:

- Valor real (*actual_value*), que se conoce inmediatamente después de la publicación de la noticia;
- Valor anterior (*prev_value*), que se conoció en la última publicación de la misma noticia;
- Valor revisado del indicador anterior, *revised_prev_value* (si se ha modificado desde la última publicación);
- Valor previsto (*forecast_value*).

Obviamente, no todos los campos deben rellenarse necesariamente. Así, el valor actual está ausente (aún no se conoce) para los eventos futuros, y la revisión de los valores pasados tampoco se produce siempre. Además, los cuatro campos solo tienen sentido para los indicadores cuantitativos, mientras que el calendario también refleja los discursos, las reuniones y los días festivos de los reguladores.

Un campo vacío (sin valor) se indica mediante la constante LONG_MIN (-9223372036854775808). Si se especifica el valor del campo (no igual a LONG_MIN), entonces corresponde al valor real del indicador multiplicado por un millón de veces, es decir, para obtener el indicador en la forma (real) habitual, es necesario dividir el valor del campo por 1 000 000.

Para comodidad del programador, la estructura define 4 métodos *Has* para comprobar que el campo está lleno, así como 4 métodos *Get* que devuelven el valor del campo correspondiente ya convertido en un número real, y en el caso de que no esté lleno, el método devolverá NaN (no es un número -Not A Number-).

A veces, para obtener valores absolutos (si son necesarios para el algoritmo), es importante analizar adicionalmente la propiedad *multiplier* en la estructura *MqlCalendarEvent*, ya que algunos valores se

especifican en múltiples unidades según la enumeración ENUM_CALENDAR_EVENT_MULTIPLIER. Además, *MqlCalendarEvent* dispone del campo *digits*, que especifica el número de dígitos significativos de los valores recibidos para su correcto formateo posterior (por ejemplo, en una llamada a *NormalizeDouble*).

El periodo de referencia (para el que se calcula el indicador publicado) se establece en el campo *period* como su primer día. Por ejemplo, si el indicador se calcula mensualmente, la fecha '2022.05.01 00:00:00' significa el mes de mayo. La duración del periodo (por ejemplo, mes, trimestre, año) se define en el campo *frequency* de la estructura relacionada *MqlCalendarEvent*: el tipo de este campo es la enumeración especial ENUM_CALENDAR_EVENT_FREQUENCY descrita anteriormente, junto con otras enumeraciones.

Especialmente interesante es el campo *impact_type*, en el que, tras la publicación de la noticia, la dirección de influencia de la divisa correspondiente en el tipo de cambio se establece automáticamente comparando los valores actuales y previstos. Esta influencia puede ser positiva (se espera que la divisa se aprecie) o negativa (se espera que la divisa se deprecie). Por ejemplo, un descenso de las ventas mayor de lo previsto se calificaría de negativo, y un descenso del desempleo mayor, de positivo. Pero esta característica no se interpreta inequívoca para todos los eventos (algunos indicadores económicos se consideran contradictorios) y, además, hay que prestar atención a las cifras relativas de los cambios.

El impacto potencial de un evento sobre el tipo de cambio de la divisa nacional se indica utilizando la enumeración ENUM_CALENDAR_EVENT_IMPACT.

Identificador	Descripción
CALENDAR_IMPACT_NA	No se establece la influencia.
CALENDAR_IMPACT_POSITIVE	Influencia positiva
CALENDAR_IMPACT_NEGATIVE	Influencia negativa

Otro concepto importante del calendario es el hecho de su cambio. Por desgracia, no existe una estructura especial para el cambio. La única propiedad que tiene un cambio es su ID único, que es un número entero asignado por el sistema cada vez que se modifica la base del calendario interno.

Como sabe, el calendario es modificado constantemente por los proveedores de información: se le añaden nuevos eventos próximos y se corrigen los indicadores y previsiones ya publicados. Por lo tanto, es muy importante realizar un seguimiento de las ediciones, cuya aparición permite detectar el aumento periódico del número de cambios.

El tiempo de edición con un identificador específico y su esencia no están disponibles en MQL5. Si es necesario, los programas MQL deben implementar ellos mismos las consultas periódicas del estado del calendario y el análisis de los registros.

Un conjunto de funciones MQL5 permite obtener información sobre países, tipos de eventos y entradas concretas del calendario, así como sobre sus cambios. Lo estudiaremos en las secciones siguientes.

> ¡Atención! Al acceder al calendario por primera vez (si no se ha abierto antes la pestaña Calendario de la barra de herramientas del terminal), la sincronización de la base de datos interna del calendario con el servidor puede tardar varios segundos.

7.3.2 Obtener la lista y las descripciones de los países disponibles

Puede obtener una lista completa de los países para los que se emiten eventos en el calendario utilizando la función *CalendarCountries*.

int CalendarCountries(MqlCalendarCountry &countries[])

La función rellena el array *countries* pasado por referencia con estructuras *MqlCalendarCountry*. El array puede ser dinámico o fijo, de tamaño suficiente.

En caso de éxito, la función devuelve el número de descripciones de países recibidas del servidor, o 0 en caso de error. Entre los posibles códigos de error en *_LastError* podemos encontrar, en concreto, 5401 (ERR_CALENDAR_TIMEOUT, límite de tiempo de solicitud excedido) o 5400 (ERR_CALENDAR_MORE_DATA, si el tamaño del array fijo es insuficiente para obtener descripciones de todos los países). En este último caso, el sistema copiará sólo lo que quepa.

Escribamos un sencillo script *CalendarCountries.mq5*, que obtiene la lista completa de países y la registra.

```
void OnStart()
{
   MqlCalendarCountry countries[];
   PRTF(CalendarCountries(countries));
   ArrayPrint(countries);
}
```

He aquí un ejemplo de resultado:

```
CalendarCountries(countries)=23 / ok
     [id]           [name] [code] [currency] [currency_symbol]       [url_name] [rese
[ 0]  554 "New Zealand"    "NZ"   "NZD"      "$"               "new-zealand"
[ 1]  999 "European Union" "EU"   "EUR"      "€"               "european-union"
[ 2]  392 "Japan"          "JP"   "JPY"      "¥"               "japan"
[ 3]  124 "Canada"         "CA"   "CAD"      "$"               "canada"
[ 4]   36 "Australia"      "AU"   "AUD"      "$"               "australia"
[ 5]  156 "China"          "CN"   "CNY"      "¥"               "china"
[ 6]  380 "Italy"          "IT"   "EUR"      "€"               "italy"
[ 7]  702 "Singapore"      "SG"   "SGD"      "R$"              "singapore"
[ 8]  276 "Germany"        "DE"   "EUR"      "€"               "germany"
[ 9]  250 "France"         "FR"   "EUR"      "€"               "france"
[10]   76 "Brazil"         "BR"   "BRL"      "R$"              "brazil"
[11]  484 "Mexico"         "MX"   "MXN"      "Mex$"            "mexico"
[12]  710 "South Africa"   "ZA"   "ZAR"      "R"               "south-africa"
[13]  344 "Hong Kong"      "HK"   "HKD"      "HK$"             "hong-kong"
[14]  356 "India"          "IN"   "INR"      "₹"               "india"
[15]  578 "Norway"         "NO"   "NOK"      "Kr"              "norway"
[16]    0 "Worldwide"      "WW"   "ALL"      ""                "worldwide"
[17]  840 "United States"  "US"   "USD"      "$"               "united-states"
[18]  826 "United Kingdom" "GB"   "GBP"      "£"               "united-kingdom"
[19]  756 "Switzerland"    "CH"   "CHF"      "₣"               "switzerland"
[20]  410 "South Korea"    "KR"   "KRW"      "₩"               "south-korea"
[21]  724 "Spain"          "ES"   "EUR"      "€"               "spain"
[22]  752 "Sweden"         "SE"   "SEK"      "Kr"              "sweden"
```

Es importante señalar que el identificador 0 (código «WW» y pseudodivisa «ALL») corresponde a eventos mundiales (relativos a muchos países, por ejemplo, las reuniones del G7, G20), y la divisa «EUR» está asociada a varios países de la UE disponibles en el calendario (como puede ver, no se presenta toda la zona euro). Además, la propia Unión Europea tiene un identificador genérico: 999.

Si está interesado en un país concreto, puede comprobar su disponibilidad mediante un código numérico según la norma ISO 3166-1. En concreto, en el registro anterior, estos códigos aparecen en la primera columna (campo *id*).

Para obtener la descripción de un país por su ID especificado en el parámetro *id*, puede utilizar la función *CalendarCountryById*.

```
bool CalendarCountryById(const long id, MqlCalendarCountry &country)
```

Si tiene éxito, la función devolverá *true* y rellenará los campos de la estructura *country*.

Si no se encuentra el país, obtendremos *false*, y en *_LastError* obtendremos un código de error 5402 (ERR_CALENDAR_NO_DATA).

Para ver un ejemplo de uso de esta función, consulte Obtener registros de eventos por país o divisa.

7.3.3 Consultar tipos de eventos por país y divisa

El calendario de eventos económicos y días festivos tiene sus propias especificidades en cada país. Un programa MQL puede consultar los tipos de eventos dentro de un país concreto, así como los tipos de eventos asociados a una divisa determinada. Esto último es relevante en los casos en que varios países utilizan la misma divisa, como, por ejemplo, la mayoría de los miembros de la Unión Europea.

int CalendarEventByCountry(const string country, MqlCalendarEvent &events[])

La función *CalendarEventByCountry* rellena un array de estructuras *MqlCalendarEvent* pasadas por referencia con descripciones de todos los tipos de eventos disponibles en el calendario para el país especificado por el código de país de dos letras (según la norma ISO 3166-1 alfa-2). Hemos visto ejemplos de este tipo de códigos en la sección anterior, en el registro: EU para la Unión Europea, US para Estados Unidos, DE para Alemania, CN para China, etc.

El array receptor puede ser dinámico o fijo de tamaño suficiente.

La función devuelve el número de descripciones recibidas, y 0 en caso de error. En particular, si el array fijo no puede contener todos los eventos, la función la rellenará con la parte ajustada de los datos disponibles y establecerá el código *_LastError*, igual a CALENDAR_MORE_DATA (5400). También es posible que se produzcan errores de asignación de memoria (4004, ERR_NOT_ENOUGH_MEMORY) o que se agote el tiempo de espera de la solicitud de calendario al servidor (5401, ERR_CALENDAR_TIMEOUT).

Si el país con el código dado no existe, se producirá un INTERNAL_ERROR (4001).

Especificando NULL o una cadena vacía «» en lugar de *country*, puede obtener una lista completa de eventos para todos los países.

Probemos el rendimiento de la función utilizando el sencillo script *CalendarEventKindsByCountry.mq5*. Tiene un único parámetro de entrada que es el código del país que nos interesa.

```
input string CountryCode = "HK";
```

A continuación, se realiza una petición de tipos de eventos llamando a *CalendarEventByCountry*, y si tiene éxito, se registran los arrays resultantes.

```
void OnStart()
{
   MqlCalendarEvent events[];
   if(PRTF(CalendarEventByCountry(CountryCode, events)))
   {
      Print("Event kinds for country: ", CountryCode);
      ArrayPrint(events);
   }
}
```

He aquí un ejemplo del resultado (debido a que las líneas son largas, se han dividido artificialmente en 2 bloques para su publicación en el libro: el primer bloque contiene los campos numéricos de las estructuras *MqlCalendarEvent*, y el segundo bloque contiene campos de cadena).

```
CalendarEventByCountry(CountryCode,events)=26 / ok
Event kinds for country: HK
[id] [type] [sector] [frequency] [time_mode] [country_id] [unit] [importance] [multip
[ 0] 344010001 1 5 2 0 344 6 1 3 1 »
[ 1] 344010002 1 5 2 0 344 1 1 0 1 »
[ 2] 344020001 1 4 2 0 344 1 1 0 1 »
[ 3] 344020002 1 2 3 0 344 1 3 0 1 »
[ 4] 344020003 1 2 3 0 344 1 2 0 1 »
[ 5] 344020004 1 6 2 0 344 1 1 0 1 »
[ 6] 344020005 1 6 2 0 344 1 1 0 1 »
[ 7] 344020006 1 6 2 0 344 2 2 3 3 »
[ 8] 344020007 1 9 2 0 344 1 1 0 1 »
[ 9] 344020008 1 3 2 0 344 1 2 0 1 »
[10] 344030001 2 12 0 1 344 0 0 0 0 »
[11] 344030002 2 12 0 1 344 0 0 0 0 »
[12] 344030003 2 12 0 1 344 0 0 0 0 »
[13] 344030004 2 12 0 1 344 0 0 0 0 »
[14] 344030005 2 12 0 1 344 0 0 0 0 »
[15] 344030006 2 12 0 1 344 0 0 0 0 »
[16] 344030007 2 12 0 1 344 0 0 0 0 »
[17] 344030008 2 12 0 1 344 0 0 0 0 »
[18] 344030009 2 12 0 1 344 0 0 0 0 »
[19] 344030010 2 12 0 1 344 0 0 0 0 »
[20] 344030011 2 12 0 1 344 0 0 0 0 »
[21] 344030012 2 12 0 1 344 0 0 0 0 »
[22] 344030013 2 12 0 1 344 0 0 0 0 »
[23] 344030014 2 12 0 1 344 0 0 0 0 »
[24] 344030015 2 12 0 1 344 0 0 0 0 »
[25] 344500001 1 8 2 0 344 0 1 0 1 »
```

Continuación del registro (fragmento derecho).

```
» [source_url] [event_code] [name]
[ 0]» "https://www.hkma.gov.hk/eng/" "foreign-exchange-reserves" "Foreign Exchange Re
[ 1]» "https://www.hkma.gov.hk/eng/" "hkma-m3-money-supply-yy" "HKMA M3 Money Supply
[ 2]» "https://www.censtatd.gov.hk/en/" "cpi-yy" "CPI y/y"
[ 3]» "https://www.censtatd.gov.hk/en/" "gdp-qq" "GDP q/q"
[ 4]» "https://www.censtatd.gov.hk/en/" "gdp-yy" "GDP y/y"
[ 5]» "https://www.censtatd.gov.hk/en/" "exports-mm" "Exports y/y"
[ 6]» "https://www.censtatd.gov.hk/en/" "imports-mm" "Imports y/y"
[ 7]» "https://www.censtatd.gov.hk/en/" "trade-balance" "Trade Balance"
[ 8]» "https://www.censtatd.gov.hk/en/" "retail-sales-yy" "Retail Sales y/y"
[ 9]» "https://www.censtatd.gov.hk/en/" "unemployment-rate-3-months" "Unemployment Ra
[10]» "https://publicholidays.hk/" "new-years-day" "New Year's Day"
[11]» "https://publicholidays.hk/" "lunar-new-year" "Lunar New Year"
[12]» "https://publicholidays.hk/" "ching-ming-festival" "Ching Ming Festival"
[13]» "https://publicholidays.hk/" "good-friday" "Good Friday"
[14]» "https://publicholidays.hk/" "easter-monday" "Easter Monday"
[15]» "https://publicholidays.hk/" "birthday-of-buddha" "The Birthday of the Buddha"
[16]» "https://publicholidays.hk/" "labor-day" "Labor Day"
[17]» "https://publicholidays.hk/" "tuen-ng-festival" "Tuen Ng Festival"
[18]» "https://publicholidays.hk/" "hksar-establishment-day" "HKSAR Establishment Day
[19]» "https://publicholidays.hk/" "day-following-mid-autumn-festival" "The Day Follo
[20]» "https://publicholidays.hk/" "national-day" "National Day"
[21]» "https://publicholidays.hk/" "chung-yeung-festival" "Chung Yeung Festival"
[22]» "https://publicholidays.hk/" "christmas-day" "Christmas Day"
[23]» "https://publicholidays.hk/" "first-weekday-after-christmas-day" "The First Wee
[24]» "https://publicholidays.hk/" "day-following-good-friday" "The Day Following Goo
[25]» "https://www.markiteconomics.com" "nikkei-pmi" "S&P Global PMI"
```

int CalendarEventByCurrency(const string currency, MqlCalendarEvent &events[])

La función *CalendarEventByCurrency* rellena el array *events* pasado con descripciones de todos los tipos de eventos del calendario que están asociados con el *currency* especificado. La denominación de tres letras de las divisas es conocida por todos los operadores de Forex.

Si se especifica un código de divisa no válido, la función devolverá 0 (sin error) y un array vacío.

Especificando NULL o una cadena vacía «» en lugar de *currency*, puede obtener una lista completa de los eventos del calendario.

Vamos a probar la función utilizando el script *CalendarEventKindsByCurrency.mq5*. El parámetro de entrada especifica el código de divisa.

```
input string Currency = "CNY";
```

En el manejador *OnStart* solicitamos eventos y los enviamos al log.

```
void OnStart()
{
   MqlCalendarEvent events[];
   if(PRTF(CalendarEventByCurrency(Currency, events)))
   {
      Print("Event kinds for currency: ", Currency);
      ArrayPrint(events);
   }
}
```

He aquí un ejemplo del resultado (dado con abreviaturas).

```
CalendarEventByCurrency(Currency,events)=40 / ok
Event kinds for currency: CNY
[id] [type] [sector] [frequency] [time_mode] [country_id] [unit] [importance] [multip
[ 0] 156010001 1 4 2 0 156 1 2 0 1 »
[ 1] 156010002 1 4 2 0 156 1 1 0 1 »
[ 2] 156010003 1 4 2 0 156 1 1 0 1 »
[ 3] 156010004 1 2 3 0 156 1 3 0 1 »
[ 4] 156010005 1 2 3 0 156 1 2 0 1 »
[ 5] 156010006 1 9 2 0 156 1 2 0 1 »
[ 6] 156010007 1 8 2 0 156 1 2 0 1 »
[ 7] 156010008 1 8 2 0 156 0 3 0 1 »
[ 8] 156010009 1 8 2 0 156 0 3 0 1 »
[ 9] 156010010 1 8 2 0 156 1 2 0 1 »
[10] 156010011 0 5 0 0 156 0 2 0 0 »
[11] 156010012 1 3 2 0 156 1 2 0 1 »
[12] 156010013 1 8 2 0 156 1 1 0 1 »
[13] 156010014 1 8 2 0 156 1 1 0 1 »
[14] 156010015 1 8 2 0 156 0 3 0 1 »
[15] 156010016 1 8 2 0 156 1 2 0 1 »
[16] 156010017 1 9 2 0 156 1 2 0 1 »
[17] 156010018 1 2 3 0 156 1 2 0 1 »
[18] 156020001 1 6 2 3 156 6 2 3 2 »
[19] 156020002 1 6 2 3 156 1 1 0 1 »
[20] 156020003 1 6 2 3 156 1 1 0 1 »
[21] 156020004 1 6 2 3 156 2 2 3 2 »
[22] 156020005 1 6 2 3 156 1 1 0 1 »
[23] 156020006 1 6 2 3 156 1 1 0 1 »
...
```

Fragmento derecho.

```
» [source_url] [event_code] [name]
[ 0]» "http://www.stats.gov.cn/english/" "cpi-mm" "CPI m/m"
[ 1]» "http://www.stats.gov.cn/english/" "cpi-yy" "CPI y/y"
[ 2]» "http://www.stats.gov.cn/english/" "ppi-yy" "PPI y/y"
[ 3]» "http://www.stats.gov.cn/english/" "gdp-qq" "GDP q/q"
[ 4]» "http://www.stats.gov.cn/english/" "gdp-yy" "GDP y/y"
[ 5]» "http://www.stats.gov.cn/english/" "retail-sales-yy" "Retail Sales y/y"
[ 6]» "http://www.stats.gov.cn/english/" "industrial-production-yy" "Industrial Produ
[ 7]» "http://www.stats.gov.cn/english/" "manufacturing-pmi" "Manufacturing PMI"
[ 8]» "http://www.stats.gov.cn/english/" "non-manufacturing-pmi" "Non-Manufacturing P
[ 9]» "http://www.stats.gov.cn/english/" "fixed-asset-investment-yy" "Fixed Asset Inv
[10]» "http://www.stats.gov.cn/english/" "nbs-press-conference-on-economic-situation"
[11]» "http://www.stats.gov.cn/english/" "unemployment-rate" "Unemployment Rate"
[12]» "http://www.stats.gov.cn/english/" "industrial-profit-yy" "Industrial Profit y/
[13]» "http://www.stats.gov.cn/english/" "industrial-profit-ytd-yy" "Industrial Profi
[14]» "http://www.stats.gov.cn/english/" "composite-pmi" "Composite PMI"
[15]» "http://www.stats.gov.cn/english/" "industrial-production-ytd-yy" "Industrial P
[16]» "http://www.stats.gov.cn/english/" "retail-sales-ytd-yy" "Retail Sales YTD y/y"
[17]» "http://www.stats.gov.cn/english/" "gdp-ytd-yy" "GDP YTD y/y"
[18]» "http://english.customs.gov.cn/" "trade-balance-usd" "Trade Balance USD"
[19]» "http://english.customs.gov.cn/" "imports-usd-yy" "Imports USD y/y"
[20]» "http://english.customs.gov.cn/" "exports-usd-yy" "Exports USD y/y"
[21]» "http://english.customs.gov.cn/" "trade-balance" "Trade Balance"
[22]» "http://english.customs.gov.cn/" "imports-yy" "Imports y/y"
[23]» "http://english.customs.gov.cn/" "exports-yy" "Exports y/y"
...
```

Un lector atento observará que el identificador del tipo de evento contiene el código del país, el número de la fuente de noticias y el número de serie dentro de la fuente (la numeración empieza por 1). Así, el formato general del identificador de tipo de evento es: CCCSSNNNN, donde CCC es el código de país, SS es la fuente, NNNN es el número. Por ejemplo, 156020001 es la primera noticia de la segunda fuente para China y 344030010 es la décima noticia de la tercera fuente para Hong Kong. La única excepción son las noticias mundiales, para las que el código «país» no es 000, sino 1000.

7.3.4 Obtener descripciones de eventos por ID

Los programas MQL reales, por regla general, solicitan eventos de calendario actuales o próximos, filtrando por intervalo de tiempo, países, divisas u otros criterios. Las funciones de la API destinadas a ello, que aún no hemos considerado, devuelven estructuras *MqlCalendarValue*, que sólo almacenan el identificador del evento en lugar de su descripción. Por lo tanto, la función *CalendarEventById* puede ser útil si necesita extraer información completa.

bool CalendarEventById(ulong id, MqlCalendarEvent &event)

La función *CalendarEventById* obtiene la descripción del evento por su ID. La función devuelve una indicación de éxito o de error.

En la siguiente sección se ofrece un ejemplo de cómo utilizar esta función.

7.3.5 Obtener registros de eventos por país o divisa

En el calendario se consultan eventos específicos de diversa índole para un intervalo de fechas determinado y se filtran por país o divisa.

int CalendarValueHistory(MqlCalendarValue &values[], datetime from, datetime to = 0, const string country = NULL, const string currency = NULL)

La función *CalendarValueHistory* rellena el array *values* pasado por referencia con entradas de calendario en el intervalo de tiempo comprendido entre *from* y *to*. Ambos parámetros pueden incluir fecha y hora. El valor *from* se incluye en el intervalo, pero el valor *to* no. En otras palabras: la función selecciona las entradas del calendario (estructuras*MqlCalendarValue*), en las que se cumple la siguiente condición compuesta para la propiedad *time*: *from* <= *time* < *to*.

La hora de inicio *from* debe especificarse, mientras que la hora de finalización *to* es opcional: si se omite o es igual a 0, todos los eventos futuros se copian en el array.

El tiempo *to* allí debe ser mayor que *from*, excepto cuando es 0. Una combinación especial para consultar todos los eventos disponibles (tanto pasados como futuros) es cuando *from* y *to* son ambos 0.

Si el array receptor es dinámico, se le asignará memoria automáticamente. Si el array tiene un tamaño fijo, el número de entradas copiadas no será mayor que el tamaño del array.

Los parámetros *country* y *currency* permiten establecer un filtrado adicional de los registros por país o divisa. El parámetro *country* acepta un código de país de dos letras ISO 3166-1 alfa-2 (por ejemplo, «DE», «FR», «EU»), y el parámetro *currency* acepta una designación de divisa de tres letras (por ejemplo, «EUR», «CNY»).

El valor por defecto NULL o una cadena vacía «» en cualquiera de los parámetros equivale a la ausencia del filtro correspondiente.

Si se especifican ambos filtros, solo se seleccionan los valores de aquellos eventos para los que se cumplen simultáneamente ambas condiciones: país y divisa. Esto puede resultar útil si el calendario incluye países con varias divisas, cada una de las cuales circula también en varios países. Por el momento no hay eventos de este tipo en el calendario. Para obtener los eventos en los países de la Eurozona, basta con especificar el código de un país concreto o «EU», y se asumirá la moneda «EUR».

La función devuelve el número de elementos copiados y puede establecer un código de error. En concreto, si se supera el tiempo de espera de la solicitud del servidor, en *_LastError* obtenemos el error 5401 (ERR_CALENDAR_TIMEOUT). Si en el array fijo no caben todos los registros, el código será igual a 5400 (ERR_CALENDAR_MORE_DATA), pero el array se llenará. Al asignar memoria a un array dinámico es posible que se produzca el error 4004 (ERR_NOT_ENOUGH_MEMORY).

> ¡Atención! El orden de los elementos de un array puede ser distinto del cronológico. Tiene que ordenar los registros por tiempo.

Utilizando la función *CalendarValueHistory* podríamos consultar los próximos eventos de la siguiente manera:

```
MqlCalendarValue values[];
if(CalendarValueHistory(values, TimeCurrent()))
{
   ArrayPrint(values);
}
```

Sin embargo, con este código obtendremos una tabla con información insuficiente, en la que los nombres de los eventos, la importancia y los códigos de divisa quedarán ocultos tras el identificador del evento en el campo *MqlCalendarValue::event_id* e, indirectamente, tras el identificador del país en el campo *MqlCalendarEvent::country_id*. Para que la salida de información sea más fácil de usar, debe

solicitar una descripción del evento mediante el código de evento, tomar el código de país de esta descripción y obtener sus atributos. Veámoslo en el script de ejemplo *CalendarForDates.mq5*.

En los parámetros de entrada ofreceremos la posibilidad de introducir el código de país y la divisa para el filtrado. Por defecto, se solicitan eventos para la Unión Europea.

```
input string CountryCode = "EU";
input string Currency = "";
```

El intervalo de fechas de los eventos contará automáticamente con un tiempo de ida y otro de vuelta. Este «algún tiempo» también se dejará a elección del usuario entre tres opciones: un día, una semana o un mes.

```
#define DAY_LONG   60 * 60 * 24
#define WEEK_LONG  DAY_LONG * 7
#define MONTH_LONG DAY_LONG * 30
#define YEAR_LONG  MONTH_LONG * 12

enum ENUM_CALENDAR_SCOPE
{
   SCOPE_DAY = DAY_LONG,
   SCOPE_WEEK = WEEK_LONG,
   SCOPE_MONTH = MONTH_LONG,
   SCOPE_YEAR = YEAR_LONG,
};

input ENUM_CALENDAR_SCOPE Scope = SCOPE_DAY;
```

Definamos nuestra estructura *MqlCalendarRecord*, derivada de *MqlCalendarValue*, y añadámosle campos para una presentación conveniente de los atributos que serán rellenados por enlaces (identificadores) de estructuras dependientes.

```
struct MqlCalendarRecord: public MqlCalendarValue
{
   static const string importances[];

   string importance;
   string name;
   string currency;
   string code;
   double actual, previous, revised, forecast;
   ...
};

static const string MqlCalendarRecord::importances[] = {"None", "Low", "Medium", "Hig
```

Entre los campos añadidos hay líneas con importancia (uno de los valores del array estático *importances*), el nombre del evento, el país y la divisa, así como cuatro valores con el formato *double*. En realidad, esto significa duplicar la información en aras de la presentación visual al imprimirla. Más adelante prepararemos un «envoltorio» más avanzado para el calendario.

Para rellenar el objeto necesitaremos un constructor paramétrico que tome la estructura original *MqlCalendarValue*. Después de copiar implícitamente todos los campos heredados en el nuevo objeto mediante el operador '=', llamamos al método especialmente preparado *extend*.

```
MqlCalendarRecord() { }

MqlCalendarRecord(const MqlCalendarValue &value)
{
   this = value;
   extend();
}
```

En el método *extend* obtenemos la descripción del evento por su identificador. A continuación, basándonos en el identificador de país de la descripción del evento, obtenemos una estructura con atributos de país. Después, podemos rellenar la primera mitad de los campos añadidos a partir de las estructuras recibidas *MqlCalendarEvent* y *MqlCalendarCountry*.

```
void extend()
{
   MqlCalendarEvent event;
   CalendarEventById(event_id, event);

   MqlCalendarCountry country;
   CalendarCountryById(event.country_id, country);

   importance = importances[event.importance];
   name = event.name;
   currency = country.currency;
   code = country.code;

   MqlCalendarValue value = this;

   actual = value.GetActualValue();
   previous = value.GetPreviousValue();
   revised = value.GetRevisedValue();
   forecast = value.GetForecastValue();
}
```

A continuación, llamamos a los métodos integrados en *Get* para rellenar cuatro campos de tipo *double* con indicadores financieros.

Ahora podemos utilizar la nueva estructura en el manejador *OnStart* principal.

```
void OnStart()
{
   MqlCalendarValue values[];
   MqlCalendarRecord records[];
   datetime from = TimeCurrent() - Scope;
   datetime to = TimeCurrent() + Scope;
   if(PRTF(CalendarValueHistory(values, from, to, CountryCode, Currency)))
   {
      for(int i = 0; i < ArraySize(values); ++i)
      {
         PUSH(records, MqlCalendarRecord(values[i]));
      }
      Print("Near past and future calendar records (extended): ");
      ArrayPrint(records);
   }
}
```

Aquí el array de estructuras estándar *MqlCalendarValue* se rellena llamando a *CalendarValueHistory* para las condiciones actuales establecidas en los parámetros de entrada. A continuación, todos los elementos se transfieren al array *MqlCalendarRecord*. Además, mientras se crean los objetos, se amplían con información adicional. Por último, el array de eventos se envía al registro.

Las entradas del registro vienen bastante largas. En primer lugar, vamos a mostrar la mitad izquierda, que es exactamente lo que veríamos si imprimiéramos un array de estructuras estándar *MqlCalendarValue*.

```
CalendarValueHistory(values,from,to,CountryCode,Currency)=6 / ok
Near past and future calendar records (extended):
[id] [event_id] [time] [period] [revision] [actual_value] [prev_value] [revised_prev_
[0] 162723 999020003 2022.06.23 03:00:00 1970.01.01 00:00:00 0 -9223372036854775808 -
[1] 162724 999020003 2022.06.24 03:00:00 1970.01.01 00:00:00 0 -9223372036854775808 -
[2] 168518 999010034 2022.06.24 11:00:00 1970.01.01 00:00:00 0 -9223372036854775808 -
[3] 168515 999010031 2022.06.24 13:10:00 1970.01.01 00:00:00 0 -9223372036854775808 -
[4] 168509 999010014 2022.06.24 14:30:00 1970.01.01 00:00:00 0 -9223372036854775808 -
[5] 161014 999520001 2022.06.24 22:30:00 2022.06.21 00:00:00 0 -9223372036854775808 -
```

Aquí está la segunda mitad con la «descodificación» de nombres, importancia y significados.

```
CalendarValueHistory(values,from,to,CountryCode,Currency)=6 / ok
Near past and future calendar records (extended):
[importance] [name] [currency] [code] [actual] [previous] [revised] [forecast]
[0] "High" "EU Leaders Summit" "EUR" "EU" nan nan nan nan
[1] "High" "EU Leaders Summit" "EUR" "EU" nan nan nan nan
[2] "Medium" "ECB Supervisory Board Member McCaul Speech" "EUR" "EU" nan nan nan nan
[3] "Medium" "ECB Supervisory Board Member Fernandez-Bollo Speech" "EUR" "EU" nan nan
[4] "Medium" "ECB Vice President de Guindos Speech" "EUR" "EU" nan nan nan nan
[5] "Low" "CFTC EUR Non-Commercial Net Positions" "EUR" "EU" nan -6.00000 nan nan
```

7.3.6 Obtener registros de eventos de un tipo específico

Si es necesario, un programa MQL tiene la capacidad de solicitar eventos de un tipo específico: para ello, basta con conocer de antemano el identificador del evento, por ejemplo, utilizando las funciones

CalendarEventByCountry o *CalendarEventByCurrency* que se presentaron en la sección Consultar tipos de eventos por país y divisa.

int CalendarValueHistoryByEvent(ulong id, MqlCalendarValue &values[], datetime from, datetime to = 0)

La función *CalendarValueHistoryByEvent* rellena el array pasado por referencia con registros de eventos de un tipo específico indicado por el identificador *id*. Los parámetros *from* y *to* permiten limitar el intervalo de fechas en el que se buscan los eventos.

Si no se especifica el parámetro opcional *to*, todas las entradas del calendario se colocarán en el array, empezando por la hora de *from* y avanzando hacia el futuro. Para consultar todos los eventos pasados, establezca *from* en 0. Si ambos parámetros *from* y *to* son 0, se devolverán todos los eventos históricos y programados. En todos los demás casos, cuando *to* no es igual a 0, debe ser mayor que *from*.

El array *values* puede ser dinámico (entonces la función lo ampliará o reducirá automáticamente según la cantidad de datos) o de tamaño fijo (entonces sólo se copiará en el array la parte que quepa).

La función devuelve el número de caracteres copiados.

Como ejemplo, considere el script *CalendarStatsByEvent.mq5*, que calcula las estadísticas (frecuencia de aparición) de eventos de diferentes tipos para un país o moneda determinados en un intervalo de tiempo dado.

Las condiciones de análisis se especifican en las variables de entrada.

```
input string CountryOrCurrency = "EU";
input ENUM_CALENDAR_SCOPE Scope = SCOPE_YEAR;
```

En función de la longitud de la cadena *CountryOrCurrency*, se interpreta como un código de país (2 caracteres) o de divisa (3 caracteres).

Para recopilar estadísticas, declararemos una estructura; sus campos almacenarán el identificador y el nombre del tipo de evento, su importancia y el contador de dichos eventos.

```
struct CalendarEventStats
{
   static const string importances[];
   ulong id;
   string name;
   string importance;
   int count;
};

static const string CalendarEventStats::importances[] = {"None", "Low", "Medium", "Hi
```

En la función *OnStart*, primero solicitamos todo tipo de eventos utilizando la función *CalendarEventByCountry* o *CalendarEventByCurrency* hasta la profundidad especificada de la historia y hacia el futuro, y luego, en un bucle a través de las descripciones de eventos recibidas en el array *events*, llamamos a *CalendarValueHistoryByEvent* para cada ID de evento. En esta aplicación no nos interesa el contenido del array *values*, ya que sólo necesitamos conocer su recuento.

```
void OnStart()
{
   MqlCalendarEvent events[];
   MqlCalendarValue values[];
   CalendarEventStats stats[];

   const datetime from = TimeCurrent() - Scope;
   const datetime to = TimeCurrent() + Scope;

   if(StringLen(CountryOrCurrency) == 2)
   {
      PRTF(CalendarEventByCountry(CountryOrCurrency, events));
   }
   else
   {
      PRTF(CalendarEventByCurrency(CountryOrCurrency, events));
   }

   for(int i = 0; i < ArraySize(events); ++i)
   {
      if(CalendarValueHistoryByEvent(events[i].id, values, from, to))
      {
         CalendarEventStats event = {events[i].id, events[i].name,
            CalendarEventStats::importances[events[i].importance], ArraySize(values)};
         PUSH(stats, event);
      }
   }

   SORT_STRUCT(CalendarEventStats, stats, count);
   ArrayReverse(stats);
   ArrayPrint(stats);
}
```

Si la llamada a la función tiene éxito, rellenamos la estructura *CalendarEventStats* y la añadimos al array de estructuras *stats*. A continuación, ordenamos la estructura de la forma que ya conocemos (la macro SORT_STRUCT se describe en la sección Comparar, ordenar y buscar en arrays).

La ejecución del script con la configuración por defecto genera algo como esto en el registro (abreviado):

```
CalendarEventByCountry(CountryOrCurrency,events)=82 / ok
          [id]                                                [name] [importance] [cc
[ 0] 999520001 "CFTC EUR Non-Commercial Net Positions"               "Low"
[ 1] 999010029 "ECB President Lagarde Speech"                        "High"
[ 2] 999010035 "ECB Executive Board Member Elderson Speech"          "Medium"
[ 3] 999030027 "Core CPI"                                            "Low"
[ 4] 999030026 "CPI"                                                 "Low"
[ 5] 999030025 "CPI excl. Energy and Unprocessed Food y/y"           "Low"
[ 6] 999030024 "CPI excl. Energy and Unprocessed Food m/m"           "Low"
[ 7] 999030010 "Core CPI m/m"                                        "Medium"
[ 8] 999030013 "CPI y/y"                                             "Low"
[ 9] 999030012 "Core CPI y/y"                                        "Low"
[10] 999040006 "Consumer Confidence Index"                           "Low"
[11] 999030011 "CPI m/m"                                             "Medium"
...
[65] 999010008 "ECB Economic Bulletin"                               "Medium"
[66] 999030023 "Wage Costs y/y"                                      "Medium"
[67] 999030009 "Labour Cost Index"                                   "Low"
[68] 999010025 "ECB Bank Lending Survey"                             "Low"
[69] 999010030 "ECB Supervisory Board Member af Jochnick Speech"     "Medium"
[70] 999010022 "ECB Supervisory Board Member Hakkarainen Speech"     "Medium"
[71] 999010028 "ECB Financial Stability Review"                      "Medium"
[72] 999010009 "ECB Targeted LTRO"                                   "Medium"
[73] 999010036 "ECB Supervisory Board Member Tuominen Speech"        "Medium"
```

Tenga en cuenta que se recibió un total de 82 tipos de eventos, pero en el array de estadísticas sólo teníamos 74. Esto se debe a que la función *CalendarValueHistoryByEvent* devuelve *false* (fallo) y código de error cero en *_LastError* si no hubo eventos de ningún tipo en el rango de fechas especificado. En la prueba anterior hay 8 entradas de este tipo que teóricamente existen pero que nunca se encontraron en el año.

7.3.7 Leer registros de eventos por ID

Al conocer el calendario de eventos para un futuro próximo, los operadores pueden ajustar sus robots en consecuencia. No hay funciones ni eventos en la API de calendario («eventos» en el sentido de funciones para procesar nueva información financiera como *OnCalendar*, por analogía con *OnTick*) para hacer un seguimiento automático de las publicaciones de noticias. El algoritmo debe hacerlo por sí mismo a cualquier frecuencia elegida. En concreto, puede averiguar el identificador del evento deseado utilizando una de las funciones comentadas anteriormente (por ejemplo, *CalendarValueHistoryByEvent*, *CalendarValueHistory*) y, a continuación, llamar a *CalendarValueById* para obtener el estado actual de los campos de la estructura *MqlCalendarValue*.

bool CalendarValueById(ulong id, MqlCalendarValue &value)

La función rellena la estructura pasada por referencia con información actual sobre un evento específico.

El resultado de la función denota un signo de éxito (*true*) o de error (*false*).

Vamos a crear un sencillo indicador sin búfer *CalendarRecordById.mq5*, que encontrará en el futuro el evento más cercano con el tipo de «indicador financiero» (es decir, un indicador numérico) y sondeará

su estado en el temporizador. Cuando se publique la noticia, los datos cambiarán (se conocerá el valor «real» del indicador), y el indicador mostrará una alerta.

La frecuencia de sondeo del calendario se establece en la variable de entrada.

```
input uint TimerSeconds = 5;
```

Ponemos en marcha el temporizador en *OnInit*.

```
void OnInit()
{
   EventSetTimer(TimerSeconds);
}
```

Para la salida conveniente al registro de descripción de eventos, utilizamos la estructura *MqlCalendarRecord* que ya conocemos del ejemplo con el script *CalendarForDates.mq5*.

Para almacenar el estado inicial de la información de las noticias, describimos la estructura *track*.

```
MqlCalendarValue track;
```

Cuando la estructura está vacía (y hay «0» en el campo *id*), el programa debe consultar los próximos eventos y encontrar entre ellos el más cercano con el tipo CALENDAR_TYPE_INDICATOR y para el que aún no se conoce el valor actual.

```
void OnTimer()
{
   if(!track.id)
   {
      MqlCalendarValue values[];
      if(PRTF(CalendarValueHistory(values, TimeCurrent(), TimeCurrent() + DAY_LONG *
      {
         for(int i = 0; i < ArraySize(values); ++i)
         {
            MqlCalendarEvent event;
            CalendarEventById(values[i].event_id, event);
            if(event.type == CALENDAR_TYPE_INDICATOR && !values[i].HasActualValue())
            {
               track = values[i];
               PrintFormat("Started monitoring %lld", track.id);
               StructPrint(MqlCalendarRecord(track), ARRAYPRINT_HEADER);
               return;
            }
         }
      }
   }
   ...
```

El evento encontrado se copia en *track* y se envía al registro. Después de eso, cada llamada a *OnTimer* se reduce a obtener información actualizada sobre el evento en la estructura *update*, que se transfiere a *CalendarValueById* con el identificador *track.id*. A continuación, las estructuras original y nueva se comparan utilizando la función auxiliar *StructCompare* (basada en *StructToCharArray* y *ArrayCompare*, véase el código fuente completo). Cualquier diferencia hace que se imprima un nuevo estado (la previsión puede haber cambiado), y si aparece el valor actual, el temporizador se detiene. Para empezar a esperar la siguiente noticia, este indicador necesita ser reiniciado: esto es para

demostración, y para controlar la situación según la lista de noticias, más adelante desarrollaremos una clase de filtro más práctica.

```
      else
      {
         MqlCalendarValue update;
         if(CalendarValueById(track.id, update))
         {
            if(fabs(StructCompare(track, update)) == 1)
            {
               Alert(StringFormat("News %lld changed", track.id));
               PrintFormat("New state of %lld", track.id);
               StructPrint(MqlCalendarRecord(update), ARRAYPRINT_HEADER);
               if(update.HasActualValue())
               {
                  Print("Timer stopped");
                  EventKillTimer();
               }
               else
               {
                  track = update;
               }
            }
         }

         if(TimeCurrent() <= track.time)
         {
            Comment("Forthcoming event time: ", track.time,
               ", remaining: ", Timing::stringify((uint)(track.time - TimeCurrent())));
         }
         else
         {
            Comment("Forthcoming event time: ", track.time,
               ", late for: ", Timing::stringify((uint)(TimeCurrent() - track.time)));
         }
      }
   }
```

Mientras se está a la espera del evento, el indicador muestra un comentario con la hora prevista de la publicación de la noticia y cuánto tiempo falta para que se produzca (o cuál es el retraso).

Comentario sobre esperar o llegar tarde a las próximas noticias

Es importante tener en cuenta que la noticia puede salir un poco antes o un poco después de la fecha prevista. Esto crea algunos problemas a la hora de simular estrategias de noticias en el historial, ya que no se proporciona la hora de actualización de las entradas del calendario en el terminal y a través de la API de MQL5. Intentaremos resolver parcialmente este problema en la próxima sección.

A continuación se muestran fragmentos de la salida logarítmica producida por el indicador con una laguna:

```
CalendarValueHistory(values,TimeCurrent(),TimeCurrent()+(60*60*24)*3)=186 / ok
Started monitoring 156045
[id] [event_id] [time] [period] [revision] »
156045 840020013 2022.06.27 15:30:00 2022.05.01 00:00:00 0 »
» [actual_value] [prev_value] [revised_prev_value] [forecast_value] [impact_type] »
» -9223372036854775808 400000 -9223372036854775808 0 0 »
» [importance] [name] [currency] [code] [actual] [previous] [revised] [forecast]
» "Medium" "Durable Goods Orders m/m" "USD" "US" nan 0.40000 nan 0.00000
...
Alert: News 156045 changed
New state of 156045
[id] [event_id] [time] [period] [revision] »
156045 840020013 2022.06.27 15:30:00 2022.05.01 00:00:00 0 »
» [actual_value] [prev_value] [revised_prev_value] [forecast_value] [impact_type] »
» 700000 400000 -9223372036854775808 0 1 »
» [importance] [name] [currency] [code] [actual] [previous] [revised] [forecast]
» "Medium" "Durable Goods Orders m/m" "USD" "US" 0.70000 0.40000 nan 0.00000
Timer stopped
```

Las noticias actualizadas tienen el valor *actual_value*.

Para no esperar demasiado durante la prueba, es aconsejable ejecutar este indicador durante las horas de trabajo de los principales mercados, cuando la densidad de publicación de noticias es elevada.

La función *CalendarValueById* no es la única, y probablemente tampoco la más flexible, con la que puede controlar los cambios en el calendario. En las secciones siguientes veremos un par de enfoques más.

7.3.8 Seguimiento de los cambios de eventos por país o divisa

Como se menciona en la sección sobre conceptos básicos del calendario, la plataforma registra todos los cambios de eventos por algún medio interno. Cada estado se caracteriza por un identificador de cambios (*change_id*). Entre las funciones de MQL5 hay dos que permiten encontrar este identificador (en un punto arbitrario en el tiempo) y luego solicitar entradas de calendario cambiado más tarde. Una de estas funciones es *CalendarValueLast*, que se analizará en esta sección. El segundo, *CalendarValueLastByEvent*, se analizará en la próxima sección.

```
int CalendarValueLast(ulong &change_id, MqlCalendarValue &values[],
 const string country = NULL, const string currency = NULL)
```

La función *CalendarValueLast* está diseñada para dos propósitos: obtener el último identificador de cambio de calendario conocido *change_id* y llenar el array *values* con los registros modificados desde la modificación anterior dada por el identificador pasado en el mismo *change_id*. En otras palabras: el parámetro *change_id* funciona como entrada y como salida. Por eso es una referencia y requiere que se especifique una variable.

Si introducimos *change_id* igual a 0 en la función, entonces la función llenará la variable con el identificador actual pero no llenará el array.

Opcionalmente, utilizando los parámetros *country* y *currency*, puede establecer registros de filtrado por país y divisa.

La función devuelve el número de elementos de calendario copiados. Dado que el array no está poblado en el primer modo de operación (*change_id = 0*), devolver 0 no es un error. También podemos obtener 0 si el calendario no se ha modificado desde el cambio especificado. Por lo tanto, para comprobar si hay un error, debe analizar *_LastError*.

Así que la forma habitual de utilizar la función es hacer un bucle a través del calendario en busca de cambios.

```
ulong change = 0;
MqlCalendarValue values[];
while(!IsStopped())
{
 // pass the last identifier known to us and get a new one if it appeared
   if(CalendarValueLast(change, values))
   {
 // analysis of added and changed records
      ArrayPrint(values);
      ...
   }
   Sleep(1000);
}
```

Esto puede hacerse en un bucle, en un temporizador o en otros eventos.

Los identificadores aumentan constantemente, pero pueden desordenarse, es decir, saltar sobre varios valores.

> Es importante tener en cuenta que cada entrada del calendario está siempre disponible en un único último estado: el historial de cambios no se proporciona en MQL5. Por regla general, esto no supone un problema, ya que el ciclo de vida de cada noticia es estándar: añadir a la base de datos con antelación suficiente y completar con datos relevantes en el momento del evento. Sin embargo, en la práctica pueden producirse varias desviaciones: editar la previsión, transferir el tiempo o revisar los valores. Es imposible averiguar exactamente a qué hora y qué se cambió en el registro a través de la API de MQL5 a partir del historial del calendario. Por lo tanto, aquellos sistemas de trading que tomen decisiones basadas en la situación momentánea requerirán guardar de forma independiente el historial de cambios e integrarlo en un Asesor Experto para ejecutarlo en el probador.

Utilizando la función *CalendarValueLast* podemos crear un servicio útil, *CalendarChangeSaver.mq5*, que comprobará si hay cambios en el calendario a los intervalos especificados y, si los hay, guardará los identificadores de cambio en el archivo junto con la hora actual del servidor. Esto permitirá seguir utilizando la información del archivo para realizar pruebas más realistas de los Asesores Expertos en el historial del calendario. Por supuesto, esto requerirá organizar la exportación/importación de toda la base de datos de calendarios, de lo que nos ocuparemos más adelante.

Vamos a proporcionar variables de entrada para especificar el nombre del archivo y el periodo entre sondeos (en milisegundos).

```
input string Filename = "calendar.chn";
input int PeriodMsc = 1000;
```

Al principio del manejador *OnStart*, abrimos el archivo binario para escribir, o más bien para añadir (si ya existe). Aquí no se comprueba el formato de un archivo existente, por lo que deberá añadir protección al incrustarlo en una aplicación real.

```
void OnStart()
{
   ulong change = 0, last = 0;
   int count = 0;
   int handle = FileOpen(Filename,
      FILE_WRITE | FILE_READ | FILE_SHARE_WRITE | FILE_SHARE_READ | FILE_BIN);
   if(handle == INVALID_HANDLE)
   {
      PrintFormat("Can't open file '%s' for writing", Filename);
      return;
   }

   const ulong p = FileSize(handle);
   if(p > 0)
   {
      PrintFormat("Resuming file %lld bytes", p);
      FileSeek(handle, 0, SEEK_END);
   }

   Print("Requesting start ID...");
   ...
```

Aquí debemos hacer una pequeña digresión:

Cada vez que se modifica el calendario, deben escribirse en el archivo al menos un par de números enteros de 8 bytes: la hora actual (*datetime*) y el ID de la noticia (*ulong*), pero puede haber más de un registro modificado al mismo tiempo. Por lo tanto, además de la fecha, el número de registros modificados se empaqueta en el primer número. Esto tiene en cuenta que las fechas caben en 0x7FFFFFFF y por lo tanto los 3 bytes superiores se dejan sin usar. Es en los dos bytes más significativos (con un desplazamiento a la izquierda de 48 bits) donde se coloca el número de identificadores que el servicio escribirá después de la marca de tiempo correspondiente. La macro PACK_DATETIME_COUNTER crea una fecha «extendida», y las otras dos, DATETIME y COUNTER, las necesitaremos más adelante cuando el archivo de cambios sea leído (por otro programa).

```
#define PACK_DATETIME_COUNTER(D,C) (D | (((ulong)(C)) << 48))
#define DATETIME(A) ((datetime)((A) & 0x7FFFFFFF))
#define COUNTER(A)  ((ushort)((A) >> 48))
```

Volvamos ahora al código del servicio principal. En un bucle que se activa cada *PeriodMsc* milisegundos, solicitamos cambios utilizando *CalendarValueLast*. Si hay cambios, escribimos en un archivo la hora actual del servidor y el array de identificadores recibidos.

```
      while(!IsStopped())
      {
         if(!TerminalInfoInteger(TERMINAL_CONNECTED))
         {
            Print("Waiting for connection...");
            Sleep(PeriodMsc);
            continue;
         }

         MqlCalendarValue values[];
         const int n = CalendarValueLast(change, values);
         if(n > 0)
         {
            string records = "[" + Description(values[0]);
            for(int i = 1; i < n; ++i)
            {
               records += "," + Description(values[i]);
            }
            records += "]";
            Print("New change ID: ", change, " ",
               TimeToString(TimeTradeServer(), TIME_DATE | TIME_SECONDS), "\n", records)
            FileWriteLong(handle, PACK_DATETIME_COUNTER(TimeTradeServer(), n));
            for(int i = 0; i < n; ++i)
            {
               FileWriteLong(handle, values[i].id);
            }
            FileFlush(handle);
            ++count;
         }
         else if(_LastError == 0)
         {
            if(!last && change)
            {
               Print("Start change ID obtained: ", change);
            }
         }

         last = change;
         Sleep(PeriodMsc);
      }
      PrintFormat("%d records added", count);
      FileClose(handle);
   }
```

Para presentar cómodamente la información sobre cada evento de noticias, hemos escrito una función de ayuda *Description*.

```
string Description(const MqlCalendarValue &value)
{
   MqlCalendarEvent event;
   MqlCalendarCountry country;
   CalendarEventById(value.event_id, event);
   CalendarCountryById(event.country_id, country);
   return StringFormat("%lld (%s/%s @ %s)",
      value.id, country.code, event.name, TimeToString(value.time));
}
```

Así, el registro mostrará no sólo el identificador, sino también el código de país, el título y la hora programada de la noticia.

> Se da por sentado que el servicio debe funcionar durante bastante tiempo para recopilar información durante un periodo suficiente para la simulación (días, semanas, meses). Lamentablemente, al igual que ocurre con el libro de órdenes, la plataforma no proporciona un historial preparado del libro de órdenes ni de las ediciones del calendario, por lo que su recopilación se deja enteramente en manos del desarrollador de programas MQL.

Veamos el servicio en acción. En el siguiente fragmento del registro (para el periodo de tiempo de 2022.06.28, 15:30 - 16:00), algunos eventos de noticias se refieren a un futuro lejano (contienen los valores del campo prev_value, que es también el campo *actual_value* del evento actual del mismo nombre). Sin embargo, hay algo más importante: la hora real de una publicación de noticias puede diferir significativamente, a veces en varios minutos, de la prevista.

```
Requesting start ID...
Start change ID obtained: 86358784
New change ID: 86359040 2022.06.28 15:30:42
[155955 (US/Wholesale Inventories m/m @ 2022.06.28 15:30)]
New change ID: 86359296 2022.06.28 15:30:45
[155956 (US/Wholesale Inventories m/m @ 2022.07.08 17:00)]
New change ID: 86359552 2022.06.28 15:30:48
[156117 (US/Goods Trade Balance @ 2022.06.28 15:30)]
New change ID: 86359808 2022.06.28 15:30:51
[156118 (US/Goods Trade Balance @ 2022.07.27 15:30)]
New change ID: 86360064 2022.06.28 15:30:54
[156231 (US/Retail Inventories m/m @ 2022.06.28 15:30)]
New change ID: 86360320 2022.06.28 15:30:57
[156232 (US/Retail Inventories m/m @ 2022.07.15 17:00)]
New change ID: 86360576 2022.06.28 15:31:00
[156255 (US/Retail Inventories excl. Autos m/m @ 2022.06.28 15:30)]
New change ID: 86360832 2022.06.28 15:31:03
[156256 (US/Retail Inventories excl. Autos m/m @ 2022.07.15 17:00)]
New change ID: 86361088 2022.06.28 15:31:07
[155956 (US/Wholesale Inventories m/m @ 2022.07.08 17:00)]
New change ID: 86361344 2022.06.28 15:31:10
[156118 (US/Goods Trade Balance @ 2022.07.27 15:30)]
New change ID: 86361600 2022.06.28 15:31:13
[156232 (US/Retail Inventories m/m @ 2022.07.15 17:00)]
New change ID: 86362368 2022.06.28 15:36:47
[158534 (US/Challenger Job Cuts y/y @ 2022.07.07 14:30)]
New change ID: 86362624 2022.06.28 15:51:23
...
New change ID: 86364160 2022.06.28 16:01:39
[154531 (US/HPI m/m @ 2022.06.28 16:00)]
New change ID: 86364416 2022.06.28 16:01:42
[154532 (US/HPI m/m @ 2022.07.26 16:00)]
New change ID: 86364672 2022.06.28 16:01:46
[154543 (US/HPI y/y @ 2022.06.28 16:00)]
New change ID: 86364928 2022.06.28 16:01:49
[154544 (US/HPI y/y @ 2022.07.26 16:00)]
New change ID: 86365184 2022.06.28 16:01:54
[154561 (US/HPI @ 2022.06.28 16:00)]
New change ID: 86365440 2022.06.28 16:01:58
[154571 (US/HPI @ 2022.07.26 16:00)]
New change ID: 86365696 2022.06.28 16:02:01
[154532 (US/HPI m/m @ 2022.07.26 16:00)]
New change ID: 86365952 2022.06.28 16:02:05
[154544 (US/HPI y/y @ 2022.07.26 16:00)]
New change ID: 86366208 2022.06.28 16:02:09
[154571 (US/HPI @ 2022.07.26 16:00)]
```

Por supuesto, esto no es importante para todas las clases de estrategias de trading, sino sólo para las que operan rápidamente en el mercado. Para ellas, el archivo creado de ediciones de calendario puede proporcionar una simulación más precisa de Asesores Expertos de noticias. En el futuro hablaremos de cómo «conectar» el calendario con el probador, pero por ahora, mostraremos cómo leer el archivo recibido.

Utilizaremos el script *CalendarChangeReader.mq5* para demostrar la funcionalidad comentada. En la práctica, el código fuente dado debe colocarse en el Asesor Experto.

Las variables de entrada permiten establecer el nombre del archivo que se va a leer y la fecha de inicio de la exploración. Si el servicio sigue funcionando (escribe el archivo), debe copiar el archivo con otro nombre o en otra carpeta (en el script de ejemplo, se cambia el nombre del archivo). Si el parámetro *Start* está en blanco, la lectura de los cambios de noticias comenzará desde el principio del día actual.

```
input string Filename = "calendar2.chn";
input datetime Start;
```

Se describe la estructura *ChangeState* para almacenar información sobre ediciones individuales.

```
struct ChangeState
{
   datetime dt;
   ulong ids[];

   ChangeState(): dt(LONG_MAX) {}
   ChangeState(const datetime at, ulong &_ids[])
   {
      dt = at;
      ArraySwap(ids, _ids);
   }

   void operator=(const ChangeState &other)
   {
      dt = other.dt;
      ArrayCopy(ids, other.ids);
   }
};
```

Se utiliza en la clase *ChangeFileReader*, que realiza la mayor parte del trabajo de leer el archivo y proporcionar a la persona que llama los cambios que son apropiados para un momento determinado.

El manejador del archivo se pasa como parámetro al constructor, al igual que la hora de inicio de la prueba. La lectura de un archivo y el relleno de la estructura *ChangeState* para una edición de calendario se realiza en el método *readState*.

```
class ChangeFileReader
{
   const int handle;
   ChangeState current;
   const ChangeState zero;

public:
   ChangeFileReader(const int h, const datetime start = 0): handle(h)
   {
      if(readState())
      {
         if(start)
         {
            ulong dummy[];
            check(start, dummy, true); // find the first edit after start
         }
      }
   }

   bool readState()
   {
      if(FileIsEnding(handle)) return false;
      ResetLastError();
      const ulong v = FileReadLong(handle);
      current.dt = DATETIME(v);
      ArrayFree(current.ids);
      const int n = COUNTER(v);
      for(int i = 0; i < n; ++i)
      {
         PUSH(current.ids, FileReadLong(handle));
      }
      return _LastError == 0;
   }
   ...
```

El método *check* lee el archivo hasta que aparezca la siguiente edición en el futuro. En este caso, todas las ediciones anteriores (por marcas de tiempo) desde la llamada anterior al método se colocan en el array de salida *records*.

```
      bool check(datetime now, ulong &records[], const bool fastforward = false)
      {
         if(current.dt > now) return false;

         ArrayFree(records);

         if(!fastforward)
         {
            ArrayCopy(records, current.ids);
            current = zero;
         }

         while(readState() && current.dt <= now)
         {
            if(!fastforward) ArrayInsert(records, current.ids, ArraySize(records));
         }

         return true;
      }
   };
```

Así se utiliza la clase en *OnStart*.

```
void OnStart()
{
   const long day = 60 * 60 * 24;
   datetime now = Start ? Start : (datetime)(TimeCurrent() / day * day);

   int handle = FileOpen(Filename,
      FILE_READ | FILE_SHARE_WRITE | FILE_SHARE_READ | FILE_BIN);
   if(handle == INVALID_HANDLE)
   {
      PrintFormat("Can't open file '%s' for reading", Filename);
      return;
   }

   ChangeFileReader reader(handle, now);

   // reading step by step, time now artificially increased in this demo
   while(!FileIsEnding(handle))
   {
      // in a real application, a call to reader.check can be made on every tick
      ulong records[];
      if(reader.check(now, records))
      {
         Print(now);          // output time
         ArrayPrint(records); // array of IDs of changed news
      }
      now += 60; // add 1 minute at a time, can be per second
   }

   FileClose(handle);
}
```

He aquí los resultados del script para los mismos cambios de calendario que fueron guardados por el servicio en el contexto del fragmento de registro anterior.

```
2022.06.28 15:31:00
155955 155956 156117 156118 156231 156232 156255
2022.06.28 15:32:00
156256 155956 156118 156232
2022.06.28 15:37:00
158534
...
2022.06.28 16:02:00
154531 154532 154543 154544 154561 154571
2022.06.28 16:03:00
154532 154544 154571
```

Los mismos identificadores se reproducen en tiempo virtual con el mismo retraso que en línea, aunque aquí se aprecia el redondeo a 1 minuto, que se produjo porque fijamos un paso artificial de este tamaño en el bucle. En teoría, por razones de eficacia, podemos aplazar las comprobaciones hasta la hora almacenada en la estructura *ChangeState current*. El código fuente adjunto define el método *getState* para obtener este tiempo.

7.3.9 Seguimiento de los cambios de eventos por tipo

La API de MQL5 le permite solicitar cambios recientes no sólo en general para todo el calendario o por país o divisa, sino también en un rango más estrecho, o mejor dicho, para un tipo específico de evento.

En teoría, podemos decir que las funciones integradas permiten filtrar los eventos en función de varias condiciones básicas: hora, país, divisa o tipo de evento. Para otros atributos, como la importancia o el sector económico, deberá aplicar su propio filtrado, del que nos ocuparemos más adelante. De momento, vamos a presentar la función *CalendarValueLastByEvent*.

int CalendarValueLastByEvent(ulong id, ulong &change_id, MqlCalendarValue &values[])

La función rellena el array *values* pasado por referencia con registros de eventos de un tipo específico con el identificador *id* que se han producido desde *change_id*. Este parámetro *change_id* es a la vez entrada y salida: el código de llamada pasa en él la etiqueta del estado pasado del calendario, tras lo cual se solicitan los cambios, y cuando vuelve el control, la función escribe la etiqueta actual del estado de la base de datos del calendario en *change_id*. Debe utilizarse la próxima vez que se llame a la función.

Si pasa null en *change_id*, entonces la función no rellena el array sino que simplemente envía el estado actual de la base de datos a través del parámetro *change_id*.

El array puede ser dinámico (entonces se ajustará automáticamente a la cantidad de datos) o de tamaño fijo (si su tamaño es insuficiente, sólo se copiarán los datos que quepan).

El valor de salida de la función es igual al número de elementos copiados en el array *values*. Si no hay cambios o se especifica *change_id = 0*, la función devolverá 0.

Para comprobar si se ha producido algún error, analice la variable integrada *_LastError*. Algunos de los posibles códigos de error son:

- 4004 - ERR_NOT_ENOUGH_MEMORY (memoria insuficiente para completar la solicitud);
- 5401 - ERR_CALENDAR_TIMEOUT (solicitud agotada);
- 5400 - ERR_CALENDAR_MORE_DATA (el tamaño del array fijo no es suficiente para obtener todos los valores).

No daremos un ejemplo aparte para *CalendarValueLastByEvent*. En lugar ello, pasemos a una tarea más compleja, pero muy demandada, de consulta y filtrado de entradas de calendario con condiciones arbitrarias sobre atributos de noticias, en la que intervendrán todas las funciones de la API «calendario». Este será el tema de la próxima sección.

7.3.10 Filtrar eventos por múltiples condiciones

Como sabemos por las secciones anteriores de este capítulo, la API de MQL5 permite solicitar eventos de calendario en función de varias condiciones:

- por países (*CalendarValueHistory*, *CalendarValueLast*)
- por frecuencias (*CalendarValueHistory*, *CalendarValueLast*)
- por ID de tipo de evento (*CalendarValueHistoryByEvent*, *CalendarValueLastByEvent*)
- por intervalo de tiempo (*CalendarValueHistory*, *CalendarValueHistoryByEvent*)
- por cambios desde la encuesta del calendario anterior (*CalendarValueLast*, *CalendarValueLastByEvent*)

- por ID de noticia específica (*CalendarValueById*)

Esto se puede resumir en la siguiente tabla de funciones (de todas las funciones de *CalendarValue*, aquí sólo falta *CalendarValueById* para obtener un valor específico).

Condiciones	Intervalo de tiempo	Últimos cambios
Países	CalendarValueHistory	CalendarValueLast
Divisas	CalendarValueHistory	CalendarValueLast
Eventos	CalendarValueHistoryByEvent	CalendarValueLastByEvent

Este conjunto de herramientas abarca los principales escenarios de análisis de calendarios, aunque no todos. Por lo tanto, en la práctica, a menudo es necesario implementar mecanismos de filtrado personalizados en MQL5, incluyendo, en concreto, las solicitudes de eventos por:

- varios países
- varias divisas
- varios tipos de eventos
- valores de propiedades arbitrarias de los eventos (importancia, sector de la economía, periodo de referencia, tipo, presencia de una previsión, impacto estimado en la tasa, subcadena en el nombre del evento, etc.)

Para resolver estos problemas, hemos creado la clase *CalendarFilter* (*CalendarFilter.mqh*).

Debido a las especificidades de las funciones integradas de la API, algunos de los atributos de las noticias tienen mayor prioridad que el resto. Esto incluye el país, la divisa y el intervalo de fechas. Se pueden especificar en el constructor de la clase, y entonces la propiedad correspondiente no se puede cambiar dinámicamente en las condiciones del filtro.

Esto se debe a que la clase de filtro se ampliará posteriormente con las capacidades de almacenamiento en caché de noticias para permitir la lectura desde el probador, y las condiciones iniciales del constructor en realidad definen el contexto de almacenamiento en caché dentro del cual es posible el filtrado posterior. Por ejemplo, si al crear un objeto especificamos el código de país «UE», obviamente no tiene sentido solicitar a través de él noticias sobre Estados Unidos o Brasil. Esto es similar al intervalo de fechas: especificarlo en el constructor hará imposible recibir noticias fuera del intervalo.

También podemos crear un objeto sin condiciones iniciales (porque todos los parámetros del constructor son opcionales), y entonces será capaz de almacenar en caché y filtrar noticias en toda la base de datos del calendario (en el momento de guardarlo).

Además, dado que los países y las divisas se muestran ahora de forma casi única (con la excepción de la Unión Europea y EUR), se pasan al constructor a través de un único parámetro *context*: si se especifica una cadena con una longitud de 2 caracteres, el código del país (o una combinación de países) está implícito, y si la longitud es de 3 caracteres, el código de la moneda está implícito. Para los códigos «UE» y «EUR», la zona del euro es un subconjunto de la «UE» (dentro de los países con tratados formales). En casos especiales, en los que interesan países de la UE no pertenecientes a la zona euro, también pueden describirse en el contexto de «UE». Si es necesario pueden añadirse al filtro condiciones más estrictas para las noticias sobre las divisas de estos países (BGN, HUF, DKK, ISK, PLN, RON, HRK, CZK, SEK) de forma dinámica utilizando métodos que presentaremos más adelante. No obstante, debido a condiciones exóticas, no hay garantías de que esas noticias entren en el calendario.

Empecemos a estudiar la clase.

```
class CalendarFilter
{
protected:
   // initial (optional) conditions set in the constructor, invariants
   string context;    // country and currency
   datetime from, to; // date range
   bool fixedDates;   // if 'from'/'to' are passed in the constructor, they cannot be

   // dedicated selectors (countries/currencies/event type identifiers)
   string country[], currency[];
   ulong ids[];

   MqlCalendarValue values[]; // filtered results

   virtual void init()
   {
      fixedDates = from != 0 || to != 0;
      if(StringLen(context) == 3)
      {
         PUSH(currency, context);
      }
      else
      {
         // even if context is NULL, we take it to poll the entire calendar base
         PUSH(country, context);
      }
   }
   ...
public:
   CalendarFilter(const string _context = NULL,
      const datetime _from = 0, const datetime _to = 0):
      context(_context), from(_from), to(_to)
   {
      init();
   }
   ...
```

Se asignan dos arrays para los países y las divisad: *country* y *currency*. Si no se rellenan desde *context* durante la creación del objeto, el programa MQL podrá añadir condiciones para varios países o divisas con el fin de realizar una consulta de noticias combinada en ellos.

Para almacenar las condiciones de todos los demás atributos de las noticias, el array *selectors* se describe en el objeto *CalendarFilter*, con la segunda dimensión igual a 3. Podemos decir que se trata de un tipo de tabla en la que cada fila tiene 3 columnas.

```
   long selectors[][3];   // [0] - property, [1] - value, [2] - condition
```

En el índice 0 se localizarán los identificadores de las propiedades de las noticias. Dado que los atributos están repartidos en tres tablas base (*MqlCalendarCountry*, *MqlCalendarEvent*, *MqlCalendarValue*) se describen utilizando los elementos de la enumeración generalizada ENUM_CALENDAR_PROPERTY (*CalendarDefines.mqh*).

```
enum ENUM_CALENDAR_PROPERTY
{                                    // +/- means support for field filtering
   CALENDAR_PROPERTY_COUNTRY_ID,       // -ulong
   CALENDAR_PROPERTY_COUNTRY_NAME,     // -string
   CALENDAR_PROPERTY_COUNTRY_CODE,     // +string (2 characters)
   CALENDAR_PROPERTY_COUNTRY_CURRENCY, // +string (3 characters)
   CALENDAR_PROPERTY_COUNTRY_GLYPH,    // -string (1 characters)
   CALENDAR_PROPERTY_COUNTRY_URL,      // -string

   CALENDAR_PROPERTY_EVENT_ID,         // +ulong (event type ID)
   CALENDAR_PROPERTY_EVENT_TYPE,       // +ENUM_CALENDAR_EVENT_TYPE
   CALENDAR_PROPERTY_EVENT_SECTOR,     // +ENUM_CALENDAR_EVENT_SECTOR
   CALENDAR_PROPERTY_EVENT_FREQUENCY,  // +ENUM_CALENDAR_EVENT_FREQUENCY
   CALENDAR_PROPERTY_EVENT_TIMEMODE,   // +ENUM_CALENDAR_EVENT_TIMEMODE
   CALENDAR_PROPERTY_EVENT_UNIT,       // +ENUM_CALENDAR_EVENT_UNIT
   CALENDAR_PROPERTY_EVENT_IMPORTANCE, // +ENUM_CALENDAR_EVENT_IMPORTANCE
   CALENDAR_PROPERTY_EVENT_MULTIPLIER, // +ENUM_CALENDAR_EVENT_MULTIPLIER
   CALENDAR_PROPERTY_EVENT_DIGITS,     // -uint
   CALENDAR_PROPERTY_EVENT_SOURCE,     // +string ("http[s]://")
   CALENDAR_PROPERTY_EVENT_CODE,       // -string
   CALENDAR_PROPERTY_EVENT_NAME,       // +string (4+ characters or wildcard '*')

   CALENDAR_PROPERTY_RECORD_ID,        // -ulong
   CALENDAR_PROPERTY_RECORD_TIME,      // +datetime
   CALENDAR_PROPERTY_RECORD_PERIOD,    // +datetime (like long)
   CALENDAR_PROPERTY_RECORD_REVISION,  // +int
   CALENDAR_PROPERTY_RECORD_ACTUAL,    // +long
   CALENDAR_PROPERTY_RECORD_PREVIOUS,  // +long
   CALENDAR_PROPERTY_RECORD_REVISED,   // +long
   CALENDAR_PROPERTY_RECORD_FORECAST,  // +long
   CALENDAR_PROPERTY_RECORD_IMPACT,    // +ENUM_CALENDAR_EVENT_IMPACT

   CALENDAR_PROPERTY_RECORD_PREVISED,  // +non-standard (previous or revised if any)

   CALENDAR_PROPERTY_CHANGE_ID,        // -ulong (reserved)
};
```

El índice 1 almacenará valores para compararlos con ellos en las condiciones de selección de registros de noticias. Por ejemplo, si desea establecer un filtro por sector de la economía, entonces escribimos CALENDAR_PROPERTY_EVENT_SECTOR en *selectors[i][0]* y uno de los valores de la enumeración estándar ENUM_CALENDAR_EVENT_SECTOR en *selectors[i][1]*.

Por último, la última columna (bajo el 2º índice) está reservada a la operación de comparación del valor del selector con el valor del atributo de la noticia: todas las operaciones admitidas se resumen en la enumeración IS.

```
enum IS
{
   EQUAL,
   NOT_EQUAL,
   GREATER,
   LESS,
   OR_EQUAL,
   ...
};
```

Vimos un planteamiento similar en *TradeFilter.mqh*. Así, podremos establecer condiciones no sólo para la igualdad de valores, sino también para la desigualdad o las relaciones más/menos. Por ejemplo, es fácil imaginar un filtro en el campo CALENDAR_PROPERTY_EVENT_IMPORTANCE, que debería ser MAYOR que CALENDAR_IMPORTANCE_LOW (se trata de un elemento de la enumeración estándar ENUM_CALENDAR_EVENT_IMPORTANCE), lo que significa una selección de noticias de importancia media y alta.

La siguiente enumeración definida específicamente para el calendario es ENUM_CALENDAR_SCOPE. Dado que el filtrado de calendarios suele asociarse a intervalos de tiempo, aquí se enumeran los más solicitados.

```
#define DAY_LONG     (60 * 60 * 24)
#define WEEK_LONG    (DAY_LONG * 7)
#define MONTH_LONG   (DAY_LONG * 30)
#define QUARTER_LONG (MONTH_LONG * 3)
#define YEAR_LONG    (MONTH_LONG * 12)

enum ENUM_CALENDAR_SCOPE
{
   SCOPE_DAY = DAY_LONG,          // Day
   SCOPE_WEEK = WEEK_LONG,        // Week
   SCOPE_MONTH = MONTH_LONG,      // Month
   SCOPE_QUARTER = QUARTER_LONG, // Quarter
   SCOPE_YEAR = YEAR_LONG,        // Year
};
```

Todas las enumeraciones se colocan en un archivo de encabezado independiente *CalendarDefines.mqh*.

Pero volvamos a la clase *CalendarFilter*. El tipo del array *selectors* es *long*, que es adecuado para almacenar valores de casi todos los tipos implicados: enumeraciones, fechas y horas, identificadores, números enteros e incluso valores de indicadores económicos porque se almacenan en el calendario en forma de números *long* (en millonésimas de valores reales). Sin embargo, ¿qué hacer con las propiedades de las cadenas?

Este problema se resuelve utilizando el array de cadenas *stringCache*, a la que se añadirán todas las líneas mencionadas en las condiciones del filtro.

```
class CalendarFilter
{
protected:
   ...
   string stringCache[];  // cache of all rows in 'selectors'
   ...
```

Entonces, en lugar del valor de la cadena en *selectors[i][1]*, podemos guardar fácilmente el índice de un elemento en el array *stringCache*.

Para rellenar el array *selectors* con las condiciones de filtrado, se proporcionan varios métodos *let*, en concreto, para las enumeraciones:

```
class CalendarFilter
{
...
public:
   // all fields of enum types are processed here
   template<typename E>
   CalendarFilter *let(const E e, const IS c = EQUAL)
   {
      const int n = EXPAND(selectors);
      selectors[n][0] = resolve(e); // by type E, returning the element ENUM_CALENDAR
      selectors[n][1] = e;
      selectors[n][2] = c;
      return &this;
   }
   ...
```

Para los valores reales de los indicadores:

```
   // the following fields are processed here:
   // CALENDAR_PROPERTY_RECORD_ACTUAL, CALENDAR_PROPERTY_RECORD_PREVIOUS,
   // CALENDAR_PROPERTY_RECORD_REVISED, CALENDAR_PROPERTY_RECORD_FORECAST,
   // and CALENDAR_PROPERTY_RECORD_PERIOD (as long)
   CalendarFilter *let(const long value, const ENUM_CALENDAR_PROPERTY property, const
   {
      const int n = EXPAND(selectors);
      selectors[n][0] = property;
      selectors[n][1] = value;
      selectors[n][2] = c;
      return &this;
   }
   ...
```

Y para las cadenas:

```
// conditions for all string properties can be found here (abbreviated)
CalendarFilter *let(const string find, const IS c = EQUAL)
{
   const int wildcard = (StringFind(find, "*") + 1) * 10;
   switch(StringLen(find) + wildcard)
   {
   case 2:
      // if the initial context is different from the country, we can supplement i
      // otherwise the filter is ignored
      if(StringLen(context) != 2)
      {
         if(ArraySize(country) == 1 && StringLen(country[0]) == 0)
         {
            country[0] = find; // narrow down "all countries" to one (may add more
         }
         else
         {
            PUSH(country, find);
         }
      }
      break;
   case 3:
      // we can set a filter for a currency only if it was not in the initial cont
      if(StringLen(context) != 3)
      {
         PUSH(currency, find);
      }
      break;
   default:
      {
         const int n = EXPAND(selectors);
         PUSH(stringCache, find);
         if(StringFind(find, "http://") == 0 || StringFind(find, "https://") == 0)
         {
            selectors[n][0] = CALENDAR_PROPERTY_EVENT_SOURCE;
         }
         else
         {
            selectors[n][0] = CALENDAR_PROPERTY_EVENT_NAME;
         }
         selectors[n][1] = ArraySize(stringCache) - 1;
         selectors[n][2] = c;
         break;
      }
   }

   return &this;
}
```

En la sobrecarga del método para cadenas, observe que las cadenas largas de 2 o 3 caracteres (si no llevan el asterisco de plantilla '*', que es un sustituto de una secuencia arbitraria de caracteres) entran en los arrays de países y símbolos, respectivamente, y todas las demás cadenas se tratan como

fragmentos del nombre o de la fuente de noticias, y en ambos campos intervienen *stringCache* y *selectors*.

De forma especial, la clase también admite el filtrado por tipo (identificador) de eventos.

```
protected:
   ulong ids[];           // filtered event types
   ...
public:
   CalendarFilter *let(const ulong event)
   {
      PUSH(ids, event);
      return &this;
   }
   ...
```

Así, el número de filtros prioritarios (que se procesan fuera del array de selectores) incluye no sólo países, divisas e intervalos de fechas, sino también identificadores de tipo de evento. Esta decisión constructiva se debe a que estos parámetros pueden pasarse como entrada a determinadas funciones de la API de calendario. Obtenemos todos los demás atributos de las noticias como valores de campo de salida en arrays de estructuras (*MqlCalendarValue*, *MqlCalendarEvent*, *MqlCalendarCountry*). Es por ellos que realizaremos el filtrado adicional, de acuerdo con las reglas en el array de selectores.

Todos los métodos de *let* devuelven un puntero a un objeto, lo que permite encadenar sus llamadas. Por ejemplo, así:

```
CalendarFilter f;
f.let(CALENDAR_IMPORTANCE_LOW, GREATER) // important and moderately important news
  .let(CALENDAR_TIMEMODE_DATETIME) // only events with exact time
  .let("DE").let("FR") // a couple of countries, or, to choose from...
  .let("USD").let("GBP") // ...a couple of currencies (but both conditions won't work
  .let(TimeCurrent() - MONTH_LONG, TimeCurrent() + WEEK_LONG) // date range "around"
  .let(LONG_MIN, CALENDAR_PROPERTY_RECORD_FORECAST, NOT_EQUAL) // there is a forecast
  .let("farm"); // full text search by news titles
```

Las condiciones del país y de la divisa pueden, en teoría, combinarse. No obstante, tenga en cuenta que los valores múltiples sólo pueden establecerse para países o divisas, pero no para ambos. Uno de estos dos aspectos del contexto (cualquiera de los dos) en la implementación actual sólo admite uno o ninguno de los valores (es decir, no tiene filtro). Por ejemplo, si se selecciona la divisa EUR, es posible limitar el contexto de búsqueda a las noticias de Alemania y Francia (códigos de país «DE» y «FR»). En consecuencia, se descartarán las noticias del BCE y de Eurostat, así como, en concreto, las de Italia y España. No obstante, la indicación de EUR en este caso es redundante, ya que no existen otras divisas en Alemania y Francia.

Dado que la clase utiliza funciones integradas en las que los parámetros *country* y *currency* se aplican a las noticias mediante la operación lógica AND, compruebe la coherencia de las condiciones de filtrado.

Una vez que el código de llamada establece las condiciones de filtrado, es necesario seleccionar las noticias en función de ellas. Esto es lo que hace el método público *select* (dado con simplificaciones).

```
public:
   bool select(MqlCalendarValue &result[])
   {
      int count = 0;
      ArrayFree(result);
      if(ArraySize(ids)) // identifiers of event types
      {
         for(int i = 0; i < ArraySize(ids); ++i)
         {
            MqlCalendarValue temp[];
            if(PRTF(CalendarValueHistoryByEvent(ids[i], temp, from, to)))
            {
               ArrayCopy(result, temp, ArraySize(result));
               ++count;
            }
         }
      }
      else
      {
         // several countries or currencies, choose whichever is more as a basis,
         // only the first element from the smaller array is used
         if(ArraySize(country) > ArraySize(currency))
         {
            const string c = ArraySize(currency) > 0 ? currency[0] : NULL;
            for(int i = 0; i < ArraySize(country); ++i)
            {
               MqlCalendarValue temp[];
               if(PRTF(CalendarValueHistory(temp, from, to, country[i], c)))
               {
                  ArrayCopy(result, temp, ArraySize(result));
                  ++count;
               }
            }
         }
         else
         {
            const string c = ArraySize(country) > 0 ? country[0] : NULL;
            for(int i = 0; i < ArraySize(currency); ++i)
            {
               MqlCalendarValue temp[];
               if(PRTF(CalendarValueHistory(temp, from, to, c, currency[i])))
               {
                  ArrayCopy(result, temp, ArraySize(result));
                  ++count;
               }
            }
         }
      }

      if(ArraySize(result) > 0)
      {
```

```
         filter(result);
      }

      if(count > 1 && ArraySize(result) > 1)
      {
         SORT_STRUCT(MqlCalendarValue, result, time);
      }

      return ArraySize(result) > 0;
   }
```

Dependiendo de cuál de los arrays de atributos de prioridad esté lleno, el método llama a diferentes funciones de la API para sondear el calendario:

- Si el array *ids* está lleno, se llama a *CalendarValueHistoryByEvent* en un bucle para todos los identificadores
- Si el array *country* está lleno y es mayor que el array de divisas, llame a *CalendarValueHistory* y recorra los países mediante un bucle.
- Si el array *currency* está lleno y es mayor o igual que el tamaño del array de países, llame a *CalendarValueHistory* y recorra las divisas mediante un bucle.

Cada llamada a una función rellena un array temporal de estructuras *MqlCalendarValue temp[]*, que se acumula secuencialmente en el array de parámetros *result*. Después de escribir en él todas las noticias relevantes según las condiciones principales (fechas, países, divisas, identificadores), si las hay, entra en juego un método auxiliar *filter*, que filtra el array basándose en las condiciones de *selectors*. Al final del método *select*, las noticias se clasifican por orden cronológico, que puede romperse combinando los resultados de varias consultas de las funciones «calendario». La ordenación se lleva a cabo mediante la macro SORT_STRUCT, de la que ya se habló en la sección Comparar, ordenar y buscar en arrays.

Para cada elemento del array de noticias, el método *filter* llama al método *match*, que devuelve un indicador booleano de si la noticia coincide con las condiciones del filtro. En caso contrario, el elemento se elimina del array.

```
protected:
   void filter(MqlCalendarValue &result[])
   {
      for(int i = ArraySize(result) - 1; i >= 0; --i)
      {
         if(!match(result[i]))
         {
            ArrayRemove(result, i, 1);
         }
      }
   }
   ...
```

Por último, el método *match* analiza nuestro array *selectors* y la compara con los campos de la estructura pasada *MqlCalendarValue*. Aquí se proporciona el código de forma abreviada.

```
bool match(const MqlCalendarValue &v)
  {
     MqlCalendarEvent event;
     if(!CalendarEventById(v.event_id, event)) return false;

     // loop through all filter conditions, except for countries, currencies, dates,
     // which have already been previously used when calling Calendar functions
     for(int j = 0; j < ArrayRange(selectors, 0); ++j)
     {
        long field = 0;
        string text = NULL;

        // get the field value from the news or its description
        switch((int)selectors[j][0])
        {
        case CALENDAR_PROPERTY_EVENT_TYPE:
           field = event.type;
           break;
        case CALENDAR_PROPERTY_EVENT_SECTOR:
           field = event.sector;
           break;
        case CALENDAR_PROPERTY_EVENT_TIMEMODE:
           field = event.time_mode;
           break;
        case CALENDAR_PROPERTY_EVENT_IMPORTANCE:
           field = event.importance;
           break;
        case CALENDAR_PROPERTY_EVENT_SOURCE:
           text = event.source_url;
           break;
        case CALENDAR_PROPERTY_EVENT_NAME:
           text = event.name;
           break;
        case CALENDAR_PROPERTY_RECORD_IMPACT:
           field = v.impact_type;
           break;
        case CALENDAR_PROPERTY_RECORD_ACTUAL:
           field = v.actual_value;
           break;
        case CALENDAR_PROPERTY_RECORD_PREVIOUS:
           field = v.prev_value;
           break;
        case CALENDAR_PROPERTY_RECORD_REVISED:
           field = v.revised_prev_value;
           break;
        case CALENDAR_PROPERTY_RECORD_PREVISED: // previous or revised (if any)
           field = v.revised_prev_value != LONG_MIN ? v.revised_prev_value : v.prev_
           break;
        case CALENDAR_PROPERTY_RECORD_FORECAST:
           field = v.forecast_value;
           break;
```

```
      ...
      }

      // compare value with filter condition
      if(text == NULL) // numeric fields
      {
         switch((IS)selectors[j][2])
         {
         case EQUAL:
            if(!equal(field, selectors[j][1])) return false;
            break;
         case NOT_EQUAL:
            if(equal(field, selectors[j][1])) return false;
            break;
         case GREATER:
            if(!greater(field, selectors[j][1])) return false;
            break;
         case LESS:
            if(greater(field, selectors[j][1])) return false;
            break;
         }
      }
      else // string fields
      {
         const string find = stringCache[(int)selectors[j][1]];
         switch((IS)selectors[j][2])
         {
         case EQUAL:
            if(!equal(text, find)) return false;
            break;
         case NOT_EQUAL:
            if(equal(text, find)) return false;
            break;
         case GREATER:
            if(!greater(text, find)) return false;
            break;
         case LESS:
            if(greater(text, find)) return false;
            break;
         }
      }
   }

   return true;
}
```

Los métodos *equal* y *greater* copian casi por completo los utilizados en nuestros desarrollos anteriores con clases de filtros.

En este caso, el problema de filtrado se resuelve en general, es decir, el programa MQL puede utilizar el objeto *CalendarFilter* de la siguiente manera:

```
CalendarFilter f;
f.let()... // a series of calls to the let method to set filtering conditions
MqlCalendarValue records[];
if(f.select(records))
{
   ArrayPrint(records);
}
```

De hecho, el método *select* puede hacer algo más importante que dejamos para un estudio opcional independiente.

En primer lugar, en la lista de noticias resultante, conviene insertar de algún modo un separador (*delimiter*) entre el pasado y el futuro, para que el ojo lo capte. En teoría, esta característica es extremadamente importante para los calendarios, pero por alguna razón, no está disponible en la interfaz de usuario de MetaTrader 5 y en el sitio web mql5.com. Nuestra implementación es capaz de insertar una estructura vacía entre el pasado y el futuro, que debemos mostrar visualmente (de lo que nos ocuparemos a continuación).

En segundo lugar, el tamaño del array resultante puede ser bastante grande (especialmente en las primeras etapas de selección de ajustes), por lo que el método *select* ofrece además la posibilidad de limitar el tamaño del array (*limit*). Para ello, se eliminan los elementos más alejados de la hora actual.

Así, el prototipo completo del método tiene este aspecto:

```
bool select(MqlCalendarValue &result[],
   const bool delimiter = false, const int limit = -1);
```

Por defecto, no se inserta ningún delimitador y el array no se trunca.

Un par de párrafos más arriba mencionamos una subtarea adicional del filtrado que es la visualización del array resultante. La clase *CalendarFilter* tiene un método especial *format*, que convierte el array de estructuras pasado *MqlCalendarValue &data[]* en un array de cadenas legibles *string &result[]*. El código del método se encuentra en el archivo adjunto *CalendarFilter.mqh*.

```
bool format(const MqlCalendarValue &data[],
   const ENUM_CALENDAR_PROPERTY &props[], string &result[],
   const bool padding = false, const bool header = false);
```

Los campos de *MqlCalendarValue* que queremos mostrar se especifican en el array *props*. Recordemos que la enumeración ENUM_CALENDAR_PROPERTY contiene campos de las tres estructuras de calendario dependientes para que un programa MQL pueda mostrar automáticamente no sólo los indicadores económicos de un registro de evento concreto, sino también su nombre, características, país o código de divisa. Todo ello se implementa mediante el método *format*.

Cada fila del array de salida *result* contiene una representación textual del valor de uno de los campos (número, descripción, elemento de enumeración). El tamaño del array *result* es igual al producto del número de estructuras en la entrada (en *data*) y el número de campos visualizados (en *props*). El parámetro opcional *header* permite añadir una fila con los nombres de los campos (columnas) al principio del array de salida. El parámetro *padding* controla la generación de espacios adicionales en el texto para que sea conveniente mostrar la tabla en un tipo de letra monoespaciado (por ejemplo, en una revista).

La clase *CalendarFilter* tiene otro método público importante: *update*.

```
bool update(MqlCalendarValue &result[]);
```

Su estructura repite casi por completo *select*. Sin embargo, en lugar de llamar a las funciones *CalendarValueHistoryByEvent* y *CalendarValueHistory*, el método llama a *CalendarValueLastByEvent* y *CalendarValueLast*. El propósito del método es obvio: consulta el calendario en busca de cambios recientes que coincidan con las condiciones de filtrado. Pero para su funcionamiento requiere una identificación de los cambios. Dicho campo sí está definido en la clase: la primera vez se rellena dentro del método *select*.

```
class CalendarFilter
{
protected:
   ...
   ulong change;
   ...
public:
   bool select(MqlCalendarValue &result[],
      const bool delimiter = false, const int limit = -1)
   {
      ...
      change = 0;
      MqlCalendarValue dummy[];
      CalendarValueLast(change, dummy);
      ...
   }
```

Algunos matices de la clase *CalendarFilter* todavía están «tras el telón», pero abordaremos algunos de ellos en las siguientes secciones.

Probemos el filtro en acción: primero en un sencillo script *CalendarFilterPrint.mq5* y después en un indicador más práctico *CalendarMonitor.mq5*.

En los parámetros de entrada del script puede establecer el contexto (código de país o divisa), el intervalo de tiempo y la cadena para la búsqueda de texto completo por nombres de eventos, así como limitar el tamaño de la tabla de noticias resultante.

```
input string Context; // Context (country - 2 characters, currency - 3 characters, em
input ENUM_CALENDAR_SCOPE Scope = SCOPE_MONTH;
input string Text = "farm";
input int Limit = -1;
```

Dados los parámetros, se crea un objeto filtro global.

```
CalendarFilter f(Context, TimeCurrent() - Scope, TimeCurrent() + Scope);
```

A continuación, en *OnStart*, configuramos un par de condiciones constantes adicionales (importancia media y alta de los eventos) y la presencia de una previsión (el campo no es igual a LONG_MIN), así como el paso y una cadena de búsqueda al objeto.

```
void OnStart()
{
   f.let(CALENDAR_IMPORTANCE_LOW, GREATER)
      .let(LONG_MIN, CALENDAR_PROPERTY_RECORD_FORECAST, NOT_EQUAL)
      .let(Text); // with '*' replacement support
      // NB: strings with the character length of 2 or 3 without '*' will be treated
      // as a country or currency code, respectively
```

A continuación, se llama al método *select* y el array resultante de estructuras *MqlCalendarValue* se formatea en una tabla con 9 columnas utilizando el método *format*.

```
  MqlCalendarValue records[];
    // apply the filter conditions and get the result
    if(f.select(records, true, Limit))
    {
       static const ENUM_CALENDAR_PROPERTY props[] =
       {
          CALENDAR_PROPERTY_RECORD_TIME,
          CALENDAR_PROPERTY_COUNTRY_CURRENCY,
          CALENDAR_PROPERTY_EVENT_NAME,
          CALENDAR_PROPERTY_EVENT_IMPORTANCE,
          CALENDAR_PROPERTY_RECORD_ACTUAL,
          CALENDAR_PROPERTY_RECORD_FORECAST,
          CALENDAR_PROPERTY_RECORD_PREVISED,
          CALENDAR_PROPERTY_RECORD_IMPACT,
          CALENDAR_PROPERTY_EVENT_SECTOR,
       };
       static const int p = ArraySize(props);

       // output the formatted result
       string result[];
       if(f.format(records, props, result, true, true))
       {
          for(int i = 0; i < ArraySize(result) / p; ++i)
          {
             Print(SubArrayCombine(result, " | ", i * p, p));
          }
       }
    }
}
```

Las celdas de la tabla se unen en filas y se envían al registro.

Con la configuración por defecto (es decir, para todos los países y divisas, con la parte «granja» en el nombre de los eventos de importancia media y alta), puede obtener algo como este calendario.

```
Selecting calendar records...
country[i]= / ok
calendarValueHistory(temp,from,to,country[i],c)=2372 / ok
Filtering 2372 records
Got 9 records
TIME | CUR⍰ | NAME | IMPORTAN⍰ | ACTU⍰ | FORE⍰ | PREV⍰ | IMPACT | SECT⍰
2022.06.02 15:15 | USD | ADP Nonfarm Employment Change | HIGH | +128 | -225 | +202 |
2022.06.02 15:30 | USD | Nonfarm Productivity q/q | MODERATE | -7.3 | -7.5 | -7.5 | P
2022.06.03 15:30 | USD | Nonfarm Payrolls | HIGH | +390 | -19 | +436 | POSITIVE | JOB
2022.06.03 15:30 | USD | Private Nonfarm Payrolls | MODERATE | +333 | +8 | +405 | POS
2022.06.09 08:30 | EUR | Nonfarm Payrolls q/q | MODERATE | +0.3 | +0.3 | +0.3 | NA |
- | - | - | - | - | - | - | - | -
2022.07.07 15:15 | USD | ADP Nonfarm Employment Change | HIGH | +nan | -263 | +128 |
2022.07.08 15:30 | USD | Nonfarm Payrolls | HIGH | +nan | -229 | +390 | NA | JOBS
2022.07.08 15:30 | USD | Private Nonfarm Payrolls | MODERATE | +nan | +51 | +333 | NA
```

Ahora echemos un vistazo al indicador *CalendarMonitor.mq5*. Su objetivo es mostrar al usuario la selección actual de eventos en el gráfico de acuerdo con los filtros especificados. Para visualizar la tabla, utilizaremos la ya conocida clase de marcador (*Tableau.mqh*, véase la sección Cálculo del margen para una orden futura). El indicador no tiene búferes ni gráficos.

Los parámetros de entrada permiten establecer el rango de la ventana de tiempo (*scope*), así como el contexto global para el objeto *CalendarFilter*, que es la divisa o el código de país en *Context* (vacío por defecto, es decir, sin restricciones) o mediante una bandera booleana *UseChartCurrencies*. Está activada por defecto, y se recomienda utilizarla para recibir automáticamente noticias de aquellas divisas que conforman la herramienta de trabajo del gráfico.

```
input string Context; // Context (country - 2 chars, currency - 3 chars, empty - all)
input ENUM_CALENDAR_SCOPE Scope = SCOPE_WEEK;
input bool UseChartCurrencies = true;
```

Se pueden aplicar filtros adicionales por tipo de evento, sector y gravedad.

```
input ENUM_CALENDAR_EVENT_TYPE_EXT Type = TYPE_ANY;
input ENUM_CALENDAR_EVENT_SECTOR_EXT Sector = SECTOR_ANY;
input ENUM_CALENDAR_EVENT_IMPORTANCE_EXT Importance = IMPORTANCE_MODERATE; // Importa
```

La importancia establece el límite inferior de la selección, no la coincidencia exacta. Así, el valor por defecto de IMPORTANCE_MODERATE captará no sólo la importancia moderada, sino también la alta.

Un lector atento observará que aquí se utilizan enumeraciones desconocidas: ENUM_CALENDAR_EVENT_TYPE_EXT, ENUM_CALENDAR_EVENT_SECTOR_EXT, ENUM_CALENDAR_EVENT_IMPORTANCE_EXT. Se encuentran en el archivo ya mencionado *CalendarDefines.mqh*, y coinciden (casi uno a uno) con enumeraciones integradas similares. La única diferencia es que han añadido un elemento que significa «cualquier» valor. Necesitamos describir tales enumeraciones para simplificar la introducción de condiciones: ahora el filtro de cada campo se configura mediante una lista desplegable en la que se puede seleccionar uno de los valores o desactivar el filtro. Si no fuera por el elemento de enumeración añadido, tendríamos que introducir una bandera lógica «on/off» en la interfaz para cada campo.

Además, los parámetros de entrada permiten consultar los eventos por la presencia en ellos de indicadores reales, de previsión y anteriores, así como por la búsqueda de una cadena de texto (*Text*).

```
input string Text;
input ENUM_CALENDAR_HAS_VALUE HasActual = HAS_ANY;
input ENUM_CALENDAR_HAS_VALUE HasForecast = HAS_ANY;
input ENUM_CALENDAR_HAS_VALUE HasPrevious = HAS_ANY;
input ENUM_CALENDAR_HAS_VALUE HasRevised = HAS_ANY;
input int Limit = 30;
```

Los objetos *CalendarFilter* y *tableau* se describen a nivel global.

```
CalendarFilter f(Context);
AutoPtr<Tableau> t;
```

Tenga en cuenta que el filtro se crea una vez, mientras que la tabla está representada por un autoselector y se volverá a crear dinámicamente en función del tamaño de los datos recibidos.

La configuración de los filtros se realiza en *OnInit* mediante llamadas consecutivas a los métodos de *let* en función de los parámetros de entrada.

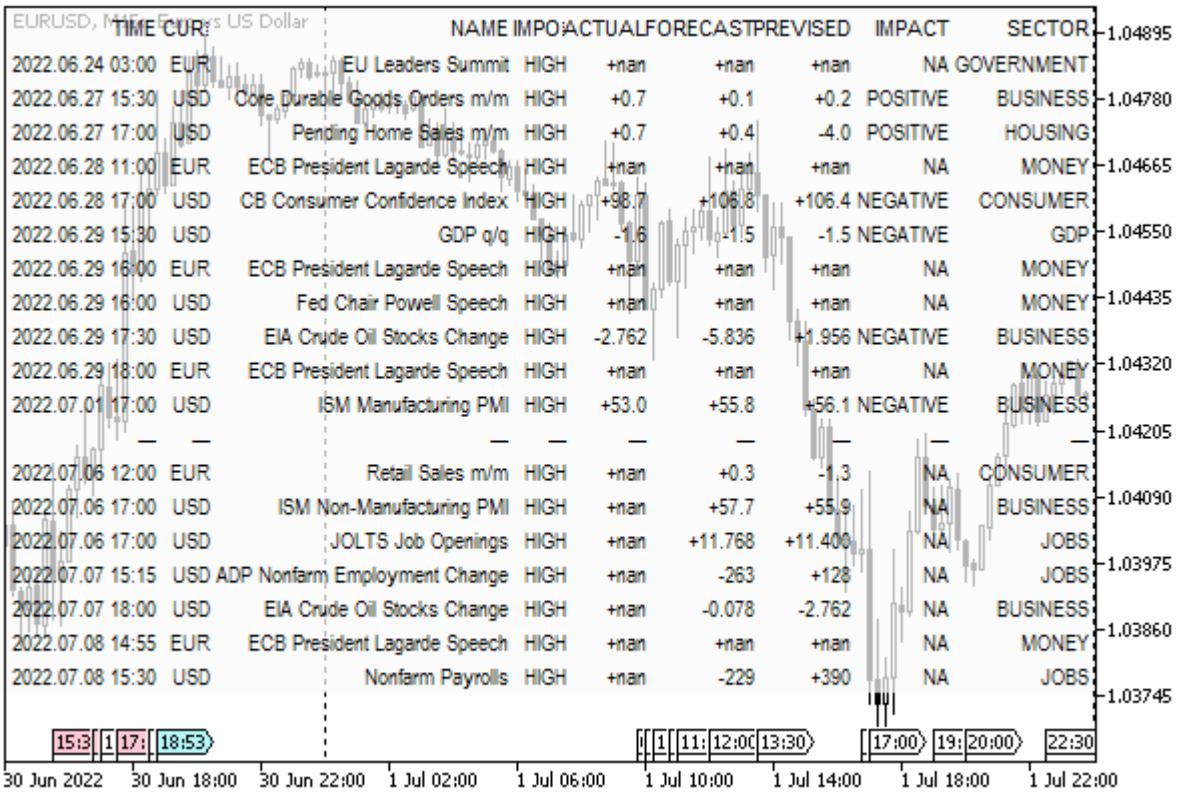
```
int OnInit()
{
   if(!f.isLoaded()) return INIT_FAILED;

   if(UseChartCurrencies)
   {
      const string base = SymbolInfoString(_Symbol, SYMBOL_CURRENCY_BASE);
      const string profit = SymbolInfoString(_Symbol, SYMBOL_CURRENCY_PROFIT);
      f.let(base);
      if(base != profit)
      {
         f.let(profit);
      }
   }

   if(Type != TYPE_ANY)
   {
      f.let((ENUM_CALENDAR_EVENT_TYPE)Type);
   }

   if(Sector != SECTOR_ANY)
   {
      f.let((ENUM_CALENDAR_EVENT_SECTOR)Sector);
   }

   if(Importance != IMPORTANCE_ANY)
   {
      f.let((ENUM_CALENDAR_EVENT_IMPORTANCE)(Importance - 1), GREATER);
   }

   if(StringLen(Text))
   {
      f.let(Text);
   }

   if(HasActual != HAS_ANY)
   {
      f.let(LONG_MIN, CALENDAR_PROPERTY_RECORD_ACTUAL,
         HasActual == HAS_SET ? NOT_EQUAL : EQUAL);
   }
   ...

   EventSetTimer(1);

   return INIT_SUCCEEDED;
}
```

Al final, se pone en marcha un segundo temporizador. Todo el trabajo se realiza en *OnTimer*.

```
void OnTimer()
{
   static const ENUM_CALENDAR_PROPERTY props[] = // table columns
   {
      CALENDAR_PROPERTY_RECORD_TIME,
      CALENDAR_PROPERTY_COUNTRY_CURRENCY,
      CALENDAR_PROPERTY_EVENT_NAME,
      CALENDAR_PROPERTY_EVENT_IMPORTANCE,
      CALENDAR_PROPERTY_RECORD_ACTUAL,
      CALENDAR_PROPERTY_RECORD_FORECAST,
      CALENDAR_PROPERTY_RECORD_PREVISED,
      CALENDAR_PROPERTY_RECORD_IMPACT,
      CALENDAR_PROPERTY_EVENT_SECTOR,
   };
   static const int p = ArraySize(props);

   MqlCalendarValue records[];

almost one to one   f.let(TimeCurrent() - Scope, TimeCurrent() + Scope); // shift the

   const ulong trackID = f.getChangeID();
   if(trackID) // if the state has already been removed, check for changes
   {
      if(f.update(records)) // request changes by filters
      {
         // if there are changes, notify the user
         string result[];
         f.format(records, props, result);
         for(int i = 0; i < ArraySize(result) / p; ++i)
         {
            Alert(SubArrayCombine(result, " | ", i * p, p));
         }
      // "fall through" further to update the table
      }
      else if(trackID == f.getChangeID())
      {
         return; // calendar without changes
      }
   }

   // request a complete set of news by filters
   f.select(records, true, Limit);

   // display the news table on the chart
   string result[];
   f.format(records, props, result, true, true);

   if(t[] == NULL || t[].getRows() != ArraySize(records) + 1)
   {
      t = new Tableau("CALT", ArraySize(records) + 1, p,
         TBL_CELL_HEIGHT_AUTO, TBL_CELL_WIDTH_AUTO,
```

```
            Corner, Margins, FontSize, FontName, FontName + " Bold",
            TBL_FLAG_ROW_0_HEADER,
            BackgroundColor, BackgroundTransparency);
      }
      const string hints[] = {};
      t[].fill(result, hints);
   }
```

Si ejecutamos el indicador en el gráfico EURUSD con la configuración predeterminada, podemos obtener la siguiente imagen.



TIME	CUR	NAME	IMPO	ACTUAL	FORECAST	PREVISED	IMPACT	SECTOR
2022.06.24 03:00	EUR	EU Leaders Summit	HIGH	+nan	+nan	+nan	NA	GOVERNMENT
2022.06.27 15:30	USD	Core Durable Goods Orders m/m	HIGH	+0.7	+0.1	+0.2	POSITIVE	BUSINESS
2022.06.27 17:00	USD	Pending Home Sales m/m	HIGH	+0.7	+0.4	-4.0	POSITIVE	HOUSING
2022.06.28 11:00	EUR	ECB President Lagarde Speech	HIGH	+nan	+nan	+nan	NA	MONEY
2022.06.28 17:00	USD	CB Consumer Confidence Index	HIGH	+98.7	+106.8	+106.4	NEGATIVE	CONSUMER
2022.06.29 15:30	USD	GDP q/q	HIGH	-1.6	-1.5	-1.5	NEGATIVE	GDP
2022.06.29 16:00	EUR	ECB President Lagarde Speech	HIGH	+nan	+nan	+nan	NA	MONEY
2022.06.29 16:00	USD	Fed Chair Powell Speech	HIGH	+nan	+nan	+nan	NA	MONEY
2022.06.29 17:30	USD	EIA Crude Oil Stocks Change	HIGH	-2.762	-5.836	+1.956	NEGATIVE	BUSINESS
2022.06.29 18:00	EUR	ECB President Lagarde Speech	HIGH	+nan	+nan	+nan	NA	MONEY
2022.07.01 17:00	USD	ISM Manufacturing PMI	HIGH	+53.0	+55.8	+56.1	NEGATIVE	BUSINESS
—	—	—	—	—	—	—	—	—
2022.07.06 12:00	EUR	Retail Sales m/m	HIGH	+nan	+0.3	-1.3	NA	CONSUMER
2022.07.06 17:00	USD	ISM Non-Manufacturing PMI	HIGH	+nan	+57.7	+55.9	NA	BUSINESS
2022.07.06 17:00	USD	JOLTS Job Openings	HIGH	+nan	+11.768	+11.400	NA	JOBS
2022.07.07 15:15	USD	ADP Nonfarm Employment Change	HIGH	+nan	-263	+128	NA	JOBS
2022.07.07 18:00	USD	EIA Crude Oil Stocks Change	HIGH	+nan	-0.078	-2.762	NA	BUSINESS
2022.07.08 14:55	EUR	ECB President Lagarde Speech	HIGH	+nan	+nan	+nan	NA	MONEY
2022.07.08 15:30	USD	Nonfarm Payrolls	HIGH	+nan	-229	+390	NA	JOBS

Noticias filtradas y formateadas en el gráfico

7.3.11 Transferir la base de datos de calendarios al probador

El calendario sólo está disponible para los programas MQL en línea, por lo que simular las estrategias de trading de noticias plantea algunas dificultades. Una de las soluciones consiste en crear de forma independiente una determinada imagen del calendario, es decir, la caché, y luego utilizarla dentro del probador. Las tecnologías de almacenamiento en caché pueden ser diferentes, como archivos o una base de datos SQLite incrustada. En esta sección mostraremos una implementación utilizando un archivo.

En cualquier caso, cuando utilice la caché de calendario, recuerde que corresponde a un punto concreto en el tiempo X. En todos los eventos «antiguos» (informes financieros) que ocurrieron antes de X, los valores reales ya están establecidos, y en los posteriores (en el «futuro», relativos a X) no hay valores reales, y no los habrá hasta que aparezca una nueva copia más reciente de la caché. En otras palabras: no tiene sentido probar indicadores y Asesores Expertos a la derecha de X. En cuanto a los que están a la izquierda de X, debe evitar mirar hacia adelante, es decir, no leer los indicadores actuales hasta el momento de la publicación de cada noticia específica.

> ¡Atención! Cuando se solicitan datos de calendario en el terminal, la hora de todos los eventos se notifica teniendo en cuenta la zona horaria actual del servidor, incluida una posible corrección por el horario de «verano» (por regla general, esto significa aumentar las marcas de tiempo en 1 hora). Esto sincroniza la publicación de noticias con los tiempos de cotización en línea. Sin embargo, los cambios de reloj pasados (hace medio año, un año o más) sólo se muestran en las cotizaciones, pero no en los eventos del calendario. Toda la base de datos del calendario se lee a través de MQL5 según la zona horaria actual del servidor. Debido a esto, cualquier archivo de calendario creado contendrá las marcas de tiempo correctas para aquellos eventos que ocurrieron con el mismo modo DST (activado o desactivado) que estaba activo en el momento del almacenamiento. Para los eventos en semestres «opuestos» se requiere realizar un ajuste de forma independiente durante una hora después de leer el archivo. En los ejemplos siguientes, se omite esta situación.

Llamemos a la clase de caché *CalendarCache* y pongámosla en un archivo llamado *CalendarCache.mqh*. Tendremos que guardar las 3 tablas de la base del calendario en el archivo (*MqlCalendarCountry*, *MqlCalendarEvent*, *MqlCalendarValue*). MQL5 proporciona las funciones *FileWriteArray* y *FileReadArray* (véase Escritura y lectura de arrays) que pueden escribir y leer directamente arrays de estructuras simples en archivos. Sin embargo, 2 de cada 3 estructuras en nuestro caso no son simples, porque tienen campos de cadena. Por lo tanto, necesitamos un mecanismo para almacenar cadenas por separado, similar al que ya utilizamos en la clase *CalendarFilter* (había un array de cadenas *stringCache*, y el índice de la cadena deseada de este array se indicaba en los filtros).

Para evitar que falten cadenas de diferentes estructuras de «calendario» en un «diccionario», prepararemos una clase de plantilla *StringRef*: el parámetro de tipo T será cualquiera de las estructuras *MqlCalendar*. Esto nos dará una caché de cadenas separada para los países, y una caché de cadenas separada para los tipos de eventos.

```
template<typename T>
struct StringRef
{
   static string cache[];
   int index;
   StringRef(): index(-1) { }

   void operator=(const string s)
   {
      if(index == -1)
      {
         PUSH(cache, s);
         index = ArraySize(cache) - 1;
      }
      else
      {
         cache[index] = s;
      }
   }

   string operator[](int x = 0) const
   {
      if(index != -1)
      {
         return cache[index];
      }
      return NULL;
   }

   static bool save(const int handle)
   {
      FileWriteInteger(handle, ArraySize(cache));
      for(int i = 0; i < ArraySize(cache); ++i)
      {
         FileWriteInteger(handle, StringLen(cache[i]));
         FileWriteString(handle, cache[i]);
      }
      return true;
   }

   static bool load(const int handle)
   {
      const int n = FileReadInteger(handle);
      for(int i = 0; i < n; ++i)
      {
         PUSH(cache, FileReadString(handle, FileReadInteger(handle)));
      }
      return true;
   }
};
```

```
template<typename T>
static string StringRef::cache[];
```

Las cadenas se almacenan en el array *cache* utilizando *operator=*, y se extraen de ella utilizando *operator[]* (con un índice ficticio que siempre se omite). Cada objeto almacena sólo el índice de la cadena en el array. El array *cache* se declara estático, por lo que acumulará todos los campos de cadena de una estructura T. Quien lo desee puede cambiar el método de almacenamiento en caché de forma que cada campo de la estructura tenga su propio array, pero esto no es importante para nosotros.

La escritura de un array en un archivo y la lectura desde un archivo se realizan mediante un par de métodos estáticos *save* y *load*: ambos toman un manejador de archivo como parámetro.

Teniendo en cuenta la clase *StringRef*, vamos a describir estructuras que duplican las estructuras de calendario estándar que utilizan objetos *StringRef* en lugar de campos de cadena. Por ejemplo, para *MqlCalendarCountry* obtenemos *MqlCalendarCountryRef*. Las estructuras estándar y modificadas se copian entre sí de forma similar mediante los operadores sobrecargados '=' y '[]'.

```
struct MqlCalendarCountryRef
{
   ulong id;
   StringRef<MqlCalendarCountry> name;
   StringRef<MqlCalendarCountry> code;
   StringRef<MqlCalendarCountry> currency;
   StringRef<MqlCalendarCountry> currency_symbol;
   StringRef<MqlCalendarCountry> url_name;

   void operator=(const MqlCalendarCountry &c)
   {
      id = c.id;
      name = c.name;
      code = c.code;
      currency = c.currency;
      currency_symbol = c.currency_symbol;
      url_name = c.url_name;
   }

   MqlCalendarCountry operator[](int x = 0) const
   {
      MqlCalendarCountry r;
      r.id = id;
      r.name = name[];
      r.code = code[];
      r.currency = currency[];
      r.currency_symbol = currency_symbol[];
      r.url_name = url_name[];
      return r;
   }
};
```

Observe que los operadores de asignación del primer método tienen la sobrecarga '=' de *StringRef*, debido a lo cual todas las líneas caen dentro del array *StringRef<MqlCalendarCountry>::cache*. En el

segundo método, las llamadas al operador '[]' obtienen de forma invisible la dirección de la cadena y devuelven de *StringRef* directamente la cadena almacenada en esa dirección en el array *cache*.

La estructura *MqlCalendarEventRef* se define de forma similar, pero sólo hay 3 campos en ella (*source_url*, *event_code*, *name*) que requieren sustituir el tipo *string* por *StringRef<MqlCalendarEvent>*. La estructura *MqlCalendarValue* no requiere tales transformaciones, ya que no contiene campos de cadena.

Con esto concluyen las etapas preparatorias, y se puede proceder a la clase principal de caché *CalendarCache*.

Por consideraciones generales, así como por compatibilidad con la clase *CalendarFilter* ya desarrollada, vamos a describir los campos de la caché que especifican el contexto (país o divisa), el intervalo de fechas para los eventos almacenados y el momento de generación de la caché (tiempo X, variable t).

```
class CalendarCache
{
   string context;
   datetime from, to;
   datetime t;
   ...

public:
   CalendarCache(const string _context = NULL,
      const datetime _from = 0, const datetime _to = 0):
      context(_context), from(_from), to(_to), t(0)
   {
      ...
   }
```

En realidad, no tiene mucho sentido establecer restricciones al crear una caché a partir de un calendario. Una caché completa es probablemente más práctica, ya que su tamaño no es crítico al tratarse de unas dos docenas de megabytes hasta mediados de 2022 (esto incluye datos históricos desde 2007 con eventos previstos hasta 2024). No obstante, las restricciones pueden ser útiles para programas de demostración con una funcionalidad artificialmente reducida.

Es obvio que se deben proporcionar arrays de estructuras de calendario en la caché para almacenar todos los datos.

```
   MqlCalendarValue values[];
   MqlCalendarEvent events[];
   MqlCalendarCountry countries[];
   ...
```

Inicialmente, estos se rellenan desde la base de datos de calendarios mediante el método *update*.

```
bool update()
{
   string country = NULL, currency = NULL;
   if(StringLen(context) == 3)
   {
      currency = context;
   }
   else if(StringLen(context) == 2)
   {
      country = context;
   }

   Print("Reading online calendar base...");

   if(!PRTF(CalendarValueHistory(values, from, to, country, currency))
      || (currency != NULL ?
         !PRTF(CalendarEventByCurrency(currency, events)) :
         !PRTF(CalendarEventByCountry(country, events)))
      || !PRTF(CalendarCountries(countries)))
   {
      // object is not ready, t = 0
   }
   else
   {
      t = TimeTradeServer();
   }
   return (bool)t;
}
```

El campo *t* es una señal de la salud de la caché, con el tiempo de llenado de los arrays.

El objeto de caché lleno puede escribirse en un archivo utilizando el método *save*. Al principio del archivo hay un encabezado CALENDAR_CACHE_HEADER: esta es la cadena «MQL5 Calendar Cache\r\nv.1.0\r\n», que le permite asegurarse de que el formato es correcto cuando se lee. A continuación, el método guarda las variables *context*, *from*, *to* y *t*, así como el array *values*, «tal cual». Antes del array propiamente dicho, anotamos su tamaño para restablecerlo al leer.

```
bool save(string filename = NULL)
{
   if(!t) return false;

   MqlDateTime mdt;
   TimeToStruct(t, mdt);
   if(filename == NULL) filename = "calendar-" +
      StringFormat("%04d-%02d-%02d-%02d-%02d.cal",
      mdt.year, mdt.mon, mdt.day, mdt.hour, mdt.min);
   int handle = PRTF(FileOpen(filename, FILE_WRITE | FILE_BIN));
   if(handle == INVALID_HANDLE) return false;

   FileWriteString(handle, CALENDAR_CACHE_HEADER);
   FileWriteString(handle, context, 4);
   FileWriteLong(handle, from);
   FileWriteLong(handle, to);
   FileWriteLong(handle, t);
   FileWriteInteger(handle, ArraySize(values));
   FileWriteArray(handle, values);
   ...
```

Con los arrays *events* y *countries* vienen nuestras estructuras de envoltorio con el sufijo «Ref». El método de ayuda *store* convierte el array *events* en un array de estructuras simples *erefs*, en la que las cadenas se sustituyen por números en el diccionario de cadenas *StringRef<MqlCalendarEvent>*. Estas estructuras simples pueden escribirse ya en un archivo de la forma habitual, pero para su posterior lectura es necesario guardar también todas las líneas del diccionario (llamando a *StringRef<MqlCalendarEvent>::save(handle)*). Las estructuras de los países se convierten y guardan en un archivo del mismo modo.

```
   MqlCalendarEventRef erefs[];
   store(erefs, events);
   FileWriteInteger(handle, ArraySize(erefs));
   FileWriteArray(handle, erefs);
   StringRef<MqlCalendarEvent>::save(handle);

   MqlCalendarCountryRef crefs[];
   store(crefs, countries);
   FileWriteInteger(handle, ArraySize(crefs));
   FileWriteArray(handle, crefs);
   StringRef<MqlCalendarCountry>::save(handle);

   FileClose(handle);
   return true;
}
```

El método *store* antes mencionado es bastante sencillo: en él, en un bucle sobre los elementos, se ejecuta un operador de asignación sobrecargado en las estructuras *MqlCalendarEventRef* o *MqlCalendarCountryRef*.

```
template<typename T1,typename T2>
void static store(T1 &array[], T2 &origin[])
{
   ArrayResize(array, ArraySize(origin));
   for(int i = 0; i < ArraySize(origin); ++i)
   {
      array[i] = origin[i];
   }
}
```

Para cargar el archivo recibido en el objeto caché se escribe un método espejo *load*. Lee los datos del archivo en variables y arrays en el mismo orden, realizando simultáneamente transformaciones inversas de los campos de cadena para tipos de eventos y países.

```
bool load(const string filename)
{
   Print("Loading calendar cache ", filename);
   t = 0;
   int handle = PRTF(FileOpen(filename, FILE_READ | FILE_BIN));
   if(handle == INVALID_HANDLE) return false;

   const string header = FileReadString(handle, StringLen(CALENDAR_CACHE_HEADER));
   if(header != CALENDAR_CACHE_HEADER) return false; // not our format

   context = FileReadString(handle, 4);
   if(!StringLen(context)) context = NULL;
   from = (datetime)FileReadLong(handle);
   to = (datetime)FileReadLong(handle);
   t = (datetime)FileReadLong(handle);
   Print("Calendar cache interval: ", from, "-", to);
   Print("Calendar cache saved at: ", t);
   int n = FileReadInteger(handle);
   FileReadArray(handle, values, 0, n);

   MqlCalendarEventRef erefs[];
   n = FileReadInteger(handle);
   FileReadArray(handle, erefs, 0, n);
   StringRef<MqlCalendarEvent>::load(handle);
   restore(events, erefs);

   MqlCalendarCountryRef crefs[];
   n = FileReadInteger(handle);
   FileReadArray(handle, crefs, 0, n);
   StringRef<MqlCalendarCountry>::load(handle);
   restore(countries, crefs);

   FileClose(handle);
   ... // something else will be here
}
```

El método auxiliar *restore* utiliza la sobrecarga del operador '[]' en un bucle sobre los elementos de las estructuras *MqlCalendarEventRef* o *MqlCalendarCountryRef* para obtener la línea en sí por número de línea y asignarla a una estructura estándar *MqlCalendarEvent* o *MqlCalendarCountry*.

```
template<typename T1,typename T2>
void static restore(T1 &array[], T2 &origin[])
{
   ArrayResize(array, ArraySize(origin));
   for(int i = 0; i < ArraySize(origin); ++i)
   {
      array[i] = origin[i][];
   }
}
```

En esta fase podríamos escribir ya un indicador de prueba sencillo basado en la clase *CalendarCache*, ejecutarlo en un gráfico en línea y guardarlo en un archivo con la caché del calendario. A continuación,

el archivo podría cargarse desde la copia del indicador en el probador, y podría recibirse el conjunto completo de eventos. Sin embargo, esto no es suficiente para desarrollos prácticos.

Y es que, para acceder rápidamente a los datos, es necesario proporcionar *indexing*, un concepto bien conocido en programación, que tocaremos más adelante, en el capítulo dedicado a las bases de datos. En teoría, podríamos utilizar el motor SQLite integrado para almacenar la caché, y entonces obtendríamos índices «gratis», pero hablaremos de ello más adelante.

El sentido de la indexación es fácil de entender si imaginamos cómo implementar eficazmente análogos de las funciones de calendario estándar en nuestra caché. Por ejemplo, el ID del evento se pasa en la función *CalendarValueById*. La enumeración directa de los registros en el array *values* llevaría mucho tiempo. Por lo tanto, es necesario complementar el array con alguna «estructura de datos» que nos permita optimizar la búsqueda. «Estructura de datos» va entre comillas porque no se trata del significado del lenguaje de programación (*struct*), sino en general de la arquitectura de construcción de datos. Puede constar de diferentes partes y basarse en distintos principios organizativos. Por supuesto, los datos adicionales requerirán memoria, pero cambiar memoria por velocidad es un enfoque habitual en programación.

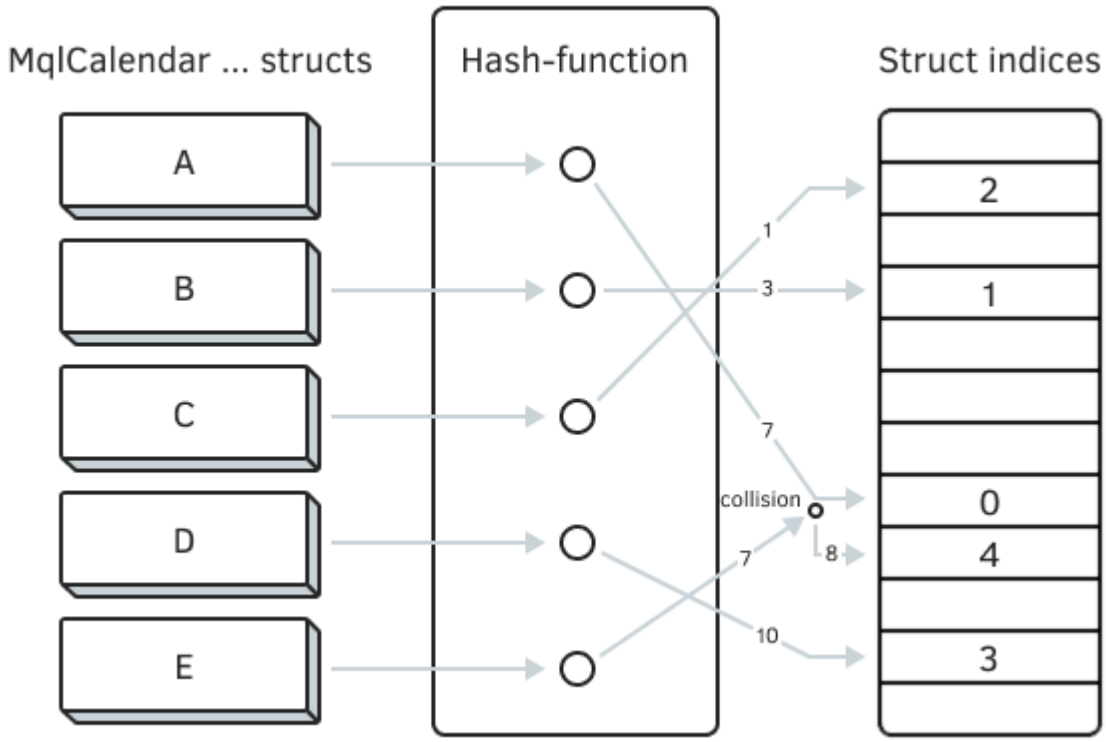
La solución más sencilla para la indexación es un array bidimensional separado, ordenado de forma ascendente para que se pueda buscar rápidamente utilizando la función *ArrayBsearch*. Para la segunda dimensión bastan dos elementos: los valores con índices *[i][0]*, por los que se realiza la ordenación, contienen identificadores, y los valores *[i][1]* contienen las posiciones ordinales en el array de estructuras.

Otro concepto utilizado con frecuencia es *hashing*, que es una transformación de los valores iniciales en algunas claves (hashes, enteros) de tal manera que proporcione el mínimo número de colisiones (coincidencias de claves para datos iniciales diferentes). La propiedad fundamental de las claves es una distribución aleatoria casi uniforme de sus valores, por lo que pueden utilizarse como índices en arrays preasignados. Calcular una función hash para un único elemento de los datos originales es un proceso rápido que en realidad proporciona la dirección del elemento en sí. Por ejemplo, las conocidas estructuras de datos de mapa hash siguen este principio.

Si los dos valores originales obtienen el mismo hash (aunque esto es poco frecuente), se alinean en una lista por su clave y se realiza una búsqueda secuencial dentro de la lista. Sin embargo, como las funciones hash se eligen para que el número de coincidencias sea pequeño, la búsqueda suele dar con el objetivo en cuanto se calcula el hash.

Para la demostración, utilizaremos ambos enfoques en la clase *CalendarCache*: hashing y búsqueda binaria.

El paquete de MetaTrader 5 incluye un conjunto de clases para crear mapas hash (MQL5/Include/Generic/HashMap.mqh), pero nosotros nos las arreglaremos con nuestra propia implementación más sencilla, en la que sólo se mantiene el principio de utilizar la función hash.



Esquema de indexación de datos mediante hashing

En nuestro caso, basta con hacer un hash sólo de los identificadores de los objetos de calendario. La función hash que elijamos tendrá que convertir el identificador en un índice dentro de un array especial: la posición del identificador en el array de estructuras de «calendario» se almacenará en una celda con este índice. Para países, tipos de eventos y noticias específicas, se asigna según su propio array.

```
int id4country[];
int id4event[];
int id4value[];
```

Sus elementos almacenarán el número de secuencia de la entrada en el array correspondiente (*countries*, *events*, *values*).

Para cada uno de los arrays de «redirección» deben asignarse al menos 2 veces más elementos que el número de estructuras correspondientes en la base de datos (y en la caché) del calendario. Gracias a esta redundancia, minimizamos el número de colisiones hash. Se cree que la mayor eficacia se consigue cuando se elige un tamaño igual a un número primo. Por lo tanto, la clase tiene un método estático *size2prime* que devuelve el tamaño recomendado del array de «cestas» hash (uno de *id4* - arrays) según el número de elementos de los datos de origen.

```
static int size2prime(const int size)
{
   static int primes[] =
   {
     17, 53, 97, 193, 389,
     769, 1543, 3079, 6151,
     12289, 24593, 49157, 98317,
     196613, 393241, 786433, 1572869,
     3145739, 6291469, 12582917, 25165843,
     50331653, 100663319, 201326611, 402653189,
     805306457, 1610612741
   };

   const int pmax = ArraySize(primes);
   for(int p = 0; p < pmax; ++p)
   {
      if(primes[p] >= 2 * size)
      {
         return primes[p];
      }
   }
   return size;
}
```

Todo el proceso de hashing del calendario se describe en el método *hash*. Veamos su inicio utilizando el ejemplo de un array de estructuras *countries*, y los otros dos arrays se tratan de forma similar.

Así que obtenemos el tamaño de índice «plano» recomendado *id4country* a partir del tamaño del array *countries* llamando a *size2prime*. Inicialmente, el array de índices se rellena con el valor -1, es decir, todos sus elementos están libres. Más adelante en el bucle a través de los países, es necesario calcular el hash para cada siguiente identificador de país y usándolo encontrar un índice libre en el array *id4country*. Este es el trabajo para el método de ayuda *place*.

```
bool hash()
{
   Print("Hashing calendar...");
   ...
   const int c = PRTF(ArraySize(countries));
   PRTF(ArrayResize(id4country, size2prime(c)));
   ArrayInitialize(id4country, -1);

   for(int i = 0; i < c; ++i)
   {
      if(place(countries[i].id, i, id4country) == -1)
      {
         return false; // failure
      }
   }
   ...
   return true; // success
}
```

La función hash dentro de *place* es la expresión *(MathSwap(id) ^ 0xEFCDAB8967452301) % n*, donde *id* es nuestro identificador, y *n* es el tamaño del array de índices. Así, el resultado de los cálculos se reduce siempre a un índice válido dentro de *array[]*. El principio de elección de una función hash es un tema aparte que queda fuera del alcance de este libro.

```
int place(const ulong id, const int index, int &array[])
{
   const int n = ArraySize(array);
   int p = (int)((MathSwap(id) ^ 0xEFCDAB8967452301) % n); // hash function
   int attempt = 0;
   while(array[p] != -1)
   {
      if(++attempt > n / 10) // number of collisions - no more than 1/10 of the nu
      {
         return -1; // error writing to index array
      }
      p = (p + attempt) % n;
   }
   array[p] = index;
   return p;
}
```

Si la celda en la posición *p* del array de índices no está ocupada (igual a -1), escribimos inmediatamente la dirección de ubicación de la estructura de calendario en el elemento *[p]*. Si la celda ya está ocupada, intentamos seleccionar la siguiente utilizando la fórmula *p = (p + attempt) % n*, donde *attempt* es un contador de intentos (esta es nuestra versión camuflada de la lista de elementos con un hash coincidente). Si el número de intentos fallidos alcanza una décima parte de los datos originales, la indexación fallará, pero esto es prácticamente imposible con nuestro tamaño de array de índices sobredimensionado y la naturaleza conocida de los datos con hash (identificadores únicos).

Como resultado del hashing del array de estructuras, obtenemos un array de índices relleno (hay espacios libres en él, pero así es como está pensado), a través del cual podemos encontrar la ubicación de la estructura correspondiente en el array de estructuras por el identificador del elemento calendario. Para ello se utiliza el método *find*, cuyo significado es opuesto al de *place*.

```
template<typename S>
int find(const ulong id, const int &array[], const S &structs[])
{
   const int n = ArraySize(array);
   if(!n) return false;
   int p = (int)((MathSwap(id) ^ 0xEFCDAB8967452301) % n); // hash function
   int attempt = 0;
   while(structs[array[p]].id != id)
   {
      if(++attempt > n / 10)
      {
         return -1; // error extracting from index array
      }
      p = (p + attempt) % n;
   }
   return array[p];
}
```

Veamos cómo se utiliza en la práctica. Las funciones estándar del calendario incluyen *CalendarCountryById* y *CalendarEventById*. Cuando necesite probar un programa MQL en el probador, no podrá acceder directamente a ellos, pero podrá cargar la caché del calendario en el objeto *CalendarCache* y, por lo tanto, deberá tener métodos similares.

```
bool calendarCountryById(ulong country_id, MqlCalendarCountry &cnt)
{
   const int index = find(country_id, id4country, countries);
   if(index == -1) return false;

   cnt = countries[index];
   return true;
}

bool calendarEventById(ulong event_id, MqlCalendarEvent &event)
{
   const int index = find(event_id, id4event, events);
   if(index == -1) return false;

   event = events[index];
   return true;
}
```

Utilizan el método *find* y los arrays de índices *id4country* y *id4event*.

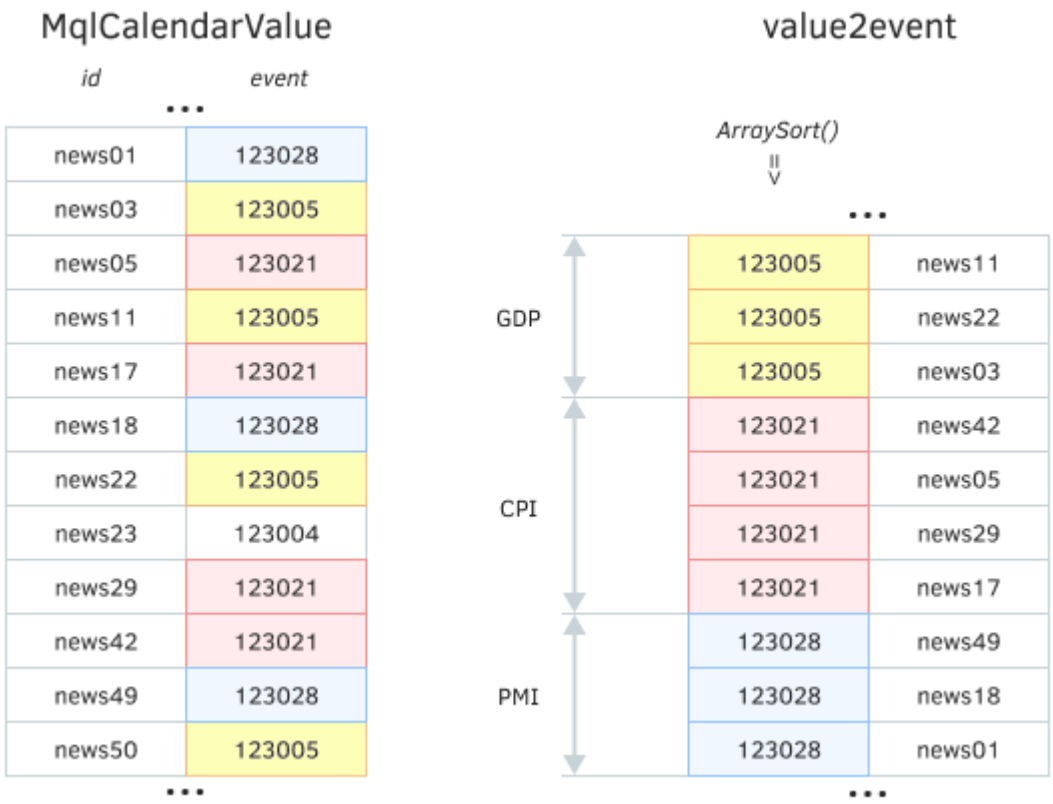
Pero éstas no son las características más deseadas del calendario. Mucho más a menudo, un programa MQL con una estrategia de noticias necesita las funciones *CalendarValueHistory*, *CalendarValueHistoryByEvent*, *CalendarValueLast* o *CalendarValueLastByEvent*. Permiten acceder rápidamente a las entradas de la agenda por hora, país o divisa.

Por lo tanto, la clase *CalendarCache* debería proporcionar métodos similares. Aquí utilizaremos el segundo método de «indexación»: mediante una búsqueda binaria en un array ordenado.

Para implementar los métodos anteriores, vamos a añadir otros 4 arrays bidimensionales a la clase para establecer una correspondencia entre las noticias y el tipo de evento, las noticias y el país, las noticias y la divisa, así como las noticias y la hora de su publicación.

```
ulong value2event[][2];    // [0] - event_id, [1] - value_id
ulong value2country[][2];  // [0] - country_id, [1] - value_id
ulong value2currency[][2]; // [0] - currency ushort[4]<->long, [1] - value_id
ulong value2time[][2];     // [0] - time, [1] - value_id
```

En el primer elemento de cada fila, es decir, bajo los índices *[i][0]* se registrará un ID de evento, país, divisa u hora, respectivamente. En el segundo elemento de la serie, bajo los índices *[i][1]* se colocarán los ID de noticias concretas. Después de llenar todos los arrays una vez, se ordenan utilizando *ArraySort* en los valores *[i][0]*. A continuación, podemos buscar por ID, por ejemplo, en *event_id*, todas las noticias de este tipo en el array *value2event*: la función *ArrayBsearch* devolverá el número del primer elemento coincidente, seguido de otros con el mismo *event_id* hasta que se encuentre un identificador distinto. El orden en la segunda «columna» no está definido (puede ser cualquiera).



Búsqueda rápida de estructuras relacionadas basada en la clasificación

Esta operación de unión mutua de estructuras de distintos tipos se lleva a cabo en el método *bind*. El tamaño de cada array «vinculante» es el mismo que el del array de noticias. Al recorrer todas las noticias en un bucle, utilizamos arrays de índices ya preparadas y el método *find* para un direccionamiento rápido.

```
bool bind()
{
   Print("Binding calendar tables...");
   const int n = ArraySize(values);
   ArrayResize(value2event, n);
   ArrayResize(value2country, n);
   ArrayResize(value2currency, n);
   ArrayResize(value2time, n);
   for(int i = 0; i < n; ++i)
   {
      value2event[i][0] = values[i].event_id;
      value2event[i][1] = values[i].id;

      const int e = find(values[i].event_id, id4event, events);
      if(e == -1) return false;

      value2country[i][0] = events[e].country_id;
      value2country[i][1] = values[i].id;

      const int c = find(events[e].country_id, id4country, countries);
      if(c == -1) return false;

      value2currency[i][0] = currencyId(countries[c].currency);
      value2currency[i][1] = values[i].id;

      value2time[i][0] = values[i].time;
      value2time[i][1] = values[i].id;
   }
   ArraySort(value2event);
   ArraySort(value2country);
   ArraySort(value2currency);
   ArraySort(value2time);
   return true;
}
```

En el caso de las divisas, se toma como identificador un número especial obtenido de la cadena mediante la función *currencyId*.

```
static ulong currencyId(const string s)
{
   union CRNC4
   {
      ushort word[4];
      ulong ul;
   } v;
   StringToShortArray(s, v.word);
   return v.ul;
}
```

Ahora podemos presentar finalmente el constructor completo de la clase *CalendarCache*.

```
CalendarCache(const string _context = NULL,
   const datetime _from = 0, const datetime _to = 0):
   context(_context), from(_from), to(_to), t(0), eventId(0)
{
   if(from > to) // label that context is a filename
   {
      load(_context);
   }
   else
   {
      if(!update() || !hash() || !bind())
      {
         t = 0;
      }
   }
}
```

Cuando se ejecuta en un gráfico en línea, el objeto creado con los parámetros predeterminados recopilará toda la información del calendario (*update*), la indexará (*hash*) y vinculará las tablas (*bind*). Si algo va mal en cualquiera de las etapas, el signo de error será 0 en la variable *t*. Si tiene éxito, el valor de la función *TimeTradeServer* permanecerá allí (recuerde, se coloca dentro de *update*). Este objeto listo para usar puede exportarse a un archivo utilizando el método *save* descrito anteriormente.

Cuando se lanza en el probador, el objeto debe crearse con una combinación especial de parámetros *from* y *to* (*from* > *to*); en este caso, el programa considerará la cadena *context* como un nombre de archivo y cargará el estado del calendario desde él. La forma más fácil de hacerlo es la siguiente:

```
CalendarCache calca("filename.cal", true);
```

Dentro del método *load* también llamaremos a *hash* y *bind* para poner el objeto en estado de funcionamiento.

```
bool load(const string filename)
{
   ... // reading the file was shown earlier
   const bool result = hash() && bind();
   if(!result) t = 0;
   return result;
}
```

Utilizando la función *CalendarValueLast* como ejemplo, mostramos una implementación equivalente del método *calendarValueLast* (con exactamente el mismo prototipo). La caché utilizará la hora actual del «servidor» como identificador de cambios, a falta de una API de software abierta para leer la tabla de cambios del calendario en línea. Hipotéticamente, podríamos utilizar la información sobre los identificadores de cambio guardados por el servicio *CalendarChangeSaver.mq5*, pero este enfoque requiere una recopilación de estadísticas a largo plazo antes de poder empezar la simulación. Por lo tanto, el tiempo de «servidor» generado por el probador se acepta como un sustituto bastante adecuado.

Cuando el programa MQL solicita cambios por primera vez con un identificador nulo, simplemente devolvemos el valor de *TimeTradeServer*.

```
int calendarValueLast(ulong &change, MqlCalendarValue &result[],
   const string code = NULL, const string currency = NULL)
{
   if(!change)
   {
      change = TimeTradeServer();
      return 0;
   }
   ...
```

Si el identificador de cambio ya es distinto de cero, continuamos con la rama principal del algoritmo.

En función del contenido de los parámetros *code* y *currency*, encontramos los identificadores del país y la divisa. Por defecto, es 0, lo que significa que busca todos los cambios.

```
   ulong country_id = 0;
   ulong currency_id = currency != NULL ? currencyId(currency) : 0;

   if(code != NULL)
   {
      for(int i = 0; i < ArraySize(countries); ++i)
      {
         if(countries[i].code == code)
         {
            country_id = countries[i].id;
            break;
         }
      }
   }
   ...
```

Más adelante, utilizando el recuento de tiempo transmitido *change* como inicio de la búsqueda, encontramos todas las noticias en *value2time* hasta el nuevo valor actual *TimeTradeServer*. Dentro del bucle utilizamos el método *find* para buscar el índice de la estructura *MqlCalendarValue* correspondiente en el array *values* y, si es necesario, comparar el país y la divisa del tipo de evento asociado a los deseados. Todas las noticias que cumplen los criterios se escriben en el array de salida *result*.

```
      const ulong past = change;
      const int index = ArrayBsearch(value2time, past);
      if(index < 0 || index >= ArrayRange(value2time, 0)) return 0;

      int i = index;
      while(value2time[i][0] <= (ulong)past && i < ArrayRange(value2time, 0)) ++i;

      if(i >= ArrayRange(value2time, 0)) return 0;

      for(int j = i; j < ArrayRange(value2time, 0)
         && value2time[j][0] <= (ulong)TimeTradeServer(); ++j)
      {
         const int p = find(value2time[j][1], id4value, values);
         if(p != -1)
         {
            change = TimeTradeServer();
            if(country_id != 0 || currency_id != 0)
            {
               const int q = find(values[p].event_id, id4event, events);
               if(country_id != 0 && country_id != events[q].country_id) continue;
               if(currency_id != 0)
               {
                  const int m = find(events[q].country_id, id4country, countries);
                  if(countries[m].currency != currency) continue;
               }
            }

            PUSH(result, values[p]);
         }
      }

      return ArraySize(result);
   }
```

Los métodos *calendarValueHistory*, *calendarValueHistoryByEvent* y *calendarValueLastByEvent* se implementan de acuerdo con un principio similar (este último en realidad delega todo el trabajo en el método *calendarValueLast* comentado anteriormente). El código fuente completo se encuentra en el archivo adjunto *CalendarCache.mqh*.

Basándose en la clase de caché, es lógico crear una clase derivada *CalendarFilter*, que, al procesar las peticiones, accedería a la caché en lugar de al calendario.

La solución final se encuentra en el archivo *CalendarFilterCached.mqh*. Debido a que la API de caché se diseñó sobre la base de la API estándar, la integración se reduce únicamente a reenviar las llamadas de filtro al objeto de caché (puntero automático *cache*).

```
class CalendarFilterCached: public CalendarFilter
{
protected:
   AutoPtr<CalendarCache> cache;

   virtual bool calendarCountryById(ulong country_id, MqlCalendarCountry &cnt) overri
   {
      return cache[].calendarCountryById(country_id, cnt);
   }

   virtual bool calendarEventById(ulong event_id, MqlCalendarEvent &event) override
   {
      return cache[].calendarEventById(event_id, event);
   }

   virtual int calendarValueHistoryByEvent(ulong event_id, MqlCalendarValue &temp[],
      datetime _from, datetime _to = 0) override
   {
      return cache[].calendarValueHistoryByEvent(event_id, temp, _from, _to);
   }

   virtual int calendarValueHistory(MqlCalendarValue &temp[],
      datetime _from, datetime _to = 0,
      const string _code = NULL, const string _coin = NULL) override
   {
      return cache[].calendarValueHistory(temp, _from, _to, _code, _coin);
   }

   virtual int calendarValueLast(ulong &_change, MqlCalendarValue &result[],
      const string _code = NULL, const string _coin = NULL) override
   {
      return cache[].calendarValueLast(_change, result, _code, _coin);
   }

   virtual int calendarValueLastByEvent(ulong event_id, ulong &_change,
      MqlCalendarValue &result[]) override
   {
      return cache[].calendarValueLastByEvent(event_id, _change, result);
   }

public:
   CalendarFilterCached(CalendarCache *_cache): cache(_cache),
      CalendarFilter(_cache.getContext(), _cache.getFrom(), _cache.getTo())
   {
   }

   virtual bool isLoaded() const override
   {
 // readiness is determined by the cache
      return cache[].isLoaded();
   }
```

```
};
```

Para probar el calendario en el probador, vamos a crear una nueva versión del indicador *CalendarMonitor.mq5 — CalendarMonitorCached.mq5*.

Las principales diferencias son las siguientes.

Suponemos que se creará o ya se ha creado algún archivo de caché con el nombre «xyz.cal» (en la carpeta *MQL5/Files*) y por lo tanto lo conectaremos al programa MQL con la directiva *tester_file*.

```
#property tester_file "xyz.cal"
```

Esta directiva garantiza la transferencia de la caché a cualquier agente, incluidos los distribuidos (que, sin embargo, es más relevante para los Asesores Expertos, en lugar de un indicador). Se puede crear un archivo de caché con este (u otro nombre) utilizando una nueva variable de entrada *CalendarCacheFile*. Si el usuario cambia el nombre por defecto por otro, entonces para trabajar en el probador necesitará corregir la directiva (¡requiere recompilación!) o transferir el archivo a la carpeta compartida de terminales (esta característica está admitida en la clase de caché, pero se queda «tras el telón»); no obstante, dicho archivo ya no está disponible para los agentes remotos.

```
input string CalendarCacheFile = "xyz.cal";
```

El objeto *CalendarFilter* se describe ahora como un puntero automático, porque dependiendo de dónde se ejecute el indicador, puede utilizar la clase original *CalendarFilter* así como la clase derivada *CalendarFilterCached*.

```
AutoPtr<CalendarFilter> fptr;
AutoPtr<CalendarCache> cache;
```

Al principio de *OnInit* hay un nuevo fragmento que se encarga de generar la caché y de leerla.

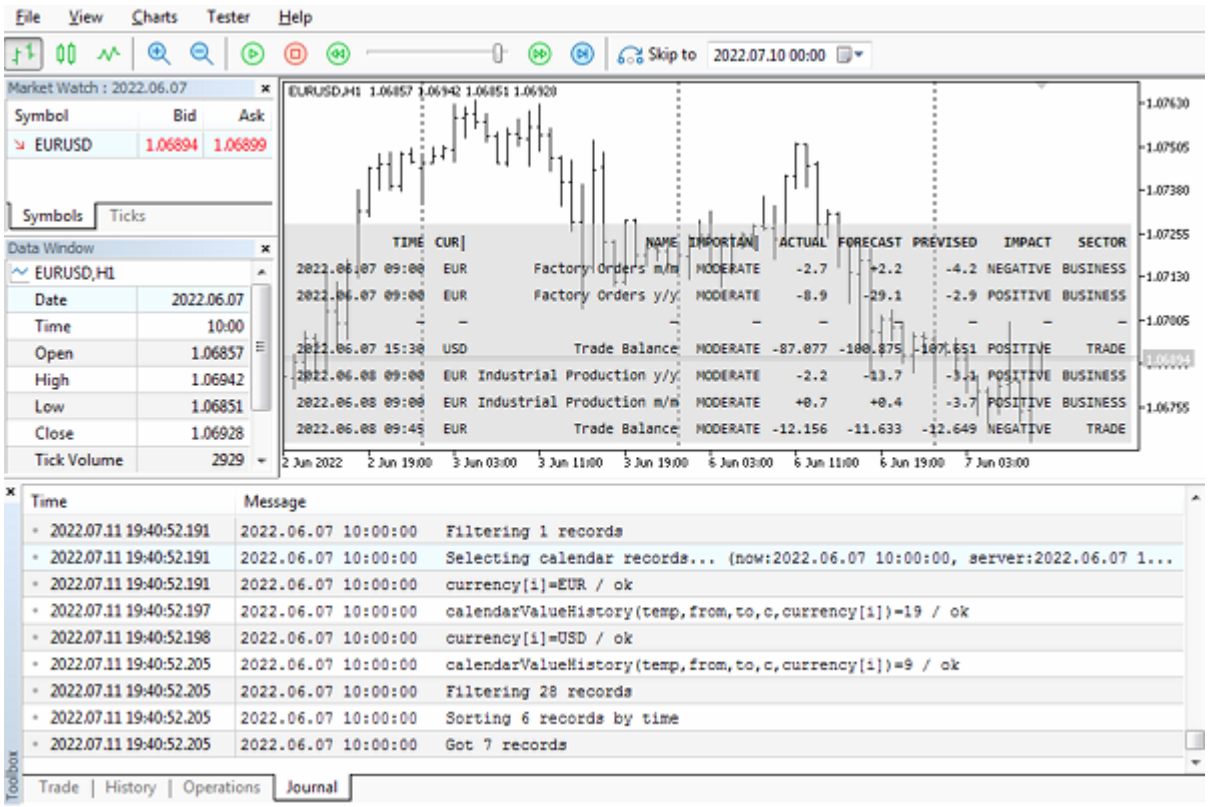
```
int OnInit()
{
   cache = new CalendarCache(CalendarCacheFile, true);
   if(cache[].isLoaded())
   {
      fptr = new CalendarFilterCached(cache[]);
   }
   else
   {
      if(MQLInfoInteger(MQL_TESTER))
      {
         Print("Can't run in the tester without calendar cache file");
         return INIT_FAILED;
      }
      else
      if(StringLen(CalendarCacheFile))
      {
         Alert("Calendar cache not found, trying to create '" + CalendarCacheFile + "
         cache = new CalendarCache();
         if(cache[].save(CalendarCacheFile))
         {
            Alert("File saved. Re-run indicator in online chart or in the tester");
         }
         else
         {
            Alert("Error: ", _LastError);
         }
         ChartIndicatorDelete(0, 0, MQLInfoString(MQL_PROGRAM_NAME));
         return INIT_PARAMETERS_INCORRECT;
      }
      Alert("Currently working in online mode (no cache)");
      fptr = new CalendarFilter(Context);
   }
   CalendarFilter *f = fptr[];
   ... // continued without changes
```

Si se ha leído el archivo de caché, obtendremos el objeto terminado *CalendarCache*, que se pasa al constructor *CalendarFilterCached*. De lo contrario, el programa comprueba si se está ejecutando en el probador o en línea. La ausencia de caché en el probador es un caso fatal. En un gráfico regular, el programa crea un nuevo objeto basado en los datos del calendario integrado y lo guarda en la caché con el nombre especificado. Pero si el nombre del archivo se deja vacío, el indicador funcionará exactamente igual que el original: directamente con el calendario.

Vamos a ejecutar el indicador en el gráfico EURUSD. Se advertirá al usuario de que no se ha encontrado el archivo especificado y se ha intentado guardarlo. Siempre que el calendario esté activado en la configuración del terminal, deberíamos obtener aproximadamente las siguientes líneas en el registro. A continuación encontrará una versión con información de diagnóstico detallada. Los detalles pueden desactivarse comentando la directiva en el código fuente *#define LOGGING*.

```
Loading calendar cache xyz.cal
FileOpen(filename,FILE_READ|FILE_BIN|flags)=-1 / CANNOT_OPEN_FILE(5004)
Alert: Calendar cache not found, trying to create 'xyz.cal'
Reading online calendar base...
CalendarValueHistory(values,from,to,country,currency)=157173 / ok
CalendarEventByCountry(country,events)=1493 / ok
CalendarCountries(countries)=23 / ok
Hashing calendar...
ArraySize(countries)=23 / ok
ArrayResize(id4country,size2prime(c))=53 / ok
Total collisions: 9, worse:3, average: 2.25 in 4
ArraySize(events)=1493 / ok
ArrayResize(id4event,size2prime(e))=3079 / ok
Total collisions: 495, worse:7, average: 1.43478 in 345
ArraySize(values)=157173 / ok
ArrayResize(id4value,size2prime(v))=393241 / ok
Total collisions: 3511, worse:1, average: 1.0 in 3511
Binding calendar tables...
FileOpen(filename,FILE_WRITE|FILE_BIN|flags)=1 / ok
Alert: File saved. Re-run indicator in online chart or in the tester
```

Ahora podemos elegir el indicador *CalendarMonitorCached.mq5* en el probador y ver en dinámica, basándonos en el historial, cómo cambia la tabla de noticias.



Indicador de noticias con caché de calendario en el probador

La presencia de la caché de calendario le permite probar estrategias de trading en las noticias. Lo haremos en la próxima sección.

7.3.12 Trading con calendario económico

Existen muchas estrategias de trading de noticias: con órdenes de mercado o pendientes, con análisis de indicadores financieros (la dirección del movimiento de los precios) y sin él (captura de la volatilidad). Además, es útil insertar un filtro antinoticias en muchos otros sistemas de trading. Es difícil optimizar y depurar todos estos programas, ya que el calendario de MQL5 no está disponible en el probador. Sin embargo, con la ayuda de la caché desarrollada en la sección anterior, podemos corregir la situación.

Intentemos crear un Asesor Experto que entre en el mercado cuando se publiquen noticias, de acuerdo con la evaluación de su impacto en el precio. El archivo de caché «xyz.cal» acaba de crearse con el indicador *CalendarMonitorCached.mq5*.

> Recordemos que la imagen del calendario en la caché corresponde siempre al momento en que se guarda y requiere precaución a la hora de leerlo: para los eventos posteriores, los indicadores reales son desconocidos, y los eventos más lejanos pueden no existir en absoluto. Debe actualizar regularmente el archivo de caché del calendario antes de la siguiente optimización o simulación.
>
> Si es necesario, tenga también en cuenta los ajustes de la hora DST durante el año: si el modo DST de los eventos es diferente del DST en el momento en que se guardó el archivo de calendario, tendrá que retrasar o adelantar la hora en 1 hora. Puede evitar estas dificultades eligiendo un bróker sin DST o construyendo una estrategia en marcos temporales superiores a H1.

El Asesor Experto *CalendarTrading.mq5* solo negociará los eventos de noticias que:

- hagan referencia al símbolo de trabajo del gráfico;
- tengan el tipo de indicador financiero (es decir, cuantitativo);
- sean de gran importancia;
- acaben de recibir el valor actual del indicador.

Esto último es importante porque para los indicadores que tienen valores previstos y reales, el sistema establece el valor del campo *impact_type* en consecuencia: servirá como señal de trading (indicará la dirección de entrada en el mercado).

> La hora exacta de la publicación de la noticia, por regla general, no coincide con la hora prevista introducida en el campo *MqlCalendarValue::time*. El calendario no registra esta hora, y no está disponible en la caché. En este sentido, la precisión de la simulación de estrategias de noticias puede verse afectada. Si desea acercar el análisis y la toma de decisiones a un proceso en línea, acumule estadísticas de publicación de noticias mediante un servicio como *CalendarChangeSaver.mq5* e incrustarlo en la caché.

De manera predeterminada, el trading se realiza con un lote mínimo, con niveles de Take Profit y Stop Loss fijados a una distancia determinada en puntos. Todo esto se refleja en los parámetros de entrada.

```
input double Volume;                // Volume (0 = minimal lot)
input int Distance2SLTP = 500;      // Distance to SL/TP in points (0 = no)
input uint MultiplePositions = 25;
```

Para las cuentas de cobertura, permitimos la existencia simultánea de varias posiciones, el valor predeterminado es 25. Este es el entorno de simulación recomendado porque le permite evaluar de forma independiente la rentabilidad del trading paralelo en noticias de distintos tipos (cada posición se crea de forma independiente y no conlleva el cierre de posiciones en otras noticias). Por otra parte, mantener una sola posición nivela automáticamente las señales contradictorias de noticias diferentes.

Opcionalmente, el Asesor Experto admite filtros para el identificador del tipo de noticia y texto para la búsqueda por título.

```
sinput ulong EventID;
sinput string Text;
```

Esto puede ser útil para futuras investigaciones sobre noticias concretas.

A nivel global, los punteros de los objetos se describen mediante el tratamiento analítico de las noticias y el seguimiento de la posición.

```
AutoPtr<CalendarFilter> fptr;
AutoPtr<CalendarCache> cache;
AutoPtr<TrailingStop> trailing[];
```

El modo de funcionamiento y el par de divisas del símbolo de trabajo actual se almacenan en las variables correspondientes. Para simplificar el ejemplo, se supone que se utiliza en Forex (en otros mercados, el trading se realizará en una sola divisa: la divisa de cotización del ticker).

```
const bool Hedging =
   AccountInfoInteger(ACCOUNT_MARGIN_MODE) == ACCOUNT_MARGIN_MODE_RETAIL_HEDGING;
const string Base = SymbolInfoString(_Symbol, SYMBOL_CURRENCY_BASE);
const string Profit = SymbolInfoString(_Symbol, SYMBOL_CURRENCY_PROFIT);
```

En el manejador *OnInit*, cargamos la caché del calendario y configuramos los filtros como se ha descrito anteriormente. La ausencia de caché está permitida en el gráfico en línea: entonces el Asesor Experto trabaja en modo de combate, directamente con el calendario. En el probador, la ausencia de un archivo de caché impedirá que se inicie el Asesor Experto.

```
int OnInit()
{
   cache = new CalendarCache("xyz.cal", true);
   if(cache[].isLoaded())
   {
      fptr = new CalendarFilterCached(cache[]);
   }
   else
   {
      if(!MQLInfoInteger(MQL_TESTER))
      {
         Print("Calendar cache file not found, fall back to online mode");
         fptr = new CalendarFilter();
      }
      else
      {
         Print("Can't proceed in the tester without calendar cache file");
         return INIT_FAILED;
      }
   }
   CalendarFilter *f = fptr[];

   if(!f.isLoaded()) return INIT_FAILED;

   // if a specific type of event is set, we look only at it
   if(EventID > 0) f.let(EventID);
   else
   {
      // otherwise follow the news on the currencies of the current symbol
      f.let(Base);
      if(Base != Profit)
      {
         f.let(Profit);
      }

      // financial indicators, high importance, actual value
      f.let(CALENDAR_TYPE_INDICATOR);
      f.let(LONG_MIN, CALENDAR_PROPERTY_RECORD_FORECAST, NOT_EQUAL);
      f.let(CALENDAR_IMPORTANCE_HIGH);

      if(StringLen(Text)) f.let(Text);
   }

   f.describe();

   if(Distance2SLTP)
   {
      ArrayResize(trailing, Hedging && MultiplePositions ? MultiplePositions : 1);
   }
   // check the news filter and start trading on it by a second timer
   EventSetTimer(1);
```

```
   return INIT_SUCCEEDED;
}
```

En el manejador *OnTimer* solicitamos cambios en las noticias según los filtros configurados.

```
void OnTimer()
{
   CalendarFilter *f = fptr[];
   MqlCalendarValue records[];

   f.let(TimeTradeServer() - SCOPE_DAY, TimeTradeServer() + SCOPE_DAY);

   if(f.update(records)) // find changes that undergo filtering
   {
      // output properties of changed news to the log
      static const ENUM_CALENDAR_PROPERTY props[] =
      {
         CALENDAR_PROPERTY_RECORD_TIME,
         CALENDAR_PROPERTY_COUNTRY_CURRENCY,
         CALENDAR_PROPERTY_COUNTRY_CODE,
         CALENDAR_PROPERTY_EVENT_NAME,
         CALENDAR_PROPERTY_EVENT_IMPORTANCE,
         CALENDAR_PROPERTY_RECORD_ACTUAL,
         CALENDAR_PROPERTY_RECORD_FORECAST,
         CALENDAR_PROPERTY_RECORD_PREVISED,
         CALENDAR_PROPERTY_RECORD_IMPACT,
      };
      static const int p = ArraySize(props);
      string result[];
      f.format(records, props, result);
      for(int i = 0; i < ArraySize(result) / p; ++i)
      {
         Print(SubArrayCombine(result, " | ", i * p, p));
      }
      ...
```

Cuando se detectan los cambios adecuados, se registran de la siguiente manera (a continuación se muestra un fragmento del registro real), indicando la hora, la divisa, el país, el nombre, los valores actual y previsto, el valor anterior y la interpretación teórica de la señal:

```
...
Filtering 5 records
2021.02.16 13:00 | EUR | EU | Employment Change q/q | HIGH | +0.3 | -0.4 | +1.0 | POS
2021.02.16 13:00 | EUR | EU | GDP q/q | HIGH | -0.6 | -0.7 | -0.7 | POSITIVE
instant buy 0.01 EURUSD at 1.21638 sl: 1.21138 tp: 1.22138 (1.21637 / 1.21638 / 1.216
deal #64 buy 0.01 EURUSD at 1.21638 done (based on order #64)
...
Filtering 3 records
2021.07.06 12:05 | EUR | DE | ZEW Economic Sentiment Indicator | HIGH | +63.3 | +84.1
instant sell 0.01 EURUSD at 1.18473 sl: 1.18973 tp: 1.17973 (1.18473 / 1.18474 / 1.18
deal #265 sell 0.01 EURUSD at 1.18473 done (based on order #265)
...
```

El impacto potencial de la noticia sobre el precio debe calcularse basándose en la evaluación sobre el terreno *impact_type*. Es importante señalar aquí que tenemos dos divisas: base y cotización. Cuando la noticia tiene un efecto positivo sobre la divisa base, se espera que el tipo suba, y si es negativo, el tipo bajará. Para la divisa cotizada, ocurre lo contrario: un efecto positivo debería aumentar el precio de la segunda divisa del par, lo que significa una disminución del tipo de cambio, mientras que uno negativo conduce a su aumento. Esta dirección normalizada del movimiento del precio se calcula en el siguiente fragmento utilizando la variable *sign*.

```
static const int impacts[3] = {0, +1, -1};
int impact = 0;
string about = "";
ulong lasteventid = 0;
for(int i = 0; i < ArraySize(records); ++i)
{
   int sign = result[i * p + 1] == Profit ? -1 : +1;
   impact += sign * impacts[records[i].impact_type];
   about += StringFormat("%+lld ", sign * (long)records[i].event_id);
   lasteventid = records[i].event_id;
}

if(impact == 0) return; // no signal
...
```

A menudo las noticias aparecen publicadas al mismo tiempo, por lo que es necesario acumular las valoraciones de todas ellas. Esto se hace en la variable *impact*. Dado que nuestra estrategia solo filtra las noticias de mayor importancia, simplemente se suman todas las señales individuales de las mismas, sin coeficientes de ponderación. La variable de cadena *about* se utiliza para preparar el texto del comentario sobre el próximo acuerdo: en él se mencionarán los identificadores de los eventos que han dado lugar a la transacción.

Si el robot se lanza en una cuenta de compensación o se ha alcanzado el número máximo de posiciones permitido, cerraremos una.

```
      PositionFilter positions;
      ulong tickets[];
      positions.let(POSITION_SYMBOL, _Symbol).select(tickets);
      const int n = ArraySize(tickets);

      if(n >= (int)(Hedging ? MultiplePositions : 1))
      {
         MqlTradeRequestSync position;
         position.close(_Symbol) && position.completed();
      }
      ...
```

Ahora puede abrir una nueva posición en una señal. Se establece un identificador de evento como número «mágico», que nos permitirá analizar posteriormente el rendimiento financiero del trading en el contexto de distintos tipos de noticias.

```
      MqlTradeRequestSync request;
      request.magic = lasteventid;
      request.comment = about;
      const double ask = SymbolInfoDouble(_Symbol, SYMBOL_ASK);
      const double bid = SymbolInfoDouble(_Symbol, SYMBOL_BID);
      const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);
      ulong ticket = 0;

      if(impact > 0)
      {
         ticket = request.buy(Lot, 0,
            Distance2SLTP ? ask - point * Distance2SLTP : 0,
            Distance2SLTP ? ask + point * Distance2SLTP : 0);
      }
      else if(impact < 0)
      {
         ticket = request.sell(Lot, 0,
            Distance2SLTP ? bid + point * Distance2SLTP : 0,
            Distance2SLTP ? bid - point * Distance2SLTP : 0);
      }

      if(ticket && request.completed() && Distance2SLTP)
      {
         for(int i = 0; i < ArraySize(trailing); ++i)
         {
            if(trailing[i][] == NULL) // looking for a free slot for the position tra
            {
               trailing[i] = new TrailingStop(ticket, Distance2SLTP, Distance2SLTP /
               break;
            }
         }
      }
   }
}
```

Movemos los Stop Loss de todas las posiciones a la llegada de los ticks.

```
void OnTick()
{
   for(int i = 0; i < ArraySize(trailing); ++i)
   {
      if(trailing[i][])
      {
         if(!trailing[i][].trail()) // position was closed
         {
            trailing[i] = NULL; // release object and slot
         }
      }
   }
}
```

Ahora viene el punto más interesante. Gracias al probador es posible analizar el éxito de la estrategia de noticias no solo en general, sino también desglosado por noticias concretas. El bloque correspondiente se implementa en nuestro manejador *OnTester*. La recogida de datos se realiza mediante el filtro de transacciones. Tras recibir de él el array de tuplas *trades*, que informa sobre el beneficio, el swap, la comisión y el número mágico de cada operación, acumulamos los resultados en tres objetos de *MapArray*: calculan por separado los beneficios, las pérdidas y el número de operaciones para cada número de *magic*.

```
double OnTester()
{
   Print("Trade profits by calendar events:");
   HistorySelect(0, LONG_MAX);
   DealFilter filter;
   int props[] = {DEAL_PROFIT, DEAL_SWAP, DEAL_COMMISSION, DEAL_MAGIC};
   filter.let(DEAL_TYPE, (1 << DEAL_TYPE_BUY) | (1 << DEAL_TYPE_SELL), IS::OR_BITWISE
      .let(DEAL_ENTRY, (1 << DEAL_ENTRY_OUT) | (1 << DEAL_ENTRY_INOUT) | (1 << DEAL_E
      IS::OR_BITWISE);
   Tuple4<double, double, double, ulong> trades[];
   MapArray<ulong,double> profits;
   MapArray<ulong,double> losses;
   MapArray<ulong,int> counts;
   if(filter.select(props, trades))
   {
      for(int i = 0; i < ArraySize(trades); ++i)
      {
         counts.inc((ulong)trades[i]._4);
         const double payout = trades[i]._1 + trades[i]._2 + trades[i]._3;
         if(payout >= 0)
         {
            profits.inc((ulong)trades[i]._4, payout);
            losses.inc((ulong)trades[i]._4, 0);
         }
         else
         {
            profits.inc((ulong)trades[i]._4, 0);
            losses.inc((ulong)trades[i]._4, payout);
         }
      }
      ...
```

Como resultado, obtenemos una tabla que muestra las estadísticas de cada tipo de evento línea por línea: su identificador, país, divisa, beneficio o pérdida total, número de operaciones (número de noticias), factor de beneficio y nombre del evento.

```
      for(int i = 0; i < profits.getSize(); ++i)
      {
         MqlCalendarEvent event;
         MqlCalendarCountry country;
         const ulong keyId = profits.getKey(i);
         if(cache[].calendarEventById(keyId, event)
            && cache[].calendarCountryById(event.country_id, country))
         {
            PrintFormat("%lld %s %s %+.2f [%d] (PF:%.2f) %s",
               event.id, country.code, country.currency,
               profits[keyId] + losses[keyId], counts[keyId],
               profits[keyId] / (losses[keyId] != 0 ? -losses[keyId] : DBL_MIN),
               event.name);
         }
         else
         {
            Print("undefined ", DoubleToString(profits.getValue(i), 2));
         }
      }
   }
   return 0;
}
```

Para probar la idea, vamos a ejecutar el Asesor Experto para el período comprendido entre principios de 2021 (hasta mediados de 2022) en el par EURUSD. A continuación se muestra un fragmento de un registro con una impresión de *OnTester*:

```
Trade profits by calendar events:
840040001 US USD -21.81 [17] (PF:0.53) ISM Manufacturing PMI
840190001 US USD -10.95 [17] (PF:0.69) ADP Nonfarm Employment Change
840200001 US USD -67.09 [78] (PF:0.60) EIA Crude Oil Stocks Change
999030003 EU EUR +14.13 [19] (PF:1.46) Retail Sales m/m
840040003 US USD -17.12 [18] (PF:0.59) ISM Non-Manufacturing PMI
840030016 US USD -1.20 [19] (PF:0.97) Nonfarm Payrolls
840030021 US USD +5.25 [14] (PF:1.21) JOLTS Job Openings
840020010 US USD -14.63 [17] (PF:0.63) Retail Sales m/m
276070001 DE EUR -22.71 [17] (PF:0.47) ZEW Economic Sentiment Indicator
840020005 US USD +10.76 [18] (PF:1.37) Building Permits
840120001 US USD -20.78 [17] (PF:0.49) Existing Home Sales
276030003 DE EUR +18.57 [17] (PF:1.87) Ifo Business Climate
840180002 US USD -3.22 [14] (PF:0.89) CB Consumer Confidence Index
840020014 US USD -8.74 [16] (PF:0.74) Core Durable Goods Orders m/m
840020008 US USD -14.54 [16] (PF:0.63) New Home Sales
250010005 FR EUR +0.66 [10] (PF:1.03) GDP q/q
840010007 US USD +0.99 [15] (PF:1.04) GDP q/q
840120003 US USD +4.53 [18] (PF:1.15) Pending Home Sales m/m
276010008 DE EUR -0.72 [10] (PF:0.97) GDP q/q
999030016 EU EUR -14.04 [14] (PF:0.59) GDP q/q
999030001 EU EUR +1.30 [2] (PF:1.35) Employment Change q/q
```

Los resultados no son muy impresionantes. Aun así, el trading de noticias está lleno de subjetividad. En primer lugar, las evaluaciones teóricas del impacto del valor real de la noticia en el curso pueden diferir

de las expectativas emocionales de la multitud o de los antecedentes informativos adicionales (que quedan fuera del calendario y no son cuantificables). En segundo lugar, ya hemos mencionado la inexactitud del tiempo de publicación del valor real. En tercer lugar, nuestra estrategia se aplica de la forma más simple, sin analizar el movimiento preliminar de los precios (cuando probablemente hubo una filtración y la noticia se «agotó» antes).

En general, esta prueba reveló que los informes de nóminas no agrícolas o PIB favoritos de los operadores no garantizan el éxito, al menos no con nuestra configuración predeterminada. Además, es necesario, de la forma habitual, analizar las transacciones individuales, averiguar qué ha fallado, seleccionar parámetros y mejorar el algoritmo, en concreto, añadir un módulo de ajuste horario para cambiar el horario de verano en la zona horaria del servidor.

Al mismo tiempo, la técnica en sí funciona bien, y podemos simplemente intentar elegir las noticias más exitosas para empezar. Por ejemplo, tomemos la noticia 276030003 (Clima de Negocios de Ifo). Al configurarlo en *EventID*, recibiremos el siguiente informe, coincidente con nuestros indicadores calculados.

Bars	9494	Ticks	2250448	Symbols	1
Initial Deposit	10 000.00				
Total Net Profit	18.57	Balance Drawdow...	7.61	Equity Drawdown ...	8.11
Gross Profit	39.97	Balance Drawdow...	11.80 (0.12%)	Equity Drawdown ...	14.13 (0.14%)
Gross Loss	-21.40	Balance Drawdow...	0.12% (11.80)	Equity Drawdown ...	0.14% (14.13)
Profit Factor	1.87	Expected Payoff	1.09	Margin Level	81581.71%
Recovery Factor	1.31	Sharpe Ratio	2.65	Z-Score	0.52 (39.69%)
AHPR	1.0001 (0.01...	LR Correlation	0.72	OnTester result	0
GHPR	1.0001 (0.01...	LR Standard Error	4.89		
Total Trades	17	Short Trades (won ...	8 (62.50%)	Long Trades (won ...	9 (33.33%)
Total Deals	34	Profit Trades (% of...	8 (47.06%)	Loss Trades (% of t...	9 (52.94%)
	Largest	profit trade	5.00	loss trade	-3.65
	Average	profit trade	5.00	loss trade	-2.38
	Maximum	consecutive wins ($)	3 (14.99)	consecutive losses ...	4 (-11.80)
	Maximal	consecutive profit ...	14.99 (3)	consecutive loss (c...	-11.80 (4)

Informe sobre trading en el probador basado en noticias Clima de Negocios de Ifo

También puede intentar operar en un grupo de eventos similares. En concreto, para responder solo a las noticias sobre el PIB (de distintos países), introduzca la cadena «*PIB*» en la variable *Text*. Los asteriscos se añaden porque, sin ellos, una cadena de 3 caracteres será tratada como divisa por la clase de filtro. Las cadenas de cualquier longitud que no sean 2 (código de país) o 3 (código de divisa) pueden especificarse tal cual, por ejemplo, «granja», «no agrícola», «ventas»: el filtro las buscará como subcadenas de nombres, teniendo en cuenta mayúsculas y minúsculas.

7.4 Criptografía

El trading algorítmico apareció en la intersección de la negociación bursátil y la tecnología de la información, lo que permite, por un lado, conectar cada vez más mercados nuevos para trabajar y, por otro, ampliar la funcionalidad de las plataformas de trading. Una tendencia tecnológica que se ha abierto camino en la mayoría de los ámbitos de actividad, incluidos los arsenales de los operadores, es la criptografía o, más en general, la seguridad de la información.

MQL5 ofrece funciones de cifrado, hash y compresión de datos: *CryptEncode* y *CryptDecode*. Ya los hemos utilizado en algunos de los ejemplos del libro: en el script *EnvSignature.mq5* (Vincular un programa a propiedades en tiempo de ejecución) y el servicio *ServiceAccount.mq5* (Servicios).

En este capítulo hablaremos con más detalle de estas funciones. Sin embargo, antes de pasar directamente a su descripción, repasemos los métodos de transformación de la información: esta dirección de programación es muy extensa, y MQL5 admite solo una parte de los estándares. Esta lista probablemente se ampliará en el futuro, pero por ahora, si no encuentra el método de cifrado necesario en la ayuda, intente encontrar una implementación ya hecha en el sitio web mql5.com (en las secciones de artículos o en la base de datos de código fuente).

7.4.1 Visión general de los métodos de transformación de la información disponibles

La protección de la información puede aplicarse con distintos fines y, por tanto, utilizar distintos métodos. En concreto, puede ser necesario ocultar por completo la esencia de la información a un observador externo o asegurar su transmisión, garantizando al mismo tiempo un estado inalterado, pero la información en sí sigue estando disponible. En el primer caso, hablamos de cifrado, y el segundo se refiere a una huella digital (hash). Así, el cifrado y el hashing se reducen a procesar los datos originales en una nueva representación utilizando, si es necesario, parámetros adicionales.

Tanto el cifrado como el hashing tienen muchas variantes.

La gradación más general divide el cifrado en cifrado de clave pública (asimétrico) y cifrado de clave privada (simétrico).

> Un esquema asimétrico implica la presencia de 2 claves -pública y privada- para cada participante en el intercambio de datos. Los pares de claves pública y privada se generan previamente mediante algoritmos especiales. Cada clave privada sólo es conocida por su propietario. Las claves públicas de todo el mundo son conocidas por todos. Las claves públicas deberán intercambiarse de un modo u otro antes de poder transmitir los datos cifrados. A continuación, el proveedor de datos utiliza su clave privada, que sólo él conoce, junto con una o varias claves públicas de los destinatarios de los datos. Éstos, a su vez, utilizan sus claves privadas y la clave pública del remitente para descifrar.

Un esquema de cifrado simétrico utiliza la misma clave secreta (privada) tanto para cifrar como para descifrar.

MQL5 admite la función de clave privada lista para usar (simétrica). Las herramientas MQL5 integradas no proporcionan una firma electrónica que utilice cifrado asimétrico.

Entre los métodos de cifrado destacan los algoritmos primitivos sin clave. Con su ayuda, los usuarios logran la ocultación condicional de información o la transformación del tipo de información. Entre ellos se encuentran, por ejemplo, ROT13 (sustitución de caracteres con un desplazamiento de sus códigos alfanuméricos por 13, utilizado, en particular, en el registro de Windows) o Base64 (traducción de archivos binarios a texto y viceversa, normalmente en proyectos web). Otra tarea popular de

transformación de datos es la compresión de datos. En cierto sentido, también se puede considerar como una encriptación, ya que los datos se vuelven ilegibles para un humano o un programa de aplicación.

También se ofrecen muchos métodos de hashing, y CRC (Cyclic Redundancy Check) es quizás el más famoso y sencillo. A diferencia del cifrado, que permite recuperar el mensaje original a partir del cifrado, el hashing sólo crea una huella digital (un conjunto característico de bytes) basada en la información original, de tal forma que su estado inalterado mientras se recalcula posteriormente garantiza (con una alta probabilidad) la invariabilidad de la información original. Por supuesto, esto supone que la información está disponible para todos los participantes/usuarios del sistema de software correspondiente. Es imposible recuperar información mediante hash. Por regla general, el tamaño del hash (el número de bytes que contiene) está limitado y estandarizado para cada método, de modo que para una cadena de 80 caracteres de longitud y para un archivo de 1 MB obtendremos un hash del mismo tamaño. Una aplicación de hashing que la mayoría de los usuarios encontrará realmente útil es el hashing de contraseñas de sitios y programas, es decir, estas últimas se almacenan en casa y se verifican durante el inicio de sesión con el hash de la contraseña, y no con la contraseña en su forma original.

Hay que señalar que ya nos encontramos con el término «hash» en el capítulo anterior: utilizamos una función de hashing para indexar estructuras de calendario económico. Ese hash simple tiene un grado de protección muy débil, que, en concreto, se expresa en una alta probabilidad de colisiones (coincidencia de resultados para datos diferentes), que procesamos especialmente en el algoritmo. Es adecuado para problemas de distribución pseudoaleatoria uniforme de datos en un número limitado de «cestas». En cambio, los estándares industriales de hashing se centran específicamente en verificar la integridad de la información y utilizan métodos de cálculo mucho más complejos. Pero la longitud del hash en este caso es de varias decenas de bytes y no de un solo número.

Los métodos de cifrado y hashing de información disponibles para los programas MQL se recopilan en la enumeración ENUM_CRYPT_METHOD.

Constante	Descripción
CRYPT_BASE64	Recodificación Base64
CRYPT_DES	Cifrado DES con una clave de 56 bits (7 bytes)
CRYPT_AES128	Cifrado AES con una clave de 128 bits (16 bytes)
CRYPT_AES256	Cifrado AES con una clave de 256 bits (32 bytes)
CRYPT_HASH_MD5	Cálculo hash MD5 (16 bytes)
CRYPT_HASH_SHA1	Cálculo hash SHA1 (20 bytes)
CRYPT_HASH_SHA256	Cálculo hash SHA256 (32 bytes)
CRYPT_ARCH_ZIP	Compresión mediante el método «deflate»

La enumeración especificada se utiliza en las dos funciones criptográficas de la API - *CryptEncode* (cifrado/hashing) y *CryptDecode* (descifrado). Se analizarán en las secciones siguientes.

Los métodos de cifrado AES y DES requieren, además de los datos, la clave de cifrado: un array de bytes de una longitud predefinida (se indica entre paréntesis en la tabla). Como ya se ha mencionado, la clave debe mantenerse en secreto y sólo debe conocerla el desarrollador del programa o el

propietario de la información. La fuerza criptográfica del cifrado, es decir, la dificultad de seleccionar una clave por parte del ordenador de un atacante, depende directamente del tamaño de la clave: cuanto más grande sea, más fiable será la protección. Por lo tanto, DES se considera obsoleto y ha sido sustituido en el sector financiero por su versión mejorada de Triple DES: implica la aplicación sucesiva de DES tres veces con tres claves diferentes, lo cual es fácil de implementar en MQL5. Existe una versión popular de Triple DES, que realiza el descifrado en la segunda iteración en lugar del cifrado con la clave número 2, es decir, restaura, digamos, los datos a una representación intermedia, deliberadamente incorrecta, antes de la tercera ronda final de DES. Pero también está previsto que Triple DES desaparezca de los estándares del sector después de 2024.

Al mismo tiempo, la fuerza criptográfica debe ser proporcional a la duración del secreto (clave e información). Si se requiere un flujo rápido de mensajes seguros, las claves más cortas que se actualizan regularmente proporcionarán un mejor rendimiento.

De los métodos de hashing, el más moderno es SHA256 (un subconjunto del estándar SHA-2). Los métodos SHA1 y MD5 se consideran inseguros, pero siguen utilizándose ampliamente para ser compatibles con los servicios existentes. Para los métodos de hashing se indica entre paréntesis el tamaño del array de bytes resultante con una huella digital de los datos. No se necesita una clave para el hashing, pero en muchas aplicaciones, la «sal» se adjunta a los datos hasheados: un componente secreto que dificulta a los atacantes la reproducción de los hashes requeridos (por ejemplo, a la hora de adivinar una contraseña).

El elemento CRYPT_ARCH_ZIP permite el archivo ZIP y la transmisión/recepción de solicitudes de datos en Internet (véase *WebRequest*).

> A pesar de que el nombre del método incluye ZIP, los datos comprimidos no son equivalentes a los archivos ZIP habituales, que, además de los contenedores «deflate», siempre contienen metadatos: encabezados especiales, una lista de archivos y sus atributos. En el sitio mql5.com, en los artículos y en la biblioteca de código fuente, puede encontrar implementaciones listas para comprimir archivos en un archivo ZIP y extraerlos desde allí. La compresión y extracción se realizan mediante las funciones *CryptEncode*/*CryptDecode*, y todas las estructuras adicionales necesarias del formato ZIP se describen y rellenan en el código MQL5.

El método Base64 está diseñado para convertir datos binarios en texto y viceversa. Los datos binarios suelen contener muchos caracteres no imprimibles y no son compatibles con las herramientas de edición y entrada, como las variables de entrada en los cuadros de diálogo de propiedades del programa MQL. Base64 puede ser útil, por ejemplo, cuando se trabaja con el popular formato de texto de intercambio de datos de objetos JSON.

Cada 3 bytes originales se codifican en Base64 con 4 caracteres, lo que da lugar a un aumento del tamaño de los datos en un tercio. El libro se acompaña de archivos de prueba con los que experimentaremos en los siguientes ejemplos; en concreto, la página web *MQL5/Files/MQL5Book/clock10.htm* y el archivo utilizado en ella con la imagen del reloj *MQL5/Files/MQL5Book/clock10.png*. Ya en esta fase introductoria puede ver claramente las posibilidades y la diferencia en la representación interna de los datos binarios y el texto Base64, mientras que mantienen una apariencia idéntica.

Página web con imagen binaria incrustada y en formato Base64

La misma imagen con la esfera de un reloj se inserta en la página como archivo externo *clock10.png*, así como su codificación Base64 en la etiqueta *img* (en su atributo *src*: es la «URL de datos»). Directamente en el texto de la propia página web tiene este aspecto (no es necesario envolver una cadena larga Base64 en un ancho de 76 caracteres, pero está permitido por la norma y se hace aquí con fines de publicación):

```
<img src="
iVBORw0KGgoAAAANSUhEUgAAABIAAAASCAQAAAD8x0bcAAAAAXNSR0IArs4c6QAAAAJiS0dEAP+
Hj8y/AAABdUlEQVQoz1WSPUiVYRiGr/f4qaRwyAMOLpVnSGuvBh3SoSnLwclwqQa3CNxaWxwcrS
D/CAQH4ShF4RBnqCWQWiQwAp1C6WBi4B/5Xg7f1+fxgXd4eC5e7ue+nxAFgADABj+Bq1wGIBsRU
ihwwCMa6eE6ga+s0sgLmv5j0ahOOOqxqq98q+qeI86o0ShRfWpFHVC105uq3lOnfa5G0UXn1Ji9
8bw7Ucf9pBbgNX9ZI2TijzjMRBf4zCWeAckbxuijgXV2aadMfd2gl4QVkhXmAegiUGKDlhyRZmC
IxyRFdomUgB1qlDlmtu6vbYqckES+U6SEtNFGjScMcUhz5tAmrTTAiGZupI79cv9crw8sXGM9Dw
WkgwtncQBf6CfoQ8rcpjcfkSPL/OYdSyRwkWHKSDiHBeQ+39hJs9Nb7qmTmdNpTapb9qexpPLu+
NGqqlVXVf1gxcGzgFNswQGn/OOB/9zypXd9n28Z6o/uBxU2kW4GuVJ3dKeW6inhKgZyVwAAAABJ
RU5ErkJggg==" title="Clock 10:00 Base64 built-it" />
```

Pronto reproduciremos esta secuencia de caracteres utilizando la función *CryptEncode*, pero por ahora, basta con señalar que, con una técnica similar, podemos generar informes HTML con gráficos incrustados desde MQL5.

7.4.2 Cifrado, hashing y empaquetado de datos: CryptEncode

La función MQL5 responsable de la encriptación, hashing y compresión de datos es *CryptEncode*. Transforma los datos del array de origen pasado *data* al array de destino *result* mediante el método especificado.

int CryptEncode(ENUM_CRYPT_METHOD method, const uchar &data[], const uchar &key[], uchar &result[])

Los métodos de cifrado también requieren pasar un array de bytes *key* con una clave privada (secreta): su longitud depende del método específico y se especifica en la tabla de métodos ENUM_CRYPT_METHOD de la sección anterior. Si el tamaño del array *key* es mayor, sólo se seguirán utilizando para la clave los primeros bytes de la cantidad requerida.

No se necesita una clave para hashing o compresión, pero hay una advertencia para CRYPT_ARCH_ZIP. El hecho es que la implementación del algoritmo «deflate» integrado en el terminal añade varios bytes a los datos resultantes para controlar la integridad: 2 bytes iniciales contienen la configuración del algoritmo «deflate», y 4 bytes al final contienen la suma de comprobación Adler32. Debido a esta

característica, el contenedor empaquetado resultante difiere del generado por los archivos ZIP para cada elemento individual del archivo (el estándar ZIP almacena CRC32, que tiene un significado similar, en sus encabezados). Por lo tanto, para poder crear y leer archivos ZIP compatibles basados en datos empaquetados por la función *CryptEncode*, MQL5 permite desactivar su propia comprobación de integridad y la generación de bytes extra utilizando un valor especial en el array *key*.

```
uchar key[] = {1, 0, 0, 0};
CryptEncode(CRYPT_ARCH_ZIP, data, key, result);
```

Se puede utilizar cualquier clave con una longitud de al menos 4 bytes. El array *result* obtenido puede enriquecerse con un título según el formato ZIP estándar (esta cuestión queda fuera del ámbito del libro) para crear un archivo accesible a otros programas.

La función devuelve el número de bytes colocados en el array de destino o 0 en caso de error. El código de error, como es habitual, se almacenará en *_LastError*.

Comprobemos el rendimiento de la función utilizando el script *CryptEncode.mq5*. Permite al usuario introducir texto (*Text*) o especificar un archivo (*File*) para su procesamiento. Para utilizar el archivo, debe borrar el campo *Text*.

Puede elegir un *Method* específico o recorrer todos los métodos a la vez para ver y comparar visualmente los distintos resultados. Para un bucle de revisión de este tipo, deje el valor predeterminado _CRYPT_ALL en el parámetro *Method*.

Por cierto, para introducir esta funcionalidad, de nuevo necesitamos extender la enumeración estándar (esta vez ENUM_CRYPT_METHOD), pero como las enumeraciones en MQL5 no pueden ser heredadas como clases, aquí se declara una nueva enumeración ENUM_CRYPT_METHOD_EXT. Una ventaja añadida es que hemos añadido nombres más fáciles de usar para los elementos (en los comentarios, con pistas que se mostrarán en el cuadro de diálogo de configuración).

```
enum ENUM_CRYPT_METHOD_EXT
{
   _CRYPT_ALL = 0xFF,                     // Try All in a Loop
   _CRYPT_DES = CRYPT_DES,                // DES    (key required, 7 bytes)
   _CRYPT_AES128 = CRYPT_AES128,          // AES128 (key required, 16 bytes)
   _CRYPT_AES256 = CRYPT_AES256,          // AES256 (key required, 32 bytes)
   _CRYPT_HASH_MD5 = CRYPT_HASH_MD5,      // MD5
   _CRYPT_HASH_SHA1 = CRYPT_HASH_SHA1,    // SHA1
   _CRYPT_HASH_SHA256 = CRYPT_HASH_SHA256, // SHA256
   _CRYPT_ARCH_ZIP = CRYPT_ARCH_ZIP,      // ZIP
   _CRYPT_BASE64 = CRYPT_BASE64,          // BASE64
};

input string Text = "Let's encrypt this message"; // Text (empty to process File)
input string File = "MQL5Book/clock10.htm"; // File (used only if Text is empty)
input ENUM_CRYPT_METHOD_EXT Method = _CRYPT_ALL;
```

De manera predeterminada, el parámetro *Text* se rellena con un mensaje que se supone cifrado. Puede sustituirlo por el suyo propio. Si borramos *Text*, el programa procesará el archivo. Al menos uno de los parámetros (*Text* o *File*) debe contener información.

Dado que el cifrado requiere una clave, las otras dos opciones permiten introducirla directamente como texto (aunque la clave no tiene por qué ser texto y puede contener cualquier dato binario, pero no se admiten en las entradas) o generar la longitud deseada, según el método de cifrado.

```
enum DUMMY_KEY_LENGTH
{
   DUMMY_KEY_0 = 0,   // 0 bytes (no key)
   DUMMY_KEY_7 = 7,   // 7 bytes (sufficient for DES)
   DUMMY_KEY_16 = 16, // 16 bytes (sufficient for AES128)
   DUMMY_KEY_32 = 32, // 32 bytes (sufficient for AES256)
   DUMMY_KEY_CUSTOM,  // use CustomKey
};

input DUMMY_KEY_LENGTH GenerateKey = DUMMY_KEY_CUSTOM; // GenerateKey (length, or frc
input string CustomKey = "My top secret key is very strong";
```

Por último, existe una opción *DisableCRCinZIP* para activar el modo de compatibilidad ZIP, que sólo afecta al método CRYPT_ARCH_ZIP.

```
input bool DisableCRCinZIP = false;
```

Para simplificar la comprobación de si el método requiere una clave de cifrado o si se calcula un hash (una conversión unidireccional irreversible), se definen 2 macros:

```
#defineKEY_REQUIRED(C) ((C) ==CRYPT_DES|| (C) ==CRYPT_AES128|| (C) ==CRYPT_AES256)
#defineIS_HASH(C) ((C) ==CRYPT_HASH_MD5|| (C) ==CRYPT_HASH_SHA1|| (C) ==CRYPT_HASH_SH
```

El comienzo de *OnStart* contiene una descripción de las variables y arrays necesarios.

```
void OnStart()
{
   ENUM_CRYPT_METHOD method = 0;
   int methods[];           // here we will collect all the elements of ENUM_CRYPT_ME
   uchar key[] = {};        // empty by default: suitable for hashing, zip, base64
   uchar zip[], opt[] = {1, 0, 0, 0}; // "options" for zip
   uchar data[], result[];  // initial data and result
```

Según la configuración de *GenerateKey* obtenemos la clave del campo *CustomKey* o simplemente rellenamos el array *key* con valores enteros monótonamente crecientes. En realidad, la clave debe ser un bloque de valores secreto, no trivial y elegido arbitrariamente.

```
   if(GenerateKey == DUMMY_KEY_CUSTOM)
   {
      if(StringLen(CustomKey))
      {
         PRTF(CustomKey);
         StringToCharArray(CustomKey, key, 0, -1, CP_UTF8);
         ArrayResize(key, ArraySize(key) - 1);
      }
   }
   else if(GenerateKey != DUMMY_KEY_0)
   {
      ArrayResize(key, GenerateKey);
      for(int i = 0; i < GenerateKey; ++i) key[i] = (uchar)i;
   }
```

Aquí y más abajo, observe el uso de *ArrayResize* después de *StringToCharArray*. Asegúrese de reducir el array en 1 elemento, porque en caso de que la función *StringToCharArray* convierta la cadena en un array de bytes, incluyendo el terminal 0, esto puede romper la ejecución esperada del programa. En

concreto, en este caso, tendremos un byte cero extra en la clave secreta, y si un programa con un artefacto similar no se utiliza en el lado receptor, entonces no será capaz de descifrar el mensaje. Estos ceros adicionales también pueden afectar a la compatibilidad con los protocolos de intercambio de datos (si se realiza una u otra integración de un programa MQL con el «mundo exterior»).

A continuación, registramos una representación en bruto de la clave resultante en formato hexadecimal: esto se hace mediante la función *ByteArrayPrint* que se utilizó en la sección Escritura y lectura de archivos en modo simplificado.

```
if(ArraySize(key))
{
   Print("Key (bytes):");
   ByteArrayPrint(key);
}
else
{
   Print("Key is not provided");
}
```

En función de la disponibilidad de *Text* o *File*, rellenamos el array *data* con caracteres de texto o con el contenido del archivo.

```
if(StringLen(Text))
{
   PRTF(Text);
   PRTF(StringToCharArray(Text, data, 0, -1, CP_UTF8));
   ArrayResize(data, ArraySize(data) - 1);
}
else if(StringLen(File))
{
   PRTF(File);
   if(PRTF(FileLoad(File, data)) <= 0)
   {
      return; // error
   }
}
```

Por último, hacemos un bucle a través de todos los métodos o realizamos la transformación una vez con un método específico.

```
const int n = (Method == _CRYPT_ALL) ?
   EnumToArray(method, methods, 0, UCHAR_MAX) : 1;
ResetLastError();
for(int i = 0; i < n; ++i)
{
   method = (ENUM_CRYPT_METHOD)((Method == _CRYPT_ALL) ? methods[i] : Method);
   Print("- ", i, " ", EnumToString(method), ", key required: ",
      KEY_REQUIRED(method));

   if(method == CRYPT_ARCH_ZIP)
   {
      if(DisableCRCinZIP)
      {
         ArrayCopy(zip, opt); // array with additional option dynamic for ArraySwa
      }
      ArraySwap(key, zip); // change key to empty or option
   }

   if(PRTF(CryptEncode(method, data, key, result)))
   {
      if(StringLen(Text))
      {
         // code page Latin (Western) to unify the display for all users
         Print(CharArrayToString(result, 0, WHOLE_ARRAY, 1252));
         ByteArrayPrint(result);
         if(method != CRYPT_BASE64)
         {
            const uchar dummy[] = {};
            uchar readable[];
            if(PRTF(CryptEncode(CRYPT_BASE64, result, dummy, readable)))
            {
               PrintFormat("Try to decode this with CryptDecode.mq5 (%s):",
                  EnumToString(method));
               // to receive encoded data back for decoding
               // via string input, apply Base64 over binary result
               Print("base64:'" + CharArrayToString(readable, 0, WHOLE_ARRAY, 1252
            }
         }
      }
      else
      {
         string parts[];
         const string filename = File + "." +
            parts[StringSplit(EnumToString(method), '_', parts) - 1];
         if(PRTF(FileSave(filename, result)))
         {
            Print("File saved: ", filename);
            if(IS_HASH(method))
            {
               ByteArrayPrint(result, 1000, "");
            }
```

```
                }
            }
        }
    }
}
```

Cuando convertimos texto, registramos el resultado, pero como casi siempre son datos binarios, con la excepción del método CRYPT_BASE64, su visualización será un completo galimatías (a decir verdad, los datos binarios no deberían registrarse, pero lo hacemos por claridad). Los símbolos no imprimibles y los símbolos con códigos superiores a 128 se muestran de forma diferente en ordenadores con idiomas distintos. Por ello, con el fin de unificar la visualización de ejemplos para todos los lectores, al formar una línea en *CharArrayToString*, utilizamos una página de código explícita (1252, lenguas de Europa Occidental). Es cierto que los tipos de letra utilizados al publicar un libro contribuirán con toda probabilidad a la forma en que se muestren determinados caracteres (el conjunto de glifos de los tipos de letra puede ser limitado).

> Es importante tener en cuenta que controlamos la elección de la página de código sólo en el método de visualización, y los bytes del array *result* no cambian por ello (por supuesto, la cadena obtenida de este modo no debe enviarse a ningún otro sitio; sólo es necesario para la visualización utilizar los bytes del propio resultado para el intercambio de datos).

Sin embargo, sigue siendo deseable que ofrezcamos al usuario alguna posibilidad de guardar el resultado cifrado para descodificarlo más tarde. La forma más sencilla es volver a transformar los datos binarios utilizando el método CRYPT_BASE64.

En el caso de la codificación de archivos, simplemente guardamos el resultado en un nuevo archivo con un nombre en el que se añade al original la extensión de la última palabra del nombre del método. Por ejemplo, aplicando CRYPT_HASH_MD5 al archivo *Example.txt*, obtendremos el archivo de salida *Example.txt.MD5* que contiene el hash MD5 del archivo fuente. Tenga en cuenta que, para el método CRYPT_ARCH_ZIP, obtendremos un archivo con extensión ZIP, pero no es un archivo ZIP estándar (debido a la falta de encabezados con meta información y tabla de contenidos).

Vamos a ejecutar el script con la configuración predeterminada: corresponden a comprobar en el bucle todos los métodos para el texto «Let's encrypt this message» («vamos a cifrar este mensaje»).

```
CustomKey=My top secret key is very strong / ok
Key (bytes):
[00] 4D | 79 | 20 | 74 | 6F | 70 | 20 | 73 | 65 | 63 | 72 | 65 | 74 | 20 | 6B | 65 |
[16] 79 | 20 | 69 | 73 | 20 | 76 | 65 | 72 | 79 | 20 | 73 | 74 | 72 | 6F | 6E | 67 |
Text=Let's encrypt this message / ok
StringToCharArray(Text,data,0,-1,CP_UTF8)=26 / ok
- 0 CRYPT_BASE64, key required: false
CryptEncode(method,data,key,result)=36 / ok
TGV0J3MgZW5jcnlwdCB0aGlzIG1lc3NhZ2U=
[00] 54 | 47 | 56 | 30 | 4A | 33 | 4D | 67 | 5A | 57 | 35 | 6A | 63 | 6E | 6C | 77 |
[16] 64 | 43 | 42 | 30 | 61 | 47 | 6C | 7A | 49 | 47 | 31 | 6C | 63 | 33 | 4E | 68 |
[32] 5A | 32 | 55 | 3D |
- 1 CRYPT_AES128, key required: true
CryptEncode(method,data,key,result)=32 / ok
¯T★ Ë[3hß Ã/-C }¬ŠÑØN¨®Ê† ‡Ñ
[00] 01 | 0B | AF | 54 | 2A | 12 | CB | 5B | 33 | 68 | DF | 0E | C3 | 2F | 2D | 43 |
[16] 19 | 7D | AC | 8A | D1 | 8F | D8 | 4E | A8 | AE | CA | 81 | 86 | 06 | 87 | D1 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=44 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_AES128):
base64:'AQuvVCoSy1szaN8Owy8tQxl9rIrRj9hOqK7KgYYGh9E='
- 2 CRYPT_AES256, key required: true
CryptEncode(method,data,key,result)=32 / ok
ø‘UL»ÉsëDC‰ô ¬.K)ŒýÁ Lá¸ +< !Dï
[00] F8 | 91 | 55 | 4C | BB | C9 | 73 | EB | 44 | 43 | 89 | F4 | 06 | 13 | AC | 2E |
[16] 4B | 29 | 8C | FD | C1 | 11 | 4C | E1 | B8 | 05 | 2B | 3C | 14 | 21 | 44 | EF |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=44 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_AES256):
base64:'+JFVTLvJc+tEQ4n0BhOsLkspjP3BEUzhuAUrPBQhRO8='
- 3 CRYPT_DES, key required: true
CryptEncode(method,data,key,result)=32 / ok
µ b &“#ÇÅ+ý°'¥ B8f¡rØ-Pè<6âì,Ë£
[00] B5 | 06 | 9D | 62 | 11 | 26 | 93 | 23 | C7 | C5 | 2B | FD | BA | 27 | A5 | 10 |
[16] 42 | 38 | 66 | A1 | 72 | D8 | 2D | 50 | E8 | 3C | 36 | E2 | EC | 82 | CB | A3 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=44 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_DES):
base64:'tQadYhEmkyPHxSv9uielEEI4ZqFy2C1Q6Dw24uyCy6M='
- 4 CRYPT_HASH_SHA1, key required: false
CryptEncode(method,data,key,result)=20 / ok
§ßö*©°ø
€|)bËbzÇÍ Û€
[00] A7 | DF | F6 | 2A | A9 | BA | F8 | 0A | 80 | 7C | 29 | 62 | CB | 62 | 7A | C7 |
[16] CD | 0E | DB | 80 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=28 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_HASH_SHA1):
base64:'p9/2Kqm6+AqAfCliy2J6x80O24A='
- 5 CRYPT_HASH_SHA256, key required: false
CryptEncode(method,data,key,result)=32 / ok
ÚZ2š€»”¾7 €… ñ-ÄÁ´˜¦“ome2r@¾ô®³”
[00] DA | 5A | 32 | 9A | 80 | BB | 94 | BE | 37 | 0C | 80 | 85 | 07 | F1 | 96 | C4 |
[16] C1 | B4 | 98 | A6 | 93 | 6F | 6D | 65 | 32 | 72 | 40 | BE | F4 | AE | B3 | 94 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=44 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_HASH_SHA256):
base64:'2loymoC7lL43DICFB/GWxMG0mKaTb21lMnJAvvSus5Q='
- 6 CRYPT_HASH_MD5, key required: false
```

```
CryptEncode(method,data,key,result)=16 / ok
zIGT… □ Fû;—3þèå
[00] 7A | 49 | 47 | 54 | 85 | 1B | 7F | 11 | 46 | FB | 3B | 97 | 33 | FE | E8 | E5 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=24 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_HASH_MD5):
base64:'eklHVIUbfxFG+zuXM/7o5Q=='
- 7 CRYPT_ARCH_ZIP, key required: false
CryptEncode(method,data,key,result)=34 / ok
x^óI-Q/VHÍK.ª,(Q(ÉÈ,VÈM-.NLO
[00] 78 | 5E | F3 | 49 | 2D | 51 | 2F | 56 | 48 | CD | 4B | 2E | AA | 2C | 28 | 51 |
[16] 28 | C9 | C8 | 2C | 56 | C8 | 4D | 2D | 2E | 4E | 4C | 4F | 05 | 00 | 80 | 07 |
[32] 09 | C2 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=48 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_ARCH_ZIP):
base64:'eF7zSS1RL1ZIzUsuqiwoUSjJyCxWyE0tLk5MTwUAgAcJwg=='
```

En este caso, la clave tiene longitud suficiente para los tres métodos de cifrado, y los demás métodos para los que no es necesaria simplemente la ignoran. Por lo tanto, todas las llamadas a funciones se han completado con éxito.

En la siguiente sección, aprenderemos a descodificar cifrados y podremos comprobar si la función *CryptDecode* devuelve el mensaje original. Tenga en cuenta esta parte del registro.

La opción *DisableCRCinZIP* activada reducirá el resultado del método CRYPT_ARCH_ZIP en unos pocos bytes de sobrecarga.

```
- 7 CRYPT_ARCH_ZIP, key required: false
CryptEncode(method,data,key,result)=28 / ok
óI-Q/VHÍK.ª,(Q(ÉÈ,VÈM-.NLO
[00] F3 | 49 | 2D | 51 | 2F | 56 | 48 | CD | 4B | 2E | AA | 2C | 28 | 51 | 28 | C9 |
[16] C8 | 2C | 56 | C8 | 4D | 2D | 2E | 4E | 4C | 4F | 05 | 00 |
CryptEncode(CRYPT_BASE64,result,dummy,readable)=40 / ok
Try to decode this with CryptDecode.mq5 (CRYPT_ARCH_ZIP):
base64:'80ktUS9WSM1LLqosKFEoycgsVshNLS5OTE8FAA=='
```

Ahora vamos a trasladar los experimentos sobre codificación a los archivos. Para ello, ejecute de nuevo el script y borre el texto del campo *Text*. Como resultado, el programa procesará el archivo *MQL5Book/clock10.htm* varias veces y creará varios archivos derivados con diferentes extensiones.

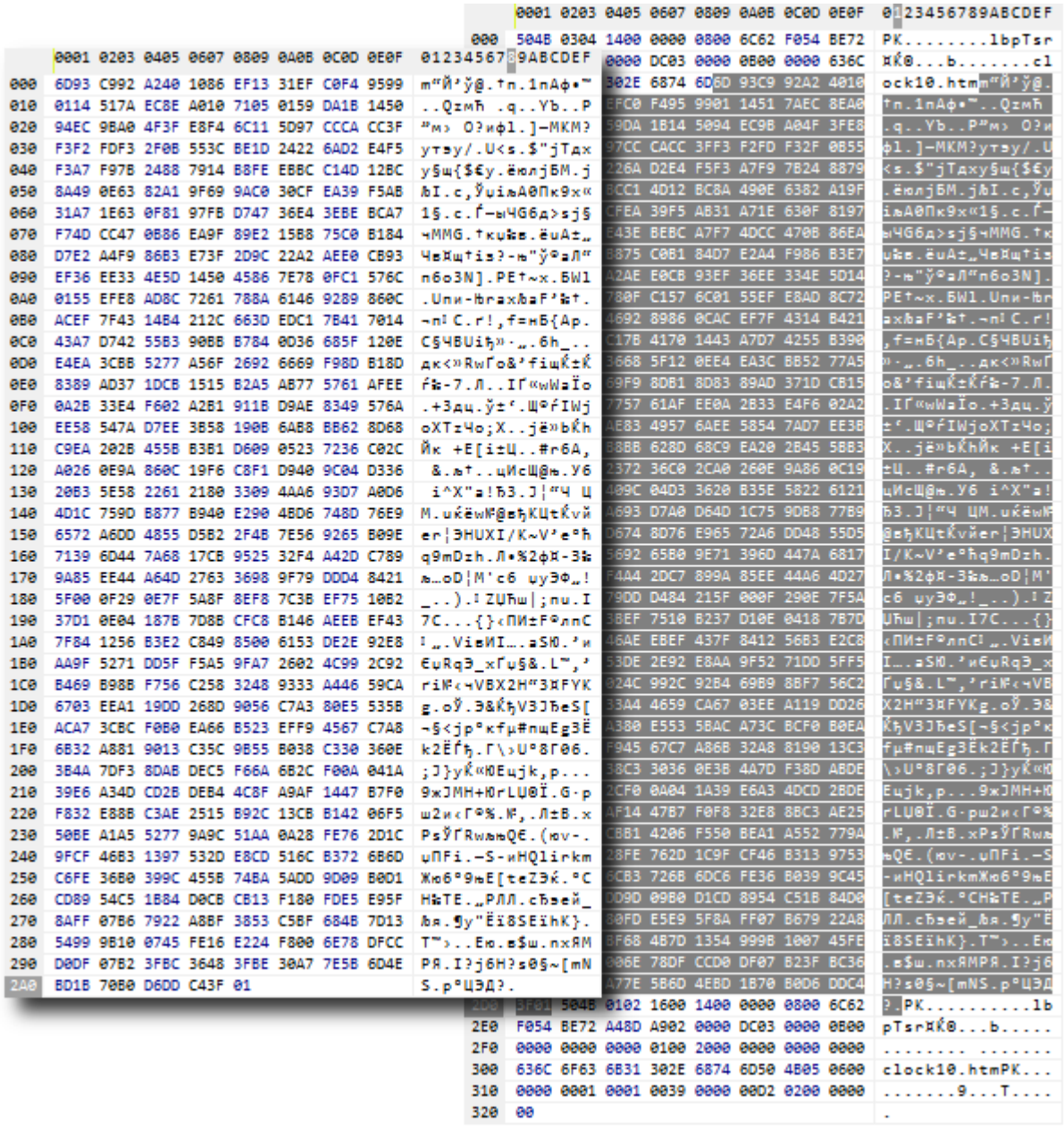
```
File=MQL5Book/clock10.htm / ok
FileLoad(File,data)=988 / ok
- 0 CRYPT_BASE64, key required: false
CryptEncode(method,data,key,result)=1320 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.BASE64
- 1 CRYPT_AES128, key required: true
CryptEncode(method,data,key,result)=992 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.AES128
- 2 CRYPT_AES256, key required: true
CryptEncode(method,data,key,result)=992 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.AES256
- 3 CRYPT_DES, key required: true
CryptEncode(method,data,key,result)=992 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.DES
- 4 CRYPT_HASH_SHA1, key required: false
CryptEncode(method,data,key,result)=20 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.SHA1
[00] 486ADFDD071CD23AB28E820B164D813A310B213F
- 5 CRYPT_HASH_SHA256, key required: false
CryptEncode(method,data,key,result)=32 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.SHA256
[00] 8990BBAC9C23B1F987952564EBCEF2078232D8C9D6F2CCC2A50784E8CDE044D0
- 6 CRYPT_HASH_MD5, key required: false
CryptEncode(method,data,key,result)=16 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.MD5
[00] 0CC4FBC899554BE0C0DBF5C18748C773
- 7 CRYPT_ARCH_ZIP, key required: false
CryptEncode(method,data,key,result)=687 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.ZIP
```

Puede mirar dentro de todos los archivos desde el gestor de archivos y asegurarse de que no queda nada en común con el contenido original. Muchos administradores de archivos tienen comandos o plugins para calcular sumas hash de modo que puedan compararse con valores hexadecimales MD5, SHA1 y SHA256 impresos en el registro.

Si intentamos codificar un texto o un archivo sin proporcionar una clave de la longitud correcta, obtendremos un error INVALID_ARRAY(4006). Por ejemplo, para un mensaje de texto predeterminado, seleccionamos AES256 en el parámetro *method* (requiere una clave de 32 bytes). Utilizando el parámetro *GenerateKey*, pedimos una clave con una longitud de 16 bytes (o puede eliminar parcial o totalmente el texto del campo *CustomKey*, dejando *GenerateKey* por defecto).

```
Key (bytes):
[00] 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
Text=Let's encrypt this message / ok
StringToCharArray(Text,data,0,-1,CP_UTF8)=26 / ok
- 0 CRYPT_AES256, key required: true
CryptEncode(method,data,key,result)=0 / INVALID_ARRAY(4006)
```

También puede comprimir el mismo archivo (como hicimos con *clock10.htm*) utilizando el método CRYPT_ARCH_ZIP o un archivador normal. Si después se mira con una utilidad de visor binario (que suele estar integrada en el administrador de archivos), ambos resultados mostrarán un bloque empaquetado común, y las diferencias estarán solo en los metadatos que lo enmarcan.



Comparación de un archivo comprimido con el método CRYPT_ARCH_ZIP (izquierda) y un archivo ZIP estándar (derecha)

Muestra que la parte central y principal del archivo es una secuencia de bytes (resaltados en oscuro) idéntica a la producida por la función *CryptEncode*.

Por último, mostraremos cómo se ha generado la representación textual *Base64* de un archivo gráfico *clock10.png*. Para ello, borre el campo *Text* y escriba *MQL5Book/clock10.png* en el parámetro *File*. Elija *Base64* en la lista desplegable *Method*.

```
File=MQL5Book/clock10.png / ok
FileLoad(File,data)=457 / ok
- 0 CRYPT_BASE64, key required: false
CryptEncode(method,data,key,result)=612 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.png.BASE64
```

Como resultado, se ha creado el archivo *clock10.png.BASE64*. En su interior, veremos la propia línea que se inserta en el código de la página web, en la etiqueta *img*.

Por cierto: el método de compresión «deflate» es la base del formato gráfico PNG, por lo que podemos utilizar *CryptEncode* para guardar mapas de bits de recursos en archivos PNG. El archivo de encabezado *PNG.mqh* se incluye con el libro, con un soporte mínimo para las estructuras internas necesarias para describir la imagen: se sugiere experimentar con su código fuente de forma independiente. Utilizando *PNG.mqh* hemos escrito un sencillo script *CryptPNG.mq5* que convierte el recurso del archivo «euro.bmp» suministrado con el terminal al archivo «my.png». La carga de archivos PNG no está implementada.

```
#resource "\\Images\\euro.bmp"

#include <MQL5Book/PNG.mqh>

void OnStart()
{
   uchar null[];      // empty key for CRYPT_ARCH_ZIP
   uchar result[];    // receiving array
   uint data[];       // original pixels
   uchar bytes[];     // original bytes
   int width, height;
   PRTF(ResourceReadImage("::Images\\euro.bmp", data, width, height));

   ArrayResize(bytes, ArraySize(data) * 3 + width); // *3 for PNG_CTYPE_TRUECOLOR (RG
   ArrayInitialize(bytes, 0);
   int j = 0;
   for(int i = 0; i < ArraySize(data); ++i)
   {
      if(i % width == 0) bytes[j++] = 0; // each line is prepended with a filter mode
      const uint c = data[i];
      // bytes[j++] = (uchar)((c >> 24) & 0xFF); // alpha, for PNG_CTYPE_TRUECOLORALP
      bytes[j++] = (uchar)((c >> 16) & 0xFF);
      bytes[j++] = (uchar)((c >> 8) & 0xFF);
      bytes[j++] = (uchar)(c & 0xFF);
   }

   PRTF(CryptEncode(CRYPT_ARCH_ZIP, bytes, null, result));

   int h = PRTF(FileOpen("my.png", FILE_BIN | FILE_WRITE));

   PNG::Image image(width, height, result); // default PNG_CTYPE_TRUECOLOR (RGB)
   image.write(h);

   FileClose(h);
}
```

7.4.3 Descifrado y descompresión de datos: CryptDecode

Para realizar operaciones de descifrado y descompresión de datos, MQL5 proporciona la función *CryptDecode*.

La función *CryptDecode* realiza una transformación inversa del array *data* al array *result* receptor utilizando el método especificado.

int CryptDecode(ENUM_CRYPT_METHOD method, const uchar &data[], const uchar &key[], uchar &result[])

Tenga en cuenta que la obtención de sumas hash realizada, en particular, por la función *CryptEncode*, es una transformación unidireccional: es imposible recuperar los datos originales a partir de los hashes.

La función devuelve el número de bytes colocados en el array de destino o 0 en caso de error. El código de error se añadirá a *_LastError*. Podría ser, por ejemplo, INVALID_PARAMETER (4003) si intentamos

descifrar el hash (*method* es igual a una de las constantes CRYPT_HASH) o INVALID_ARRAY (4006) si la clave de descifrado no es lo suficientemente larga o está ausente.

Si la clave es incorrecta (diferente de la utilizada en el cifrado), obtendremos como resultado un galimatías en lugar de los datos fuente codificados, pero el código de error es cero. Este es el comportamiento normal de la función.

Comprobemos el trabajo de *CryptDecode* utilizando el mismo script *CryptDecode.mq5*.

En los parámetros de entrada puede especificar el texto o el archivo que desea convertir. El texto siempre está implícito en la codificación *Base64*, ya que todos los datos codificados están en formato binario y no se admiten en los parámetros *input*. El método de conversión se selecciona en la lista *Method*.

```
input string Text; // Text (base64, or empty to process File)
input string File = "MQL5Book/clock10.htm.BASE64";
input ENUM_CRYPT_METHOD_EXT Method = _CRYPT_BASE64;
```

Los métodos de cifrado requieren una clave que puede especificarse como una cadena en el campo *CustomKey* si *GenerateKey* contiene la opción DUMMY_KEY_CUSTOM. También puede generar una clave demo de la longitud requerida a partir de la enumeración DUMMY_KEY_LENGTH (es la misma que en el script *CryptEncode.mq5*).

```
input DUMMY_KEY_LENGTH GenerateKey = DUMMY_KEY_CUSTOM; // GenerateKey (length, or fro
input string CustomKey = "My top secret key is very strong";
input bool DisableCRCinZIP = false;
```

En *GenerateKey* y *CustomKey*, debe elegir los mismos valores que al lanzar *CryptEncode.mq5*.

El algoritmo en *OnStart* comienza con una descripción de los arrays necesarios y la obtención de una clave a partir de una cadena o por simple generación (sólo para una demostración, utilice software o algoritmos especiales para generar una clave cripto-resistente que funcione).

```
void OnStart()
{
   ENUM_CRYPT_METHOD method = 0;
   int methods[];
   uchar key[] = {};        // default empty key suitable for zip and base64
   uchar data[], result[];
   uchar zip[], opt[] = {1, 0, 0, 0};

   if(GenerateKey == DUMMY_KEY_CUSTOM)
   {
      if(StringLen(CustomKey))
      {
         PRTF(CustomKey);
         StringToCharArray(CustomKey, key, 0, -1, CP_UTF8);
         ArrayResize(key, ArraySize(key) - 1);
      }
   }
   else if(GenerateKey != DUMMY_KEY_0)
   {
      ArrayResize(key, GenerateKey);
      for(int i = 0; i < GenerateKey; ++i) key[i] = (uchar)i;
   }

   if(ArraySize(key))
   {
      Print("Key (bytes):");
      ByteArrayPrint(key);
   }
   else
   {
      Print("Key is not provided");
   }
```

A continuación, leemos el contenido del archivo o descodificamos *Base64* desde el campo *Text* (dependiendo de lo que se haya rellenado) para obtener los datos que se van a procesar.

```
   method = (ENUM_CRYPT_METHOD)Method;
   Print("- ", EnumToString(method), ", key required: ", KEY_REQUIRED(method));
   if(StringLen(Text))
   {
      if(method != CRYPT_BASE64)
      {
         // since all methods except Base64 produce binary results,
         // they are additionally converted to CryptEncode.mq5 using Base64 to text,
         // so here we want to recover binary data from text input
         // before decryption
         uchar base64[];
         const uchar dummy[] = {};
         PRTF(Text);
         PRTF(StringToCharArray(Text, base64, 0, -1, CP_UTF8));
         ArrayResize(base64, ArraySize(base64) - 1);
         Print("Text (bytes):");
         ByteArrayPrint(base64);
         if(!PRTF(CryptDecode(CRYPT_BASE64, base64, dummy, data)))
         {
            return; // error
         }

         Print("Raw data to decipher (after de-base64):");
         ByteArrayPrint(data);
      }
      else
      {
         PRTF(StringToCharArray(Text, data, 0, StringLen(Text), CP_UTF8));
         ArrayResize(data, ArraySize(data) - 1);
      }
   }
   else if(StringLen(File))
   {
      PRTF(File);
      if(PRTF(FileLoad(File, data)) <= 0)
      {
         return; // error
      }
   }
```

Si el usuario intenta recuperar datos del hash, mostraremos una advertencia.

```
   if(IS_HASH(method))
   {
      Print("WARNING: hashes can not be used to restore data! CryptDecode will fail."
   }
```

Por último, realizamos directamente la desencriptación o descompresión (unpacking). En el caso de un texto, el resultado simplemente se registra. En el caso de un archivo, añadimos la extensión «.dec» al nombre y escribimos un nuevo archivo: puede compararse con el original, que se procesó mediante el script *CryptEncode.mq5*.

```
   ResetLastError();
   if(PRTF(CryptDecode(method, data, key, result)))
   {
      if(StringLen(Text))
      {
         Print("Text restored:");
         Print(CharArrayToString(result, 0, WHOLE_ARRAY, CP_UTF8));
      }
      else // File
      {
         const string filename = File + ".dec";
         if(PRTF(FileSave(filename, result)))
         {
            Print("File saved: ", filename);
         }
      }
   }
```

Si ejecuta el script con la configuración predeterminada, intentará descodificar el archivo *MQL5Book/clock10.htm.BASE64*. Se supone que se creó durante los experimentos de la sección anterior, por lo que el proceso debería tener éxito.

```
- CRYPT_BASE64, key required: false
File=MQL5Book/clock10.htm.BASE64 / ok
FileLoad(File,data)=1320 / ok
CryptDecode(method,data,key,result)=988 / ok
FileSave(filename,result)=true / ok
File saved: MQL5Book/clock10.htm.BASE64.dec
```

El archivo *clock10.htm.BASE64.dec* obtenido es completamente idéntico al original *clock10.htm*. Lo mismo debería ocurrir si descifra archivos con extensiones AES128, AES256 o DES, siempre que especifique la misma clave que la utilizada al cifrar.

Para mayor claridad, comprobemos el descifrado del texto. Anteriormente, el cifrado de una frase conocida mediante el método AES128 producía un binario que, por comodidad, se convertía en la cadena *Base64* siguiente:

```
AQuvVCoSy1szaN8Owy8tQxl9rIrRj9hOqK7KgYYGh9E=
```

Vamos a introducirla en el campo *Text* y a seleccionar AES128 en la lista desplegable *Method*. Veremos los siguientes registros:

```
CustomKey=My top secret key is very strong / ok
Key (bytes):
[00] 4D | 79 | 20 | 74 | 6F | 70 | 20 | 73 | 65 | 63 | 72 | 65 | 74 | 20 | 6B | 65 |
[16] 79 | 20 | 69 | 73 | 20 | 76 | 65 | 72 | 79 | 20 | 73 | 74 | 72 | 6F | 6E | 67 |
- CRYPT_AES128, key required: true
Text=AQuvVCoSy1szaN8Owy8tQxl9rIrRj9hOqK7KgYYGh9E= / ok
StringToCharArray(Text,base64,0,-1,CP_UTF8)=44 / ok
Text (bytes):
[00] 41 | 51 | 75 | 76 | 56 | 43 | 6F | 53 | 79 | 31 | 73 | 7A | 61 | 4E | 38 | 4F |
[16] 77 | 79 | 38 | 74 | 51 | 78 | 6C | 39 | 72 | 49 | 72 | 52 | 6A | 39 | 68 | 4F |
[32] 71 | 4B | 37 | 4B | 67 | 59 | 59 | 47 | 68 | 39 | 45 | 3D |
CryptDecode(CRYPT_BASE64,base64,dummy,data)=32 / ok
Raw data to decipher (after de-base64):
[00] 01 | 0B | AF | 54 | 2A | 12 | CB | 5B | 33 | 68 | DF | 0E | C3 | 2F | 2D | 43 |
[16] 19 | 7D | AC | 8A | D1 | 8F | D8 | 4E | A8 | AE | CA | 81 | 86 | 06 | 87 | D1 |
CryptDecode(method,data,key,result)=32 / ok
Text restored:
Let's encrypt this message
```

El mensaje se ha descifrado correctamente.

Si, con el mismo texto de entrada, elige generar una clave arbitraria (aunque de longitud suficiente), obtendrá un galimatías en lugar de un mensaje.

```
Key (bytes):
[00] 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
- CRYPT_AES128, key required: true
Text=AQuvVCoSy1szaN8Owy8tQxl9rIrRj9hOqK7KgYYGh9E= / ok
StringToCharArray(Text,base64,0,-1,CP_UTF8)=44 / ok
Text (bytes):
[00] 41 | 51 | 75 | 76 | 56 | 43 | 6F | 53 | 79 | 31 | 73 | 7A | 61 | 4E | 38 | 4F |
[16] 77 | 79 | 38 | 74 | 51 | 78 | 6C | 39 | 72 | 49 | 72 | 52 | 6A | 39 | 68 | 4F |
[32] 71 | 4B | 37 | 4B | 67 | 59 | 59 | 47 | 68 | 39 | 45 | 3D |
CryptDecode(CRYPT_BASE64,base64,dummy,data)=32 / ok
Raw data to decipher (after de-base64):
[00] 01 | 0B | AF | 54 | 2A | 12 | CB | 5B | 33 | 68 | DF | 0E | C3 | 2F | 2D | 43 |
[16] 19 | 7D | AC | 8A | D1 | 8F | D8 | 4E | A8 | AE | CA | 81 | 86 | 06 | 87 | D1 |
CryptDecode(method,data,key,result)=32 / ok
Text restored:
??? ?L?? ??J Q+?]?v?9?????n?N?Ű
```

El programa se comportará de forma similar si confunde el método de encriptación.

No tiene sentido elegir métodos de "unhashing": INVALID_PARAMETER (4003).

```
- CRYPT_HASH_MD5, key required: false
File=MQL5Book/clock10.htm.MD5 / ok
FileLoad(File,data)=16 / ok
WARNING: hashes can not be used to restore data! CryptDecode will fail.
CryptDecode(method,data,key,result)=0 / INVALID_PARAMETER(4003)
```

Un intento de descomprimir (CRYPT_ARCH_ZIP) algo que no es un bloque «deflate» comprimido dará como resultado INTERNAL_ERROR (4001). Se puede obtener el mismo error si se activa la opción de omitir CRC para el «archivo» sin él o, a la inversa, se descomprimen los datos sin la opción, aunque el empaquetado se haya realizado con ella.

7.5 Funciones de red

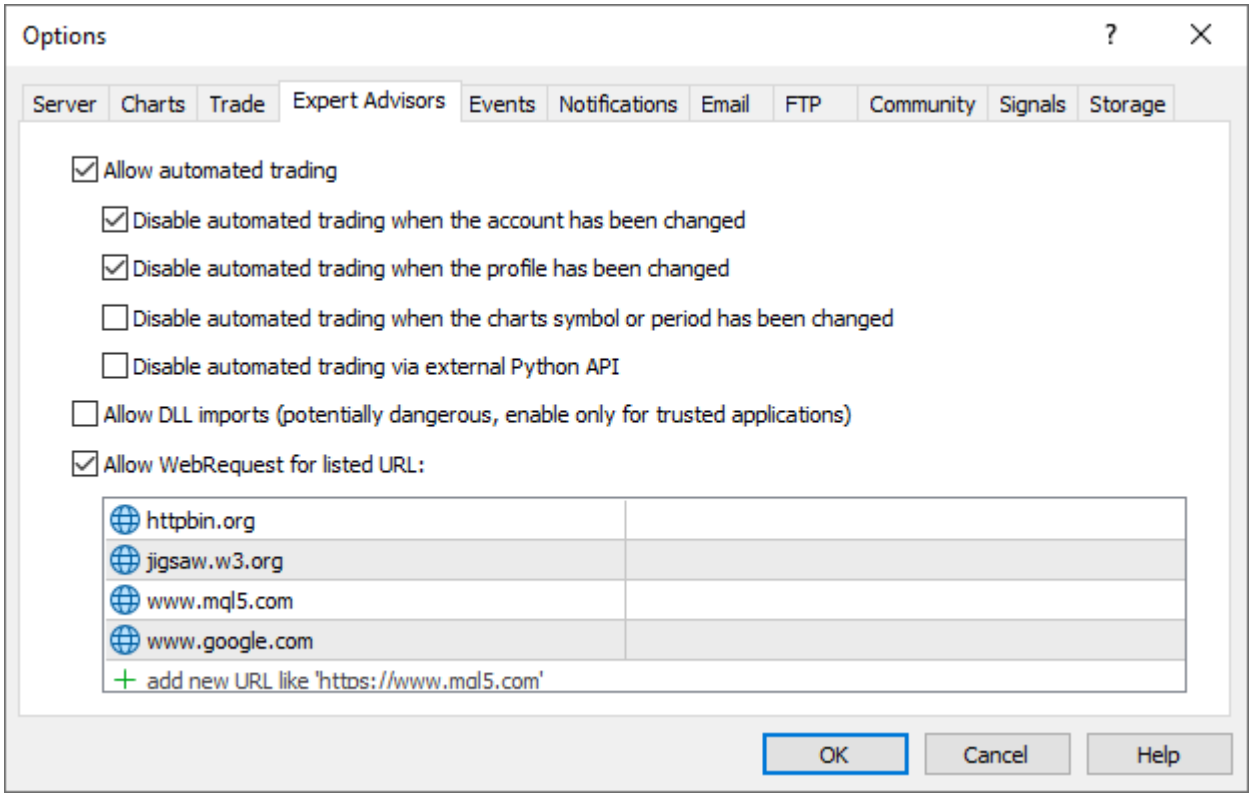
Los programas MQL pueden comunicarse con otros ordenadores de una red distribuida o con servidores de Internet mediante diversos protocolos. Las funciones admiten operaciones con sitios web y servicios (HTTP/HTTPS), transferencia de archivos (FTP), envío de correo electrónico (SMTP) y notificaciones push.

Las funciones de red pueden dividirse en tres grupos:

- *SendFTP*, *SendMail* y *SendNotification* son las funciones más básicas para enviar archivos, correos electrónicos y notificaciones móviles.
- La función *WebRequest* está diseñada para trabajar con recursos web y permite enviar fácilmente solicitudes HTTP (incluyendo GET y POST).
- El conjunto de funciones *Socket* permite crear una conexión TCP (incluida una conexión TLS segura) con un host remoto a través de sockets del sistema.

La secuencia en la que se enumeran los grupos corresponde a la transición entre las funciones de alto nivel que ofrecen mecanismos ya hechos para la interacción entre el cliente y el servidor, a las de bajo nivel que permiten la implementación de un protocolo de aplicación arbitrario según los requisitos de un servicio público concreto (por ejemplo, una bolsa de criptomonedas o un servicio de señales de trading). Por supuesto, una aplicación de este tipo requiere mucho esfuerzo.

Por seguridad del usuario final, la lista de direcciones web permitidas a las que puede conectarse un programa MQL mediante las funciones *Socket* y *WebRequest* debe especificarse explícitamente en el cuadro de diálogo de configuración, en la pestaña *Expert Advisors*. Aquí puede especificar dominios, la ruta completa a páginas web (no sólo el sitio, sino también otros fragmentos de la URL, como carpetas o un número de puerto) o direcciones IP. A continuación se muestra una captura de pantalla de la configuración de algunos de los dominios de los ejemplos de este capítulo.



Permisos de acceso a los recursos de red en la configuración del terminal

Esta lista no se puede editar mediante programación. Si intenta acceder a un recurso de red que no está en esta lista, el programa MQL recibirá un error y la solicitud será rechazada.

Es importante señalar que todas las funciones de red proporcionan solamente una conexión de cliente a un servidor en particular, es decir, es imposible organizar un servidor utilizando MQL5 para esperar y procesar las solicitudes entrantes. Para ello, será necesario integrar el terminal con un programa externo o un servicio de Internet (por ejemplo, con uno en la nube).

7.5.1 Envío de notificaciones push

Como sabe, el terminal le permite enviar notificaciones push de los servicios MetaQuotes, del propio terminal y de los programas MQL a un dispositivo móvil con sistemas operativos iOS o Android. Esta tecnología utiliza MetaQuotes ID, un identificador único de un usuario (véase la nota). MetaQuotes ID se asigna al instalar la versión móvil del terminal en el dispositivo del usuario, tras lo cual se debe especificar el ID en los ajustes del terminal, en la pestaña Notificaciones (se pueden especificar varios identificadores separados por comas). Después, la funcionalidad de envío de notificaciones push estará disponible para los programas MQL.

> De hecho, MetaQuotes ID no identifica a un usuario, sino a una instalación concreta de un terminal móvil; un usuario puede tener varias instalaciones. Por defecto, el ID no está asociado al registro en la comunidad mql5.com, aunque puede especificarse tal vinculación en el sitio. No confunda el registro de usuario en la comunidad con el ID de MetaQuotes. Para trabajar con las notificaciones no es necesario que el usuario del terminal se conecte a la comunidad.

bool SendNotification(const string text)

La función *SendNotification* envía notificaciones push con el texto especificado a todos los terminales móviles que tengan un MetaQuotes ID de la configuración del terminal. La longitud del mensaje no puede superar los 255 caracteres.

Si la notificación se envía correctamente desde el terminal, la función devuelve *true*, y devuelve *false* si se produce un error. Los posibles códigos de error en *_LastError* incluyen:

- 4515 - ERR_NOTIFICATION_SEND_FAILED - problemas de comunicación
- 4516 - ERR_NOTIFICATION_WRONG_PARAMETER - un parámetro no válido, por ejemplo, una cadena vacía
- 4517 - ERR_NOTIFICATION_WRONG_SETTINGS - MetaQuotes ID está mal configurado o no existe
- 4518 - ERR_NOTIFICATION_TOO_FREQUENT - llamadas a funciones demasiado frecuentes

Si hay conexión con el servidor, el mensaje se envía al instante. Si el dispositivo del usuario está en línea, el mensaje debería llegar al destinatario, pero no se puede garantizar la entrega en el caso general. No hay notificación de retorno al programa sobre la entrega del mensaje. No se guarda el historial de mensajes push en el servidor para entrega diferida.

La función tiene restricciones en cuanto a la frecuencia de uso: no más de 2 llamadas por segundo y no más de 10 por minuto.

En el probador de estrategias, la función *SendNotification* no se ejecuta.

En el libro se incluye un sencillo script *NetNotification.mq5* que envía una notificación de prueba cuando la configuración es correcta.

```
void OnStart()
{
   const string message = MQLInfoString(MQL_PROGRAM_NAME)
      + " runs on " + AccountInfoString(ACCOUNT_SERVER)
      + " " + (string)AccountInfoInteger(ACCOUNT_LOGIN);
   Print("Sending notification: " + message);
   PRTF(SendNotification(NULL));    // INVALID_PARAMETER(4003)
   PRTF(SendNotification(message)); // NOTIFICATION_WRONG_SETTINGS(4517) or 0 (succes
}
```

7.5.2 Envío de notificaciones por correo electrónico

El terminal le permite enviar correos electrónicos a la dirección de correo electrónico especificada en la pestaña Correo electrónico del cuadro de diálogo de configuración. Para este caso, MQL5 proporciona la función *SendMail*.

bool SendMail(const string subject, const string text)

Los parámetros de la función establecen el título y el texto (el cuerpo del mensaje).

La función devuelve *true* si el mensaje está en cola para su envío en el servidor de correo; en caso contrario, devuelve *false*. Es posible que se produzcan errores si el trabajo con el correo está desactivado en la configuración o si los datos de correo (servidor SMTP, puerto, inicio de sesión, contraseña) contienen un error o no están especificados.

En el probador de estrategias, la función *SendMail* no se ejecuta.

> MQL5 no admite la comprobación del correo electrónico entrante ni su lectura (es decir, los protocolos POP, IMAP).

En el libro se incluye el script *NetMail.mq5*, que intenta enviar un mensaje de prueba.

```
void OnStart()
{
   const string message = "Hello from "
      + AccountInfoString(ACCOUNT_SERVER)
      + " " + (string)AccountInfoInteger(ACCOUNT_LOGIN);
   Print("Sending email: " + message);
   PRTF(SendMail(MQLInfoString(MQL_PROGRAM_NAME),
      message)); // MAIL_SEND_FAILED(4510) or 0 (success)
}
```

7.5.3 Envío de archivos a un servidor FTP

MetaTrader 5 admite el envío de archivos a un servidor FTP. Para que esta función pueda operar, debe introducir los datos FTP necesarios en el cuadro de diálogo de configuración de la pestaña FTP: dirección del servidor FTP, nombre de usuario, contraseña y, opcionalmente, la ruta para colocar los archivos en el servidor. Si su ordenador está en la red de un ISP que no le ha asignado una dirección IP pública, probablemente tendrá que activar el modo pasivo.

La función *SendFTP* permite enviar archivos directamente desde un programa MQL.

bool SendFTP(const string filename, const string path = NULL)

La función envía un archivo con el nombre especificado al servidor FTP desde la configuración del terminal. Si es necesario, puede especificar una ruta diferente a la configurada de antemano. Si no se especifica el parámetro *path*, se utiliza el directorio descrito en la configuración.

El archivo cargado debe encontrarse en la carpeta *MQL5/Files* o en sus subcarpetas.

La función devuelve un indicador de éxito (*true*) o de error (*false*). Los errores potenciales en *_LastError* incluyen:

- 4514 - ERR_FTP_SEND_FAILED - Error al enviar un archivo por FTP
- 4519 - ERR_FTP_NOSERVER - Servidor FTP no especificado
- 4520 - ERR_FTP_NOLOGIN - No se ha especificado el login FTP
- 4521 - ERR_FTP_FILE_ERROR - el archivo especificado no se encontró en el directorio MQL5/Files
- 4522 - ERR_FTP_CONNECT_FAILED - se ha producido un error al conectar con el servidor FTP
- 4523 - ERR_FTP_CHANGEDIR - el directorio para subir el archivo no se encontró en el servidor FTP
- 4524 - ERR_FTP_CLOSED - se ha cerrado la conexión con el servidor FTP

La función bloquea la ejecución del programa MQL hasta que se completa la operación. A este respecto, no se permite utilizar la función en los indicadores.

Así mismo, en el probador de estrategias no se ejecuta la función *SendFTP*.

> El terminal sólo admite el envío de un único archivo a un servidor FTP. El resto de comandos FTP no están disponibles desde MQL5.

El script de ejemplo *NetFtp.mq5* toma una captura de pantalla del gráfico actual e intenta enviarla por FTP.

```
void OnStart()
{
   const string filename = _Symbol + "-" + PeriodToString() + "-"
      + (string)(ulong)TimeTradeServer() + ".png";
   PRTF(ChartScreenShot(0, filename, 300, 200));
   Print("Sending file: " + filename);
   PRTF(SendFTP(filename, "/upload")); // 0 (success) or FTP_CONNECT_FAILED(4522), FT
}
```

7.5.4 Intercambio de datos con un servidor web a través de HTTP/HTTPS

MQL5 permite integrar programas con servicios web y solicitar datos de Internet. Los datos pueden enviarse y recibirse a través de los protocolos HTTP/HTTPS utilizando la función *WebRequest*, que tiene dos versiones: una para una interacción simplificada y la otra para una avanzada con servidores web.

int WebRequest(const string method, const string url, const string cookie, const string referer,
 int timeout, const char &data[], int size, char &result[], string &response)
int WebRequest(const string method, const string url, const string headers, int timeout,
 const char &data[], char &result[], string &response)

La principal diferencia entre las dos funciones es que la versión simplificada sólo permite especificar dos tipos de encabezados en la petición: un *cookie* y un *referer*, es decir, la dirección desde la que se

realiza la transición (aquí no hay ningún error tipográfico: históricamente, la palabra «referrer» se escribe en las cabeceras HTTP mediante una 'r'). La versión extendida toma un parámetro genérico *headers* para enviar un conjunto arbitrario de encabezados. Los encabezados de solicitud tienen la forma «nombre: valor» y van unidas por un salto de línea «\r\n» si hay más de una.

Si asumimos que la cadena *cookie* debe contener «nombre1=valor1; nombre2=valor2» y el enlace *referer* es igual a «google.com», entonces, para llamar a la segunda versión de la función con el mismo efecto que la primera, necesitamos añadir lo siguiente en el parámetro *headers*: «Cookie: nombre1=valor1; nombre2=valor2\r\nReferer: google.com».

El parámetro *method* especifica uno de los métodos de protocolo, «HEAD», «GET» o «POST». La dirección del recurso o servicio solicitado se pasa en el parámetro *url*. Según la especificación HTTP, la longitud de un identificador de recurso de red está limitada a 2048 bytes, pero en el momento de escribir el libro, MQL5 tenía un límite de 1024 bytes.

La duración máxima de una solicitud viene determinada por la dirección *timeout* en milisegundos.

Ambas versiones de la función transfieren datos del array *data* al servidor. La primera opción requiere además especificar el tamaño de este array en bytes (*size*).

Para enviar peticiones sencillas con valores de varias variables, puede combinarlas en una cadena como «nombre1=valor1&nombre2=valor2&...» y añadirlas a la dirección de la petición GET, después del carácter delimitador '?' o ponerlas en el array *data* para una petición POST utilizando el encabezado «Content-Type: application/x-www-form-urlencoded». Para casos más complejos, como la carga de archivos, utilice una solicitud POST y «Content-Type: multipart/form-data».

El array receptor *result* obtiene el cuerpo de la respuesta del servidor (si existe). Los encabezados de respuesta del servidor se colocan en la cadena *response*.

La función devuelve el código de respuesta HTTP del servidor, o -1 en caso de error del sistema (por ejemplo, problemas de comunicación o errores de parámetros). Los posibles códigos de error que pueden aparecer en *_LastError* incluyen:

- 5200 - ERR_WEBREQUEST_INVALID_ADDRESS - URL no válida
- 5201 - ERR_WEBREQUEST_CONNECT_FAILED - no se pudo conectar a la URL especificada
- 5202 - ERR_WEBREQUEST_TIMEOUT - se ha superado el tiempo de espera para recibir una respuesta del servidor
- 5203 - ERR_WEBREQUEST_REQUEST_FAILED - cualquier otro error como resultado de la solicitud

Recordemos que, aunque la solicitud se haya ejecutado sin errores en el nivel MQL5, el código de respuesta HTTP del servidor puede contener un error de aplicación (por ejemplo, se requiere autorización, formato de datos no válido, página no encontrada, etc.). En este caso, el resultado estará vacío, y las instrucciones para resolver la situación, por regla general, se aclaran analizando los encabezados *response* recibidos.

Para utilizar la función *WebRequest*, las direcciones del servidor deben añadirse a la lista de URL permitidas en la pestaña *Expert Advisors* de la configuración del terminal. El puerto del servidor se selecciona automáticamente en función del protocolo especificado: 80 para «http://" y 443 para "https://».

> La función f*WebRequest* es sincrónica, es decir, detiene la ejecución del programa a la espera de una respuesta del servidor. A este respecto, no se permite llamar a la función desde los indicadores, ya que trabajan en flujos comunes para cada carácter. Un retraso en la ejecución de un indicador detendrá la actualización de todos los gráficos para este símbolo.

Cuando se trabaja en el probador de estrategias, la función *WebRequest* no se ejecuta.

Empecemos con un simple script *WebRequestTest.mq5* que ejecuta una única petición. En los parámetros de entrada, proporcionaremos una opción para el método (por defecto «GET»), la dirección de la página web de prueba, encabezados adicionales (opcional), y también el tiempo de espera.

```
input string Method = "GET"; // Method (GET,POST)
input string Address = "https://httpbin.org/headers";
input string Headers;
input int Timeout = 5000;
```

La dirección se introduce como en la línea del navegador: todos los caracteres que la especificación HTTP prohíbe utilizar directamente en las direcciones (incluidos los caracteres del alfabeto local) quedan «enmascarados» automáticamente por la función *WebRequest* antes de enviarlos según el algoritmo *urlencode* (el navegador hace exactamente lo mismo, pero nosotros no lo vemos, ya que esta vista está destinada a ser transmitida a través de la infraestructura de red, no a los humanos).

También añadiremos la opción *DumpDataToFiles*: cuando sea igual a *true*, el script guardará la respuesta del servidor en un archivo aparte, ya que puede ser bastante grande. El valor *false* indica que los datos se envíen directamente al registro.

```
input bool DumpDataToFiles = true;
```

> Hay que decir de entrada que para probar este tipo de scripts se necesita un servidor. Los interesados pueden instalar un servidor web local, por ejemplo, node.js, pero esto requiere una preparación propia o la instalación de scripts del lado del servidor (en este caso, la conexión de módulos JavaScript). Una forma más sencilla es utilizar los servidores web públicos de prueba disponibles en Internet. Puede utilizar, por ejemplo, *httpbin.org*, *httpbingo.org*, *webhook site*, *putsreq.com*, *www.mockable.io* o *reqbin.com*. Ofrecen un conjunto diferente de prestaciones; elija o encuentre el más adecuado para usted (cómodo y comprensible, o lo más flexible posible).

En el parámetro *Address*, la dirección por defecto es el *endpoint* de la API del servidor *httpbin.org*. Esta «página web» dinámica devuelve al cliente los encabezados HTTP de su solicitud (en formato JSON). Así, podremos ver en nuestro programa qué es exactamente lo que ha llegado al servidor web desde el terminal.

> No olvide añadir el dominio «httpbin.org» a la lista de permitidos en la configuración del terminal.

El formato de texto JSON es el estándar de facto para los servicios web. En el sitio *mql5.com* se pueden encontrar implementaciones ya hechas de clases para analizar JSON, pero por ahora nos limitaremos a mostrar el JSON «tal cual».

En el manejador *OnStart*, llamamos a *WebRequest* con los parámetros dados y procesamos el resultado si el código de error es no negativo. Los encabezados de respuesta del servidor (*response*) se registran siempre.

```
void OnStart()
{
   uchar data[], result[];
   string response;

   int code = PRTF(WebRequest(Method, Address, Headers, Timeout, data, result, respon
   if(code > -1)
   {
      Print(response);
      if(ArraySize(result) > 0)
      {
         PrintFormat("Got data: %d bytes", ArraySize(result));
         if(DumpDataToFiles)
         {
            string parts[];
            URL::parse(Address, parts);

            const string filename = parts[URL_HOST] +
               (StringLen(parts[URL_PATH]) > 1 ? parts[URL_PATH] : "/_index_.htm");
            Print("Saving ", filename);
            PRTF(FileSave(filename, result));
         }
         else
         {
            Print(CharArrayToString(result, 0, 80, CP_UTF8));
         }
      }
   }
}
```

Para formar el nombre del archivo, utilizamos la clase auxiliar URL del archivo de encabezado *URL.mqh* (que no se describirá completamente aquí). El método *URL::parse* analiza la cadena pasada en componentes de URL de acuerdo con la especificación, ya que la forma general de la URL es siempre «protocol://domain.com:port/path?query#hash»; tenga en cuenta que muchos fragmentos son opcionales. Los resultados se colocan en el array receptor, cuyos índices corresponden a partes específicas de la URL y se describen en la enumeración URL_PARTS:

```
enum URL_PARTS
{
   URL_COMPLETE,    // full address
   URL_SCHEME,      // protocol
   URL_USER,        // username/password (deprecated, not supported)
   URL_HOST,        // server
   URL_PORT,        // port number
   URL_PATH,        // path/directories
   URL_QUERY,       // query string after '?'
   URL_FRAGMENT,    // fragment after '#' (not highlighted)
   URL_ENUM_LENGTH
};
```

Así, cuando los datos recibidos deben escribirse en un archivo, el script lo crea en una carpeta con el nombre del servidor (*parts[URL_HOST]*) y así sucesivamente, conservando la jerarquía de rutas en la

URL (*parts[URL_PATH]*): en el caso más sencillo, será simplemente el nombre del «endpoint». Cuando se solicita la página de inicio de un sitio (la ruta contiene únicamente una barra '/'), el archivo se denomina «_index_.htm».

Intentemos ejecutar el script con los parámetros predeterminados, recordando primero permitir este servidor en la configuración del terminal. En el registro veremos las siguientes líneas (encabezados HTTP de la respuesta del servidor y un mensaje sobre el guardado correcto del archivo):

```
WebRequest(Method,Address,Headers,Timeout,data,result,response)=200 / ok
Date: Fri, 22 Jul 2022 08:45:03 GMT
Content-Type: application/json
Content-Length: 291
Connection: keep-alive
Server: gunicorn/19.9.0
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true

Got data: 291 bytes
Saving httpbin.org/headers
FileSave(filename,result)=true / ok
```

El archivo *httpbin.org/headers* contiene los encabezados de nuestra petición tal y como las ve el servidor (el propio servidor ha añadido el formato JSON al respondernos).

```
{
  "headers":
  {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Accept-Language": "ru,en",
    "Host": "httpbin.org",
    "User-Agent": "MetaTrader 5 Terminal/5.3333 (Windows NT 10.0; Win64; x64)",
    "X-Amzn-Trace-Id": "Root=1-62da638f-2554..." // <- this is added by the reverse p
  }
}
```

Así, el terminal informa de que está preparado para aceptar datos de cualquier tipo, con compatibilidad para compresión por métodos específicos y una lista de idiomas preferidos. Además, aparece en el campo User-Agent como MetaTrader 5. Esto último puede resultar indeseable cuando se trabaja con algunos sitios optimizados para funcionar exclusivamente con navegadores. Entonces podemos especificar un nombre ficticio en el parámetro de entrada *headers*, por ejemplo, «User-Agent: Mozilla/5.0 (Windows NT 10.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/103.0.0.0 Safari/537.36».

> Algunos de los sitios de prueba mencionados anteriormente le permiten organizar un entorno de prueba temporal en el servidor con un nombre aleatorio para su experimento personal: para ello, tiene que ir al sitio desde un navegador y obtener un enlace único que suele funcionar durante 24 horas. Entonces podrá utilizar este enlace como dirección para las solicitudes de MQL5 y monitorizar el comportamiento de las mismas directamente desde el navegador. Allí también puede configurar las respuestas del servidor, en particular, el intento de envío de formularios.

Hagamos este ejemplo un poco más difícil. El servidor puede requerir acciones adicionales del cliente para dar respuesta a la solicitud; en concreto, autorizar, realizar una «redirección» (ir a una dirección diferente), reducir la frecuencia de las solicitudes, etc. Todas estas «señales» se denotan mediante

códigos HTTP especiales devueltos por la función *WebRequest*. Por ejemplo, los códigos 301 y 302 significan redirigir por motivos diferentes, y *WebRequest* lo ejecuta internamente de forma automática, volviendo a solicitar la página en la dirección especificada por el servidor (por lo tanto, los códigos de redirección nunca acaban en el código del programa MQL). El código 401 requiere que el cliente proporcione un nombre de usuario y una contraseña, y aquí toda la responsabilidad recae en nosotros. Hay muchas formas de enviar estos datos. Un nuevo script *WebRequestAuth.mq5* demuestra el manejo de dos opciones de autorización que el servidor solicita utilizando encabezados de respuesta HTTP: «WWW-Authenticate: Basic» o «WWW-Authenticate: Digest». En los encabezados podría tener este aspecto:

```
WWW-Authenticate:Basic realm="DemoBasicAuth"
```

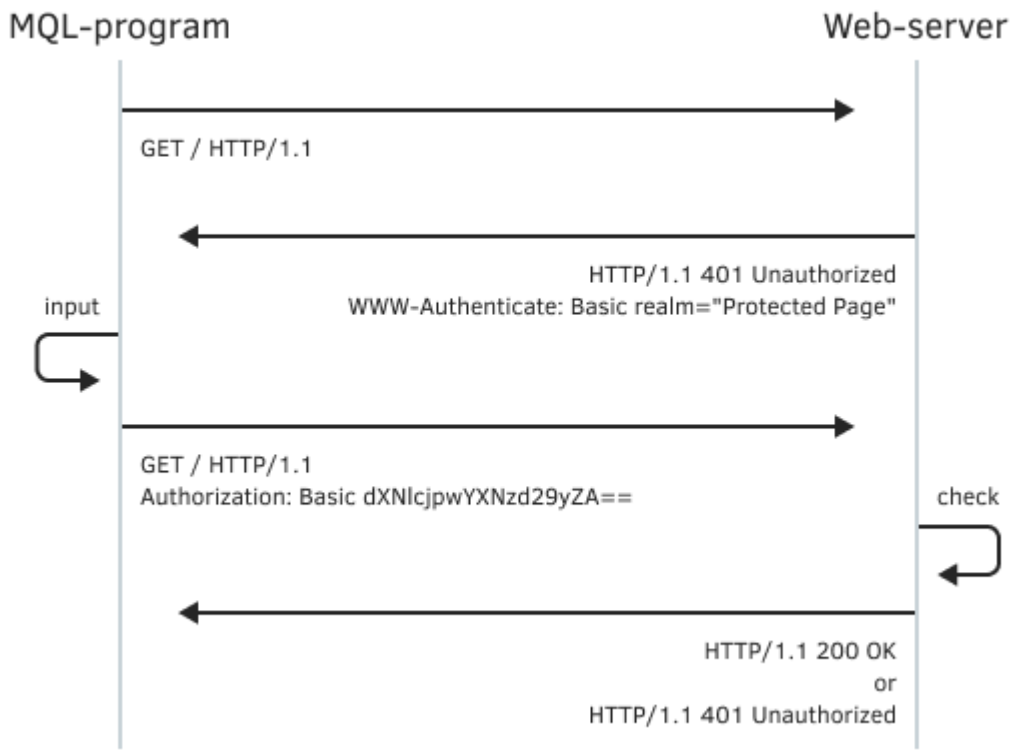
O este:

```
WWW-Authenticate:Digest realm="DemoDigestAuth",qop="auth", »
» nonce="cuFAuHbb5UDvtFGkZEb2mNxjqEG/DjDr",opaque="fyNjGC4x8Zgt830PpzbXRvoqExsZeQSDZ
```

La primera de ellas es la más sencilla y menos segura, por lo que prácticamente no se utiliza: se ofrece en el libro por lo fácil que es aprenderla en una primera fase. El resultado final de su trabajo es generar la siguiente petición HTTP en respuesta a una solicitud del servidor añadiendo un encabezado especial:

```
Authorization: Basic dXNlcjpwYXNzd29yZA==
```

Aquí, la palabra clave «Basic» va seguida de la cadena codificada en Base64 «user:password» con el nombre de usuario y la contraseña reales, y el carácter ':' se inserta a continuación «tal cual» como bloque de enlace. En la imagen se muestra más claramente el proceso de interacción:



Esquema sencillo de autorización en un servidor web

El esquema de autorización *Digest* se considera más avanzado. En este caso, el servidor proporciona información adicional en su respuesta:

- *realms* - el nombre del sitio (zona del sitio) en el que se realiza la entrada
- *qop* - una variación del método Digest (sólo consideraremos «auth»)
- *nonce* - una cadena aleatoria que se utilizará para generar los datos de autorización
- *opaque* - una cadena aleatoria que devolveremos «tal cual» en nuestros encabezados
- *algorithm* - un nombre opcional del algoritmo de hashing, MD5 se asume de manera predeterminada

Para la autorización, debe realizar los siguientes pasos:

1. Genere su propia cadena aleatoria *cnonce*
2. Inicialice o incremente su contador de solicitudes *nc*
3. Calcule *hash1 = MD5(user:realm:password)*
4. Calcule *hash2 = MD5(method:uri)*, aquí *uri* es la ruta y el nombre de la página
5. Calcule *response = MD5(hash1:nonce:nc:cnonce:qop:hash2)*

Después, el cliente puede repetir la solicitud al servidor, añadiendo una línea como ésta a sus encabezados:

```
Authorization: Digest username="user",realm="realm",nonce="...", »
»  uri="/path/to/page",qop=auth,nc=00000001,cnonce="...",response="...",opaque="..."
```

Como el servidor tiene la misma información que el cliente, podrá repetir los cálculos y comprobar que los hashes coinciden.

Añadamos variables a los parámetros del script para introducir el nombre de usuario y la contraseña. De manera predeterminada, el parámetro *Address* incluye la dirección del endpoint *digest-auth*, que puede solicitar autorización con los parámetros *qop* («auth»), login («test») y password («pass»). Todo esto es opcional en la ruta del endpoint (puede probar otros métodos y credenciales de usuario, como por ejemplo así: «https://httpbin.org/digest-auth/auth-int/mql5client/mql5password»).

```
const string Method = "GET";
input string Address = "https://httpbin.org/digest-auth/auth/test/pass";
input string Headers = "User-Agent: noname";
input int Timeout = 5000;
input string User = "test";
input string Password = "pass";
input bool DumpDataToFiles = true;
```

Hemos especificado un nombre de navegador ficticio en el parámetro *Headers* para demostrar la función.

En la función *OnStart*, añadimos el procesamiento del código HTTP 401. Si no se proporciona un nombre de usuario y una contraseña, no podremos continuar.

```
void OnStart()
{
   string parts[];
   URL::parse(Address, parts);
   uchar data[], result[];
   string response;
   int code = PRTF(WebRequest(Method, Address, Headers, Timeout, data, result, respon
   Print(response);
   if(code == 401)
   {
      if(StringLen(User) == 0 || StringLen(Password) == 0)
      {
         Print("Credentials required");
         return;
      }
      ...
```

El siguiente paso es analizar los encabezados recibidos del servidor. Por comodidad, hemos escrito la clase *HttpHeader* (*HttpHeader.mqh*). El texto completo se pasa a su constructor, así como el separador de elementos (en este caso, el carácter de nueva línea '\n') y el carácter utilizado entre el nombre y el valor dentro de cada elemento (en este caso, los dos puntos ':'). Durante su creación, el objeto «analiza» el texto, y luego los elementos se ponen a disposición a través del operador sobrecargado [], siendo el tipo de su argumento una cadena. Como resultado, podemos comprobar si existe un requisito de autorización con el nombre «WWW-Authenticate». Si tal elemento existe en el texto y es igual a «Basic», formamos el encabezado de respuesta «Authorization: Basic» con el nombre de usuario y la contraseña codificados en Base64.

```
      code = -1;
      HttpHeader header(response, '\n', ':');
      const string auth = header["WWW-Authenticate"];
      if(StringFind(auth, "Basic ") == 0)
      {
         string Header = Headers;
         if(StringLen(Header) > 0) Header += "\r\n";
         Header += "Authorization: Basic ";
         Header += HttpHeader::hash(User + ":" + Password, CRYPT_BASE64);
         PRTF(Header);
         code = PRTF(WebRequest(Method, Address, Header, Timeout, data, result, respo
         Print(response);
      }
      ...
```

Para la autorización Digest, todo es un poco más complicado, siguiendo el algoritmo descrito anteriormente.

```
else if(StringFind(auth, "Digest ") == 0)
{
   HttpHeader params(StringSubstr(auth, 7), ',', '=');
   string realm = HttpHeader::unquote(params["realm"]);
   if(realm != NULL)
   {
      string qop = HttpHeader::unquote(params["qop"]);
      if(qop == "auth")
      {
         string h1 = HttpHeader::hash(User + ":" + realm + ":" + Password);
         string h2 = HttpHeader::hash(Method + ":" + parts[URL_PATH]);
         string nonce = HttpHeader::unquote(params["nonce"]);
         string counter = StringFormat("%08x", 1);
         string cnonce = StringFormat("%08x", MathRand());
         string h3 = HttpHeader::hash(h1 + ":" + nonce + ":" + counter + ":" +
            cnonce + ":" + qop + ":" + h2);

         string Header = Headers;
         if(StringLen(Header) > 0) Header += "\r\n";
         Header += "Authorization: Digest ";
         Header += "username=\"" + User + "\",";
         Header += "realm=\"" + realm + "\",";
         Header += "nonce=\"" + nonce + "\",";
         Header += "uri=\"" + parts[URL_PATH] + "\",";
         Header += "qop=" + qop + ",";
         Header += "nc=" + counter + ",";
         Header += "cnonce=\"" + cnonce + "\",";
         Header += "response=\"" + h3 + "\",";
         Header += "opaque=" + params["opaque"] + "";
         PRTF(Header);
         code = PRTF(WebRequest(Method, Address, Header, Timeout, data, result,
         Print(response);
      }
   }
}
```

El método estático *HttpHeader::hash* obtiene una cadena con una representación hash hexadecimal (por defecto MD5) para todas las cadenas compuestas requeridas. A partir de estos datos se forma el encabezado para la siguiente llamada a *WebRequest*. El método estático *HttpHeader::unquote* elimina las comillas.

El resto del script se mantuvo sin cambios. Una petición HTTP repetida puede tener éxito, y entonces obtendremos el contenido de la página segura, o se denegará la autorización, y el servidor escribirá algo como «Acceso denegado».

Dado que los parámetros predeterminados contienen los valores correctos («/digest-auth/auth/test/pass» corresponde al usuario «test» y a la contraseña «pass»), deberíamos obtener el siguiente resultado al ejecutar el script (se registran todos los pasos y datos principales).

```
WebRequest(Method,Address,Headers,Timeout,data,result,response)=401 / ok
Date: Fri, 22 Jul 2022 10:45:56 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 0
Connection: keep-alive
Server: gunicorn/19.9.0
WWW-Authenticate: Digest realm="me@kennethreitz.com" »
»  nonce="87d28b529a7a8797f6c3b81845400370", qop="auth",
»  opaque="4cb97ad7ea915a6d24cf1ccbf6feeaba", algorithm=MD5, stale=FALSE
...
```

La primera llamada a *WebRequest* ha finalizado con el código 401, y entre los encabezados de respuesta hay una solicitud de autorización («WWW-Authenticate») con los parámetros requeridos. A partir de ellos, calculamos la respuesta correcta y preparamos los encabezados para una nueva solicitud.

```
Header=User-Agent: noname
Authorization: Digest username="test",realm="me@kennethreitz.com" »
»  nonce="87d28b529a7a8797f6c3b81845400370",uri="/digest-auth/auth/test/pass",
»  qop=auth,nc=00000001,cnonce="00001c74",
»  response="c09e52bca9cc90caf9a707d046b567b2",opaque="4cb97ad7ea915a6d24cf1ccbf6feea
...
```

La segunda solicitud devuelve 200 y una carga útil que escribimos en el archivo.

```
WebRequest(Method,Address,Header,Timeout,data,result,response)=200 / ok
Date: Fri, 22 Jul 2022 10:45:56 GMT
Content-Type: application/json
Content-Length: 47
Connection: keep-alive
Server: gunicorn/19.9.0
...
Got data: 47 bytes
Saving httpbin.org/digest-auth/auth/test/pass
FileSave(filename,result)=true / ok
```

Dentro del archivo *MQL5/Files/httpbin.org/digest-auth/auth/test/pass* se encuentra la «página web», o mejor dicho, el estado de la autorización correcta en formato JSON.

```
{
  "authenticated": true,
  "user": "test"
}
```

Si especifica una contraseña incorrecta al ejecutar el script, recibiremos una respuesta vacía del servidor y el archivo no se escribirá.

> Utilizando *WebRequest* entramos automáticamente en el campo de los sistemas de software distribuido, en los que el correcto funcionamiento depende no sólo de nuestro código MQL cliente, sino también del servidor (por no hablar de los enlaces intermedios, como un proxy). Por lo tanto, hay que estar preparado para que se produzcan errores ajenos. En concreto, en el momento de escribir el libro en la implementación del endpoint *digest-auth* en *httpbin.org*, había un problema: el nombre de usuario introducido en la solicitud no participaba en la comprobación de autorización, por lo que cualquier inicio de sesión conducía a una autorización correcta si se especificaba la

contraseña correcta. Aun así, para comprobar nuestro script, utilice otros servicios, por ejemplo, algo como *httpbingo.org/digest-auth/auth/test/pass*. También puede configurar el script a la dirección *jigsaw.w3.org/HTTP/Digest/* - espera el login/contraseña «guest»/«invitado».

En la práctica, la mayoría de los sitios implementan la autorización utilizando formularios incrustados directamente en las páginas web: dentro del código HTML, son esencialmente la etiqueta contenedora *form* con un conjunto de campos de entrada, que son rellenados por el usuario y enviados al servidor utilizando el método POST. A este respecto, tiene sentido analizar el ejemplo del envío de un formulario. Sin embargo, antes de entrar en detalles, conviene destacar una técnica más.

La cuestión es que la interacción entre el cliente y el servidor suele ir acompañada de un cambio en el estado tanto del cliente como del servidor. Utilizando el ejemplo de la autorización, esto se puede entender más claramente, ya que antes de la autorización el usuario era desconocido para el sistema, y después de eso, el sistema ya conoce el inicio de sesión y puede aplicar la configuración preferida para el sitio (por ejemplo, idioma, color, método de visualización del foro), y también permitir el acceso a aquellas páginas en las que los visitantes no autorizados no pueden entrar (el servidor detiene tales intentos devolviendo el estado HTTP 403, Prohibido).

La compatibilidad y la sincronización del estado consistente de las partes cliente y servidor de una aplicación web distribuida se proporciona utilizando el mecanismo de cookies que implica variables con nombre y sus valores en los encabezados HTTP. El término se remonta a las «galletas de la suerte», porque *cookies* también contiene pequeños mensajes invisibles para el usuario.

Cualquiera de las partes, servidor y cliente, puede añadir *cookie* al encabezado HTTP. El servidor hace esto con una línea como esta:

```
Set-Cookie: name=value; [Domain=domain; Path=path; Expires=date; Max-Age=number_of_se
```

Solo el nombre y el valor son obligatorios y el resto de los atributos son opcionales; los principales son *Domain*, *Path*, *Expires* y *Max age*, pero en situaciones reales hay más.

Una vez recibida dicho encabezado (o varios encabezados), el cliente debe recordar el nombre y el valor de la variable y enviarlos al servidor en todas las solicitudes que se dirijan a los correspondientes *Domain* y *Path* dentro de este dominio hasta la fecha de vencimiento (*Expires* o *Max-Age*).

En una solicitud HTTP saliente de un cliente, las *cookie*s se pasan como una cadena:

```
Cookie: name(№)=value(№) [; name(i)=value(i) ...]opt
```

Aquí, separados por un punto y coma y un espacio, se enumeran todos los pares nombre=valor; son establecidos por el servidor y conocidos por este cliente, coinciden con la solicitud actual por el dominio y la ruta, y no han vencido.

El servidor y el cliente intercambian todas las cookies necesarias con cada solicitud HTTP, por lo que este estilo arquitectónico de sistemas distribuidos se denomina REST (*Representational State Transfer*). Por ejemplo, después de que un usuario se conecte con éxito al servidor, éste establece (a través del encabezado «Set-Cookie:») una «cookie» especial con el identificador del usuario, tras lo cual el navegador web (o, en nuestro caso, un terminal con un programa MQL) la enviará en solicitudes posteriores (añadiendo la línea adecuada al encabezado «Cookie:»).

La función *WebRequest* hace silenciosamente todo este trabajo por nosotros: recoge las cookies de los encabezados entrantes y añade las cookies apropiadas a las solicitudes HTTP salientes.

Las cookies son almacenadas por el terminal y entre sesiones, según su configuración. Para comprobarlo, basta con solicitar dos veces una página web a un sitio que utilice cookies.

> Atención: las cookies se almacenan en relación con el sitio y por lo tanto se sustituyen imperceptiblemente en los encabezados salientes de todos los programas MQL que utilizan *WebRequest* para el mismo sitio.

Para simplificar las solicitudes secuenciales, tiene sentido formalizar las acciones populares en una clase especial *HTTPRequest* (*HTTPRequest.mqh*). En ella almacenaremos encabezados HTTP comunes, que probablemente serán necesarios para todas las solicitudes (por ejemplo, idiomas admitidos, instrucciones para proxies, etc.). Además, también es común un ajuste como el tiempo de espera. Ambas opciones se pasan al constructor del objeto.

```
class HTTPRequest: public HttpCookie
{
protected:
   string common_headers;
   int timeout;

public:
   HTTPRequest(const string h, const int t = 5000):
      common_headers(h), timeout(t) { }
   ...
```

De manera predeterminada, el tiempo de espera está fijado en 5 segundos. El principal método, en cierto sentido universal, de la clase es *request*.

```
   int request(const string method, const string address,
      string headers, const uchar &data[], uchar &result[], string &response)
   {
      if(headers == NULL) headers = common_headers;

      ArrayResize(result, 0);
      response = NULL;
      Print(">>> Request:\n", method + " " + address + "\n" + headers);

      const int code = PRTF(WebRequest(method, address, headers, timeout, data, resul
      Print("<<< Response:\n", response);
      return code;
   }
};
```

Vamos a describir un par de métodos más para consultas de tipos específicos.

Las solicitudes GET sólo utilizan encabezados y el cuerpo del documento (a menudo se utiliza el término *payload*) está vacío.

```
   int GET(const string address, uchar &result[], string &response,
      const string custom_headers = NULL)
   {
      uchar nodata[];
      return request("GET", address, custom_headers, nodata, result, response);
   }
```

En las solicitudes POST suele haber una carga útil.

```
int POST(const string address, const uchar &payload[],
   uchar &result[], string &response, const string custom_headers = NULL)
{
   return request("POST", address, custom_headers, payload, result, response);
}
```

Los formularios pueden enviarse en distintos formatos. El más sencillo es «application/x-www-form-urlencoded». Implica que la carga útil será una cadena (quizá muy larga, ya que las especificaciones no imponen restricciones, y todo depende de la configuración de los servidores web). Para tales formularios, proporcionaremos una sobrecarga más conveniente del método POST con el parámetro de cadena de carga útil.

```
int POST(const string address, const string payload,
   uchar &result[], string &response, const string custom_headers = NULL)
{
   uchar bytes[];
   const int n = StringToCharArray(payload, bytes, 0, -1, CP_UTF8);
   ArrayResize(bytes, n - 1); // remove terminal zero
   return request("POST", address, custom_headers, bytes, result, response);
}
```

Escribamos un sencillo script para probar el motor web de nuestro cliente *WebRequestCookie.mq5*. Su tarea consistirá en solicitar la misma página web dos veces: la primera vez el servidor ofrecerá con toda probabilidad instalar sus cookies, y luego serán sustituidas automáticamente en la segunda solicitud. En los parámetros de entrada, especifique la dirección de la página para la prueba: deje que sea el sitio web *mql5.com*. También simularemos los encabezados por defecto mediante la cadena «User-Agent» corregida.

```
input string Address = "https://www.mql5.com";
input string Headers = "User-Agent: Mozilla/5.0 (Windows NT 10.0) Chrome/103.0.0.0";
```

En la función principal del script, describimos el objeto *HTTPRequest* y ejecutamos dos peticiones GET en un bucle.

> ¡Atención! Esta prueba funciona bajo el supuesto de que los programas MQL aún no han visitado el sitio www.mql5.com y no han recibido cookies del mismo. Después de ejecutar el script una vez, las cookies permanecerán en la caché del terminal, y será imposible reproducir el ejemplo: en ambas iteraciones del bucle, obtendremos las mismas entradas de registro.
>
> No olvide añadir el dominio «www.mql5.com» a la lista de permitidos en la configuración del terminal.

```
void OnStart()
{
   uchar result[];
   string response;
   HTTPRequest http(Headers);

   for(int i = 0; i < 2; ++i)
   {
      if(http.GET(Address, result, response) > -1)
      {
         if(ArraySize(result) > 0)
         {
            PrintFormat("Got data: %d bytes", ArraySize(result));
            if(i == 0) // show the beginning of the document only the first time
            {
               const string s = CharArrayToString(result, 0, 160, CP_UTF8);
               int j = -1, k = -1;
               while((j = StringFind(s, "\r\n", j + 1)) != -1) k = j;
               Print(StringSubstr(s, 0, k));
            }
         }
      }
   }
}
```

La primera iteración del bucle generará las siguientes entradas de registro (con abreviaturas):

```
>>> Request:
GET https://www.mql5.com
User-Agent: Mozilla/5.0 (Windows NT 10.0) Chrome/103.0.0.0
WebRequest(method,address,headers,timeout,data,result,response)=200 / ok
<<< Response:
Server: nginx
Date: Sun, 24 Jul 2022 19:04:35 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: keep-alive
Cache-Control: no-cache,no-store
Content-Encoding: gzip
Expires: -1
Pragma: no-cache
Set-Cookie: sid=CfDJ8O2AwC...Ne2yP5QXpPKA2; domain=.mql5.com; path=/; samesite=lax; h
Vary: Accept-Encoding
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
Content-Security-Policy: default-src 'self'; script-src 'self' ...
Generate-Time: 2823
Agent-Type: desktop-ru-en
X-Cache-Status: MISS
Got data: 184396 bytes

<!DOCTYPE html>
<html lang="ru">
<head>
  <meta http-equiv="X-UA-Compatible" content="IE=edge" />
```

Hemos recibido una nueva cookie con el nombre *sid*. Para comprobar su eficacia, se pasa a ver la segunda parte del registro, para la segunda iteración del bucle.

```
>>> Request:
GET https://www.mql5.com
User-Agent: Mozilla/5.0 (Windows NT 10.0) Chrome/103.0.0.0
WebRequest(method,address,headers,timeout,data,result,response)=200 / ok
<<< Response:
Server: nginx
Date: Sun, 24 Jul 2022 19:04:36 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: keep-alive
Cache-Control: no-cache, no-store, must-revalidate, no-transform
Content-Encoding: gzip
Expires: -1
Pragma: no-cache
Vary: Accept-Encoding
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
Content-Security-Policy: default-src 'self'; script-src 'self' ...
Generate-Time: 2950
Agent-Type: desktop-ru-en
X-Cache-Status: MISS
```

Por desgracia, aquí no vemos los encabezados salientes completos formados dentro de *WebRequest*, pero la instancia de la cookie que se está enviando al servidor usando el encabezado «Cookie:» está probada por el hecho de que el servidor en su segunda respuesta ya no pide establecerla.

En teoría, esta cookie simplemente identifica al visitante (como hacen la mayoría de los sitios) pero no significa su autorización. Por lo tanto, volvamos al ejercicio de enviar el formulario de forma general, es decir, en el futuro la tarea privada de introducir un nombre de usuario y una contraseña.

Recordemos que para enviar el formulario podemos utilizar el método POST con un parámetro de cadena *payload*. El principio de la preparación de datos según la norma «x-www-form-urlencoded» es que las variables con nombre y sus valores se escriben en una línea continua (algo parecido a las cookies).

```
name(№)=value(№)[&name(i)=value(i)...]opt
```

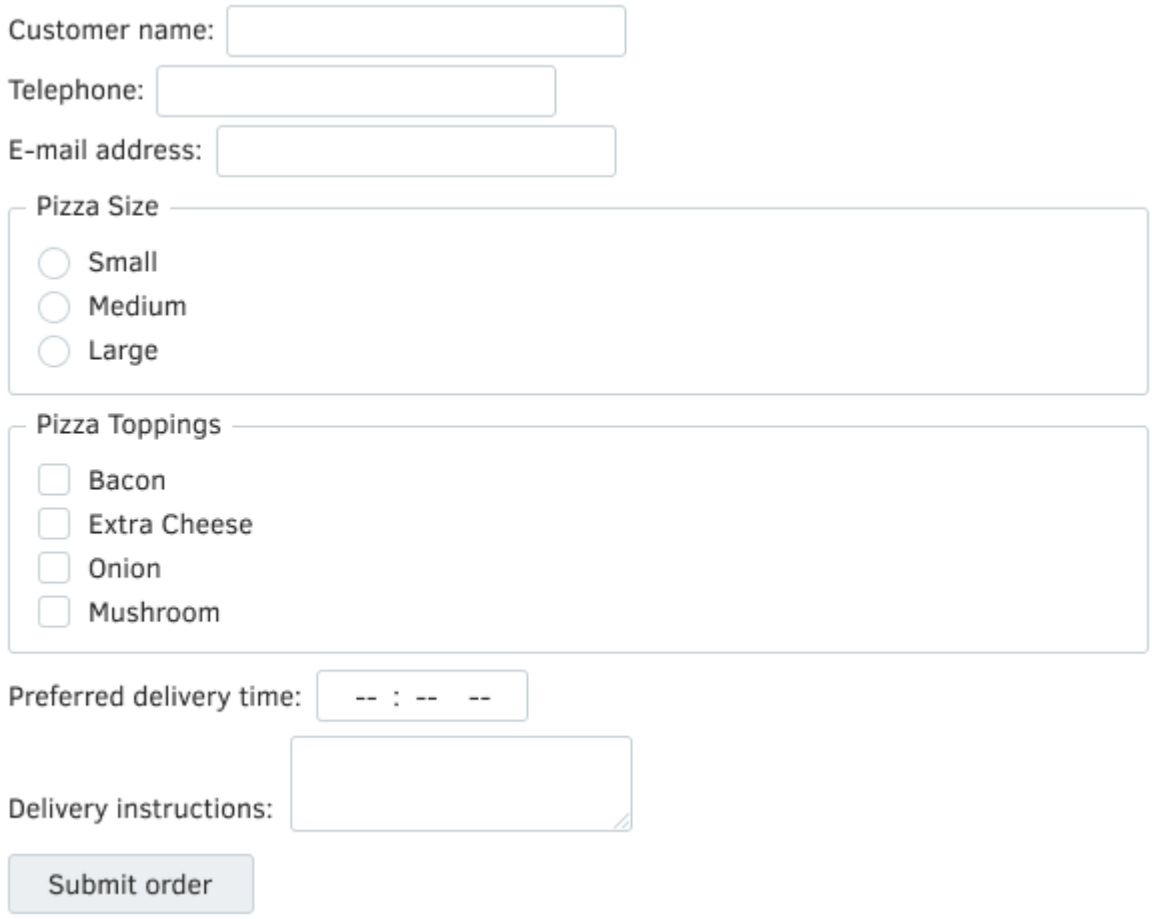
El nombre y el valor se conectan con el signo '=', y los pares se unen mediante el carácter ampersand '&'. Puede que falte el valor. Por ejemplo:

```
Name=John&Age=33&Education=&Address=
```

Es importante tener en cuenta que, desde un punto de vista técnico, esta cadena debe convertirse según el algoritmo antes de enviar *urlencode* (de ahí viene el nombre del formato); sin embargo, *WebRequest* hace esta transformación por nosotros.

Los nombres de las variables vienen determinados por el formulario web (el contenido de la etiqueta *form* en una página web) o la lógica de la aplicación web; en cualquier caso, el servidor web debe ser capaz de interpretar los nombres y valores. Por lo tanto, con el fin de familiarizarnos con la tecnología, necesitamos un servidor de prueba con un formulario.

El formulario de la prueba está disponible en *https://httpbin.org/forms/post*. Se trata de un diálogo para pedir una pizza.



Formulario web de prueba

Su estructura interna y su comportamiento se describen en el código HTML que aparece más abajo. En él, nos interesan principalmente las etiquetas *input*, que establecen las variables esperadas por el servidor. Además, hay que prestar atención al atributo *action* de la etiqueta *form*, ya que define la dirección a la que debe enviarse la petición POST, y en este caso es «/post», que junto con el dominio da la cadena «httpbin.org/post». Esto es lo que utilizaremos en el programa MQL.

```
<!DOCTYPE html>
<html>
  <body>
  <form method="post" action="/post">
    <p><label>Customer name: <input name="custname"></label></p>
    <p><label>Telephone: <input type=tel name="custtel"></label></p>
    <p><label>E-mail address: <input type=email name="custemail"></label></p>
    <fieldset>
      <legend> Pizza Size </legend>
      <p><label> <input type=radio name=size value="small"> Small </label></p>
      <p><label> <input type=radio name=size value="medium"> Medium </label></p>
      <p><label> <input type=radio name=size value="large"> Large </label></p>
    </fieldset>
    <fieldset>
      <legend> Pizza Toppings </legend>
      <p><label> <input type=checkbox name="topping" value="bacon"> Bacon </label></p
      <p><label> <input type=checkbox name="topping" value="cheese"> Extra Cheese </l
      <p><label> <input type=checkbox name="topping" value="onion"> Onion </label></p
      <p><label> <input type=checkbox name="topping" value="mushroom"> Mushroom </lab
    </fieldset>
    <p><label>Preferred delivery time: <input type=time min="11:00" max="21:00" step=
    <p><label>Delivery instructions: <textarea name="comments"></textarea></label></p
    <p><button>Submit order</button></p>
  </form>
  </body>
</html>
```

En el script *WebRequestForm.mq5* hemos preparado variables de entrada similares para que sean especificadas por el usuario antes de ser enviadas al servidor.

```
input string Address = "https://httpbin.org/post";

input string Customer = "custname=Vincent Silver";
input string Telephone = "custtel=123-123-123";
input string Email = "custemail=email@address.org";
input string PizzaSize = "size=small"; // PizzaSize (small,medium,large)
input string PizzaTopping = "topping=bacon"; // PizzaTopping (bacon,cheese,onion,mush
input string DeliveryTime = "delivery=";
input string Comments = "comments=";
```

Las cadenas ya configuradas se muestran sólo para probarlas con un clic: puede sustituirlas por las suyas propias, pero tenga en cuenta que dentro de cada cadena sólo debe editarse el valor a la derecha de '=', y debe conservarse el nombre a la izquierda de '=' (los nombres desconocidos serán ignorados por el servidor).

En la función *OnStart*, describimos el encabezado HTTP «Content-Type:» y preparamos una cadena concatenada con todas las variables.

```
void OnStart()
{
   uchar result[];
   string response;
   string header = "Content-Type: application/x-www-form-urlencoded";
   string form_fields;
   StringConcatenate(form_fields,
      Customer, "&",
      Telephone, "&",
      Email, "&",
      PizzaSize, "&",
      PizzaTopping, "&",
      DeliveryTime, "&",
      Comments);
   HTTPRequest http;
   if(http.POST(Address, form_fields, result, response) > -1)
   {
      if(ArraySize(result) > 0)
      {
         PrintFormat("Got data: %d bytes", ArraySize(result));
         // NB: UTF-8 is implied for many content-types,
 // but some may be different, analyze the response headers
         Print(CharArrayToString(result, 0, WHOLE_ARRAY, CP_UTF8));
      }
   }
}
```

A continuación ejecutamos el método POST y registramos la respuesta del servidor. He aquí un ejemplo de resultado:

```
>>> Request:
POST https://httpbin.org/post
Content-Type: application/x-www-form-urlencoded
WebRequest(method,address,headers,timeout,data,result,response)=200 / ok
<<< Response:
Date: Mon, 25 Jul 2022 08:41:41 GMT
Content-Type: application/json
Content-Length: 780
Connection: keep-alive
Server: gunicorn/19.9.0
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true

Got data: 721 bytes
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "comments": "",
    "custemail": "email@address.org",
    "custname": "Vincent Silver",
    "custtel": "123-123-123",
    "delivery": "",
    "size": "small",
    "topping": "bacon"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Accept-Language": "ru,en",
    "Content-Length": "127",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "MetaTrader 5 Terminal/5.3333 (Windows NT 10.0; x64)",
    "X-Amzn-Trace-Id": "Root=1-62de5745-25bd1d823a9609f01cff04ad"
  },
  "json": null,
  "url": "https://httpbin.org/post"
}
```

El servidor de prueba acusa recibo de los datos como copia JSON. En la práctica, el servidor, por supuesto, no devolverá los datos en sí, sino que simplemente informará de un estado de éxito y posiblemente redirigirá a otra página web en la que los datos hayan tenido efecto (por ejemplo, mostrar el número del pedido).

Con la ayuda de este tipo de solicitudes POST, pero de menor tamaño, se suele realizar también la autorización. Sin embargo, a decir verdad, la mayoría de los servicios web complican este proceso en exceso de forma deliberada por motivos de seguridad y obligan a calcular primero varias sumas hash a partir de los datos del usuario. Las API públicas especialmente desarrolladas suelen tener descripciones de todos los algoritmos necesarios en la documentación, pero no siempre es así. En concreto, no

podremos conectarnos utilizando *WebRequest* en *mql5.com* porque el sitio no tiene una interfaz de programación abierta.

Cuando envíe solicitudes a servicios web, respete siempre la norma de no exceder la frecuencia de las solicitudes: normalmente, cada servicio especifica sus propios límites, y la violación de los mismos conllevará el consiguiente bloqueo de su programa cliente, cuenta o dirección IP.

7.5.5 Establecer y romper una conexión de socket de red

En las secciones anteriores nos hemos familiarizado con las funciones de red de alto nivel de MQL5: cada una de ellas proporciona soporte para un protocolo de aplicación específico. Por ejemplo, SMTP se utiliza para enviar correos electrónicos (*SendMail*), FTP para transferir archivos (*SendFTP*) y HTTP permite recibir documentos web (*WebRequest*). Todas las normas mencionadas se basan en una capa inferior, la de transporte TCP (Transmission Control Protocol). No es el último de la jerarquía, ya que también los hay inferiores, pero no los discutiremos aquí.

La implementación estándar de protocolos de aplicación esconde muchos matices técnicos en su interior y elimina la necesidad de que el programador siga rutinariamente las especificaciones durante horas. Sin embargo, carece de flexibilidad y no tiene en cuenta las funciones avanzadas que incrustadas en los estándares. Por lo tanto, a veces es necesario programar la comunicación de red a nivel TCP, es decir, a nivel de socket.

Un socket puede considerarse análogo a un archivo en un disco: un socket también se describe mediante un descriptor entero por el que se pueden leer o escribir datos, pero esto ocurre en una infraestructura de red distribuida. A diferencia de los archivos, el número de sockets en un ordenador es limitado, por lo que el descriptor de socket debe solicitarse previamente al sistema antes de asociarlo a un recurso de red (dirección, URL). Digamos también antes de nada que el acceso a la información a través de un socket es streaming, es decir, que es imposible «rebobinar» un determinado «puntero» hasta el principio, como en un archivo.

Los hilos de escritura y lectura no se cruzan, pero pueden afectar a futuros datos de lectura o escritura, ya que la información transmitida suele ser interpretada por los servidores y programas cliente como comandos de control. Los estándares de los protocolos definen si un flujo contiene comandos o datos.

La función *SocketCreate* permite la creación de un descriptor de socket «vacío» en MQL5.

```
int SocketCreate(uint flags = 0)
```

Su único parámetro se reserva para el futuro con el fin de especificar el patrón de bits de las banderas que determinan el modo del socket, pero por el momento sólo se admite una bandera «stub»: SOCKET_DEFAULT corresponde al modo actual y puede omitirse. A nivel de sistema, esto equivale a un socket en modo de bloqueo (esto puede interesar a los programadores de redes).

Si tiene éxito, la función devuelve el manejador del socket. En caso contrario, devuelve INVALID_HANDLE.

Se puede crear un máximo de 128 sockets desde un programa MQL. Cuando se supera el límite, se registra el error 5271 (ERR_NETSOCKET_TOO_MANY_OPENED) en *_LastError*.

Una vez abierto el socket, debe asociarse a una dirección de red.

bool SocketConnect(int socket, const string server, uint port, uint timeout)

La función *SocketConnect* establece una conexión de socket con el servidor en la dirección y el puerto especificados (por ejemplo, los servidores web suelen funcionar en los puertos 80 o 443 para HTTP y HTTPS, respectivamente, y SMTP en el puerto 25). La dirección puede ser un nombre de dominio o una dirección IP.

El parámetro *timeout* permite establecer un tiempo de espera en milisegundos para esperar una respuesta del servidor.

La función devuelve una señal de conexión correcta (*true*) o de error (*false*). El código de error se escribe en *_LastError*, por ejemplo, 5272 (ERR_NETSOCKET_CANNOT_CONNECT).

> Tenga en cuenta que la dirección de conexión debe añadirse a la lista de direcciones permitidas en la configuración del terminal (cuadro de diálogo *Service -> Settings -> Advisors*).

Cuando haya terminado de trabajar con la red, debe liberar el socket con *SocketClose*.

bool SocketClose(const int socket)

La función *SocketClose* cierra el socket por su asa, abierta anteriormente mediante la función *SocketCreate*. Si el socket estaba previamente conectado a través de *SocketConnect*, la conexión se romperá.

La función también devuelve un indicador de éxito (*true*) o de error (*false*). En particular, cuando se pasa un manejador no válido a *_LastError*, se registra el error 5270 (ERR_NETSOCKET_INVALIDHANDLE).

Recordemos que todas las funciones de esta sección y de las siguientes están prohibidas en los indicadores: allí, un intento de trabajar con sockets dará como resultado el error 4014 (ERR_FUNCTION_NOT_ALLOWED, «No se permite llamar a la función del sistema»).

Consideremos un ejemplo introductorio, el script *SocketConnect.mq5*. En los parámetros de entrada puede especificar la dirección y el puerto del servidor. Se supone que debemos empezar las pruebas con servidores web normales como mql5.com.

```
input string Server = "www.mql5.com";
input uint Port = 443;
```

En la función *OnStart* simplemente creamos un socket y lo vinculamos a un recurso de red.

```
void OnStart()
{
   PRTF(Server);
   PRTF(Port);
   const int socket = PRTF(SocketCreate());
   if(PRTF(SocketConnect(socket, Server, Port, 5000)))
   {
      PRTF(SocketClose(socket));
   }
}
```

Si todos los ajustes del terminal son correctos y está conectado a Internet, obtendremos el siguiente «informe».

```
Server=www.mql5.com / ok
Port=443 / ok
SocketCreate()=1 / ok
SocketConnect(socket,Server,Port,5000)=true / ok
SocketClose(socket)=true / ok
```

7.5.6 Comprobar el estado del socket

Cuando se trabaja con un socket, se hace necesario comprobar su estado porque las redes distribuidas no son tan fiables como un sistema de archivos. En concreto, la conexión puede perderse por un motivo u otro. La función *SocketIsConnected* permite averiguarlo.

bool SocketIsConnected(const int socket)

La función comprueba si el socket con el manejador especificado (obtenido de *SocketCreate*) está conectado a su recurso de red (especificado en *Conexión de socket*) y devuelve *true* en caso de éxito.

Otra función, *SocketIsReadable*, permite saber si hay datos para leer en el búfer del sistema asociado al socket. Esto significa que el ordenador, al que nos conectamos en la dirección de red, nos envió (y puede seguir enviándonos) datos.

uint SocketIsReadable(const int socket)

La función devuelve el número de bytes que se pueden leer del socket. En caso de error, se devuelve 0.

> Los programadores familiarizados con las API del sistema de sockets de Windows/Linux saben que un valor 0 también puede ser un estado normal cuando no hay datos entrantes en el búfer interno del socket. Sin embargo, esta función se comporta de manera diferente en MQL5. Con un búfer de socket del sistema vacío, devuelve especulativamente 1, aplazando la comprobación real de la disponibilidad de datos hasta la siguiente llamada a una de las funciones de lectura. En concreto, esta situación con un resultado ficticio de 1 byte se produce, por regla general, la primera vez que se llama a una función en un socket cuando el búfer interno receptor aún está vacío.

Al ejecutar esta función puede producirse un error, lo que significa que la conexión establecida a través de *SocketConnect* se ha roto (en *_LastError* obtendremos el código 5273, ERR_NETSOCKET_IO_ERROR).

La función *SocketIsReadable* es útil en programas diseñados para la lectura «no bloqueante» de datos utilizando *SocketRead*. La cuestión es que la función *SocketRead*, cuando no haya datos en el búfer de recepción, esperará su llegada, suspendiendo la ejecución del programa (por el valor de timeout especificado).

Por otro lado, una lectura de bloqueo es más fiable en el sentido de que su programa se «despertará» en cuanto lleguen nuevos datos, pero la comprobación de su presencia con *SocketIsReadable* debe hacerse periódicamente, en función de algunos otros eventos (normalmente, en un temporizador o en un bucle).

Debe prestarse especial atención al utilizar la función *SocketIsReadable* en Modo seguro TLS. La función devuelve la cantidad de datos «en bruto», que en el modo TLS es un bloque cifrado. Si los datos «en bruto» aún no se han acumulado en el tamaño del bloque de descifrado, la llamada posterior de la función de lectura *SocketTlsRead* bloqueará la ejecución del programa, a la espera del fragmento que falta. Si los datos «en bruto» ya contienen un bloque listo para el descifrado, la función de lectura devolverá menos bytes descifrados que el número de bytes «brutos». En este sentido, con TLS activado, se recomienda utilizar siempre la función *SocketIsReadable* junto con *SocketTlsReadAvailable*.

De lo contrario, el comportamiento del programa diferirá de lo esperado. Por desgracia, MQL5 no proporciona la función *SocketTlsIsReadable*, que es compatible con el modo TLS y no impone las convenciones descritas.

La función similar *SocketIsWritable* comprueba si se puede escribir en el socket dado en el momento actual.

bool SocketIsWritable(const int socket)

La función devuelve un indicador de éxito (*true*) o de error (*false*). En este último caso, la conexión establecida a través de *SocketConnect* se romperá.

He aquí un sencillo script *SocketIsConnected.mq5* para probar las funciones. En los parámetros de entrada, tendremos la oportunidad de introducir la dirección y el puerto.

```
input string Server = "www.mql5.com";
input uint Port = 443;
```

En el manejador *OnStart* creamos un socket, nos conectamos al sitio y empezamos a comprobar el estado del socket en un bucle. Después de la segunda iteración, cerramos forzosamente el socket, y esto debería conducir a una salida del bucle.

```
void OnStart()
{
   PRTF(Server);
   PRTF(Port);
   const int socket = PRTF(SocketCreate());
   if(PRTF(SocketConnect(socket, Server, Port, 5000)))
   {
      int i = 0;
      while(PRTF(SocketIsConnected(socket)) && !IsStopped())
      {
         PRTF(SocketIsReadable(socket));
         PRTF(SocketIsWritable(socket));
         Sleep(1000);
         if(++i >= 2)
         {
            PRTF(SocketClose(socket));
         }
      }
   }
}
```

En el registro se muestran las siguientes entradas:

```
Server=www.mql5.com / ok
Port=443 / ok
SocketCreate()=1 / ok
SocketConnect(socket,Server,Port,5000)=true / ok
SocketIsConnected(socket)=true / ok
SocketIsReadable(socket)=0 / ok
SocketIsWritable(socket)=true / ok
SocketIsConnected(socket)=true / ok
SocketIsReadable(socket)=0 / ok
SocketIsWritable(socket)=true / ok
SocketClose(socket)=true / ok
SocketIsConnected(socket)=false / NETSOCKET_INVALIDHANDLE(5270)
```

7.5.7 Establecer tiempos de espera de envío y recepción de datos para sockets

Dado que las conexiones de red no son fiables, todas las operaciones con funciones *Socket* admiten un ajuste centralizado del tiempo de espera. Si la lectura o el envío de datos no se completa con éxito en el tiempo especificado, la función dejará de intentar realizar la acción correspondiente.

Con la función *SocketTimeouts* puede establecer tiempos de espera para la recepción y el envío de datos.

bool SocketTimeouts(int socket, uint timeout_send, uint timeout_receive)

Ambos tiempos de espera se indican en milisegundos y afectan a todas las funciones del socket especificado a nivel de sistema.

La función *SocketRead* tiene su propio parámetro *timeout*, con el que puede controlar adicionalmente el tiempo de espera durante una llamada concreta de la función *SocketRead*.

SocketTimeouts devuelve *true* si tiene éxito y *false* en caso contrario.

De manera predeterminada, no hay tiempos de espera, lo que significa esperar indefinidamente a que se reciban o envíen todos los datos.

7.5.8 Leer y escribir datos a través de una conexión de socket insegura

Históricamente, los sockets proporcionan transferencia de datos a través de una simple conexión por defecto. La transmisión de datos de forma abierta permite a los medios técnicos analizar todo el tráfico. En los últimos años, las cuestiones de seguridad se han tomado más en serio y, por ello, en casi todas partes se ha implantado la tecnología TLS (Transport Layer Security): proporciona cifrado sobre la marcha de todos los datos entre el remitente y el destinatario. En concreto, para las conexiones a Internet, la diferencia radica en los protocolos HTTP (conexión simple) y HTTPS (segura).

MQL5 proporciona diferentes conjuntos de funciones *Socket* para trabajar con conexiones simples y seguras. En esta sección nos familiarizaremos con el modo simple, y más adelante pasaremos al protegido.

Para leer datos de un socket, utilice la función *SocketRead*.

int SocketRead(int socket, uchar &buffer[], uint maxlen, uint timeout)

El descriptor de socket se obtiene de *SocketCreate* y se conecta a un recurso de red mediante *Conexión de socket*.

El parámetro *buffer* es una referencia al array en el que se leerán los datos. Si el array es dinámico, su tamaño aumenta en función del número de bytes leídos, pero no puede superar INT_MAX (2147483647). Puede limitar el número de bytes leídos en el parámetro *maxlen*. Los datos que no quepan permanecerán en el búfer interno del socket: pueden obtenerse mediante la siguiente llamada *SocketRead*. El valor de *maxlen* debe estar comprendido entre 1 e INT_MAX (2147483647).

El parámetro *timeout* especifica el tiempo (en milisegundos) que hay que esperar para que se complete la lectura. Si no se recibe ningún dato en este tiempo, los intentos terminan y la función sale con el resultado -1.

También se devuelve -1 en caso de error, mientras que el código de error en *_LastError*, por ejemplo, 5273 (ERR_NETSOCKET_IO_ERROR), significa que la conexión establecida a través de *SocketConnect* está interrumpida.

Si tiene éxito, la función devuelve el número de bytes leídos.

Cuando se ajusta el tiempo de espera de lectura a 0, se utiliza el valor por defecto de 120000 (2 minutos).

Para escribir datos en un socket, utilice la función *SocketSend*.

> Por desgracia, los nombres de las funciones *SocketRead* y *SocketSend* no son «simétricos»: la operación inversa para «leer» es «escribir», y para «enviar» es «recibir». Esto puede resultar desconocido para los desarrolladores con experiencia que hayan trabajado con API de redes en otras plataformas.

int SocketSend(int socket, const uchar &buffer[], uint maxlen)

El primer parámetro es un manejador de un socket previamente creado y abierto. Cuando se pasa un manejador no válido, *_LastError* recibe el error 5270 (ERR_NETSOCKET_INVALIDHANDLE). El array *buffer* contiene los datos que se van a enviar, cuyo tamaño se especifica en el parámetro *maxlen* (el parámetro se introdujo para facilitar el envío de parte de los datos de un array fijo).

La función devuelve el número de bytes escritos en el socket en caso de éxito, y -1 en caso de error.

Los errores a nivel de sistema (5273, ERR_NETSOCKET_IO_ERROR) indican una desconexión.

El script *SocketReadWriteHTTP.mq5* demuestra cómo se pueden utilizar los sockets para implementar el trabajo sobre el protocolo HTTP, es decir, solicitar información sobre una página a un servidor web. Esta es una pequeña parte de lo que la función *WebRequest* hace por nosotros «entre bastidores».

Dejemos la dirección por defecto en los parámetros de entrada: el sitio «www.mql5.com». El número de puerto elegido es 80 porque es el valor por defecto para conexiones HTTP no seguras (aunque algunos servidores pueden utilizar un puerto diferente): 81, 8080, etc.). Los puertos reservados para conexiones seguras (en particular, el más popular, 443) no se admiten todavía en este ejemplo. Además, en el parámetro *Server*, es importante introducir el nombre del dominio y no una página concreta, ya que el script sólo puede solicitar la página principal, es decir, la ruta raíz «/».

```
input string Server = "www.mql5.com";
input uint Port = 80;
```

En la función principal del script crearemos un socket y abriremos una conexión en él con los parámetros especificados (el tiempo de espera es de 5 segundos).

```
void OnStart()
{
   PRTF(Server);
   PRTF(Port);
   const int socket = PRTF(SocketCreate());
   if(PRTF(SocketConnect(socket, Server, Port, 5000)))
   {
      ...
   }
}
```

Veamos cómo funciona el protocolo HTTP. El cliente envía las solicitudes en forma de encabezados especialmente diseñados (cadenas con nombres y valores predefinidos), que incluyen, en concreto, la dirección de la página web, y el servidor envía como respuesta la página web completa o el estado de la operación, utilizando también para ello encabezados especiales. El cliente puede solicitar una página web con una solicitud GET, enviar algunos datos con una solicitud POST o comprobar el estado de la página web con una frugal solicitud HEAD. En teoría, hay muchos más métodos HTTP: puede descubrirlos en la especificación del protocolo HTTP.

Así, el script debe generar y enviar un encabezado HTTP a través de la conexión de socket. En su forma más simple, la solicitud HEAD siguiente permite obtener metainformación sobre la página (podríamos sustituir HEAD por GET para solicitar la página completa, pero existen algunas complicaciones; hablaremos de ello más adelante).

```
HEAD / HTTP/1.1
Host: _server_
User-Agent: MetaTrader 5
                                        // <- two newlines in a row \r\n\r\n
```

La barra oblicua después de «HEAD» (u otro método) es la ruta más corta posible en cualquier servidor al directorio raíz, lo que suele dar lugar a que se muestre la página principal. Si quisiéramos una página web concreta, podríamos escribir algo como «GET /en/forum/ HTTP/1.1» y obtener la tabla de contenidos de los foros en inglés de *mql5.com*. Especifique un dominio real en lugar de la cadena «_server_».

Aunque la presencia de «User-Agent:» es opcional, permite al programa «presentarse» al servidor, sin lo cual algunos servidores pueden rechazar la solicitud.

Fíjese en las dos líneas vacías: marcan el final del encabezado. En nuestro script es conveniente formar el título con la siguiente expresión:

```
StringFormat("HEAD / HTTP/1.1\r\nHost: %s\r\n\r\n", Server)
```

Ahora sólo tenemos que enviarlo al servidor. Para ello, hemos escrito una sencilla función *HTTPSend*, que recibe un descriptor de socket y una línea de encabezado.

```
bool HTTPSend(int socket, const string request)
{
   char req[];
   int len = StringToCharArray(request, req, 0, WHOLE_ARRAY, CP_UTF8) - 1;
   if(len < 0) return false;
   return SocketSend(socket, req, len) == len;
}
```

Internamente, convertimos la cadena en un array de bytes y llamamos a *SocketSend*.

A continuación, necesitamos aceptar la respuesta del servidor, para lo cual hemos escrito la función *HTTPRecv*. También espera un descriptor de socket y una referencia a una cadena donde colocar los datos, pero es más complejo.

```
bool HTTPRecv(int socket, string &result, const uint timeout)
{
   char response[];
   int len;         // signed integer needed for error flag -1
   uint start = GetTickCount();
   result = "";

   do
   {
      ResetLastError();
      if(!(len = (int)SocketIsReadable(socket)))
      {
         Sleep(10); // wait for data or timeout
      }
      else          // read the data in the available volume
      if((len = SocketRead(socket, response, len, timeout)) > 0)
      {
         result += CharArrayToString(response, 0, len); // NB: without CP_UTF8 only '
         const int p = StringFind(result, "\r\n\r\n");
         if(p > 0)
         {
            // HTTP header ends with a double newline, use this
            // to make sure the entire header is received
            Print("HTTP-header found");
            StringSetLength(result, p); // cut off the body of the document (in case
            return true;
         }
      }
   }
   while(GetTickCount() - start < timeout && !IsStopped() && !_LastError);

   if(_LastError) PRTF(_LastError);

   return StringLen(result) > 0;
}
```

Aquí estamos comprobando en un bucle la aparición de datos dentro del tiempo de espera especificado y leyéndolos en el búfer *response*. La aparición de un error pone fin al bucle.

Los bytes del búfer se convierten inmediatamente en una cadena y se concatenan en una respuesta completa en la variable *result*. Es importante tener en cuenta que sólo podemos utilizar la función *CharArrayToString* con la codificación por defecto para el encabezado HTTP, ya que en ella sólo se permiten letras latinas y unos pocos caracteres especiales de ANSI.

Para recibir un documento web completo, que, por regla general, tiene codificación UTF-8 (pero potencialmente tiene otra no latina, que se indica sólo en el encabezado HTTP), será necesario un procesamiento más complicado: primero, hay que recoger todos los bloques enviados en un búfer común y luego convertirlo todo en una cadena que indique CP_UTF8 (de lo contrario, cualquier carácter codificado en dos bytes puede «cortarse» al ser enviado, y llegará en bloques diferentes; por eso no podemos esperar un flujo de bytes UTF-8 correcto en fragmento individual). Mejoraremos este ejemplo en las secciones siguientes.

Teniendo las funciones *HTTPSend* y *HTTPRecv*, completamos el código *OnStart*.

```
void OnStart()
{
      ...
      if(PRTF(HTTPSend(socket, StringFormat("HEAD / HTTP/1.1\r\nHost: %s \r\n"
         "User-Agent: MetaTrader 5\r\n\r\n", Server))))
      {
         string response;
         if(PRTF(HTTPRecv(socket, response, 5000)))
         {
            Print(response);
         }
      }
      ...
}
```

En el encabezado HTTP recibido del servidor pueden resultar interesantes las siguientes líneas:

- 'Content-Length:' - la longitud total del documento en bytes
- 'Content-Language:' - idioma del documento (por ejemplo, «de-DE, ru»)
- 'Content-Type:' - codificación del documento (por ejemplo, «text/html; charset=UTF-8»)
- 'Last-Modified:' - la hora de la última modificación del documento, para no descargar lo que ya está (en principio, podemos añadir el encabezado 'If-Modified-Since:' en nuestra solicitud HTTP).

Hablaremos de averiguar la longitud del documento (tamaño de los datos) con más detalle porque casi todos los encabezados son opcionales, es decir, el servidor los comunica a voluntad y, en su ausencia, se utilizan mecanismos alternativos. El tamaño es importante para saber cuándo hay que cerrar la conexión, es decir, para asegurarse de que se han recibido todos los datos.

La ejecución del script con los parámetros por defecto produce el siguiente resultado:

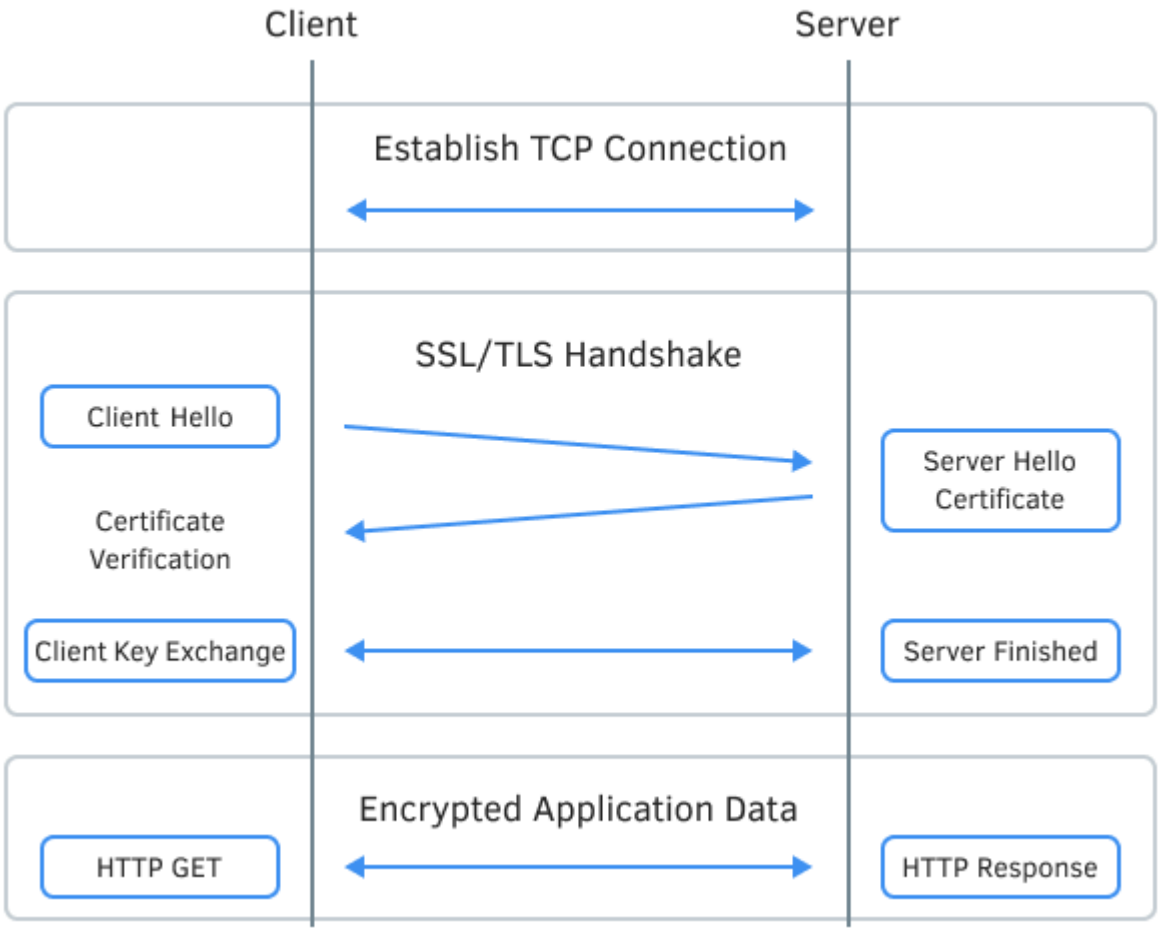
```
Server=www.mql5.com / ok
Port=80 / ok
SocketCreate()=1 / ok
SocketConnect(socket,Server,Port,5000)=true / ok
HTTPSend(socket,StringFormat(HEAD / HTTP/1.1
Host: %s
,Server))=true / ok
HTTP-header found
HTTPRecv(socket,response,5000)=true / ok
HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Sun, 31 Jul 2022 10:24:00 GMT
Content-Type: text/html
Content-Length: 162
Connection: keep-alive
Location: https://www.mql5.com/
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
X-Frame-Options: SAMEORIGIN
```

Tenga en cuenta que este sitio, como la mayoría de los sitios actuales, redirige nuestra solicitud a una conexión segura: esto se consigue con el código de estado «301 Moved Permanently» y la nueva dirección «Location: https://www.mql5.com/» (el protocolo es importante aquí «https»). Para reintentar una solicitud habilitada para TLS, se deben utilizar otras funciones, que discutiremos más adelante.

7.5.9 Preparar una conexión de socket segura

Para transferir una conexión de socket a un estado protegido y comprobarlo, MQL6 proporciona las siguientes funciones: *SocketTlsHandshake* y *SocketTlsCertificate*, respectivamente. Por regla general, no necesitamos activar «manualmente» la protección llamando a *SocketTlsHandshake* si la conexión se establece en el puerto 443. El hecho es que es estándar para HTTPS (TLS).

La protección se basa en el cifrado del flujo de datos entre el cliente y el servidor, para lo cual se utiliza inicialmente un par de claves asimétricas: pública y privada. Ya hemos tratado este tema en la sección Visión general de los métodos de transformación de la información disponibles. Cada sitio decente adquiere un certificado digital de una de las autoridades de certificación (CA) en las que confía la comunidad de la red. El certificado contiene la clave pública del sitio y está firmado digitalmente por el centro. Los navegadores y otras aplicaciones cliente almacenan (o pueden importar) las claves públicas de las CA y, por tanto, pueden verificar la calidad de un certificado concreto.



Establecer una conexión TLS segura
(imagen de internet)

Además, al preparar una conexión segura, el navegador o la aplicación genera un cierto «secreto», lo cifra con la clave pública del sitio y le envía la clave, y el sitio lo descifra con la clave privada que sólo él conoce. Esta etapa parece más complicada en la práctica, pero como resultado, tanto el cliente como el servidor disponen de la clave de cifrado para la sesión (conexión) actual. Esta clave la utilizan los dos participantes en la comunicación para cifrar las solicitudes y respuestas posteriores en un extremo y descifrarlas en el otro.

La función *SocketTlsHandshake* inicia una conexión TLS segura con el host especificado utilizando el protocolo handshake de TLS. En este caso, el cliente y el servidor acuerdan los parámetros de conexión: la versión del protocolo utilizado y el método de cifrado de los datos.

bool SocketTlsHandshake(int socket, const string host)

En los parámetros de la función se pasan el manejador de socket y la dirección del servidor con el que se establece la conexión (de hecho, es el mismo nombre que se especificó en *SocketConnect*).

Antes de establecer una conexión segura, el programa debe establecer primero una conexión TCP normal con el host utilizando *SocketConnect*.

La función devuelve *true* si tiene éxito; en caso contrario, devuelve *false*. En caso de error, se escribe el código 5274 (ERR_NETSOCKET_HANDSHAKE_FAILED) en *_LastError*.

La función *SocketTlsCertificate* obtiene información sobre el certificado utilizado para proteger la conexión de red.

int SocketTlsCertificate(int socket, string &subject, string &issuer, string &serial, string &thumbprint, datetime &expiration)

Si se establece una conexión segura para el socket (ya sea tras una llamada explícita y satisfactoria a *SocketTlsHandshake* o tras conectarse a través del puerto 443), esta función rellena todas las demás variables de referencia del descriptor del socket con la información correspondiente: el nombre del propietario del certificado (*subject*), el nombre del emisor del certificado (*issuer*), el número de serie (*serial*), la huella digital (*thumbprint*) y el periodo de validez del certificado (*expiration*).

La función devuelve *true* en caso de recepción satisfactoria de la información sobre el certificado, o *false* como resultado de un error. El código de error es 5275 (ERR_NETSOCKET_NO_CERTIFICATE). Permite determinar si la conexión abierta por *SocketConnect* se encuentra en modo protegido de forma inmediata. Lo utilizaremos en un ejemplo en la siguiente sección.

7.5.10 Leer y escribir datos a través de una conexión de socket segura

Una conexión segura tiene su propio conjunto de funciones de intercambio de datos entre el cliente y el servidor. Los nombres y el concepto de operación de las funciones coinciden prácticamente con los de las funciones *SocketRead* y *SocketSend* consideradas anteriormente.

int SocketTlsRead(int socket, uchar &buffer[], uint maxlen)

La función *SocketTlsRead* lee datos de una conexión TLS segura abierta en el socket especificado. Los datos se introducen en el array *buffer* pasado por referencia. Si es dinámico, su tamaño se incrementará según la cantidad de datos pero no más de INT_MAX (2147483647) bytes.

El parámetro *maxlen* especifica el número de bytes descifrados que se recibirán (su número es siempre inferior a la cantidad de datos cifrados «en bruto» que llegan al búfer interno del socket). Los datos que no caben en el array permanecen en el socket y pueden ser recibidos por la siguiente llamada a *SocketTlsRead*.

La función se ejecuta hasta que recibe la cantidad de datos especificada o hasta que se produce el tiempo de espera especificado en *SocketTimeouts*.

En caso de éxito, la función devuelve el número de bytes leídos; en caso de error, devuelve -1, mientras que el código 5273 (ERR_NETSOCKET_IO_ERROR) se escribe en *_LastError*. La presencia de un error indica que la conexión ha finalizado.

int SocketTlsReadAvailable(int socket, uchar &buffer[], const uint maxlen)

La función *SocketTlsReadAvailable* lee todos los datos descifrados disponibles de una conexión TLS segura, pero no más de *maxlen* bytes. A diferencia de *SocketTlsRead*, *SocketTlsReadAvailable* no espera la presencia obligatoria de una cantidad determinada de datos y devuelve inmediatamente sólo los que están presentes. Así, si el búfer interno del socket está «vacío» (aún no se ha recibido nada del servidor, ya se ha leído o aún no se ha formado un bloque listo para el descifrado), la función devolverá 0 y no se registrará nada en el array de recepción *buffer*. Esta es una situación habitual.

El valor de *maxlen* debe estar comprendido entre 1 e INT_MAX (2147483647).

int SocketTlsSend(int socket, const uchar &buffer[], uint bufferlen)

La función *SocketTlsSend* envía datos desde el array *buffer* a través de una conexión segura abierta en el socket especificado. El principio de funcionamiento es el mismo que el de la función descrita anteriormente *SocketSend*, mientras que la única diferencia es el tipo de conexión.

Vamos a crear un nuevo script *SocketReadWriteHTTPS.mq5* basado en el *SocketReadWriteHTTP.mq5* anteriormente considerado, y a añadir flexibilidad en cuanto a la elección de un método HTTP (GET de manera predeterminada, no HEAD), estableciendo un tiempo de espera y admitiendo conexiones seguras. El puerto predeterminado es 443.

```
input string Method = "GET"; // Method (HEAD,GET)
input string Server = "www.google.com";
input uint Port = 443;
input uint Timeout = 5000;
```

El servidor predeterminado es www.google.com. No olvide añadirlo (y cualquier otro servidor que introduzca) a la lista de permitidos en la configuración del terminal.

Para determinar si la conexión es segura o no, utilizaremos la función *SocketTlsCertificate*: si tiene éxito, el servidor ha proporcionado un certificado y el modo TLS está activo. Si la función devuelve *false* y lanza el código de error NETSOCKET_NO_CERTIFICATE(5275), significa que estamos utilizando una conexión normal pero el error puede ser ignorado y reiniciado, ya que estamos satisfechos con una conexión no segura.

```
void OnStart()
{
   PRTF(Server);
   PRTF(Port);
   const int socket = PRTF(SocketCreate());
   if(socket == INVALID_HANDLE) return;
   SocketTimeouts(socket, Timeout, Timeout);
   if(PRTF(SocketConnect(socket, Server, Port, Timeout)))
   {
      string subject, issuer, serial, thumbprint;
      datetime expiration;
      bool TLS = false;
      if(PRTF(SocketTlsCertificate(socket, subject, issuer, serial, thumbprint, expir
      {
         PRTF(subject);
         PRTF(issuer);
         PRTF(serial);
         PRTF(thumbprint);
         PRTF(expiration);
         TLS = true;
      }
      ...
```

El resto de la función *OnStart* se ejecuta según el plan anterior: enviar una petición mediante la función *HTTPSend* y aceptar la respuesta mediante *HTTPRecv*. Pero esta vez, pasamos adicionalmente la bandera TLS a estas funciones, y deben implementarse de forma ligeramente diferente.

```
   if(PRTF(HTTPSend(socket, StringFormat("%s / HTTP/1.1\r\nHost: %s\r\n"
      "User-Agent: MetaTrader 5\r\n\r\n", Method, Server), TLS)))
   {
      string response;
      if(PRTF(HTTPRecv(socket, response, Timeout, TLS)))
      {
         Print("Got ", StringLen(response), " bytes");
         // for large documents, we will save to a file
         if(StringLen(response) > 1000)
         {
            int h = FileOpen(Server + ".htm", FILE_WRITE | FILE_TXT | FILE_ANSI, 0
            FileWriteString(h, response);
            FileClose(h);
         }
         else
         {
            Print(response);
         }
      }
   }
```

Del ejemplo con *HTTPSend*, se puede ver que dependiendo de la bandera TLS, utilizamos *SocketTlsSend* o *SocketSend*.

```
bool HTTPSend(int socket, const string request, const bool TLS)
{
   char req[];
   int len = StringToCharArray(request, req, 0, WHOLE_ARRAY, CP_UTF8) - 1;
   if(len < 0) return false;
   return (TLS ? SocketTlsSend(socket, req, len) : SocketSend(socket, req, len)) == l
}
```

Las cosas son un poco más complicadas con *HTTPRecv*. Dado que ofrecemos la posibilidad de descargar toda la página (no sólo los encabezados), necesitamos alguna forma de saber si hemos recibido todos los datos. Incluso después de que se haya transmitido todo el documento, el socket suele dejarse abierto para optimizar futuras solicitudes previstas. Pero nuestro programa no sabrá si la transmisión se detuvo normalmente, o tal vez hubo una «congestión» temporal en algún lugar de la infraestructura de red (este tipo de carga de página relajada e intermitente puede observarse a veces en los navegadores). O viceversa, en caso de fallo de conexión, podemos creer erróneamente que hemos recibido el documento completo.

El hecho es que los propios sockets sólo actúan como medio de comunicación entre programas y trabajan con bloques abstractos de datos: desconocen el tipo de datos, su significado y su conclusión lógica. Todas estas cuestiones se gestionan mediante protocolos de aplicación como HTTP. Por lo tanto, tendremos que profundizar en las especificaciones e implementar las comprobaciones nosotros mismos.

```
bool HTTPRecv(int socket, string &result, const uint timeout, const bool TLS)
{
   uchar response[]; // accumulate the data as a whole (headers + body of the web doc
   uchar block[];    // separate read block
   int len;          // current block size (signed integer for error flag -1)
   int lastLF = -1;  // position of the last line feed found LF(Line-Feed)
   int body = 0;     // offset where document body starts
   int size = 0;     // document size according to title
   result = "";      // set an empty result at the beginning
   int chunk_size = 0, chunk_start = 0, chunk_n = 1;
   const static string content_length = "Content-Length:";
   const static string crlf = "\r\n";
   const static int crlf_length = 2;
   ...
```

El método más sencillo para determinar el tamaño de los datos recibidos se basa en analizar el encabezado «Content-Length:». Aquí necesitamos tres variables: *lastLF*, *size* y *content_length*. Sin embargo, este encabezado no siempre está presente, y operamos con «trozos»: se introducen las variables *chunk_size*, *chunk_start*, *crlf* y *crlf_length* para detectarlos.

Para demostrar diversas técnicas de recepción de datos, utilizamos en este ejemplo una función *SocketTlsReadAvailable* «no bloqueante». Sin embargo, no existe una función similar para una conexión insegura, por lo que tendremos que escribirla nosotros mismos (un poco más adelante). El esquema general del algoritmo es sencillo: se trata de un bucle con intentos de recibir nuevos bloques de datos de 1024 (o menos) bytes de tamaño. Si conseguimos leer algo, lo acumulamos en el array de respuesta. Si el búfer de entrada del socket está vacío, las funciones devolverán 0 y haremos una pequeña pausa. Por último, si se produce un error o se agota el tiempo de espera, el bucle se romperá.

```
   uint start = GetTickCount();
   do
   {
      ResetLastError();
      if((len = (TLS ? SocketTlsReadAvailable(socket, block, 1024) :
         SocketReadAvailable(socket, block, 1024))) > 0)
      {
         const int n = ArraySize(response);
         ArrayCopy(response, block, n); // put all the blocks together
         ...
         // main operation here
      }
      else
      {
         if(len == 0) Sleep(10); // wait a bit for the arrival of a portion of data
      }
   }
   while(GetTickCount() - start < timeout && !IsStopped() && !_LastError);
   ...
```

En primer lugar, hay que esperar a que se complete la cabecera HTTP en el flujo de datos de entrada. Como ya hemos visto en el ejemplo anterior, los encabezados se separan del documento mediante una doble «línea nueva», es decir, mediante la secuencia de caracteres «\r\n\r\n». Es fácil de detectar por dos símbolos «\n» (LF) situados uno detrás de otro.

El resultado de la búsqueda será el desplazamiento en bytes desde el inicio de los datos, donde termina el encabezado y comienza el documento. Lo almacenaremos en la variable *body*.

```
if(body == 0) // look for the completion of the headers until we find it
{
   for(int i = n; i < ArraySize(response); ++i)
   {
      if(response[i] == '\n') // LF
      {
         if(lastLF == i - crlf_length) // found sequence "\r\n\r\n"
         {
            body = i + 1;
            string headers = CharArrayToString(response, 0, i);
            Print("* HTTP-header found, header size: ", body);
            Print(headers);
            const int p = StringFind(headers, content_length);
            if(p > -1)
            {
               size = (int)StringToInteger(StringSubstr(headers,
                  p + StringLen(content_length)));
               Print("* ", content_length, size);
            }
            ...
            break; // header/body boundary found
         }
         lastLF = i;
      }
   }
}

if(size == ArraySize(response) - body) // entire document
{
   Print("* Complete document");
   break;
}
...
```

Esto busca inmediatamente el encabezado «Content-Length:» y extrae de él el tamaño. La variable *size* rellenada permite escribir una sentencia condicional adicional para salir del bucle de recepción de datos cuando se haya recibido todo el documento.

Algunos servidores dan el contenido en partes llamadas «trozos». En estos casos, la línea «Transfer-Encoding: chunked» está presente en el encabezado HTTP, y falta la línea «Content-Length:». Cada trozo comienza con un número hexadecimal que indica el tamaño del trozo, seguido de una nueva línea y el número especificado de bytes de datos. El trozo termina con otra nueva línea. El último trozo que marca el final del documento tiene un tamaño cero.

Tenga en cuenta que la división en dichos segmentos la realiza el servidor, basándose en sus propias «preferencias» actuales para optimizar el envío, y no tiene nada que ver con los bloques (paquetes) de datos en los que se divide la información a nivel de socket para su transmisión por la red. En otras palabras: los trozos tienden a fragmentarse arbitrariamente y el límite entre paquetes de red puede darse incluso entre dígitos de un tamaño de trozo.

Esquemáticamente, esto se puede representar de la siguiente manera (a la izquierda están los trozos del documento, y a la derecha, los bloques de datos del búfer del socket):

Fragmentación de un documento web durante la transmisión a los niveles HTTP y TCP

En nuestro algoritmo, los paquetes entran en el array *block* en cada iteración, pero no tiene sentido analizarlos uno por uno, y todo el trabajo principal va con el array de respuesta común.

Así, si el encabezado HTTP se recibe completamente pero no se encuentra en ella la cadena «Content-Length:», pasamos a la rama del algoritmo con el modo «Transfer-Encoding: chunked». Por la posición

actual de *body* en el array *response* (inmediatamente después de la finalización de los encabezados HTTP), el fragmento de cadena se selecciona y convierte en un número asumiendo el formato hexadecimal: esto es realizado por la función auxiliar *HexStringToInteger* (véase el código fuente adjunto). Si realmente hay un número, lo escribimos en *chunk_size*, marcamos la posición como inicio del «trozo» en *chunk_start* y eliminamos los bytes con el número y las nuevas líneas de encuadre de *response*.

```
...
if(lastLF == i - crlf_length) // found sequence "\r\n\r\n"
{
   body = i + 1;
   ...
   const int p = StringFind(headers, content_length);
   if(p > -1)
   {
      size = (int)StringToInteger(StringSubstr(headers,
         p + StringLen(content_length)));
      Print("* ", content_length, size);
   }
   else
   {
      size = -1; // server did not provide document length
      // try to find chunks and the size of the first one
      if(StringFind(headers, "Transfer-Encoding: chunked") > 0)
      {
         // chunk syntax:
         // <hex-size>\r\n<content>\r\n...
         const string preview = CharArrayToString(response, body, 2
         chunk_size = HexStringToInteger(preview);
         if(chunk_size > 0)
         {
            const int d = StringFind(preview, crlf) + crlf_length;
            chunk_start = body;
            Print("Chunk: ", chunk_size, " start at ", chunk_start,
            ArrayRemove(response, body, d);
         }
      }
   }
   break; // header/body boundary found
}
lastLF = i;
...
```

Ahora, para comprobar la integridad del documento, es necesario analizar no sólo la variable *size* (que, como hemos visto, en realidad puede desactivarse asignando -1 en ausencia de «Content-Length:»), sino también nuevas variables para los trozos: *chunk_start* y *chunk_size*. El esquema de actuación es el mismo que tras los encabezados HTTP: por desplazamiento en el array *response*, donde terminó el trozo anterior, aislamos el tamaño del siguiente «trozo». Continuamos el proceso hasta encontrar un trozo de tamaño cero.

```
...
if(size == ArraySize(response) - body) // entire document
{
   Print("* Complete document");
   break;
}
else if(chunk_size > 0 && ArraySize(response) - chunk_start >= chunk_size)
{
   Print("* ", chunk_n, " chunk done: ", chunk_size, " total: ", ArraySize(r
   const int p = chunk_start + chunk_size;
   const string preview = CharArrayToString(response, p, 20);
   if(StringLen(preview) > crlf_length              // there is '\r\n...\r\n
      && StringFind(preview, crlf, crlf_length) > crlf_length)
   {
      chunk_size = HexStringToInteger(preview, crlf_length);
      if(chunk_size > 0)
      {                                 // twice '\r\n': before and after chunk
         int d = StringFind(preview, crlf, crlf_length) + crlf_length;
         chunk_start = p;
         Print("Chunk: ", chunk_size, " start at ", chunk_start, " -", d);
         ArrayRemove(response, chunk_start, d);
         ++chunk_n;
      }
      else
      {
         Print("* Final chunk");
         ArrayRemove(response, p, 5); // "\r\n0\r\n"
         break;
      }
   } // otherwise wait for more data
}
```

Así, proporcionamos una salida del bucle basada en los resultados del análisis del flujo entrante de dos formas diferentes (además de la salida por tiempo de espera y por error). Al final normal del bucle, convertimos esa parte del array en la cadena *response*, que comienza en la posición *body* y contiene todo el documento. De lo contrario, simplemente devolvemos todo lo que hemos conseguido obtener, junto con los encabezados, para su «análisis».

```
bool HTTPRecv(int socket, string &result, const uint timeout, const bool TLS)
{
   ...
   do
   {
      ResetLastError();
      if((len = (TLS ? SocketTlsReadAvailable(socket, block, 1024) :
         SocketReadAvailable(socket, block, 1024))) > 0)
      {
         ... // main operation here - discussed above
      }
      else
      {
         if(len == 0) Sleep(10); // wait a bit for the arrival of a portion of data
      }
   }
   while(GetTickCount() - start < timeout && !IsStopped() && !_LastError);

   if(_LastError) PRTF(_LastError);

   if(ArraySize(response) > 0)
   {
      if(body != 0)
      {
         // TODO: Desirable to check 'Content-Type:' for 'charset=UTF-8'
         result = CharArrayToString(response, body, WHOLE_ARRAY, CP_UTF8);
      }
      else
      {
         // to analyze wrong cases, return incomplete headers as is
         result = CharArrayToString(response);
      }
   }

   return StringLen(result) > 0;
}
```

La única función restante es *SocketReadAvailable*, que es el análogo de *SocketTlsReadAvailable* para conexiones no seguras.

```
int SocketReadAvailable(int socket, uchar &block[], const uint maxlen = INT_MAX)
{
   ArrayResize(block, 0);
   const uint len = SocketIsReadable(socket);
   if(len > 0)
      return SocketRead(socket, block, fmin(len, maxlen), 10);
   return 0;
}
```

El script está listo para trabajar.

Nos costó bastante esfuerzo implementar una simple petición de página web utilizando sockets. Esto sirve para demostrar hasta qué punto la compatibilidad de los protocolos de red a bajo nivel suele ser

una tarea ardua. Por supuesto, en el caso de HTTP, es más fácil y más correcto para nosotros utilizar la implementación integrada de WebRequest, pero no incluye todas las características de HTTP (además, hemos tocado HTTP 1.1 de pasada, pero también existe HTTP / 2), y el número de otros protocolos de aplicación es enorme. Por lo tanto, se requieren las funciones *Socket* para integrarlos en MetaTrader 5.

Vamos a ejecutar *SocketReadWriteHTTPS.mq5* con la configuración por defecto.

```
Server=www.google.com / ok
Port=443 / ok
SocketCreate()=1 / ok
SocketConnect(socket,Server,Port,Timeout)=true / ok
SocketTlsCertificate(socket,subject,issuer,serial,thumbprint,expiration)=true / ok
subject=CN=www.google.com / ok
issuer=C=US, O=Google Trust Services LLC, CN=GTS CA 1C3 / ok
serial=00c9c57583d70aa05d12161cde9ee32578 / ok
thumbprint=1EEE9A574CC92773EF948B50E79703F1B55556BF / ok
expiration=2022.10.03 08:25:10 / ok
HTTPSend(socket,StringFormat(%s / HTTP/1.1
Host: %s
,Method,Server),TLS)=true / ok
* HTTP-header found, header size: 1080
HTTP/1.1 200 OK
Date: Mon, 01 Aug 2022 20:48:35 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Set-Cookie: 1P_JAR=2022-08-01-20; expires=Wed, 31-Aug-2022 20:48:35 GMT;
   path=/; domain=.google.com; Secure
...
Accept-Ranges: none
Vary: Accept-Encoding
Transfer-Encoding: chunked
Chunk: 22172 start at 1080 -6
* 1 chunk done: 22172 total: 24081
Chunk: 30824 start at 23252 -8
* 2 chunk done: 30824 total: 54083
* Final chunk
HTTPRecv(socket,response,Timeout,TLS)=true / ok
Got 52998 bytes
```

Como podemos ver, el documento se transfiere en trozos y se ha guardado en un archivo temporal (puede encontrarlo en *MQL5/Files/www.mql5.com.htm*).

Ahora vamos a ejecutar el script para el sitio «www.mql5.com» y el puerto 80. Por la sección anterior, sabemos que el sitio en este caso emite una redirección a su versión protegida, pero esta «redirección» no está vacía: tiene un documento stub, y ahora podemos obtenerlo completo. Lo que nos importa aquí es que el encabezado «Content-Length:» se utiliza correctamente en este caso.

```
Server=www.mql5.com / ok
Port=80 / ok
SocketCreate()=1 / ok
SocketConnect(socket,Server,Port,Timeout)=true / ok
HTTPSend(socket,StringFormat(%s / HTTP/1.1
Host: %s
,Method,Server),TLS)=true / NETSOCKET_NO_CERTIFICATE(5275)
* HTTP-header found, header size: 291
HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Sun, 31 Jul 2022 19:28:57 GMT
Content-Type: text/html
Content-Length: 162
Connection: keep-alive
Location: https://www.mql5.com/
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
X-Frame-Options: SAMEORIGIN
* Content-Length:162
* Complete document
HTTPRecv(socket,response,Timeout,TLS)=true / ok
<html>
<head><title>301 Moved Permanently</title></head>
<body>
<center><h1>301 Moved Permanently</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

Otro gran ejemplo del uso de sockets en la práctica lo consideraremos en el capítulo Proyectos.

7.6 Base de datos SQLite

MetaTrader 5 proporciona compatibilidad nativa para la base de datos SQLite. Se trata de un sistema de gestión de bases de datos (SGBD) ligero pero plenamente funcional. Tradicionalmente, este tipo de sistemas se centran en el tratamiento de tablas de datos, en las que se almacenan registros del mismo tipo con un conjunto común de atributos, y se pueden establecer distintas correspondencias (enlaces o relaciones) entre registros de distintos tipos (es decir, tablas), por lo que este tipo de bases de datos también se denominan relacionales. Ya hemos considerado ejemplos de este tipo de conexiones entre estructuras del calendario económico, pero la base de datos del calendario se almacena dentro del terminal, y las funciones de esta sección le permitirán crear bases de datos arbitrarias a partir de programas MQL.

La especialización del SGBD en estas estructuras de datos permite optimizar (acelerar y simplificar) muchas operaciones habituales, como ordenar, buscar, filtrar, sumar o calcular otras funciones agregadas para grandes cantidades de datos.

Sin embargo, esto tiene otra cara: La programación del SGBD requiere su propio SQL (Structured Query Language), y el conocimiento de MQL5 puro no será suficiente. A diferencia de MQL5, que se refiere a los lenguajes *imperative* (aquellos que utilizan operadores que indican qué, cómo y en qué secuencia hacer), SQL es *declarative*, es decir, describe los datos iniciales y el resultado deseado, sin

especificar cómo y en qué secuencia realizar los cálculos. El significado del algoritmo en SQL se describe en forma de consultas SQL. Una consulta es un análogo de un operador MQL5 separado, formado como una cadena utilizando una sintaxis especial.

En lugar de programar complejos bucles y comparaciones, podemos simplemente llamar a funciones SQLite (por ejemplo, *DatabaseExecute* o *Database Prepare*) pasándoles consultas SQL. Para obtener los resultados de la consulta en una estructura MQL5 ya preparada, puede utilizar la función *DatabaseReadBind*. Esto le permitirá leer todos los campos del registro (estructura) a la vez en una sola llamada.

Con la ayuda de las funciones de base de datos, es fácil crear tablas, añadirles registros, realizar modificaciones y efectuar selecciones según condiciones complejas, por ejemplo, para tareas como:

- Obtener el historial de trading y cotizaciones
- Guardar los resultados de la optimización y la simulación
- Preparar e intercambiar datos con otros paquetes de análisis
- Analizar los datos del calendario económico
- Almacenar ajustes y estados de programas MQL5

Además, en las consultas SQL puede utilizarse una amplia gama de funciones comunes, estadísticas y matemáticas. Además, las expresiones con su participación pueden calcularse incluso sin crear una tabla.

SQLite no requiere una aplicación, configuración y administración independientes, no exige muchos recursos y admite la mayoría de los comandos del popular estándar SQL92. Una ventaja añadida es que toda la base de datos reside en un único archivo en el disco duro del ordenador del usuario y puede transferirse o copiarse fácilmente. Sin embargo, para acelerar las operaciones de lectura, escritura y modificación, la base de datos también puede abrirse/crearse en RAM con el indicador DATABASE_OPEN_MEMORY; sin embargo, en este caso, dicha base de datos sólo estará disponible para este programa concreto y no podrá utilizarse para el trabajo conjunto de varios programas.

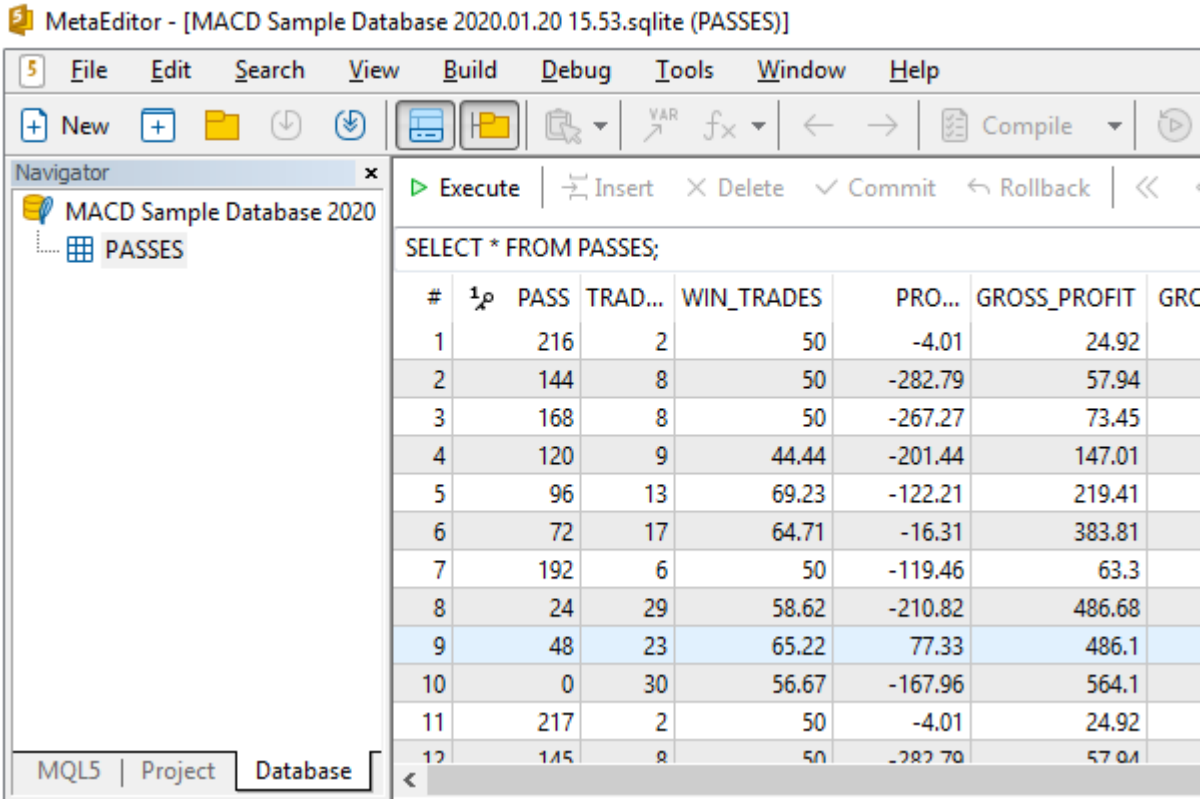
> Es importante señalar que la relativa simplicidad de SQLite, en comparación con los SGBD completos, conlleva algunas limitaciones. En particular, SQLite no tiene un proceso dedicado (servicio del sistema o aplicación) que proporcione acceso centralizado a la base de datos y a la API de gestión de tablas, por lo que no está garantizado el acceso paralelo y compartido a la misma base de datos (archivo) desde distintos procesos. Por lo tanto, si necesita leer y escribir simultáneamente en la base de datos desde agentes de optimización que ejecutan instancias del mismo Asesor Experto, tendrá que escribir código en él para sincronizar el acceso (de lo contrario, los datos que se escriban y lean estarán en un estado inconsistente: al fin y al cabo, el orden de escritura, modificación, borrado y lectura de procesos concurrentes no sincronizados es aleatorio). Además, los intentos de modificar la base de datos al mismo tiempo pueden hacer que el programa MQL reciba errores de «base de datos ocupada» (y no se realice la operación solicitada). El único escenario que no requiere sincronización de operaciones paralelas con SQLite es cuando sólo se trata de operaciones de lectura.

Sólo presentaremos los fundamentos de SQL en la medida necesaria para empezar a aplicarlo. Una descripción completa de la sintaxis y el funcionamiento de SQL queda fuera del alcance de este libro. Consulte la documentación en el sitio web de SQLite. No obstante, tenga en cuenta que MQL5 y MetaEditor apoyan un subconjunto limitado de comandos y construcciones sintácticas SQL.

MQL Wizard en MetaEditor tiene una opción incrustada para crear una base de datos, que inmediatamente ofrece crear la primera tabla definiendo una lista de sus campos. Asimismo, *Navigator* ofrece una pestaña independiente para trabajar con bases de datos.

Utilizando *Wizard* o el menú contextual de *Navigator* puede crear una base de datos vacía (un archivo en disco, situado por defecto, en el directorio *MQL5/Files*) de formatos compatibles (*.db, *.sql, *.sqlite y otros). Además, en el menú contextual, puede importar toda la base de datos desde un archivo sql o tablas individuales desde archivos csv.

Una base de datos existente o creada puede abrirse fácilmente a través del mismo menú. A continuación, sus tablas aparecerán en *Navigator*, y en la zona derecha de la ventana se mostrará un panel con herramientas para depurar consultas SQL y una tabla con los resultados. Por ejemplo, al hacer doble clic en el nombre de una tabla se realiza una consulta rápida de todos los campos del registro, lo que corresponde a la sentencia «SELECT * FROM 'tabla'» que aparece en el campo de entrada de la parte superior.



#	PASS	TRAD...	WIN_TRADES	PRO...	GROSS_PROFIT	GRC
1	216	2	50	-4.01	24.92	
2	144	8	50	-282.79	57.94	
3	168	8	50	-267.27	73.45	
4	120	9	44.44	-201.44	147.01	
5	96	13	69.23	-122.21	219.41	
6	72	17	64.71	-16.31	383.81	
7	192	6	50	-119.46	63.3	
8	24	29	58.62	-210.82	486.68	
9	48	23	65.22	77.33	486.1	
10	0	30	56.67	-167.96	564.1	
11	217	2	50	-4.01	24.92	
12	145	8	50	-282.79	57.94	

Visualización de la base de datos SQLite en MetaEditor

Puede editar la solicitud y hacer clic en el botón *Execute* para activarla. Los posibles errores de sintaxis SQL se muestran en el registro.

Para obtener más detalles sobre *Wizard*, la importación/exportación de bases de datos y el trabajo interactivo con ellas, consulte Documentación de MetaEditor.

7.6.0 Principios de las operaciones de base de datos en MQL5

Las bases de datos almacenan información en forma de tablas. La obtención, modificación y adición de nuevos datos se realiza mediante consultas en lenguaje SQL. Describiremos sus particularidades en las secciones siguientes. Mientras tanto, utilicemos el script *DatabaseRead.mq5*, que no tiene nada que ver con el trading, y veamos cómo crear una base de datos sencilla y obtener información de ella. Todas las funciones mencionadas aquí se describirán en detalle más adelante. Ahora es importante imaginar los principios generales.

La creación y cierre de una base de datos incorporada mediante funciones *DatabaseOpen/DatabaseClose* integradas se realiza de forma similar a como se hace con los archivos, ya que también creamos un descriptor para la base de datos, lo comprobamos y lo cerramos al final.

```
void OnStart()
{
   string filename = "company.sqlite";
   // create or open a database
   int db = DatabaseOpen(filename, DATABASE_OPEN_READWRITE | DATABASE_OPEN_CREATE);
   if(db == INVALID_HANDLE)
   {
      Print("DB: ", filename, " open failed with code ", _LastError);
      return;
   }
   ...// further work with the database
   // close the database
   DatabaseClose(db);
}
```

Después de abrir la base de datos, nos aseguraremos de que no haya ninguna tabla en ella con el nombre que necesitamos. Si la tabla ya existe, al intentar insertar en ella los mismos datos que en nuestro ejemplo, se producirá un error, por lo que utilizaremos la función *DatabaseTableExists*.

La eliminación y creación de una tabla se realiza mediante consultas que se envían a la base de datos con dos llamadas a la función *DatabaseExecute* y acompañadas de una comprobación de errores.

```
   ...
   // if the table COMPANY exists, then delete it
   if(DatabaseTableExists(db, "COMPANY"))
   {
      if(!DatabaseExecute(db, "DROP TABLE COMPANY"))
      {
         Print("Failed to drop table COMPANY with code ", _LastError);
         DatabaseClose(db);
         return;
      }
   }
   // creating table COMPANY
   if(!DatabaseExecute(db, "CREATE TABLE COMPANY("
     "ID      INT     PRIMARY KEY NOT NULL,"
     "NAME    TEXT    NOT NULL,"
     "AGE     INT     NOT NULL,"
     "ADDRESS CHAR(50),"
     "SALARY  REAL );"))
   {
      Print("DB: ", filename, " create table failed with code ", _LastError);
      DatabaseClose(db);
      return;
   }
   ...
```

Vamos a explicar la esencia de las consultas SQL. En la tabla EMPRESA tenemos solo 5 campos: ID de registro, nombre, edad, dirección y salario. Aquí, el campo ID es una clave, es decir, un índice único.

Los índices permiten identificar unívocamente cada registro y pueden utilizarse en todas las tablas para vincularlas entre sí. Esto es similar a cómo el ID de posición vincula todas las operaciones y órdenes que pertenecen a una posición en particular.

Ahora necesita llenar la tabla con datos, lo que se hace mediante la consulta «INSERT»:

```
// insert data into table
if(!DatabaseExecute(db,
   "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) VALUES (1,'Paul',32,'Californ
   "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) VALUES (2,'Allen',25,'Texas',
   "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) VALUES (3,'Teddy',23,'Norway'
   "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) VALUES (4,'Mark',25,'Rich-Mon
{
   Print("DB: ", filename, " insert failed with code ", _LastError);
   DatabaseClose(db);
   return;
}
...
```

Aquí se añaden 4 registros a la tabla EMPRESA; para cada registro hay una lista de campos, y se indican los valores que se escribirán en estos campos. Los registros se insertan mediante consultas «INSERT...» separadas, que se combinan en una sola línea, mediante un carácter delimitador especial ';', pero podríamos insertar cada registro en la tabla con una llamada separada a *DatabaseExecute*.

Como al final del script la base de datos se guardará en el archivo «empresa.sqlite», la próxima vez que se ejecute intentaríamos escribir los mismos datos en la tabla EMPRESA con el mismo ID. Esto provocaría un error, razón por la cual previamente borramos la tabla para empezar de cero cada vez que se ejecutara el script.

Ahora obtenemos todos los registros de la tabla EMPRESA con el campo *SALARY > 15000*. Para ello se utiliza la función *DatabasePrepare*, que «compila» el texto de la solicitud y devuelve su controlador para su posterior uso en las funciones *DatabaseRead* o *DatabaseReadBind*.

```
// prepare a request with a descriptor
int request = DatabasePrepare(db, "SELECT * FROM COMPANY WHERE SALARY>15000");
if(request == INVALID_HANDLE)
{
   Print("DB: ", filename, " request failed with code ", _LastError);
   DatabaseClose(db);
   return;
}
...
```

Una vez que la solicitud se ha creado correctamente, necesitamos obtener los resultados de su ejecución. Esto puede hacerse utilizando la función *DatabaseRead*, que en la primera llamada ejecutará la consulta y saltará al primer registro de los resultados. En cada llamada posterior, leerá el siguiente registro hasta llegar al final. En este caso, devolverá *false*, lo que significa «no hay más registros».

```
   // printing all records with salary over 15000
   int id, age;
   string name, address;
   double salary;
   Print("Persons with salary > 15000:");
   for(int i = 0; DatabaseRead(request); i++)
   {
      // read the values of each field from the received record by its number
      if(DatabaseColumnInteger(request, 0, id) && DatabaseColumnText(request, 1, name
         DatabaseColumnInteger(request, 2, age) && DatabaseColumnText(request, 3, add
         DatabaseColumnDouble(request, 4, salary))
         Print(i, ":  ", id, " ", name, " ", age, " ", address, " ", salary);
      else
      {
         Print(i, ": DatabaseRead() failed with code ", _LastError);
         DatabaseFinalize(request);
         DatabaseClose(db);
         return;
      }
   }
   // deleting handle after use
   DatabaseFinalize(request);
```

El resultado de la ejecución será:

```
Persons with salary > 15000:
0:  1 Paul 32 California 25000.0
1:  3 Teddy 23 Norway 20000.0
2:  4 Mark 25 Rich-Mond  65000.0
```

La función *DatabaseRead* permite recorrer todos los registros del resultado de la consulta y, a continuación, obtener información completa sobre cada columna de la tabla resultante a través de las funciones *DatabaseColumn*. Estas funciones están diseñadas para trabajar de forma universal con los resultados de cualquier consulta, pero el coste es un código redundante.

Si se conoce de antemano la estructura de los resultados de la consulta, es mejor utilizar la función *DatabaseReadBind*, que permite leer todo el registro de una vez en una estructura. Podemos rehacer así el ejemplo anterior y presentarlo con un nuevo nombre *DatabaseReadBind.mq5*. En primer lugar, declaremos la estructura *Person*:

```
struct Person
{
   int    id;
   string name;
   int    age;
   string address;
   double salary;
};
```

A continuación, restaremos cada registro de los resultados de la consulta con *DatabaseReadBind(request, person)* en un bucle mientras la función devuelva *true*:

```
Person person;
Print("Persons with salary > 15000:");
for(int i = 0; DatabaseReadBind(request, person); i++)
   Print(i, ":  ", person.id, " ", person.name, " ", person.age,
      " ", person.address, " ", person.salary);
DatabaseFinalize(request);
```

De este modo, obtenemos inmediatamente los valores de todos los campos del registro actual y no necesitamos leerlos por separado.

Este ejemplo introductorio se ha extraído del artículo SQLite: trabajo nativo con bases de datos en SQL en MQL5, donde, además, se contemplan varias opciones de aplicación de la base de datos para operadores de trading. En concreto, puede encontrar allí la restauración del historial de posiciones de las operaciones, el análisis de un informe de trading en términos de estrategias, símbolos de trabajo u horas de trading preferidas, así como técnicas para trabajar con los resultados de la optimización.

Para dominar este material pueden ser necesarios algunos conocimientos básicos de SQL, por lo que los trataremos brevemente en las siguientes secciones.

7.6.1 Conceptos básicos de SQL

Todas las tareas realizadas en SQLite presuponen la presencia de una base de datos en funcionamiento (una o varias), por lo que crear y abrir una base de datos (de forma similar a un archivo) son operaciones marco obligatorias que establecen el necesario entorno de programación. En SQLite no existe la posibilidad de borrar la base de datos mediante programación, ya que se supone que basta con borrar el archivo de base de datos del disco.

Las acciones disponibles en el contexto de una base abierta pueden dividirse condicionalmente en los siguientes grupos principales:

- Creación y eliminación de tablas, así como modificación de sus esquemas, es decir, las descripciones de las columnas, incluida la identificación de tipos, nombres y restricciones
- Creación (adición), lectura, edición y eliminación de registros en tablas; estas operaciones se suelen designar con la abreviatura común CRUD (Create, Read, Update, Delete)
- Creación de consultas para seleccionar registros de una tabla o de una combinación de varias tablas conforme a condiciones complejas
- Optimización de algoritmos mediante la creación de índices en columnas seleccionadas, uso de vistas (vista), inclusión de acciones por lotes en transacciones, declaración de desencadenadores de procesamiento de eventos y otras herramientas avanzadas

En las bases de datos SQL, todas estas acciones se realizan mediante comandos (o sentencias) SQL reservados. Debido a las particularidades de la integración con MQL5, algunas de las acciones se realizan mediante funciones MQL5 integradas. Por ejemplo, la apertura, aplicación o cancelación de una transacción se realiza mediante la trinidad de funciones *DatabaseTransaction*, aunque el estándar SQL (y la implementación pública de SQLite) tiene los comandos SQL correspondientes (BEGIN TRANSACTION, COMMIT y ROLLBACK).

La mayoría de los comandos SQL también están disponibles en programas MQL: se pasan al motor de ejecución SQLite como parámetros de cadena de las funciones *DatabaseExecute* o *DatabasePrepare*. La diferencia entre estas dos opciones radica en varios matices.

DatabasePrepare permite preparar una consulta para su posterior ejecución cíclica masiva con diferentes valores de parámetros en cada iteración (los parámetros en sí, es decir, sus nombres en la consulta, son los mismos). Además, estas consultas preparadas proporcionan un mecanismo para leer los resultados utilizando *DatabaseRead* y *DatabaseReadBind*. Por lo tanto, puede utilizarlos para operaciones con un conjunto de registros seleccionados.

Por el contrario, la función *DatabaseExecute* ejecuta la consulta única pasada de forma unilateral: el comando entra en el motor SQLite, realiza algunas acciones sobre los datos, pero no devuelve nada. Suele utilizarse para crear tablas o modificar datos por lotes.

En el futuro, a menudo tendremos que operar con varios conceptos básicos. Vamos a presentarlos:

Tabla - un conjunto estructurado de datos, formado por filas y columnas. Cada fila es un registro de datos independiente con campos (propiedades) descritos mediante el nombre y el tipo de las columnas correspondientes. Todas las tablas de la base de datos se almacenan físicamente en el archivo de la base de datos y están disponibles para lectura y escritura (si no se restringieron los derechos al abrir la base de datos).

Vista - un tipo de tabla virtual calculada por el motor SQLite a partir de una consulta SQL dada u otras tablas o vistas. Las vistas son de sólo lectura. A diferencia de las tablas (incluidas las temporales que SQL permite crear en memoria durante una sesión de programa), las vistas se recalculan de forma dinámica cada vez que se accede a ellas.

Índice - una estructura de datos de servicio (el árbol equilibrado, árbol B) para la búsqueda rápida de registros por los valores de campos predefinidos (propiedades) o sus combinaciones.

Disparador - una subrutina de una o más sentencias SQL asignada para ejecutarse automáticamente en respuesta a eventos (antes o después) de añadir, modificar o eliminar un registro en una tabla determinada.

He aquí una breve lista de las sentencias SQL más populares y las acciones que realizan:

- CREATE - crea un objeto de base de datos (tabla, vista, índice, disparador);
- ALTER - modifica un objeto (tabla);
- DROP - elimina un objeto (tabla, vista, índice, disparador);
- SELECT - selecciona registros o calcula valores que satisfacen las condiciones dadas;
- INSERT - añade nuevos datos (uno o un conjunto de registros);
- UPDATE - modifica los registros existentes;
- DELETE - elimina registros de la tabla;

La lista sólo muestra las palabras clave que inician la construcción del lenguaje SQL correspondiente. A continuación se muestra una sintaxis más detallada. Su aplicación práctica se mostrará en los siguientes ejemplos.

Cada sentencia puede abarcar varias líneas (se ignoran los caracteres de salto de línea y los espacios adicionales). Si es necesario, puede enviar varios comandos a SQLite a la vez. En este caso, después de cada comando, debe utilizar el carácter de terminación de comando ';' (punto y coma).

El texto de los comandos es analizado por el sistema sin tener en cuenta mayúsculas y minúsculas, pero en SQL es habitual escribir las palabras clave en mayúsculas.

Al crear una tabla, debemos especificar su nombre, así como una lista de columnas entre paréntesis, separadas por comas. A cada columna se le asigna un nombre, un tipo y, opcionalmente, una restricción. La forma más sencilla:

```
CREATE TABLE table_name
  ( column_name type [ constraints ] [, column_name type [ constraints ...] ...]);
```

Veremos las restricciones en SQL en la sección siguiente. Mientras tanto, veamos un ejemplo claro (con distintos tipos y opciones):

```
CREATE TABLE IF NOT EXISTS example_table
   (id INTEGER PRIMARY KEY,
    name TEXT,
    timestamp INTEGER DEFAULT CURRENT_STAMP,
    income REAL,
    data BLOB);
```

La sintaxis para crear un índice es:

```
CREATE [ UNIQUE ] INDEX index_name
  ON table_name( column_name [, column_name ...]);
```

Los índices existentes se utilizan automáticamente en consultas con condiciones de filtro en las columnas correspondientes. Sin índices, el proceso es más lento.

Borrar una tabla (junto con los datos, si se ha escrito algo en ella) es bastante sencillo:

```
DROP TABLE table_name;
```

Puede insertar datos en una tabla de la siguiente manera:

```
INSERT INTO table_name [ ( column_name [, column_name ...] ) ]
  VALUES( value [, value ...]);
```

La primera lista entre paréntesis incluye los nombres de las columnas y es opcional (véase la explicación más abajo). Debe coincidir con la segunda lista con valores para ellos. Por ejemplo:

```
INSERT INTO example_table (name, income) VALUES ('Morning Flat Breakout', 1000);
```

Tenga en cuenta que los literales de cadena se encierran entre comillas simples en SQL.

Si se omiten los nombres de las columnas en la sentencia INSERT, se supone que la palabra clave VALUES va seguida de los valores de todas las columnas de la tabla, y en el orden exacto en que se describen en la tabla.

También existen formas más complejas del operador, que permiten, en particular, insertar registros de otras tablas o resultados de consultas.

La selección de registros por condición, con una limitación opcional de la lista de campos devueltos (columnas), se realiza mediante el comando SELECT.

```
SELECT column_name [, column_name ...] FROM table_name [WHERE condition ];
```

Si desea devolver todos los registros coincidentes en su totalidad (todas las columnas), utilice la notación estrella:

```
SELECT *FROM table_name [WHERE condition ];
```

Cuando la condición no está presente, el sistema devuelve todos los registros de la tabla.

Como condición, puede sustituir una expresión lógica que incluya nombres de columnas y varios operadores de comparación, así como funciones SQL integradas y los resultados de una consulta SELECT anidada (dichas consultas se escriben entre paréntesis). Entre los operadores de comparación se incluyen:

- AND lógico
- OR lógico
- IN para un valor de la lista
- NOT IN para un valor fuera de la lista
- BETWEEN para un valor del intervalo
- LIKE - similar en ortografía a un patrón con caracteres comodín especiales ('%', '_')
- EXISTS - comprueba que los resultados de la consulta anidada no estén vacíos

Por ejemplo, una selección de nombres de registros con un ingreso de al menos 1000 y no más de un año de antigüedad (redondeado preliminarmente al mes más próximo):

```
SELECT name FROM example_table
  WHERE income >= 1000 AND timestamp > datetime('now', 'start of month', '-1 year');
```

Además, la selección puede ordenarse de forma ascendente o descendente (ORDER BY), agruparse por características (GROUP BY) y filtrarse por grupos (HAVING). También podemos limitar el número de registros que contiene (LIMIT, OFFSET). Para cada grupo, puede devolver el valor de cualquier función agregada, en particular, COUNT, SUM, MIN, MAX y AVG, calculado sobre todos los registros del grupo.

```
SELECT [ DISTINCT ] column_name [, column_name...](i) FROM table_name
  [ WHERE condition ]
  [ORDER BY column_name [ ASC | DESC ]
     [ LIMIT quantity OFFSET start_offset ] ]
  [ GROUP BY column_name [ HAVING condition ] ];
```

La palabra clave opcional DISTINCT permite eliminar duplicados (si se encuentran en los resultados según los criterios de selección actuales). Sólo tiene sentido en ausencia de agrupación.

LIMIT sólo dará resultados reproducibles si la clasificación está presente.

Si es necesario, la selección SELECT puede hacerse no a partir de una tabla, sino de varias, combinándolas según la combinación de campos requerida. Para ello se utiliza la palabra clave JOIN.

```
SELECT [...] FROM table name_1
  [ INNER | OUTER | CROSS ] JOIN table_name_2
  ON boolean_condition
```

o

```
SELECT [...] FROM table name_1
  [ INNER | OUTER | CROSS ] JOIN table_name_2
  USING ( common_column_name [, common_column_name ...] )
```

SQLite admite tres tipos de JOINs: INNER JOIN, OUTER JOIN y CROSS JOIN. El libro proporciona una idea general de ellos a partir de ejemplos, mientras que usted puede profundizar en los detalles por su cuenta.

Por ejemplo, con JOIN, puede construir todas las combinaciones de registros de una tabla con registros de otra tabla o comparar transacciones de la tabla de transacciones (llamémoslas «transacciones») con transacciones de la misma tabla según el principio de coincidencia de identificadores de posición, pero de tal forma que la dirección de las transacciones (entrada al mercado/salida del mercado) fuera la opuesta, dando como resultado una tabla virtual de operaciones.

```
SELECT // list the columns of the results table with aliases (after 'as')
  d1.time as time_in, d1.position_id as position, d1.type as type, // table d1
   d1.volume as volume, d1.symbol as symbol, d1.price as price_in,
  d2.time as time_out, d2.price as price_out,                      // table d2
   d2.swap as swap, d2.profit as profit,
  d1.commission + d2.commission as commission                      // combination
  FROM deals d1 INNER JOIN deals d2      // d1 and d2 - aliases of one table "deals"
  ON d1.position_id = d2.position_id     // merge condition by position
  WHERE d1.entry = 0 AND d2.entry = 1    // selection condition "entry/exit"
```

Esta es una consulta SQL de la ayuda de MQL5, donde hay ejemplos de JOIN en las descripciones de las funciones *DatabaseExecute* y *DatabasePrepare*.

La propiedad fundamental de SELECT es que siempre devuelve resultados al programa que la llama, a diferencia de otras consultas como CREATE, INSERT, etc. Sin embargo, a partir de SQLite 3,35, las sentencias INSERT, UPDATE y DELETE también tienen la capacidad de devolver valores, si es necesario, utilizando la palabra clave adicional RETURNING. Por ejemplo:

```
INSERT INTO example_table (name, income) VALUES ('Morning Flat Breakout', 1000)
   RETURNING id;
```

En cualquier caso, en MQL5 se accede a los resultados de las consultas a través de las funciones *DatabaseColumn*, *DatabaseRead* y *DatabaseReadBind*.

Además, SELECT permite evaluar los resultados de expresiones y devolverlos tal cual o combinarlos con resultados de tablas. Las expresiones pueden incluir la mayoría de los operadores que conocemos de expresiones MQL5, así como funciones SQL integradas. Consulte la documentación de SQLite para obtener una lista completa. Por ejemplo, así es como puede encontrar la versión actual de SQLite en su terminal y en el editor, lo que puede ser importante para saber qué opciones están disponibles.

```
SELECT sqlite_version();
```

En este caso, toda la expresión consiste en una única llamada a la función *sqlite_version*. De forma similar a la selección de varias columnas de una tabla, puede evaluar varias expresiones separadas por comas.

También están disponibles varias funciones estadísticas y matemáticas populares.

Los registros deben editarse con una sentencia UPDATE.

```
UPDATE table_name SET column_name = value [, column_name = value ...]
  WHERE condition;
```

La sintaxis del comando de eliminación es la siguiente:

```
DELETE FROM table_name WHERE condition;
```

7.6.2 Estructura de tablas: tipos de datos y restricciones

Al describir los campos de la tabla, es necesario especificar los tipos de datos para ellos, pero el concepto de un tipo de datos en SQLite es muy diferente de MQL5.

MQL5 es un lenguaje fuertemente tipado: cada variable o campo de estructura siempre conserva el tipo de datos según la declaración. SQL, por su parte, es un lenguaje poco tipado: los tipos que especificamos en la descripción de la tabla no son más que una recomendación. El programa puede escribir un valor de un tipo arbitrario en cualquier «celda» (un campo del registro), y la «celda» cambiará de tipo, lo que, en concreto, puede ser detectado por la función MQL integrada *DatabaseColumnType*.

Por supuesto, en la práctica, la mayoría de los usuarios tienden a ceñirse a los tipos de columna de «respeto».

La segunda diferencia significativa en el mecanismo de tipos de SQL es la presencia de un gran número de palabras clave que describen los tipos, pero todas estas palabras se reducen en última instancia a cinco clases de almacenamiento. Al ser una versión simplificada de SQL, SQLite en la mayoría de los casos no distingue entre palabras clave del mismo grupo (por ejemplo, en la descripción de una cadena con un límite de longitud VARCHAR(80), este límite no se controla, y la descripción es equivalente a la clase de almacenamiento TEXT), por lo que es más lógico describir el tipo por el nombre del grupo. Los tipos específicos se dejan solo por compatibilidad con otros SGBD (pero esto no es importante para nosotros).

En la siguiente tabla se enumeran los tipos MQL5 y sus correspondientes «afinidades» (que significan características generalizadoras de los tipos SQL).

Tipos MQL5	Tipos SQL genéricos
NULL (no es un tipo en MQL5)	NULL (sin valor)
bool, char, short, int, long, uchar, ushort, uint, ulong, datetime, color, enum	INTEGER
float, double	REAL
(número real de precisión fija, no analógico en MQL5)	NUMERIC
cadena	TEXT
(datos «en bruto» arbitrarios, análogo de uchar[] array u otros)	BLOB (objeto binario de gran tamaño), NINGUNO

Al escribir un valor en la base de datos SQL determina su tipo según varias reglas:

- La ausencia de comillas, punto decimal o exponente da INTEGER

- La presencia de un punto decimal y un exponente significa REAL
- El enmarcado de comillas simples o dobles señala el tipo TEXT
- Un valor NULL sin comillas corresponde a la clase NULL
- Los literales (constantes) con datos binarios se escriben como una cadena hexadecimal prefijada con 'x'

La función SQL especial *typeof* permite comprobar el tipo de un valor. Por ejemplo, se puede ejecutar la siguiente consulta en MetaEditor.

```
SELECT typeof(100), typeof(10.0), typeof('100'), typeof(x'1000'), typeof(NULL);
```

Enviará lo siguiente a la tabla de resultados:

```
integer       |      real   |      text   |      blob   |      null
```

No puede comprobar valores para NULL comparando '=' (porque el resultado también dará NULL), debe utilizar el operador especial NOT NULL.

SQLite impone algunos límites a los datos almacenados: algunos de ellos son difíciles de alcanzar (y por tanto los omitiremos aquí), pero otros pueden tenerse en cuenta a la hora de diseñar un programa. Así, el número máximo de columnas de la tabla es 2000, y el tamaño de una fila, BLOB, y en general de un registro, no puede superar el millón de bytes. Se elige el mismo valor que el límite de longitud de la consulta SQL.

En cuanto a fechas y horas, SQL puede en teoría almacenarlas en tres formatos, pero sólo el primero coincide con *datetime* en MQL5:

- INTEGER - el número de segundos desde 1970.01.01 (también conocido como «Unix epoch»)
- REAL - el número de días (con fracciones) desde el 24 de noviembre de 4714 a.C.
- TEXT - fecha y hora con precisión al milisegundo en el formato «AAAA-MM-DD HH:mm:SS.sss», opcionalmente con la zona horaria, para lo cual se añade el sufijo «[±]HH:mm» con un desfase respecto a UTC.

Un tipo de almacenamiento de fecha real (también llamado día juliano, para el que existe una función SQL integrada *Julianday*) es interesante porque permite almacenar el tiempo con una precisión de milisegundos. En teoría, esto también se puede hacer como una cadena de formato 'AAAA-MM-DDTHH:mm:SS.sssZ', pero dicho almacenamiento es muy poco económico. La conversión del «día» en el número de segundos con una parte fraccionaria, a partir de la fecha familiar 1970.01.01 00:00:00, se realiza según la fórmula: *julianday('now') - 2440587.5) * 86400.0*. 'Ahora' denota aquí la hora UTC actual, pero puede cambiarse a otros valores descritos en la documentación de SQLite. La constante 2440587.5 es exactamente igual al número de días «naturales» para la fecha «cero» especificada: el punto de inicio de la «época Unix».

Además del tipo, cada campo puede tener una o varias restricciones, que se escriben con palabras clave especiales después del tipo. Una restricción describe los valores que puede tomar el campo e incluso permite automatizar la cumplimentación de acuerdo con la finalidad predefinida del campo.

Consideremos las principales limitaciones.

```
... DEFAULT expression
```

Al añadir un nuevo registro, si no se especifica el valor del campo, el sistema introducirá automáticamente el valor (constante) especificado aquí o calculará la expresión (función).

```
... CHECK ( boolean_expression )
```

Al añadir un nuevo registro, el sistema comprobará que la expresión, que puede contener nombres de campo como variables, es verdadera. Si la expresión es falsa, el registro no se insertará y el sistema devolverá un error.

```
... UNIQUE
```

El sistema comprueba que todos los registros de la tabla tengan valores diferentes para este campo. Si se intenta añadir una entrada con un valor que ya existe, se producirá un error y la adición no tendrá lugar.

Para controlar la unicidad, el sistema crea implícitamente un índice para el campo especificado.

```
... PRIMARY KEY
```

Un campo marcado con este atributo es utilizado por el sistema para identificar registros en una tabla y enlaces a ellos desde otras tablas (así es como se forman las relaciones relacionales, que dan nombre a las bases de datos relacionales en cuestión como SQLite). Obviamente, esta función también incluye un índice único.

Si la tabla no tiene un campo de tipo INTEGER con el atributo PRIMARY KEY, el sistema crea automáticamente de forma implícita una columna de este tipo denominada *rowid*. Si su tabla tiene un campo entero declarado como clave primaria, también estará disponible con el alias *rowid*.

Si se añade a la tabla un registro con un *rowid* omitido o NULL, SQLite le asignará automáticamente el siguiente número entero (de 64 bits, correspondiente a *long* en MQL5), mayor que el máximo *rowid* de la tabla en 1. El valor inicial es 1.

Normalmente, el contador sólo se incrementa en 1 cada vez, pero si el número de registros insertados alguna vez en una tabla (y posiblemente borrados después) supera *long*, el contador saltará al principio y el sistema intentará encontrar números libres. Pero esto es poco probable. Por ejemplo, si escribe ticks en una tabla a una velocidad media de 1 tick por milisegundo, el desbordamiento se producirá en 292 millones de años.

Solo puede haber una clave primaria, pero puede constar de varias columnas, lo que se hace utilizando una sintaxis distinta de las restricciones directamente en la descripción de la tabla.

```
CREATE TABLE table_name (
  column_name type [ restrictions ]
  [, column_name type [ restrictions ] ...]
  , PRIMARY KEY ( column_name [, column_name ...] ) );
```

Volvamos a las restricciones.

```
... AUTOINCREMENT
```

Esta restricción solo puede especificarse como complemento de la CLAVE PRIMARIA, lo que garantiza que los identificadores se incrementen en todo momento. Esto significa que los ID anteriores, incluso los utilizados en las entradas eliminadas, no se volverán a seleccionar. No obstante, este mecanismo se implementa en SQLite de forma menos eficiente que una simple CLAVE PRIMARIA en términos de recursos informáticos y, por tanto, no se recomienda su uso.

```
... NOT NULL
```

Esta restricción prohíbe añadir un registro a la tabla en la que este campo no esté rellenado. De manera predeterminada, cuando no hay ninguna restricción, cualquier campo no único puede omitirse en el registro añadido y se establecerá como NULL.

```
... CURRENT_TIME
... CURRENT_DATE
... CURRENT_TIMESTAMP
```

Estas instrucciones permiten rellenar automáticamente un campo con la hora (sin fecha), la fecha (sin hora) o la hora UTC completa en el momento en que se insertó el registro (siempre que la sentencia SQL INSERT no escriba explícitamente nada en este campo, ni siquiera NULL). SQLite no sabe detectar automáticamente la hora de un cambio de registro de forma similar; para ello tendrá que escribir un disparador (lo que está fuera del alcance del libro).

Desafortunadamente, las restricciones del grupo CURRENT_TIMESTAMP se implementan en SQLite con una omisión: la marca de tiempo no se aplica si el campo es NULL. Esto distingue a SQLite de otros motores SQL y de cómo el propio SQLite maneja los NULL en los campos de clave primaria. Resulta que para el etiquetado automático no se puede escribir todo el objeto en la base de datos, sino que es necesario especificar de forma explícita todos los campos excepto el campo con la fecha y la hora. Para resolver el problema, necesitamos una opción alternativa en la que la función SQL STRFTIME('%s') se sustituya en la consulta compilada para las columnas correspondientes.

7.6.3 POO (MQL5) e integración SQL: concepto de ORM

El uso de una base de datos en un programa MQL implica que el algoritmo se divide en 2 partes: la parte de control se escribe en MQL5, y la parte de ejecución se escribe en SQL. Como resultado, el código fuente puede empezar a parecer un mosaico y requerir atención para mantener la coherencia. Para evitarlo, los lenguajes orientados a objetos han desarrollado el concepto de mapeo objeto-relacional (ORM, por sus siglas en inglés), es decir, el mapeo de objetos a registros de tablas relacionales y viceversa.

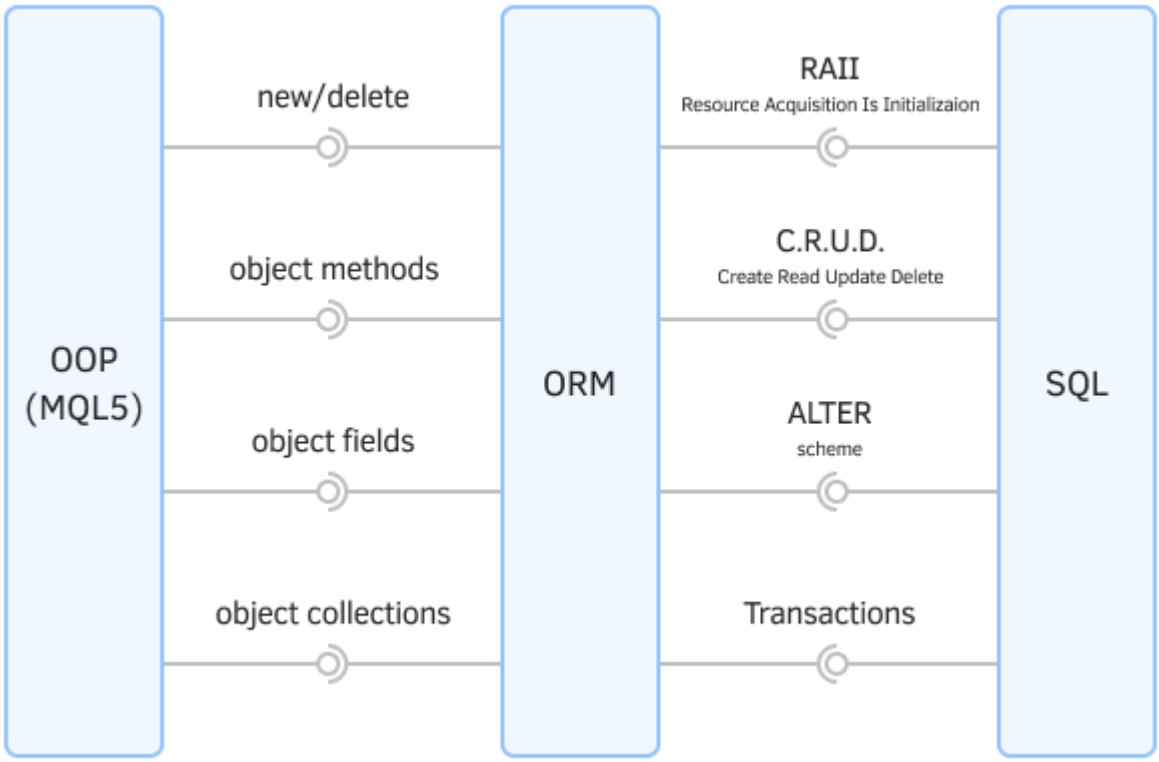
La esencia del enfoque consiste en encapsular todas las acciones del lenguaje SQL en clases/estructuras de una capa especial. Como resultado, la parte de aplicación del programa puede escribirse en un lenguaje puramente POO (programación orientada a objetos), como por ejemplo, MQL5, sin distraerse con los matices de SQL.

En presencia de una implementación ORM completa (en forma de «caja negra» con un conjunto de todos los comandos), un desarrollador de aplicaciones generalmente tiene la oportunidad de no aprender SQL.

Además, ORM permite cambiar «imperceptiblemente» el «motor» del SGBD si es necesario. Esto no es particularmente relevante para MQL5, porque solo la base de datos SQLite está integrada en él, pero algunos desarrolladores prefieren utilizar DBMS completos y conectarlos a MetaTrader 5 mediante la importación de DLLs.

El uso de objetos con constructores y destructores es muy útil cuando necesitamos adquirir y liberar recursos automáticamente. Hemos tratado este concepto (RAII, Resource Acquisition Is Initialization) en la sección Gestión de descriptores de archivos; sin embargo, como veremos más adelante, el trabajo con la base de datos también se basa en la asignación y liberación de distintos tipos de descriptores.

En la siguiente imagen se muestra esquemáticamente la interacción de diferentes capas de software al integrar la programación orientada a objetos (POO) y SQL en forma de ORM.



ORM, Mapeo Objeto-Relacional

Además, un «envoltorio» de objetos (no sólo un ORM específico de la base de datos) automatizará la preparación y transformación de los datos, además de comprobar su corrección para evitar algunos errores.

En las siguientes secciones, mientras recorremos las funciones integradas para trabajar con la base, implementaremos los ejemplos, construyendo gradualmente nuestra propia capa ORM sencilla. Debido a algunas especificidades de MQL5, nuestras clases no podrán proporcionar un universalismo que cubra el 100 % de las tareas, pero serán útiles para muchos proyectos.

7.6.4 Crear, abrir y cerrar bases de datos

Las funciones *DatabaseOpen* y *DatabaseClose* permiten crear y abrir bases de datos.

int DatabaseOpen(const string filename, uint flags)

La función abre o crea una base de datos en un archivo denominado *filename*. El parámetro puede contener no sólo el nombre, sino también la ruta con subcarpetas relativa a *MQL5/Files* (de una instancia de terminal específica o en una carpeta compartida, véanse las banderas más abajo). La extensión puede omitirse, lo que añade «.sqlite» al nombre por defecto.

Si se especifica NULL o una cadena vacía «» en el parámetro *filename*, la base de datos se crea en un archivo temporal, que se borrará automáticamente después de cerrar la base de datos.

Si se especifica la cadena «:memory:» en el parámetro *filename*, la base de datos se creará en memoria. Dicha base temporal se borrará automáticamente tras el cierre.

El parámetro *flags* contiene una combinación de banderas que describen condiciones adicionales para crear o abrir una base de datos de la enumeración ENUM_DATABASE_OPEN_FLAGS.

Identificador	Descripción
DATABASE_OPEN_READONLY	Abrir sólo para lectura
DATABASE_OPEN_READWRITE	Abrir para lectura y escritura
DATABASE_OPEN_CREATE	Crear un archivo en disco si no existe
DATABASE_OPEN_MEMORY	Crear una base de datos en memoria
DATABASE_OPEN_COMMON	El archivo se encuentra en la carpeta compartida de todos los terminales

Si no se especifica ninguno de los indicadores DATABASE_OPEN_READONLY o DATABASE_OPEN_READWRITE en el parámetro *flags*, se utilizará el indicador DATABASE_OPEN_READWRITE.

En caso de éxito, la función devuelve un manejador a la base de datos, que luego se utiliza como parámetro para que otras funciones accedan a ella. En caso contrario, se devuelve INVALID_HANDLE, y el código de error se puede encontrar en *_LastError*.

void DatabaseClose(int database)

La función *DatabaseClose* cierra la base de datos por su manejador, recibido previamente de la función *DatabaseOpen*.

Después de llamar a *DatabaseClose*, todos los manejadores de consulta que aprenderemos a crear para una base abierta en las siguientes secciones se eliminan e invalidan automáticamente.

La función no devuelve nada. Sin embargo, si se le pasa un manejador incorrecto, establecerá *_LastError* en ERR_DATABASE_INVALID_HANDLE.

Empecemos a desarrollar un envoltorio orientado a objetos para bases de datos en un archivo *DBSQLite.mqh*.

La clase DBSQ*lite* garantizará la creación, apertura y cierre de bases de datos. Lo ampliaremos más adelante.

```
class DBSQLite
{
protected:
   const string path;
   const int handle;
   const uint flags;

public:
   DBSQLite(const string file, const uint opts =
      DATABASE_OPEN_CREATE | DATABASE_OPEN_READWRITE):
      path(file), flags(opts), handle(DatabaseOpen(file, opts))
   {
   }

   ~DBSQLite(void)
   {
      if(handle != INVALID_HANDLE)
      {
         DatabaseClose(handle);
      }
   }

   int getHandle() const
   {
      return handle;
   }

   bool isOpen() const
   {
      return handle != INVALID_HANDLE;
   }
};
```

Tenga en cuenta que la base de datos se crea o se abre automáticamente cuando se crea el objeto, y se cierra cuando se destruye el objeto.

Usando esta clase, vamos a escribir un script *DBinit.mq5* sencillo, que creará o abrirá la base de datos especificada.

```
input string Database = "MQL5Book/DB/Example1";

void OnStart()
{
   DBSQLite db(Database);                    // create or open the base in the constru
   PRTF(db.getHandle());                     // 65537 / ok
   PRTF(FileIsExist(Database + ".sqlite")); // true / ok
} // the base is closed in the destructor
```

Tras la primera ejecución, con la configuración por defecto, deberíamos obtener un nuevo archivo *MQL5/Files/MQL5Book/DB/Example1.sqlite*. Esto se confirma en el código comprobando la existencia del archivo. En ejecuciones posteriores con el mismo nombre, el script simplemente abre la base de datos y registra el descriptor actual (un número entero).

7.6.5 Ejecutar consultas sin enlace de datos MQL5

Algunas consultas SQL son comandos que basta con enviar al motor tal cual. No requieren ni entrada de variables ni resultados. Por ejemplo, si nuestro programa MQL necesita crear una tabla, índice o vista con una determinada estructura y nombre en la base de datos, podemos escribirlo como una cadena constante con la sentencia «CREATE...». Además, es conveniente utilizar este tipo de consultas para el tratamiento por lotes de registros o su combinación (fusión, cálculo de indicadores agregados y modificaciones del mismo tipo). Es decir, con una sola consulta, puede convertir los datos de toda la tabla o rellenar otras tablas basándose en ella. Estos resultados pueden analizarse en las consultas posteriores.

En todos estos casos, sólo es importante obtener la confirmación del éxito de la acción. Las solicitudes de este tipo se realizan mediante la función *DatabaseExecute*.

bool DatabaseExecute(int database, const string sql)

La función ejecuta una consulta en la base de datos especificada por el descriptor *database*. La propia solicitud se envía como una cadena lista *sql*.

La función devuelve un indicador de éxito (*true*) o de error (*false*).

Por ejemplo, podemos complementar nuestra clase *DBSQLite* con este método (el descriptor ya está dentro del objeto).

```
class DBSQLite
{
   ...
   bool execute(const string sql)
   {
      return DatabaseExecute(handle, sql);
   }
};
```

Entonces el script que crea una nueva tabla (y, si es necesario, previamente, la propia base de datos) puede tener este aspecto (*DBcreateTable.mq5*):

```
input string Database = "MQL5Book/DB/Example1";
input string Table = "table1";

void OnStart()
{
   DBSQLite db(Database);
   if(db.isOpen())
   {
      PRTF(db.execute(StringFormat("CREATE TABLE %s (msg text)", Table))); // true
   }
}
```

Después de ejecutar el script, intente abrir la base de datos especificada en MetaEditor y asegúrese de que contiene una tabla vacía con un único campo de texto «msg». Pero también puede hacerse mediante programación (véase la sección siguiente).

Si ejecutamos el script por segunda vez con los mismos parámetros, obtendremos un error (aunque no crítico, sin forzar el cierre del programa).

```
database error, table table1 already exists
db.execute(StringFormat(CREATE TABLE %s (msg text),Table))=false / DATABASE_ERROR(560
```

Esto se debe a que no se puede volver a crear una tabla existente. Pero SQL permite suprimir este error y crear una tabla sólo si aún no existe, de lo contrario, permite no hacer casi nada y devolver un indicador de éxito. Para ello, basta con añadir «IF NOT EXISTS» delante del nombre en la consulta.

```
db.execute(StringFormat("CREATE TABLE IF NOT EXISTS %s (msg text)", Table));
```

En la práctica, las tablas son necesarias para almacenar información sobre objetos del área de aplicación, como cotizaciones, operaciones y señales de trading. Por lo tanto, es deseable automatizar la creación de tablas basadas en la descripción de objetos en MQL5. Como veremos más adelante, las funciones SQLite ofrecen la posibilidad de vincular los resultados de las consultas a estructuras MQL5 (pero no a clases). En este sentido, en el marco del envoltorio ORM, desarrollaremos un mecanismo para generar la consulta SQL «CREATE TABLE» de acuerdo con la descripción *struct* del tipo específico en MQL5.

Para ello es necesario registrar de algún modo los nombres y tipos de los campos de estructura en la lista general en el momento de la compilación y, a continuación, ya en la fase de ejecución del programa, se pueden generar consultas SQL a partir de esta lista.

En la fase de compilación se analizan varias categorías de entidades MQL5, que pueden utilizarse para identificar tipos y nombres:

- macros
- herencia
- plantillas

En primer lugar, hay que recordar que las descripciones de los campos recopilados están relacionadas con el contexto de una estructura concreta y no deben mezclarse, porque el programa puede contener muchas estructuras diferentes con nombres y tipos potencialmente coincidentes. En otras palabras: es deseable acumular información en listas separadas para cada tipo de estructura. Para ello es ideal un tipo de plantilla cuyo parámetro (S) será la estructura de la aplicación. Llamemos a la plantilla *DBEntity*.

```
template<typename S>
struct DBEntity
{
   static string prototype[][3]; // 0 - type, 1 - name, 2 - constraints
   ...
};

template<typename T>
static string DBEntity::prototype[][3];
```

Dentro de la plantilla hay un array multidimensional *prototype*, en el que escribiremos la descripción de los campos. Para interceptar el tipo y el nombre del campo aplicado, tendrá que declarar otra estructura de plantilla, *DBField*, dentro de *DBEntity*: esta vez su parámetro T es el tipo del propio campo. En el constructor, tenemos información sobre este tipo (*typename(T)*), y también obtenemos el nombre del campo (y opcionalmente, la restricción) como parámetros.

```
template<typename S>
struct DBEntity
{
   ...
   template<typename T>
   struct DBField
   {
      T f;
      DBField(const string name, const string constraints = "")
      {
         const int n = EXPAND(prototype);
         prototype[n][0] = typename(T);
         prototype[n][1] = name;
         prototype[n][2] = constraints;
      }
   };
```

El campo *f* no se utiliza, pero es necesario porque las estructuras no pueden estar vacías.

Supongamos que tenemos una estructura de aplicación *Data* (*DBmetaProgramming.mq5*).

```
struct Data
{
   long id;
   string name;
   datetime timestamp;
   double income;
};
```

Podemos hacer que su análogo heredado de *DBEntity<DataDB>*, pero con campos sustituidos basados en *DBField*, sea idéntico al conjunto original.

```
struct DataDB: public DBEntity<DataDB>
{
   DB_FIELD(long, id);
   DB_FIELD(string, name);
   DB_FIELD(datetime, timestamp);
   DB_FIELD(double, income);
} proto;
```

Al sustituir el nombre de la estructura en el parámetro de la plantilla padre, la estructura proporciona al programa información sobre sus propias propiedades.

Preste atención a la definición única de la variable *proto* junto con la declaración de la estructura. Esto es necesario porque, en las plantillas, cada tipo parametrizado específico sólo se compila si se crea al menos un objeto de este tipo en el código fuente. Es importante para nosotros que la creación de este proto-objeto se produzca al principio del lanzamiento del programa, en el momento de la inicialización de las variables globales.

Una macro se oculta bajo el identificador DB_FIELD:

```
#define DB_FIELD(T,N) struct T##_##N: DBField<T> { T##_##N() : DBField<T>(#N) { } } \
   _##T##_##N;
```

Así es como se expande para un solo campo:

```
struct Type_Name: DBField<Type>
{
   Type_Name() : DBField<Type>(Name) { }
} _Type_Name;
```

Aquí la estructura no sólo se define, sino que se crea al instante: de hecho, sustituye al campo original.

Dado que la estructura *DBField* contiene una única variable *f* del tipo deseado, las dimensiones y la representación binaria interna de *Data* y *DataDB* son idénticas. Esto puede comprobarse fácilmente ejecutando el script *DBmetaProgramming.mq5*.

```
void OnStart()
{
   PRTF(sizeof(Data));
   PRTF(sizeof(DataDB));
   ArrayPrint(DataDB::prototype);
}
```

Éste envía al registro lo siguiente:

```
DBEntity<Data>::DBField<long>::DBField<long>(const string,const string)
long id
DBEntity<Data>::DBField<string>::DBField<string>(const string,const string)
string name
DBEntity<Data>::DBField<datetime>::DBField<datetime>(const string,const string)
datetime timestamp
DBEntity<Data>::DBField<double>::DBField<double>(const string,const string)
double income
sizeof(Data)=36 / ok
sizeof(DataDB)=36 / ok
            [,0]        [,1]        [,2]
[0,] "long"      "id"        ""
[1,] "string"    "name"      ""
[2,] "datetime"  "timestamp" ""
[3,] "double"    "income"    ""
```

Sin embargo, para acceder a los campos, tendría que escribir algo inconveniente: *data._long_id.f*, *data._string_name.f*, *data._datetime_timestamp.f*, *data._double_income.f*.

No lo haremos, no sólo y no tanto por inconveniencia, sino porque esta forma de construir metaestructuras no es compatible con los principios de vinculación de datos a consultas SQL. En las siguientes secciones exploraremos las funciones de *database* que permiten obtener registros de tablas y resultados de consultas SQL en estructuras MQL5. No obstante, sólo se permite utilizar estructuras simples sin herencia y miembros estáticos de tipos de objeto. Por lo tanto, es necesario modificar ligeramente el principio de revelación de la metainformación.

Tendremos que dejar sin cambios los tipos de estructuras originales y repetir de hecho la descripción para la base de datos, asegurándonos de que no haya discrepancias (erratas). Esto no es muy conveniente, pero no hay otra manera en este momento.

Trasladaremos la declaración de instancias *DBEntity* y *DBField* más allá de las estructuras de aplicación. En este caso, la macro DB_FIELD recibirá un parámetro adicional (S), en el que será necesario pasar el tipo de la estructura de la aplicación (antes se tomaba implícitamente declarándolo dentro de la propia estructura).

```
#define DB_FIELD(S,T,N) \
   struct S##_##T##_##N: DBEntity<S>::DBField<T> \
   { \
      S##_##T##_##N() : DBEntity<S>::DBField<T>(#N) {} \
   }; \
   const S##_##T##_##N _##S##_##T##_##N;
```

Dado que las columnas de la tabla pueden tener restricciones, también habrá que pasarlas al constructor de *DBField* si es necesario. Para ello, vamos a añadir un par de macros con los parámetros adecuados (en teoría, una columna puede tener varias restricciones, pero normalmente no más de dos).

```
#define DB_FIELD_C1(S,T,N,C1) \
   struct S##_##T##_##N: DBEntity<S>::DBField<T> \
   {
      S##_##T##_##N() : DBEntity<S>::DBField<T>(#N, C1) {} \
   }; \
   const S##_##T##_##N _##S##_##T##_##N;

#define DB_FIELD_C2(S,T,N,C1,C2) \
   struct S##_##T##_##N: DBEntity<S>::DBField<T> \
   { \
      S##_##T##_##N() : DBEntity<S>::DBField<T>(#N, C1 + " " + C2) {} \
   }; \
   const S##_##T##_##N _##S##_##T##_##N;
```

Las tres macros, así como otros desarrollos, se añaden al archivo de encabezado *DBSQLite.mqh*.

> Es importante tener en cuenta que esta vinculación «propia» de los objetos a una tabla sólo es necesaria para introducir datos en la base de datos, ya que la lectura de datos de una tabla a un objeto se implementa en MQL5 mediante la función *DatabaseReadBind*.

Mejoremos también la aplicación de *DBField*. Los tipos MQL5 no se corresponden exactamente con las clases de almacenamiento SQL, por lo que es necesario realizar una conversión al rellenar el elemento *prototype[n][0]*. Para ello se utiliza el método estático *affinity*.

```
template<typename T>
struct DBField
{
   T f;
   DBField(const string name, const string constraints = "")
   {
      const int n = EXPAND(prototype);
      prototype[n][0] = affinity(typename(T));
      ...
   }

   static string affinity(const string type)
   {
      const static string ints[] =
      {
         "bool", "char", "short", "int", "long",
         "uchar", "ushort", "uint", "ulong", "datetime",
         "color", "enum"
      };
      for(int i = 0; i < ArraySize(ints); ++i)
      {
         if(type == ints[i]) return DB_TYPE::INTEGER;
      }

      if(type == "float" || type == "double") return DB_TYPE::REAL;
      if(type == "string") return DB_TYPE::TEXT;
      return DB_TYPE::BLOB;
   }
};
```

Las constantes de texto de los tipos genéricos SQL utilizados aquí se colocan en un espacio de nombres separado: pueden ser necesarios en diferentes lugares en los programas MQL en algún momento, y es necesario asegurarse de que no hay conflictos de nombres.

```
namespace DB_TYPE
{
   const string INTEGER = "INTEGER";
   const string REAL = "REAL";
   const string TEXT = "TEXT";
   const string BLOB = "BLOB";
   const string NONE = "NONE";
   const string _NULL = "NULL";
}
```

Los preajustes de las posibles restricciones también se describen en su grupo para mayor comodidad (a modo de pista).

```
namespace DB_CONSTRAINT
{
   const string PRIMARY_KEY = "PRIMARY KEY";
   const string UNIQUE = "UNIQUE";
   const string NOT_NULL = "NOT NULL";
   const string CHECK = "CHECK (%s)"; // requires an expression
   const string CURRENT_TIME = "CURRENT_TIME";
   const string CURRENT_DATE = "CURRENT_DATE";
   const string CURRENT_TIMESTAMP = "CURRENT_TIMESTAMP";
   const string AUTOINCREMENT = "AUTOINCREMENT";
   const string DEFAULT = "DEFAULT (%s)"; // requires an expression (constants, funct
}
```

Dado que algunas de las restricciones requieren parámetros (los lugares para ellos están marcados con el modificador de formato habitual «%s»), añadamos una comprobación de su presencia. Esta es la forma final del constructor *DBField*.

```
template<typename T>
struct DBField
{
   T f;
   DBField(const string name, const string constraints = "")
   {
      const int n = EXPAND(prototype);
      prototype[n][0] = affinity(typename(T));
      prototype[n][1] = name;
      if(StringLen(constraints) > 0       // avoiding error STRING_SMALL_LEN(5035)
         && StringFind(constraints, "%") >= 0)
      {
         Print("Constraint requires an expression (skipped): ", constraints);
      }
      else
      {
         prototype[n][2] = constraints;
      }
   }
```

Debido a que la combinación de macros y objetos auxiliares *DBEntity<S>* y *DBField<T>* rellena un array de prototipos, dentro de la clase *DBSQlite*, se hace posible implementar la generación automática de una consulta SQL para crear una tabla de estructuras.

El método *createTable* se modela con un tipo de estructura de aplicación y contiene un stub de consulta («CREATE TABLE %s %s (%s);»). Su primer argumento es la instrucción opcional «IF NOT EXISTS». El segundo parámetro es el nombre de la tabla, que por defecto se toma como el tipo del parámetro de plantilla *typename(S)*, pero se puede sustituir por otra cosa si es necesario utilizando el nombre del parámetro de entrada (si no es NULL). Por último, el tercer argumento entre paréntesis es la lista de columnas de la tabla: la forma el método de ayuda *columns* a partir del array *DBEntity<S>::prototype*.

```
class DBSQLite
{
   ...
   template<typename S>
   bool createTable(const string name = NULL,
      const bool not_exist = false, const string table_constraints = "") const
   {
      const static string query = "CREATE TABLE %s %s (%s);";
      const string fields = columns<S>(table_constraints);
      if(fields == NULL)
      {
         Print("Structure '", typename(S), "' with table fields is not initialized");
         SetUserError(4);
         return false;
      }
      // attempt to create an already existing table will give an error,
      // if not using IF NOT EXISTS
      const string sql = StringFormat(query,
         (not_exist ? "IF NOT EXISTS" : ""),
         StringLen(name) ? name : typename(S), fields);
      PRTF(sql);
      return DatabaseExecute(handle, sql);
   }

   template<typename S>
   string columns(const string table_constraints = "") const
   {
      static const string continuation = ",\n";
      string result = "";
      const int n = ArrayRange(DBEntity<S>::prototype, 0);
      if(!n) return NULL;
      for(int i = 0; i < n; ++i)
      {
         result += StringFormat("%s%s %s %s",
            i > 0 ? continuation : "",
            DBEntity<S>::prototype[i][1], DBEntity<S>::prototype[i][0],
            DBEntity<S>::prototype[i][2]);
      }
      if(StringLen(table_constraints))
      {
         result += continuation + table_constraints;
      }
      return result;
   }
};
```

Para cada columna, la descripción consta de un nombre, un tipo y una restricción opcional. Además, es posible pasar una restricción general a la tabla (*table_constraints*).

Antes de enviar la consulta SQL generada a la función *DatabaseExecute*, el método *createTable* produce una salida de depuración del texto de la consulta al registro (toda salida de este tipo en las clases ORM puede desactivarse de forma centralizada sustituyendo la macro PRTF).

Ahora todo está listo para escribir un script de prueba *DBcreateTableFromStruct.mq5*, que, por declaración de estructura, crearía la tabla correspondiente en SQLite. En el parámetro de entrada, establecemos sólo el nombre de la base de datos, y el programa elegirá el nombre de la propia tabla en función del tipo de estructura.

```
#include <MQL5Book/DBSQLite.mqh>

input string Database = "MQL5Book/DB/Example1";

struct Struct
{
   long id;
   string name;
   double income;
   datetime time;
};

DB_FIELD_C1(Struct, long, id, DB_CONSTRAINT::PRIMARY_KEY);
DB_FIELD(Struct, string, name);
DB_FIELD(Struct, double, income);
DB_FIELD(Struct, string, time);
```

En la función principal *OnStart*, creamos una tabla llamando a *createTable* con la configuración predeterminada. Si no queremos recibir una señal de error cuando intentemos crearlo la próxima vez, tenemos que pasar *true* como primer parámetro (*db.createTable<Struct>(true)*).

```
void OnStart()
{
   DBSQLite db(Database);
   if(db.isOpen())
   {
      PRTF(db.createTable<Struct>());
      PRTF(db.hasTable(typename(Struct)));
   }
}
```

El método *hasTable* comprueba la presencia de una tabla en la base de datos por el nombre de la tabla. Examinaremos la aplicación de este método en la sección siguiente. Ahora, vamos a ejecutar el script. Tras la primera ejecución, la tabla se crea correctamente y puede ver la consulta SQL en el registro (se muestra con saltos de línea, tal y como la formamos en el código).

```
sql=CREATE TABLE  Struct (id INTEGER PRIMARY KEY,
name TEXT ,
income REAL ,
time TEXT ); / ok
db.createTable<Struct>()=true / ok
db.hasTable(typename(Struct))=true / ok
```

La segunda ejecución devolverá un error de la llamada a *DatabaseExecute*, porque esta tabla ya existe, lo que se indica adicionalmente en el resultado de *hasTable*.

```
sql=CREATE TABLE  Struct (id INTEGER PRIMARY KEY,
name TEXT ,
income REAL ,
time TEXT ); / ok
database error, table Struct already exists
db.createTable<Struct>()=false / DATABASE_ERROR(5601)
db.hasTable(typename(Struct))=true / ok
```

7.6.6 Comprobar si una tabla existe en la base de datos

La función integrada *DatabaseTableExists* permite comprobar la existencia de una tabla por su nombre.

bool DatabaseTableExists(int database, const string table)

El descriptor de la base de datos y el nombre de la tabla se especifican en los parámetros. El resultado de la llamada a la función es *true* si la tabla existe.

Vamos a ampliar la clase *DBSQLite* añadiendo el método *hasTable*.

```
class DBSQLite
{
   ...
   bool hasTable(const string table) const
   {
      return DatabaseTableExists(handle, table);
   }
```

El script *DBcreateTable.mq5* comprobará si la tabla ha aparecido.

```
void OnStart()
{
   DBSQLite db(Database);
   if(db.isOpen())
   {
      PRTF(db.execute(StringFormat("CREATE TABLE %s (msg text)", Table)));
      PRTF(db.hasTable(Table));
   }
}
```

Una vez más, no se preocupe por la posibilidad de obtener un error al intentar volver a crear. Esto no afecta en modo alguno a la existencia de la tabla.

```
database error, table table1 already exists
db.execute(StringFormat(CREATE TABLE %s (msg text),Table))=false / DATABASE_ERROR(560
db.hasTable(Table)=true / ok
```

Dado que estamos escribiendo una clase de ayuda genérica *DBSQLite*, proporcionaremos un mecanismo para eliminar tablas en ella. SQL tiene el comando DROP para este propósito.

```
class DBSQLite
{
   ...
   bool deleteTable(const string name) const
   {
      const static string query = "DROP TABLE '%s';";
      if(!DatabaseTableExists(handle, name)) return true;
      if(!DatabaseExecute(handle, StringFormat(query, name))) return false;
      return !DatabaseTableExists(handle, name)
         && ResetLastErrorOnCondition(_LastError == DATABASE_NO_MORE_DATA);
   }

   static bool ResetLastErrorOnCondition(const bool cond)
   {
      if(cond)
      {
         ResetLastError();
         return true;
      }
      return false;
   }
```

Antes de ejecutar la consulta, comprobamos la existencia de la tabla y salimos inmediatamente si no existe.

Después de ejecutar la consulta, comprobamos además si la tabla se ha borrado volviendo a llamar a *DatabaseTableExists*. Dado que la ausencia de una tabla se marcará con el código de error DATABASE_NO_MORE_DATA, que es el resultado esperado para este método, borramos el código de error con *ResetLastErrorOnCondition*.

Puede ser más eficaz utilizar las capacidades de SQL para excluir un intento de eliminar una tabla inexistente: basta con añadir la frase «IF EXISTS» a la consulta. Por lo tanto, la versión final del método *deleteTable* está simplificada:

```
   bool deleteTable(const string name) const
   {
      const static string query = "DROP TABLE IF EXISTS '%s';";
      return DatabaseExecute(handle, StringFormat(query, name));
   }
```

Puede intentar escribir un script de prueba para borrar la tabla, pero tenga cuidado de no borrar una tabla de trabajo por error. Las tablas se borran inmediatamente con todos los datos, sin confirmación y sin posibilidad de recuperación. Para proyectos importantes, guarde copias de seguridad de la base de datos.

7.6.7 Preparar consultas vinculadas: DatabasePrepare

En muchos casos, es necesario incluir parámetros en las consultas SQL. Dado que la consulta SQL es «originalmente» una cadena que responde a una sintaxis especial, puede formarse mediante una simple llamada a *StringFormat* o por concatenación, añadiendo los valores de los parámetros en los lugares adecuados. Ya hemos utilizado esta técnica en consultas para crear una tabla («CREATE TABLE %s '% s' (%s);»), pero aquí sólo una parte de los parámetros contenía datos (la lista de valores se sustituía

por %s entre paréntesis), y el resto representaba una opción y un nombre de tabla. En esta sección, nos centraremos exclusivamente en la sustitución de datos en una consulta. Hacer esto de una manera SQL nativa es importante por varias razones.

En primer lugar, la consulta SQL sólo se pasa al motor SQLite como una cadena, y allí se analiza en componentes, se comprueba si es correcta y se «compila» de cierta manera (por supuesto, no se trata de un compilador MQL5). A continuación, la base de datos ejecuta la consulta compilada. Por eso entrecomillamos la palabra «originalmente».

Cuando hay que ejecutar la misma consulta con distintos parámetros (por ejemplo, al insertar muchos registros en una tabla; poco a poco nos acercamos a esta tarea), compilar y comprobar por separado la consulta para cada registro resulta bastante ineficaz. Es más correcto compilar la consulta una vez, y luego ejecutarla en bloque, simplemente sustituyendo valores diferentes.

Esta operación de compilación se denomina preparación de la consulta y la realiza la función *DatabasePrepare*.

Las consultas preparadas tienen un propósito más: con su ayuda, el motor SQLite devuelve los resultados de la ejecución de la consulta al código MQL5 (encontrará más información al respecto en las secciones Ejecutar consultas preparadas y Lectura por separado de los campos del registro de resultados de la consulta).

El último, pero no por ello menos importante, momento asociado a las consultas parametrizadas es que protegen su programa de posibles ataques de hackers denominados inyección SQL. En primer lugar, esto es crítico para las bases de datos de sitios públicos, donde la información introducida por los usuarios se registra en la base de datos incrustándola en consultas SQL: si en este caso se utiliza una simple sustitución de formato «%s», el usuario podrá introducir alguna cadena larga en lugar de los datos esperados con comandos SQL adicionales, y pasará a formar parte de la consulta SQL original, distorsionando su significado. Pero si la consulta SQL se compila, no puede modificarse por los datos de entrada: siempre se tratan como datos.

Aunque el programa MQL no es un programa servidor, puede almacenar la información recibida del usuario en la base de datos.

int DatabasePrepare(int database, const string sql, ...)

La función *DatabasePrepare* crea un manejador en la base de datos especificada para la consulta en la cadena *sql*. La *database* debe ser abierta previamente por la función *DatabaseOpen*.

Las ubicaciones de los parámetros de consulta se especifican en la cadena *sql* mediante fragmentos '?1', '?2', '?3', etc. La numeración significa el índice del parámetro utilizado en el futuro al asignarle un valor de entrada, en las funciones *DatabaseBind*. No es necesario que los números de la cadena *sql* vayan en orden y pueden repetirse si es necesario insertar el mismo parámetro en distintos lugares de la consulta.

> ¡Atención! La indexación en los fragmentos sustituidos '?n' comienza en 1, mientras que en las funciones *DatabaseBind* comienza en 0. Por ejemplo, el parámetro '?1' del cuerpo de la consulta obtendrá el valor al llamar a *DatabaseBind* en el índice 0, el parámetro '?2' en el índice 1, y así sucesivamente. Este desplazamiento constante de 1 se mantiene aunque haya huecos (accidentales o intencionados) en la numeración de los parámetros '?n'.

Si tiene previsto vincular todos los parámetros estrictamente en orden, puede utilizar una notación abreviada: en lugar de cada parámetro, indique simplemente el símbolo '?' sin número: en este caso, los parámetros se numeran automáticamente. Cualquier parámetro '?' sin número obtiene el número

que es en 1 mayor que el máximo de los parámetros leídos a la izquierda (con números explícitos o calculados según el mismo principio, y el primero obtendrá el número 1, es decir, '?1').

Así, la solicitud

```
SELECT * FROM table WHERE risk > ?1 AND signal = ?2
```

es equivalente a:

```
SELECT * FROM table WHERE risk > ? AND signal = ?
```

Si algunos de los parámetros son constantes o la consulta se está preparando para ejecutarse una sola vez con el fin de obtener un resultado, los valores de los parámetros se pueden pasar a la función *DatabasePrepare* como una lista separada por comas en lugar de una elipsis (igual que en *Print* o *Comment*).

> Los parámetros de consulta sólo pueden utilizarse para establecer valores en las columnas de la tabla (al escribir, modificar o filtrar condiciones). Los nombres de tablas, columnas, opciones y palabras clave SQL no pueden pasarse a través de parámetros '?'/'?n'.

La función *DatabasePrepare* por sí misma no satisface la consulta. El manejador que devuelve debe pasarse a las llamadas a función *DatabaseRead* o *DatabaseReadBind*. Estas funciones ejecutan la consulta y ponen el resultado a disposición para su lectura (puede ser un registro o muchos). Por supuesto, si hay marcadores de posición de parámetros ('?' o '?n') en la consulta, y los valores para ellos no se especificaron en *DatabasePrepare*, antes de ejecutar la consulta, es necesario vincular los parámetros y los datos utilizando las funciones *DatabaseBind* apropiadas.

Si no se asigna un valor a un parámetro, se sustituye por NULL durante la ejecución de la consulta.

En caso de error, la función *DatabasePrepare* devolverá INVALID_HANDLE.

En las secciones siguientes se presentará un ejemplo de utilización de *DatabasePrepare*, después de explorar otras características relacionadas con las consultas preparadas.

7.6.8 Borrar y reiniciar consultas preparadas

Dado que las consultas preparadas pueden ejecutarse varias veces, en un bucle para diferentes valores de parámetros, es necesario restablecer la consulta al estado inicial en cada iteración. De ello se encarga la función *DatabaseReset*. Pero no tiene sentido llamarlo si la consulta preparada se ejecuta una vez.

bool DatabaseReset(int request)

La función restablece las estructuras internas de consulta compiladas al estado inicial, de forma similar a cuando se llama a *DatabasePrepare*. Sin embargo, *DatabaseReset* no recompila la consulta, por lo que es muy rápido.

También es importante que la función no anule las vinculaciones de datos ya establecidas en la consulta si se ha realizado alguna. Así, si es necesario, puede cambiar el valor de un único parámetro o de un número reducido de ellos. Entonces, después de llamar a *DatabaseReset*, puede simplemente llamar a las funciones *DatabaseBind* sólo para los parámetros modificados.

> En el momento de escribir el libro, la API de MQL5 no proporcionaba una función para restablecer el enlace de datos, un análogo de la función *sqlite_clear_bindings* en la distribución estándar SQLite.

En el parámetro *request*, especifique el manejador válido de la consulta obtenido anteriormente de *DatabasePrepare*. Si se pasa un manejador de la consulta que se ha eliminado previamente con *DatabaseFinalize* (ver más abajo), se devolverá un error.

La función devuelve un indicador de éxito (*true*) o de error (*false*).

El principio general de trabajo con consultas recurrentes se muestra en el pseudocódigo que se muestra más abajo. Las funciones *DatabaseBind* y *DatabaseRead* se describirán en las secciones siguientes y se «empaquetarán» en clases ORM.

```
struct Data                                          // structure example
{
   long count;
   double value;
   string comment;
};
Data data[];
...                                                  // getting data array
int r =
     DatabasePrepare(db, "INSERT... (?, ?, ?)")); // compile query with parameters
for(int i = 0; i < ArraySize(data); ++i)             // data loop
{
   DatabaseBind(r, 0, data[i].count);                // make data binding to parameters
   DatabaseBind(r, 1, data[i].value);
   DatabaseBind(r, 2, data[i].comment);
   DatabaseRead(r);                                  // execute request
   ...                                               // analyze or save results
   DatabaseReset(r);                                 // initial state at each iteration
}
DatabaseFinalize(r);
```

Una vez que la consulta preparada ya no sea necesaria, deberá liberar los recursos informáticos que ocupa mediante *DatabaseFinalize*.

void DatabaseFinalize(int request)

La función elimina la consulta con el manejador especificado, creada en *DatabasePrepare*.

Si se pasa un descriptor incorrecto, la función registrará ERR_DATABASE_INVALID_HANDLE en *_LastError*.

Al cerrar la base de datos con *DatabaseClose*, todos los gestores de consulta creados para él se eliminan e invalidan automáticamente.

Vamos a complementar nuestra capa ORM (*DBSQLite.mqh*) con una nueva clase *DBQuery* para trabajar con consultas preparadas. Por ahora, sólo contendrá la funcionalidad de inicialización y desinicialización inherente al concepto RAII, pero pronto la ampliaremos.

```
class DBQuery
{
protected:
   const string sql;  // query
   const int db;      // database handle (constructor argument)
   const int handle;  // prepared request handle

public:
   DBQuery(const int owner, const string s): db(owner), sql(s),
      handle(PRTF(DatabasePrepare(db, sql)))
   {
   }

   ~DBQuery()
   {
      DatabaseFinalize(handle);
   }

   bool isValid() const
   {
      return handle != INVALID_HANDLE;
   }

   virtual bool reset()
   {
      return DatabaseReset(handle);
   }
   ...
};
```

En la clase *DBSQLite*, iniciamos la preparación de la solicitud en el método *prepare* creando una instancia de *DBQuery*. Todos los objetos de consulta se almacenarán en el array interno *queries* en forma de punteros automáticos, lo que permite que el código de llamada no siga su eliminación explícita.

```
class DBSQLite
{
   ...
protected:
   AutoPtr<DBQuery> queries[];
public:
   DBQuery *prepare(const string sql)
   {
      return PUSH(queries, new DBQuery(handle, sql));
   }
   ...
};
```

7.6.9 Vincular datos a parámetros de consulta: DatabaseBind/Array

Después de que la consulta SQL haya sido compilada por la función *DatabasePrepare* puede utilizar el manejador de consulta recibido para vincular datos a los parámetros de consulta, que es para lo que sirven las funciones *DatabaseBind* y *DatabaseBindArray*. Ambas funciones pueden invocarse no sólo inmediatamente después de crear una consulta en *DatabasePrepare*, sino también después de restablecer la solicitud a su estado inicial con *DatabaseReset* (si la solicitud se ejecuta muchas veces en un bucle).

El paso de vinculación de datos no siempre es necesario porque las consultas preparadas pueden no tener parámetros. Por regla general, esta situación se produce cuando una consulta devuelve datos de SQL a MQL5, y por lo tanto se requiere un descriptor de consulta: cómo leer los resultados de la consulta por sus manejadores se describe en las secciones sobre las funciones *DatabaseRead/DatabaseReadBind* y *DatabaseColumn*.

bool DatabaseBind(int request, int index, T value)

La función *DatabaseBind* establece el valor del parámetro *index* para la consulta con el manejador *request*. De manera predeterminada, la numeración empieza por 0 si los parámetros de la consulta están marcados con símbolos de sustitución '?' (sin número). Sin embargo, los parámetros pueden especificarse en la cadena de consulta y con un número (?1, '?5', ?21): en este caso, los índices reales que se pasen a la función deben ser 1 menos que el número correspondiente de la cadena. Esto se debe a que la numeración de la cadena de consulta empieza por 1.

Por ejemplo, la siguiente consulta requiere un parámetro (índice 0):

```
int r = DatabasePrepare(db, "SELECT * FROM table WHERE id=?");
DatabaseBind(r, 0, 1234);
```

Si se utilizara la sustitución «... id=?10» en la cadena de consulta, sería necesario llamar a *DatabaseBind* con el índice 9.

El *value* en el prototipo *DatabaseBind* puede ser de cualquier tipo simple o cadena. Si un parámetro necesita asignar datos de tipo compuesto (estructuras) o datos binarios arbitrarios que puedan representarse como un array de bytes, utilice la función *DatabaseBindArray*.

La función devuelve *true* si tiene éxito. En caso contrario, devuelve *false*.

bool DatabaseBindArray(int request, int index, T &array[])

La función *DatabaseBindArray* establece el valor del parámetro *index* como un array de tipo simple o de estructuras simples (incluidas cadenas) para la consulta con el manejador *request*. Esta función permite escribir BLOB y NULL (la ausencia de un valor que se considera un tipo independiente en SQL y no es igual a 0) a la base de datos.

Ahora volvamos a la clase *DBQuery* en el archivo *DBSQLite.mqh* y agreguemos compatibilidad con vinculación de datos.

```
class DBQuery
{
   ...
public:
   template<typename T>
   bool bind(const int index, const T value)
   {
      return PRTF(DatabaseBind(handle, index, value));
   }
   template<typename T>
   bool bindBlob(const int index, const T &value[])
   {
      return PRTF(DatabaseBindArray(handle, index, value));
   }

   bool bindNull(const int index)
   {
      static const uchar null[] = {};
      return bindBlob(index, null);
   }
   ...
};
```

BLOB es adecuado para transferir cualquier archivo a la base de datos sin cambios; por ejemplo, si primero lo lee en un array de bytes utilizando la función *FileLoad*.

La necesidad de vincular explícitamente un valor nulo no es tan obvia. Al insertar nuevos registros en la base de datos, el programa de llamada suele pasar sólo los campos que conoce, y todos los que falten (si no están marcados con la restricción NOT NULL o no tienen un valor DEFAULT diferente en la descripción de la tabla) serán automáticamente dejados igual a NULL por el motor. No obstante, cuando se utiliza el enfoque ORM, es conveniente escribir todo el objeto en la base de datos, incluido el campo con una clave primaria única (PRIMARY KEY). El nuevo objeto aún no tiene este identificador, ya que la propia base de datos lo añade cuando el objeto se escribe por primera vez, por lo que es importante vincular este campo en el nuevo objeto al valor NULL.

7.6.10 Ejecutar consultas preparadas: DatabaseRead/Bind

Las consultas preparadas se ejecutan mediante las funciones *DatabaseRead* y *DatabaseReadBind*. La primera función extrae los resultados de la base de datos de forma que posteriormente se puedan leer campos individuales de cada registro recibido a su vez como respuesta, y la segunda extrae cada registro coincidente en su totalidad, en forma de estructura.

bool DatabaseRead(int request)

En la primera llamada, después de *Database Prepare* o *DatabaseReset*, la función *DatabaseRead* ejecuta la consulta y establece el puntero interno de resultado de la consulta en el primer registro recuperado (si la consulta espera que se devuelvan registros). Las funciones *DatabaseColumn* permiten leer los valores de los campos del registro, es decir, las columnas especificadas en la consulta.

En las siguientes llamadas, la función *DatabaseRead* salta al siguiente registro de los resultados de la consulta hasta llegar al final.

La función devuelve *true* una vez completada con éxito. El valor *false* se utiliza como indicador de un error (por ejemplo, la base de datos puede estar bloqueada u ocupada), así como cuando se alcanza normalmente el final de los resultados, por lo que debe analizar el código en *_LastError*. En concreto, el valor ERR_DATABASE_NO_MORE_DATA (5126) indica que los resultados han finalizado.

> ¡Atención! Si *DatabaseRead* se utiliza para ejecutar consultas que no devuelven datos, como INSERT, UPDATE, etc., la función devuelve inmediatamente *false* y establece el código de error ERR_DATABASE_NO_MORE_DATA si la solicitud se ha realizado correctamente.

El patrón habitual de uso de la función se ilustra con el siguiente pseudocódigo (*DatabaseColumn* las funciones para los distintos tipos se presentan en la sección siguiente).

```
int r = DatabasePrepare(db, "SELECT... WHERE...?",
   param));                              //compiling the query(optional with parameters
while(DatabaseRead(r))                   // query execution (on the first iteration)
{                                        //    and loop through result records
   int count;
   DatabaseColumnInteger(r, 0, count); // read one field from the current record
   double number;
   DatabaseColumnDouble(r, 1, number); // read another field from the current record
   ...                                   // column types and numbers in record are dete
                                         // process the received values of count, numbe
}                                        // loop is interrupted when the end of the res
DatabaseFinalize(r);
```

Obsérvese que, dado que la consulta (lectura de datos condicionales) sólo se ejecuta una vez (en la primera iteración), no es necesario llamar a *DatabaseReset*, como hicimos al registrar los datos cambiantes. No obstante, si queremos volver a ejecutar la consulta y «recorrer» los nuevos resultados, sería necesario llamar a *DatabaseReset*.

bool DatabaseReadBind(int request, void &object)

La función *DatabaseReadBind* opera de forma similar a *DatabaseRead*: la primera llamada ejecuta la consulta SQL y, en caso de éxito (hay datos adecuados en el resultado), rellena la estructura *object* pasada por referencia con los campos del primer registro; las llamadas posteriores continúan moviendo el puntero interno a través de los registros de los resultados de la consulta, rellenando la estructura con los datos del siguiente registro.

La estructura debe tener sólo tipos numéricos y/o cadenas como miembros (no se permiten arrays), no puede heredar de ni contener miembros estáticos de tipos objeto.

El número de campos de la estructura *object* no debe superar el número de columnas de los resultados de la consulta; de lo contrario, obtendremos un error. El número de columnas puede determinarse dinámicamente mediante la función *DatabaseColumnsCount*; sin embargo, el llamante suele necesitar «conocer» de antemano la configuración de datos esperada según la solicitud original.

Si el número de campos de la estructura es inferior al número de campos del registro, se realizará una lectura parcial. El resto de los datos pueden obtenerse utilizando las funciones *DatabaseColumn* correspondientes.

Se supone que los tipos de campo de la estructura coinciden con los tipos de datos de las columnas de resultados. De lo contrario, se realizará una conversión automática implícita, que puede tener consecuencias inesperadas (por ejemplo, una cadena leída en un campo numérico dará 0).

En el caso más sencillo, cuando calculamos un determinado valor total para los registros de la base de datos, por ejemplo, llamando a una función agregada como *SUM(column)*, *COUNT(column)* o *AVERAGE(column)*, el resultado de la consulta será un único registro con un único campo.

```
SELECT SUM(swap) FROM trades;
```

Dado que la lectura de los resultados está relacionada con las funciones de *DatabaseColumn*, aplazaremos el desarrollo del ejemplo hasta la siguiente sección, donde éstas se presentan.

7.6.11 Leer campos por separado: funciones DatabaseColumn

Como resultado de la ejecución de la consulta mediante las funciones *DatabaseRead* o *DatabaseReadBind*, el programa tiene la oportunidad de desplazarse por los registros seleccionados según las condiciones especificadas. En cada iteración, en las estructuras internas del motor SQLite se asigna un registro específico, cuyos campos (columnas) están disponibles a través del grupo de funciones *DatabaseColumn*.

int DatabaseColumnsCount(int request)

Basándose en el descriptor de la consulta, la función devuelve el número de campos (columnas) en los resultados de la consulta. En caso de error, devuelve -1.

Puede averiguar el número de campos de la consulta creada en *DatabasePrepare* incluso antes de llamar a la función *DatabaseRead*. Para otras funciones de *DatabaseColumn* debe llamar inicialmente a *DatabaseRead* (al menos una vez).

Utilizando el número original de un campo en los resultados de la consulta, el programa puede encontrar el nombre del campo (*DatabaseColumnName*), el tipo (*DatabaseColumnType*), el tamaño (*DatabaseColumnSize*) y el valor del tipo correspondiente (cada tipo tiene su función).

bool DatabaseColumnName(int request, int column, string &name)

La función rellena el parámetro de cadena pasado por referencia (*name*) con el nombre de la columna especificada por número (*column*) en los resultados de la consulta (*request*).

La numeración de los campos comienza en 0 y no puede superar el valor de *DatabaseColumnsCount() - 1*. Esto se aplica no sólo a esta función, sino también a todas las demás funciones de la sección.

La función devuelve *true* si tiene éxito, o *false* en caso de error.

ENUM_DATABASE_FIELD_TYPE DatabaseColumnType(int request, int column)

La función *DatabaseColumnType* devuelve el tipo del valor de la columna especificada en el registro actual de los resultados de la consulta. Los tipos posibles se recopilan en la enumeración ENUM_DATABASE_FIELD_TYPE.

Identificador	Descripción
DATABASE_FIELD_TYPE_INVALID	Error al obtener tipo, código de error en *_LastError*
DATABASE_FIELD_TYPE_INTEGER	Número entero
DATABASE_FIELD_TYPE_FLOAT	Número real
DATABASE_FIELD_TYPE_TEXT	Cadena
DATABASE_FIELD_TYPE_BLOB	Datos binarios
DATABASE_FIELD_TYPE_NULL	Vacío (tipo especial NULL)

Se ofrecieron más detalles sobre los tipos SQL y su correspondencia con los tipos MQL5 en la sección Estructura (esquema) de las tablas: tipos de datos y restricciones.

int DatabaseColumnSize(int request, int column)

La función devuelve el tamaño del valor en bytes para el campo con el índice *column* en el registro actual de resultados de la consulta *request*. Por ejemplo, los valores enteros pueden ser representados por un número diferente de bytes (lo sabemos por los tipos MQL5, en concreto, *short/int/long*).

El siguiente grupo de funciones permite obtener el valor de un tipo concreto a partir del campo correspondiente del registro. Para leer los valores del siguiente registro, hay que volver a llamar a *DatabaseRead*.

bool DatabaseColumnText(int request, int column, string &value)
bool DatabaseColumnInteger(int request, int column, int &value)
bool DatabaseColumnLong(int request, int column, long &value)
bool DatabaseColumnDouble(int request, int column, double &value)
bool DatabaseColumnBlob(int request, int column, void &data[])

Todas las funciones devuelven *true* en caso de éxito y ponen el valor del campo en la variable receptora *value*. El único caso especial es la función *DatabaseColumnBlob*, que pasa un array de un tipo simple arbitrario o estructuras simples como variable de salida. Especificando el array *uchar[]* como la opción más versátil, puede leer la representación en bytes de cualquier valor (incluidos los archivos binarios marcados con el tipo DATABASE_FIELD_TYPE_BLOB).

> El motor SQLite no comprueba que para una columna se llame a una función correspondiente a su tipo. Si los tipos, de forma inadvertida o intencionada, son diferentes, el sistema convertirá automática e implícitamente el valor del campo al tipo de la variable receptora.

Ahora, tras familiarizarnos con la mayoría de las funciones de *Database*, podemos completar el desarrollo de un conjunto de clases SQL en el archivo *DBSQLite.mqh* y pasar a los ejemplos prácticos.

7.6.12 Ejemplos de operaciones CRUD en SQLite mediante objetos ORM

Hemos estudiado todas las funciones necesarias para la implementación del ciclo de vida completo de la información en la base de datos, es decir, CRUD (Create, Read, Update, Delete). Pero antes de pasar a la práctica, necesitamos completar la capa ORM.

De los apartados anteriores ya se deduce que la unidad de trabajo con la base de datos es un registro: puede ser un registro de una tabla de la base de datos o un elemento de los resultados de una consulta.

Para leer un único registro a nivel ORM, introduzcamos la clase DBRow. Cada registro es generado por una consulta SQL, por lo que su manejador se pasa al constructor.

Como sabemos, un registro puede constar de varias columnas, cuyo número y tipos nos permiten encontrar funciones *DatabaseColumn*. Para exponer esta información a un programa MQL utilizando *DBRow*, reservamos las variables pertinentes: *columns* y un array de estructuras *DBRowColumn* (la última contiene tres campos para almacenar el nombre, tipo y tamaño de la columna).

Además, los objetos de *DBRow* pueden, si es necesario, almacenar en caché en sí mismos los valores obtenidos de la base de datos. Para ello se utiliza el array *data* de tipo *MqlParam*. Como no sabemos de antemano qué tipo de valores habrá en una columna concreta, utilizamos *MqlParam* como una especie de tipo universal *Variant* disponible en otros entornos de programación.

```
class DBRow
{
protected:
   const int query;
   int columns;
   DBRowColumn info[];
   MqlParam data[];
   const bool cache;
   int cursor;
   ...
public:
   DBRow(const int q, const bool c = false):
      query(q), cache(c), columns(0), cursor(-1)
   {
   }

   int length() const
   {
      return columns;
   }
   ...
};
```

La variable *cursor* realiza un seguimiento del número de registro actual de los resultados de la consulta. Hasta que se complete la solicitud, *cursor* es igual a -1.

El método virtual *DBread* se encarga de ejecutar la consulta; llama a *DatabaseRead*.

```
protected:
   virtual bool DBread()
   {
      return PRTF(DatabaseRead(query));
   }
```

Más adelante veremos por qué necesitábamos un método virtual. El método público *next*, que utiliza *DBread*, proporciona el «desplazamiento» por los registros de resultados y tiene el siguiente aspecto:

```
public:
   virtual bool next()
   {
      ...
      const bool success = DBread();
      if(success)
      {
         if(cursor == -1)
         {
            columns = DatabaseColumnsCount(query);
            ArrayResize(info, columns);
            if(cache) ArrayResize(data, columns);
            for(int i = 0; i < columns; ++i)
            {
               DatabaseColumnName(query, i, info[i].name);
               info[i].type = DatabaseColumnType(query, i);
               info[i].size = DatabaseColumnSize(query, i);
               if(cache) data[i] = this[i]; // overload operator[](int)
            }
         }
         ++cursor;
      }
      return success;
   }
```

Si se accede a la consulta por primera vez, asignamos memoria y leemos la información de la columna. Si se ha solicitado el almacenamiento en caché, rellenamos adicionalmente el array *data*. Para ello, se llama al operador sobrecargado '[]' para cada columna. En ella, dependiendo del tipo de valor, llamamos a la función *DatabaseColumn* adecuada y ponemos el valor resultante en uno u otro campo de la estructura *MqlParam*.

```
virtual MqlParam operator[](const int i = 0) const
{
   MqlParam param = {};
   if(i < 0 || i >= columns) return param;
   if(ArraySize(data) > 0 && cursor != -1) // if there is a cache, return from it
   {
      return data[i];
   }
   switch(info[i].type)
   {
   case DATABASE_FIELD_TYPE_INTEGER:
      switch(info[i].size)
      {
      case 1:
         param.type = TYPE_CHAR;
         break;
      case 2:
         param.type = TYPE_SHORT;
         break;
      case 4:
         param.type = TYPE_INT;
         break;
      case 8:
      default:
         param.type = TYPE_LONG;
         break;
      }
      DatabaseColumnLong(query, i, param.integer_value);
      break;
   case DATABASE_FIELD_TYPE_FLOAT:
      param.type = info[i].size == 4 ? TYPE_FLOAT : TYPE_DOUBLE;
      DatabaseColumnDouble(query, i, param.double_value);
      break;
   case DATABASE_FIELD_TYPE_TEXT:
      param.type = TYPE_STRING;
      DatabaseColumnText(query, i, param.string_value);
      break;
   case DATABASE_FIELD_TYPE_BLOB: // return base64 only for information we can't
      {                           // return binary data in MqlParam - exact
         uchar blob[];            // representation of binary fields is given by g
         DatabaseColumnBlob(query, i, blob);
         uchar key[], text[];
         if(CryptEncode(CRYPT_BASE64, blob, key, text))
         {
            param.string_value = CharArrayToString(text);
         }
      }
      param.type = TYPE_BLOB;
      break;
   case DATABASE_FIELD_TYPE_NULL:
      param.type = TYPE_NULL;
```

```
         break;
      }
      return param;
   }
```

El método *getBlob* se proporciona para leer completamente datos binarios de campos BLOB (utilice el tipo *uchar* como S para obtener un array de bytes si no hay información más específica sobre el formato del contenido).

```
   template<typename S>
   int getBlob(const int i, S &object[])
   {
      ...
      return DatabaseColumnBlob(query, i, object);
   }
```

Para los métodos descritos, el proceso de ejecución de una consulta y de lectura de sus resultados puede representarse mediante el siguiente pseudocódigo (deja entre bastidores las clases *DBSQLite* y *DBQuery* existentes, pero pronto las uniremos todas):

```
int query = ...
DBRow *row = new DBRow(query);
while(row.next())
{
   for(int i = 0; i < row.length(); ++i)
   {
      StructPrint(row[i]); // print the i-th column as an MqlParam structure
   }
}
```

No es elegante escribir explícitamente un bucle a través de las columnas en cada ocasión, por lo que la clase proporciona un método para obtener los valores de todos los campos del registro.

```
   void readAll(MqlParam &params[]) const
   {
      ArrayResize(params, columns);
      for(int i = 0; i < columns; ++i)
      {
         params[i] = this[i];
      }
   }
```

Además, la clase recibió por conveniencia sobrecargas del operador '[]' y el método *getBlob* para leer campos por sus nombres en lugar de índices. Por ejemplo:

```
class DBRow
{
   ...
public:
   int name2index(const string name) const
   {
      for(int i = 0; i < columns; ++i)
      {
         if(name == info[i].name) return i;
      }
      Print("Wrong column name: ", name);
      SetUserError(3);
      return -1;
   }

   MqlParam operator[](const string name) const
   {
      const int i = name2index(name);
      if(i != -1) return this[i]; // operator()[int] overload
      static MqlParam param = {};
      return param;
   }
   ...
};
```

De esta manera podrá acceder a las columnas seleccionadas.

```
int query = ...
DBRow *row = new DBRow(query);
for(int i = 1; row.next(); )
{
   Print(i++, " ", row["trades"], " ", row["profit"], " ", row["drawdown"]);
}
```

No obstante, obtener los elementos del registro individualmente, como un array *MqlParam*, no puede ser llamado un auténtico enfoque POO. Sería preferible leer el registro de la tabla de la base de datos entera en un objeto, una estructura de aplicación. Recordemos que la API de MQL5 proporciona una función adecuada: *DatabaseReadBind*. Aquí es donde obtenemos la ventaja de la capacidad de describir una clase derivada *DBRow* y anular su método virtual *DBRead*.

Esta clase de *DBRowStruct* es una plantilla y espera como parámetro S una de las estructuras simples que se permiten enlazar en *DatabaseReadBind*.

```
template<typename S>
class DBRowStruct: public DBRow
{
protected:
   S object;

   virtual bool DBread() override
   {
      // NB: inherited structures and nested structures are not allowed;
      // count of structure fields should not exceed count of columns in table/query
      return PRTF(DatabaseReadBind(query, object));
   }

public:
   DBRowStruct(const int q, const bool c = false): DBRow(q, c)
   {
   }

   S get() const
   {
      return object;
   }
};
```

Con una clase derivada, podemos obtener objetos de la base casi sin problemas.

```
int query = ...
DBRowStruct<MyStruct> *row = new DBRowStruct<MyStruct>(query);
MyStruct structs[];
while(row.next())
{
   PUSH(structs, row.get());
}
```

Ahora es el momento de convertir el pseudocódigo en código funcional enlazando *DBRow/DBRowStruct* con *DBQuery*. En *DBQuery*, añadimos un puntero automático al objeto *DBRow*, que contendrá datos sobre el registro actual a partir de los resultados de la consulta (si se ha ejecutado). El uso de un puntero automático libera al código de llamada de preocuparse por liberar los objetos *DBRow*: se eliminan con *DBQuery* o cuando se vuelven a crear debido al reinicio de la consulta (si es necesario). La inicialización del objeto *DBRow* o *DBRowStruct* se completa mediante un método de plantilla *start*.

```
class DBQuery
{
protected:
   ...
   AutoPtr<DBRow> row;    // current entry
public:
   DBQuery(const int owner, const string s): db(owner), sql(s),
      handle(PRTF(DatabasePrepare(db, sql)))
   {
      row = NULL;
   }

   template<typename S>
   DBRow *start()
   {
      DatabaseReset(handle);
      row = typename(S) == "DBValue" ? new DBRow(handle) : new DBRowStruct<S>(handle)
      return row[];
   }
```

El tipo *DBValue* es una estructura ficticia que sólo se necesita para indicar al programa que cree el objeto *DBRow* subyacente, sin violar la compilabilidad de la línea con la llamada a *DatabaseReadBind*.

Con el método *start*, todos los fragmentos de pseudocódigo anteriores pasan a funcionar gracias a la siguiente preparación de la solicitud:

```
DBSQLite db("MQL5Book/DB/Example1");                              // open base
DBQuery *query = db.prepare("PRAGMA table_xinfo('Struct')");      // prepare the reques
DBRowStruct<DBTableColumn> *row = query.start<DBTableColumn>(); // get object cursor
DBTableColumn columns[];                                          // receiving array of
while(row.next())             // loop while there are records in the query result
{
   PUSH(columns, row.get());  // getting an object from the current record
}
ArrayPrint(columns);
```

Este ejemplo lee meta-información sobre la configuración de una tabla concreta de la base de datos (la creamos en el ejemplo *DBcreateTableFromStruct.mq5* en la sección Ejecutar consultas sin enlace de datos MQL5): cada columna se describe mediante un registro independiente con varios campos (estándar SQLite), que se formaliza en la estructura *DBTableColumn*.

```
struct DBTableColumn
{
   int cid;              // identifier (serial number)
   string name;          // name
   string type;          // type
   bool not_null;        // attribute NOT NULL (yes/no)
   string default_value; // default value
   bool primary_key;     // PRIMARY KEY sign (yes/no)
};
```

Para evitar que el usuario tenga que escribir cada vez un bucle con la traducción de los registros de resultados a objetos de estructura, la clase *DBQuery* proporciona un método de plantilla *readAll* que

rellena un array de estructuras referenciada con información de los resultados de la consulta. Un método similar *readAll* rellena un array de punteros a objetos *DBRow* (esto es más adecuado para recibir los resultados de consultas sintéticas con columnas de diferentes tablas).

En un cuarteto de operaciones, el método CRUD *DBRowStruct::get* es el responsable de la letra R (Read). Para que la lectura de un objeto sea funcionalmente más completa, soportaremos la recuperación puntual de un objeto de la base de datos por su identificador.

La gran mayoría de las tablas en bases de datos SQLite tienen una clave primaria *rowid* (a menos que el desarrollador por una razón u otra haya utilizado la opción «WITHOUT ROWID» en la descripción), por lo que el nuevo método *read* tomará un valor de clave como parámetro. Por defecto, se asume que el nombre de la tabla es igual al tipo de la estructura receptora, pero puede cambiarse por otro alternativo a través del parámetro *table*. Teniendo en cuenta que una solicitud de este tipo es una solicitud de una sola vez y debe devolver un registro, tiene sentido colocar el método *read* directamente en la clase *DBSQLite* y gestionar los objetos de vida corta *DBQuery* y *DBRowStruct<S>* en su interior.

```
class DBSQLite
{
   ...
public:
   template<typename S>
   bool read(const long rowid, S &s, const string table = NULL,
      const string column = "rowid")
   {
      const static string query = "SELECT * FROM '%s' WHERE %s=%ld;";
      const string sql = StringFormat(query,
         StringLen(table) ? table : typename(S), column, rowid);
      PRTF(sql);
      DBQuery q(handle, sql);
      if(!q.isValid()) return false;
      DBRowStruct<S> *r = q.start<S>();
      if(r.next())
      {
         s = r.get();
         return true;
      }
      return false;
   }
};
```

El trabajo principal lo realiza la consulta SQL «SELECT * FROM '%s' WHERE %s=%ld;», que devuelve un registro con todos los campos de la tabla especificada al coincidir con la clave *rowid*.

Ahora puede crear un objeto específico de la base de datos de la siguiente manera (se supone que el identificador que nos interesa debe estar almacenado en alguna parte).

```
DBSQLite db("MQL5Book/DB/Example1");
long rowid = ... // ill in the identifier
Struct s;
if(db.read(rowid, s))
   StructPrint(s);
```

Por último, en algunos casos complejos en los que se requiere la máxima flexibilidad en la consulta (por ejemplo, una combinación de varias tablas, normalmente un SELECT con un JOIN, o consultas anidadas), todavía tenemos que permitir un comando SQL explícito para obtener una selección, aunque esto viola el principio ORM. Esta posibilidad se abre con el método *DBSQLite::prepare*, que ya hemos presentado en el contexto de la gestión de consultas preparadas

Hemos considerado todas las formas principales de lectura.

Sin embargo, aún no tenemos nada que leer de la base de datos, porque nos hemos saltado el paso de añadir registros.

Intentemos implementar la creación de objetos (C). Recordemos que, en nuestro concepto de objeto, los tipos de estructura definen semiautomáticamente tablas de base de datos (mediante macros DB_FIELD). Por ejemplo, la estructura *Struct* permitía crear una tabla «Struct» en la base de datos con un conjunto de columnas correspondientes a los campos de la estructura. Para ello, hemos incluido un método de plantilla *createTable* en la clase *DBSQLite*. Ahora, por analogía, necesita escribir un método de plantilla *insert*, que añadiría un registro a esta tabla.

Se pasa al método un objeto de una estructura, para cuyo tipo debe existir el array *DBEntity<S>::prototype* <S> rellenado (se rellena con macros). Gracias a este array, podemos formar una lista de parámetros (más concretamente, sus sustitutos '?n'): esto se hace mediante el método estático *qlist*. Sin embargo, la preparación de la consulta sigue siendo media batalla. En el código siguiente, tendremos que vincular los datos de entrada en función de las propiedades del objeto.

Se ha añadido una sentencia «RETURNING rowid» al comando «INSERT», de modo que cuando la consulta tiene éxito, esperamos una única fila de resultados con un valor: *rowid* nuevo.

```
class DBSQLite
{
   ...
public:
   template<typename S>
   long insert(S &object, const string table = NULL)
   {
      const static string query = "INSERT INTO '%s' VALUES(%s) RETURNING rowid;";
      const int n = ArrayRange(DBEntity<S>::prototype, 0);
      const string sql = StringFormat(query,
         StringLen(table) ? table : typename(S), qlist(n));
      PRTF(sql);
      DBQuery q(handle, sql);
      if(!q.isValid()) return 0;
      DBRow *r = q.start<DBValue>();
      if(object.bindAll(q))
      {
         if(r.next()) // the result should be one record with one new rowid value
         {
            return object.rowid(r[0].integer_value);
         }
      }
      return 0;
   }

   static string qlist(const int n)
   {
      string result = "?1";
      for(int i = 1; i < n; ++i)
      {
         result += StringFormat(",?%d", (i + 1));
      }
      return result;
   }
};
```

El código fuente del método *insert* tiene un punto al que hay que prestar especial atención. Para vincular valores a parámetros de consulta, llamamos al método *object.bindAll(q)*. Esto significa que en la estructura de la aplicación que desea integrar con la base, es necesario implementar un método de este tipo que proporciona todas las variables de miembro para el motor.

Además, para identificar los objetos, se supone que existe un campo con una clave primaria, y sólo el objeto «sabe» cuál es este campo. Así, la estructura dispone del método *rowid*, que cumple una doble acción: en primer lugar, transfiere al objeto el identificador de registro asignado en la base de datos y, en segundo lugar, permite averiguar este identificador a partir del objeto, si ya ha sido asignado anteriormente.

El método *DBSQLite::update* (U) para modificar un registro es similar en muchos aspectos a *insert*, por lo que se propone familiarizarse con él. Su base es la consulta SQL «UPDATE '%s' SET (%s)=(%s) WHERE rowid=%ld;», que se supone que pasa todos los campos de la estructura (objeto*bindAll()*) y la clave (objeto*rowid()*).

Por último, mencionamos que la eliminación puntual (D) de un registro por un objeto se implementa en el método *DBSQLite::remove* (la palabra *delete* es un operador MQL5).

Vamos a mostrar todos los métodos en un script de ejemplo *DBfillTableFromStructArray.mq5*, donde se define la nueva estructura *Struct*.

Haremos varios valores de tipos de uso común como campos de la estructura.

```
struct Struct
{
   long id;
   string name;
   double number;
   datetime timestamp;
   string image;
   ...
};
```

En el campo de cadena *image*, el código de llamada especificará el nombre del recurso gráfico o el nombre del archivo, y en el momento de la vinculación a la base de datos, los datos binarios correspondientes se copiarán como BLOB. Posteriormente, cuando leamos datos de la base de datos en objetos *Struct*, los datos binarios acabarán en la cadena *image* pero, por supuesto, con distorsiones (porque la línea se romperá en el primer byte nulo). Para extraer con precisión los BLOB de la base de datos, deberá llamar al método *DBRow::getBlob* (basado en *DatabaseColumnBlob*).

La creación de metainformación sobre los campos de la estructura *Struct* proporciona las siguientes macros. Basándose en ellos, un programa MQL puede crear automáticamente una tabla en la base de datos para los objetos *Struct*, así como iniciar la vinculación de los datos pasados a las consultas basándose en las propiedades de los objetos (esta vinculación no debe confundirse con la vinculación inversa para obtener los resultados de las consultas, es decir, *DatabaseReadBind*).

```
DB_FIELD_C1(Struct, long, id, DB_CONSTRAINT::PRIMARY_KEY);
DB_FIELD(Struct, string, name);
DB_FIELD(Struct, double, number);
DB_FIELD_C1(Struct, datetime, timestamp, DB_CONSTRAINT::CURRENT_TIMESTAMP);
DB_FIELD(Struct, blob, image);
```

Para rellenar un pequeño array de prueba, el script dispone de variables de entrada: especifican un trío de divisas cuyas cotizaciones entrarán en el campo *number*. También hemos incrustado dos imágenes estándar en el script para probar el trabajo con BLOBs: «irán» al campo *image*. El campo *timestamp* será rellenado automáticamente por nuestras clases ORM con la marca de tiempo de modificación o inserción actual del registro. La clave primaria del campo *id* tendrá que ser rellenada por el propio SQLite.

```
#resource "\\Images\\euro.bmp"
#resource "\\Images\\dollar.bmp"

input string Database = "MQL5Book/DB/Example2";
input string EURUSD = "EURUSD";
input string USDCNH = "USDCNH";
input string USDJPY = "USDJPY";
```

Dado que los valores de las variables de consulta de entrada (esas mismas '?n') se ligan, en última instancia, mediante las funciones *DatabaseBind* o *DatabaseBindArray* bajo los números, nuestra

estructura *bindAll* en el método debe establecer una correspondencia entre los números y sus campos: se supone una numeración simple en el orden de declaración.

```
struct Struct
{
   ...
   bool bindAll(DBQuery &q) const
   {
      uint pixels[] = {};
      uint w, h;
      if(StringLen(image))                // load binary data
      {
         if(StringFind(image, "::") == 0) // this is a resource
         {
            ResourceReadImage(image, pixels, w, h);
            // debug/test example (not BMP, no header)
            FileSave(StringSubstr(image, 2) + ".raw", pixels);
         }
         else                             // it's a file
         {
            const string res = "::" + image;
            ResourceCreate(res, image);
            ResourceReadImage(res, pixels, w, h);
            ResourceFree(res);
         }
      }
      // when id = NULL, the base will assign a new rowid
      return (id == 0 ? q.bindNull(0) : q.bind(0, id))
         && q.bind(1, name)
         && q.bind(2, number)
         // && q.bind(3, timestamp) // this field will be autofilled CURRENT_TIMESTAM
         && q.bindBlob(4, pixels);
   }
   ...
};
```

El método *rowid* es muy sencillo.

```
struct Struct
{
   ...
   long rowid(const long setter = 0)
   {
      if(setter) id = setter;
      return id;
   }
};
```

Una vez definida la estructura, describimos un array de prueba de 4 elementos. Sólo 2 de ellos tienen imágenes adjuntas. Todos los objetos tienen cero identificadores porque aún no están en la base de datos.

```
Struct demo[] =
{
   {0, "dollar", 1.0, 0, "::Images\\dollar.bmp"},
   {0, "euro", SymbolInfoDouble(EURUSD, SYMBOL_ASK), 0, "::Images\\euro.bmp"},
   {0, "yuan", 1.0 / SymbolInfoDouble(USDCNH, SYMBOL_BID), 0, NULL},
   {0, "yen", 1.0 / SymbolInfoDouble(USDJPY, SYMBOL_BID), 0, NULL},
};
```

En la función principal *OnStart*, creamos o abrimos una base de datos (por defecto *MQL5Book/DB/Example2.sqlite*). Por si acaso, intentamos eliminar la tabla «Struct» para garantizar la reproducibilidad de los resultados y la depuración cuando se repita el script, entonces crearemos una tabla para la estructura *Struct*.

```
void OnStart()
{
   DBSQLite db(Database);
   if(!PRTF(db.isOpen())) return;
   PRTF(db.deleteTable(typename(Struct)));
   if(!PRTF(db.createTable<Struct>(true))) return;
   ...
```

En lugar de añadir objetos de uno en uno, utilizamos un bucle:

```
  // -> this option (set aside)
   for(int i = 0; i < ArraySize(demo); ++i)
   {
      PRTF(db.insert(demo[i])); // get a new rowid on each call
   }
```

En este bucle utilizaremos una implementación alternativa del método *insert*, que toma como entrada un array de objetos a la vez y los procesa en una única solicitud, lo cual es más eficiente (pero el foso general del método es el método *insert* considerado anteriormente para un objeto).

```
   db.insert(demo);  // new rowids are placed in objects
   ArrayPrint(demo);
   ...
```

Ahora vamos a intentar seleccionar registros de la base de datos según algunas condiciones; por ejemplo, aquellos que no tienen una imagen asignada. Para ello, vamos a preparar una consulta SQL envuelta en el objeto *DBQuery*, y luego obtendremos sus resultados de dos maneras: a través de la vinculación a estructuras *Struct* o a través de las instancias de la clase genérica *DBRow*.

```
DBQuery *query = db.prepare(StringFormat("SELECT * FROM %s WHERE image IS NULL",
   typename(Struct)));

// approach 1: application type of the Struct structure
Struct result[];
PRTF(query.readAll(result));
ArrayPrint(result);

query.reset(); // reset the query to try again

// approach 2: generic DBRow record container with MqlParam values
DBRow *rows[];
query.readAll(rows); // get DBRow objects with cached values
for(int i = 0; i < ArraySize(rows); ++i)
{
   Print(i);
   MqlParam fields[];
   rows[i].readAll(fields);
   ArrayPrint(fields);
}
...
```

Ambas opciones deberían dar el mismo resultado, aunque presentado de forma diferente (véase el registro más abajo).

A continuación, nuestro script hace una pausa de 1 segundo para que podamos notar los cambios en las marcas de tiempo de las próximas entradas que cambiaremos.

```
Print("Pause...");
Sleep(1000);
...
```

A los objetos del array *result[]* les asignamos la imagen «yuan.bmp» que se encuentra en la carpeta junto al script. A continuación, actualizamos los objetos en la base de datos.

```
for(int i = 0; i < ArraySize(result); ++i)
{
   result[i].image = "yuan.bmp";
   db.update(result[i]);
}
...
```

Después de ejecutar el script, puede asegurarse de que los cuatro registros tienen BLOBs en el navegador de base de datos incorporado en MetaEditor, así como la diferencia en las marcas de tiempo de los dos primeros y los dos últimos registros.

Vamos a demostrar la extracción de datos binarios. Primero veremos cómo se asigna un BLOB al campo de cadena *image* (los datos binarios no son para el registro, sólo lo hacemos a efectos de demostración).

```
const long id1 = 1;
Struct s;
if(db.read(id1, s))
{
   Print("Length of string with Blob: ", StringLen(s.image));
   Print(s.image);
}
...
```

A continuación, leemos los datos completos con *getBlob* (la longitud total es mayor que la de la línea anterior).

```
DBRow *r;
if(db.read(id1, r, "Struct"))
{
   uchar bytes[];
   Print("Actual size of Blob: ", r.getBlob("image", bytes));
   FileSave("temp.bmp.raw", bytes); // not BMP, no header
}
```

Necesitamos obtener el archivo *temp.bmp.raw*, idéntico a *MQL5/Files/Images/dollar.bmp.raw*, que se crea en el método *Struct::bindAll* con fines de depuración. Así, es fácil verificar la correspondencia exacta de los datos binarios escritos y leídos.

> Tenga en cuenta que, dado que almacenamos el contenido binario del recurso en la base de datos, no se trata de un archivo fuente BMP: los recursos producen la normalización del color y almacenar un array de píxeles sin encabezado con metainformación sobre la imagen.

Mientras se ejecuta, el script genera un registro detallado. En concreto, la creación de una base de datos y una tabla se marca con las siguientes líneas.

```
db.isOpen()=true / ok
db.deleteTable(typename(Struct))=true / ok
sql=CREATE TABLE IF NOT EXISTS Struct (id INTEGER PRIMARY KEY,
name TEXT ,
number REAL ,
timestamp INTEGER CURRENT_TIMESTAMP,
image BLOB ); / ok
db.createTable<Struct>(true)=true / ok
```

La consulta SQL para insertar un array de objetos se prepara una vez y luego se ejecuta varias veces con previnculación de datos diferentes (aquí sólo se muestra una iteración). El número de llamadas a la función *DatabaseBind* coincide con las variables '?n' de la consulta ('?4' es sustituido automáticamente por nuestras clases con la llamada a la función SQL STRFTIME('%s') para obtener la marca de tiempo UTC actual).

```
sql=INSERT INTO 'Struct' VALUES(?1,?2,?3,STRFTIME('%s'),?5) RETURNING rowid; / ok
DatabasePrepare(db,sql)=131073 / ok
DatabaseBindArray(handle,index,value)=true / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBindArray(handle,index,value)=true / ok
DatabaseRead(query)=true / ok
...
```

A continuación, un array de estructuras con claves primarias ya asignadas *rowid* se envía al registro en la primera columna.

```
    [id]   [name] [number]         [timestamp]               [image]
[0]    1 "dollar"  1.00000 1970.01.01 00:00:00 "::Images\dollar.bmp"
[1]    2 "euro"    1.00402 1970.01.01 00:00:00 "::Images\euro.bmp"
[2]    3 "yuan"    0.14635 1970.01.01 00:00:00 null
[3]    4 "yen"     0.00731 1970.01.01 00:00:00 null
```

La selección de registros sin imágenes da el siguiente resultado (ejecutamos esta consulta dos veces con métodos diferentes: la primera vez rellenamos el array de estructuras *Struct*, y la segunda es el array *DBRow*, del que para cada campo obtenemos el «valor» en forma de *MqlParam*).

```
DatabasePrepare(db,sql)=196609 / ok
DatabaseReadBind(query,object)=true / ok
DatabaseReadBind(query,object)=true / ok
DatabaseReadBind(query,object)=false / DATABASE_NO_MORE_DATA(5126)
query.readAll(result)=true / ok
    [id] [name] [number]         [timestamp] [image]
[0]    3 "yuan"  0.14635 2022.08.20 13:14:38 null
[1]    4 "yen"   0.00731 2022.08.20 13:14:38 null
DatabaseRead(query)=true / ok
DatabaseRead(query)=true / ok
DatabaseRead(query)=false / DATABASE_NO_MORE_DATA(5126)
0
    [type] [integer_value] [double_value] [string_value]
[0]      4               3        0.00000 null
[1]     14               0        0.00000 "yuan"
[2]     13               0        0.14635 null
[3]     10      1661001278        0.00000 null
[4]      0               0        0.00000 null
1
    [type] [integer_value] [double_value] [string_value]
[0]      4               4        0.00000 null
[1]     14               0        0.00000 "yen"
[2]     13               0        0.00731 null
[3]     10      1661001278        0.00000 null
[4]      0               0        0.00000 null
...
```

La segunda parte del script actualiza un par de registros encontrados sin imágenes y les añade BLOBs.

```
Pause...
sql=UPDATE 'Struct' SET (id,name,number,timestamp,image)=
   (?1,?2,?3,STRFTIME('%s'),?5) WHERE rowid=3; / ok
DatabasePrepare(db,sql)=262145 / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBindArray(handle,index,value)=true / ok
DatabaseRead(handle)=false / DATABASE_NO_MORE_DATA(5126)
sql=UPDATE 'Struct' SET (id,name,number,timestamp,image)=
   (?1,?2,?3,STRFTIME('%s'),?5) WHERE rowid=4; / ok
DatabasePrepare(db,sql)=327681 / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBind(handle,index,value)=true / ok
DatabaseBindArray(handle,index,value)=true / ok
DatabaseRead(handle)=false / DATABASE_NO_MORE_DATA(5126)
...
```

Por último, al obtener datos binarios de dos formas - incompatible, a través del campo de cadena *image* como resultado de la lectura de todo el objeto *DatabaseReadBind* (esto sólo se hace para visualizar la secuencia de bytes en el registro), y compatible, a través de *DatabaseRead* y *DatabaseColumnBlob* - obtenemos resultados diferentes: por supuesto, el segundo método es correcto, pues se restablecen la longitud y el contenido del BLOB en 4096 bytes.

```
sql=SELECT * FROM 'Struct' WHERE rowid=1; / ok
DatabasePrepare(db,sql)=393217 / ok
DatabaseReadBind(query,object)=true / ok
Length of string with Blob: 922
□7?□7?□7?□7?□7?□7?ł7?ő6?ű6?Ĭ5???5?¦5?Ĭ5?ļ6?ő6?Ő7?ł7?ł7?□7?□7?□7?□7?□7?□7?□7?□7?□7?
sql=SELECT * FROM 'Struct' WHERE rowid=1; / ok
DatabasePrepare(db,sql)=458753 / ok
DatabaseRead(query)=true / ok
Actual size of Blob: 4096
```

Resumiendo el resultado intermedio del desarrollo de nuestro propio envoltorio ORM, presentamos un esquema generalizado de sus clases.

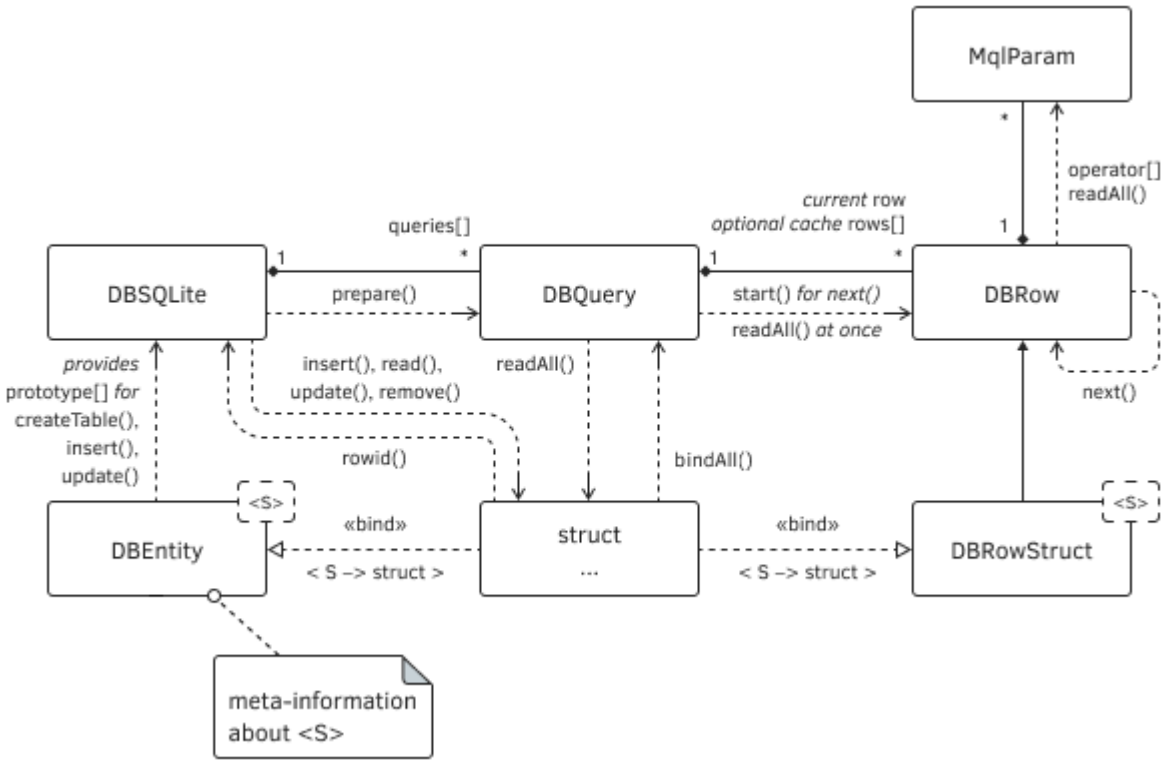


Diagrama de Clases ORM (MQL5<->SQL)

7.6.13 Transacciones

SQLite admite *transactions*: conjuntos de acciones relacionadas lógicamente que pueden realizarse por completo o no realizarse en absoluto, lo que garantiza la coherencia de los datos en la base de datos.

> El concepto de *transaction* tiene un nuevo significado en el contexto de las bases de datos, diferente del que solíamos describir en transacciones de trading Una operación de trading es una operación independiente sobre las entidades de una cuenta de trading, incluidas las órdenes, las transacciones y las posiciones.

Las transacciones proporcionan 4 características principales de los cambios en bases de datos:

- Atómico (indivisible) - al finalizar con éxito la transacción, todos los cambios incluidos en ella llegarán a la base de datos, y en caso de error, no llegará nada.
- Consistente - el estado actual correcto de la base sólo puede cambiar a otro estado correcto (los estados intermedios, según la lógica de la aplicación, están excluidos).
- Aislado - los cambios en la transacción de la conexión actual no son visibles hasta el final de esta transacción en otras conexiones a la misma base de datos y viceversa, los cambios de otras conexiones no son visibles en la conexión actual mientras haya una transacción incompleta.
- Duradero - se garantiza que los cambios de una transacción correcta se almacenan en la base de datos.

Los términos para estas características - Atómico, Consistente, Aislado y Duradero - forman el acrónimo ACID (por sus siglas en inglés), bien conocido en la teoría de bases de datos.

Aunque el curso normal del programa se interrumpa debido a un fallo del sistema, la base de datos conservará su estado de funcionamiento.

La mayoría de las veces, el uso de las transacciones se ilustra con el ejemplo de un sistema bancario, en el que se transfieren fondos de la cuenta de un cliente a la cuenta de otro. Debe afectar a dos registros con saldos de clientes: en uno, el saldo se reduce por el importe de la transferencia, y en el otro, se incrementa. Una situación en la que sólo se aplicara uno de estos cambios alteraría el saldo de las cuentas bancarias: dependiendo de qué operación falle, el importe transferido podría desaparecer o, por el contrario, surgir de la nada.

Es posible ofrecer un ejemplo más cercano a la práctica de trading, pero basado en el principio «opuesto». El hecho es que el sistema de contabilización de órdenes, transacciones y posiciones en MetaTrader 5 no es transaccional.

En particular, como sabemos por el capítulo Crear Asesores Expertos, una orden activada (de mercado o pendiente), que falte en la lista de activas, puede no aparecer inmediatamente en la lista de posiciones. Por lo tanto, para analizar el resultado real, es necesario implementar en el programa MQL la expectativa de actualización del entorno de trading. Si el sistema contable se basara en transacciones, la ejecución de una orden, el registro de una transacción en el historial y la aparición de una posición se englobarían en una transacción y se coordinarían entre sí. Los desarrolladores del terminal han optado por un enfoque diferente: devolver cualquier modificación del entorno de trading lo más rápida y asíncronamente posible, y su integridad debe ser supervisada por un programa MQL.

Cualquier comando SQL que cambie la base (es decir, de hecho, todo excepto SELECT) se envolverá automáticamente en una transacción si esto no se hizo explícitamente de antemano.

La API de MQL5 proporciona 3 funciones para gestionar las transacciones: *DatabaseTransactionBegin*, *DatabaseTransactionCommit* y *DatabaseTransactionRollback*. Todas las funciones devuelven *true* en caso de éxito, o *false* en caso de error.

bool DatabaseTransactionBegin(int database)

La función *DatabaseTransactionBegin* inicia la ejecución de una transacción en la base de datos con el descriptor especificado obtenido de *DatabaseOpen*.

Todos los cambios posteriores realizados en la base de datos se acumulan en la caché interna de transacciones y no llegan a la base de datos hasta que se llama a la función *DatabaseTransactionCommit*.

Las transacciones en MQL5 no se pueden anidar: si una transacción ya se ha iniciado, entonces volver a llamar a *DatabaseTransactionBegin* devolverá una bandera de error y la salida de un mensaje en el registro.

```
database error, cannot start a transaction within a transaction
DatabaseTransactionBegin(db)=false / DATABASE_ERROR(5601)
```

Respectivamente, no puede intentar completar la transacción varias veces.

bool DatabaseTransactionCommit(int database)

La función *DatabaseTransactionCommit* finaliza una transacción iniciada previamente en la base de datos con el manejador especificado y aplica todos los cambios acumulados (los guarda). Si un programa MQL inicia una transacción pero no la aplica antes de cerrar la base de datos, se perderán todos los cambios.

Si es necesario, el programa puede deshacer la transacción y, por tanto, todos los cambios desde el inicio de la transacción.

bool DatabaseTransactionRollback(int database)

La función *DatabaseTransactionRollback* realiza un «retroceso» de todas las acciones incluidas en la transacción iniciada previamente para la base de datos con el manejador *database*.

Vamos a completar los métodos de la clase *DBSQLite* para trabajar con transacciones, teniendo en cuenta la restricción en su anidamiento, que calcularemos en la variable *transaction*. Si es 0, el método *begin* inicia una transacción llamando a *DatabaseTransactionBegin*. Todos los intentos posteriores de iniciar una transacción simplemente aumentan el contador. En el método *commit*, reducimos el contador, y cuando llega a 0 llamamos a *DatabaseTransactionCommit*.

```
class DBSQLite
{
protected:
   int transaction;
   ...
public:
   bool begin()
   {
      if(transaction > 0)    // already in transaction
      {
         transaction++;      // keep track of the nesting level
         return true;
      }
      return (bool)(transaction = PRTF(DatabaseTransactionBegin(handle)));
   }

   bool commit()
   {
      if(transaction > 0)
      {
         if(--transaction == 0) // outermost transaction
            return PRTF(DatabaseTransactionCommit(handle));
      }
      return false;
   }
   bool rollback()
   {
      if(transaction > 0)
      {
         if(--transaction == 0)
            return PRTF(DatabaseTransactionRollback(handle));
      }
      return false;
   }
};
```

Además, vamos a crear la clase *DBTransaction*, que permitirá describir objetos dentro de bloques (por ejemplo, funciones) que aseguren el inicio automático de una transacción con su posterior aplicación (o cancelación) cuando el programa salga del bloque.

```
class DBTransaction
{
   DBSQLite *db;
   const bool autocommit;
public:
   DBTransaction(DBSQLite &owner, const bool c = false): db(&owner), autocommit(c)
   {
      if(CheckPointer(db) != POINTER_INVALID)
      {
         db.begin();
      }
   }

   ~DBTransaction()
   {
      if(CheckPointer(db) != POINTER_INVALID)
      {
         autocommit ? db.commit() : db.rollback();
      }
   }

   bool commit()
   {
      if(CheckPointer(db) != POINTER_INVALID)
      {
         const bool done = db.commit();
         db = NULL;
         return done;
      }
      return false;
   }
};
```

La política de utilizar este tipo de objetos elimina la necesidad de procesar varias opciones para salir de un bloque (función).

```
void DataFunction(DBSQLite &db)
{
   DBTransaction tr(db);
   DBQuery *query = db.prepare("UPDATE..."); // batch changes
   ... // base modification
   if(... /* error1 */) return;             // automatic rollback
   ... // base modification
   if(... /* error2 */) return;             // automatic rollback
   tr.commit();
}
```

Para que un objeto aplique automáticamente los cambios en cualquier fase, pase *true* en el segundo parámetro de su constructor.

```
void DataFunction(DBSQLite &db)
{
   DBTransaction tr(db, true);
   DBQuery *query = db.prepare("UPDATE...");  // batch changes
   ... // base modification
   if(... /* condition1 */) return;           // automatic commit
   ... // base modification
   if(... /* condition2 */) return;           // automatic commit
   ...
}                                             // automatic commit
```

Puede describir el objeto *DBTransaction* dentro del bucle y entonces, en cada iteración, se iniciará y cerrará una transacción independiente.

Se hará una demostración de las transacciones en la sección Ejemplo de búsqueda de una estrategia de trading mediante SQLite.

7.6.14 Importar y exportar tablas de bases de datos

MQL5 permite exportar e importar tablas individuales de bases de datos a/desde archivos CSV. No se proporciona la exportación/importación de la base de datos completa, como un archivo con comandos SQL.

long DatabaseImport(int database, const string table, const string filename, uint flags,
 const string separator, ulong skip_rows, const string comment_chars)

La función *DatabaseImport* importa datos del archivo especificado a la tabla. Los dos primeros parámetros indican el descriptor de la base de datos abierta y el nombre de la tabla.

Si la tabla *table* no existe, se creará automáticamente. Los nombres y tipos de campos de la tabla se reconocerán automáticamente en función de los datos contenidos en el archivo.

El archivo importado puede ser no sólo un archivo CSV ya preparado, sino también un archivo ZIP con un archivo CSV. El nombre del archivo puede contener una ruta. El archivo se busca en relación con el directorio *MQL5/Files*.

Los indicadores válidos que pueden combinarse por bits se describen en la enumeración ENUM_DATABASE_IMPORT_FLAGS:

- DATABASE_IMPORT_HEADER - la primera línea contiene los nombres de los campos de la tabla.
- DATABASE_IMPORT_CRLF - para los saltos de línea, se utiliza la secuencia de caracteres CRLF.
- DATABASE_IMPORT_APPEND - añadir datos a una tabla existente
- DATABASE_IMPORT_QUOTED_STRINGS - valores de cadena entre comillas dobles
- DATABASE_IMPORT_COMMON_FOLDER - carpeta común de terminales

El parámetro *separator* establece el carácter delimitador en el archivo CSV.

El parámetro *skip_rows* omite el número especificado de líneas iniciales del archivo.

El parámetro *comment_chars* contiene los caracteres utilizados en el archivo como bandera de comentario. Las líneas que empiecen por cualquiera de estos caracteres se considerarán comentarios y no se importarán.

La función devuelve el número de filas importadas, o -1 en caso de error.

long DatabaseExport(int database, const string table_or_sql, const string filename, uint flags, const string separator)

La función *DatabaseExport* exporta una tabla o el resultado de una consulta SQL a un archivo CSV. El manejador de la base de datos, así como el nombre de la tabla o el texto de la consulta, se especifican en los dos primeros parámetros.

Si se exportan los resultados de la consulta, la consulta SQL debe empezar por «SELECT» o «select». En otras palabras: una consulta SQL no puede cambiar el estado de la base de datos; de lo contrario, *DatabaseExport* finalizará con un error.

El nombre *filename* del archivo puede contener una ruta dentro del directorio *MQL5/Files* de la instancia actual del terminal o la carpeta compartida de terminales, dependiendo de las banderas.

El parámetro *flags* permite especificar una combinación de banderas que controla el formato y la ubicación del archivo.

- DATABASE_EXPORT_HEADER - salida de una cadena con nombres de campo
- DATABASE_EXPORT_INDEX - mostrar números de línea
- DATABASE_EXPORT_NO_BOM - no insertar etiqueta BOM al principio del archivo (BOM se inserta por defecto)
- DATABASE_EXPORT_CRLF - usar CRLF para romper una línea (LF por defecto)
- DATABASE_EXPORT_APPEND - añadir datos al final de un archivo existente (por defecto, el archivo se sobrescribe), si el archivo no existe, se creará.
- DATABASE_EXPORT_QUOTED_STRINGS - salida de valores de cadena entre comillas dobles
- DATABASE_EXPORT_COMMON_FOLDER - el archivo CSV se creará en la carpeta común de todos los terminales. *MetaQuotes/Terminal/Common/File*

El parámetro *separator* especifica el carácter separador de columnas. Si es NULL, se utilizará el carácter de tabulación '\t' como separador. La cadena vacía «» se considera un delimitador válido, pero el archivo CSV resultante no podrá leerse como una tabla y será un conjunto de filas.

Los campos de texto de la base de datos pueden contener líneas nuevas ('\r' o '\r\n'), así como el carácter delimitador especificado en el parámetro separador. En este caso, es necesario utilizar el indicador DATABASE_EXPORT_QUOTED_STRINGS en el parámetro *flags*. Si esta bandera está presente, todas las cadenas de salida se encerrarán entre comillas dobles, y si la cadena contiene comillas dobles, se sustituirá por dos comillas dobles.

La función devuelve el número de registros exportados o un valor negativo en caso de error.

7.6.15 Imprimir tablas y consultas SQL en registros

Si es necesario, un programa MQL puede mostrar el contenido de una tabla o los resultados de una consulta SQL en un registro mediante la función *DatabasePrint*.

long DatabasePrint(int database, const string table_or_sql, uint flags)

El manejador de la base de datos se pasa en el primer parámetro, seguido del nombre de la tabla o del texto de la consulta (*table_or_sql*). La consulta SQL debe empezar por «SELECT» o «select», es decir,

no debe cambiar el estado de la base de datos. De lo contrario, la función *DatabasePrint* finalizará con un error.

El parámetro *flags* especifica una combinación de banderas que determinan el formato de la salida.

- DATABASE_PRINT_NO_HEADER - no mostrar los nombres de columna de la tabla (nombres de campo)
- DATABASE_PRINT_NO_INDEX - no mostrar números de línea
- DATABASE_PRINT_NO_FRAME - no mostrar un marco que separe el encabezado de los datos.
- DATABASE_PRINT_STRINGS_RIGHT - alinear cadenas a la derecha

Si *flags* = *0*, entonces se muestran las columnas y las filas, el encabezado y los datos están separados por un marco, y las filas están alineadas a la izquierda.

La función devuelve el número de registros visualizados, o -1 en caso de error.

Utilizaremos la función en la siguiente sección.

Lamentablemente, la función no permite una salida de consultas preparadas con parámetros. Si hay parámetros, tendrán que incrustarse en el texto de la consulta en el nivel MQL5.

7.6.16 Ejemplo de búsqueda de una estrategia de trading mediante SQLite

Intentemos utilizar SQLite para resolver problemas prácticos. Importaremos estructuras a la base de datos *MqlRates* con el historial de cotizaciones y las analizaremos para identificar patrones y buscar posibles estrategias de trading. Por supuesto, cualquier lógica elegida también se puede implementar en MQL5, pero SQL le permite hacerlo de una manera diferente, en muchos casos de manera más eficiente y utilizando muchas funciones SQL integradas interesantes. El tema del libro, orientado al aprendizaje de MQL5, no permite profundizar en esta tecnología, pero la mencionamos como digna de la atención de un operador de trading algorítmico.

El script para convertir el historial de cotizaciones en un formato de base de datos se llama *DBquotesImport.mq5*. En los parámetros de entrada, puede establecer el prefijo del nombre de la base de datos y el tamaño de la transacción (el número de registros de una transacción).

```
input string Database = "MQL5Book/DB/Quotes";
input int TransactionSize = 1000;
```

Para añadir estructuras *MqlRates* a la base de datos utilizando nuestra capa ORM, el script define una estructura *MqlRatesDB* auxiliar que proporciona las reglas para vincular campos de estructura a columnas base. Dado que nuestro script sólo escribe datos en la base de datos y no los lee desde allí, no es necesario vincularlo mediante la función *DatabaseReadBind*, que impondría una restricción a la «simplicidad» de la estructura. La ausencia de una restricción permite derivar la estructura *MqlRatesDB* de *MqlRates* (y no repetir la descripción de los campos).

```
struct MqlRatesDB: public MqlRates
{
   /* for reference:

      datetime time;
      double   open;
      double   high;
      double   low;
      double   close;
      long     tick_volume;
      int      spread;
      long     real_volume;
   */

   bool bindAll(DBQuery &q) const
   {
      return q.bind(0, time)
         && q.bind(1, open)
         && q.bind(2, high)
         && q.bind(3, low)
         && q.bind(4, close)
         && q.bind(5, tick_volume)
         && q.bind(6, spread)
         && q.bind(7, real_volume);
   }

   long rowid(const long setter = 0)
   {
 // rowid is set by us according to the bar time
      return time;
   }
};

DB_FIELD_C1(MqlRatesDB, datetime, time, DB_CONSTRAINT::PRIMARY_KEY);
DB_FIELD(MqlRatesDB, double, open);
DB_FIELD(MqlRatesDB, double, high);
DB_FIELD(MqlRatesDB, double, low);
DB_FIELD(MqlRatesDB, double, close);
DB_FIELD(MqlRatesDB, long, tick_volume);
DB_FIELD(MqlRatesDB, int, spread);
DB_FIELD(MqlRatesDB, long, real_volume);
```

El nombre de la base de datos se forma a partir del prefijo *Database*, el nombre y el marco temporal del gráfico actual en el que se está ejecutando el script. Se crea una única tabla «MqlRatesDB» en la base de datos con la configuración de campos especificada por las macros DB_FIELD. Tenga en cuenta que la clave primaria no será generada por la base de datos, sino que se toma directamente de las barras, del campo *time* (hora de apertura de la barra).

```
void OnStart()
{
   Print("");
   DBSQLite db(Database + _Symbol + PeriodToString());
   if(!PRTF(db.isOpen())) return;

   PRTF(db.deleteTable(typename(MqlRatesDB)));

   if(!PRTF(db.createTable<MqlRatesDB>(true))) return;
   ...
```

A continuación, utilizando paquetes de barras *TransactionSize*, solicitamos barras del historial y las añadimos a la tabla. Esta es una tarea de la función auxiliar *ReadChunk*, llamada en bucle mientras haya datos (la función devuelve *true*) o el usuario no detenga el script manualmente. El código de la función se muestra a continuación:

```
   int offset = 0;
   while(ReadChunk(db, offset, TransactionSize) && !IsStopped())
   {
      offset += TransactionSize;
   }
```

Una vez finalizado el proceso, pedimos a la base de datos el número de registros generados en la tabla y lo enviamos al registro.

```
   DBRow *rows[];
   if(db.prepare(StringFormat("SELECT COUNT(*) FROM %s",
      typename(MqlRatesDB))).readAll(rows))
   {
      Print("Records added: ", rows[0][0].integer_value);
   }
}
```

La función *ReadChunk* tiene el siguiente aspecto:

```
bool ReadChunk(DBSQLite &db, const int offset, const int size)
{
   MqlRates rates[];
   MqlRatesDB ratesDB[];
   const int n = CopyRates(_Symbol, PERIOD_CURRENT, offset, size, rates);
   if(n > 0)
   {
      DBTransaction tr(db, true);
      Print(rates[0].time);
      ArrayResize(ratesDB, n);
      for(int i = 0; i < n; ++i)
      {
         ratesDB[i] = rates[i];
      }

      return db.insert(ratesDB);
   }
   else
   {
      Print("CopyRates failed: ", _LastError, " ", E2S(_LastError));
   }
   return false;
}
```

Llama a la función integrada *CopyRates* a través de la cual se rellena el array de barras *rates*. A continuación, las barras se transfieren al array *ratesDB*, de modo que con una sola sentencia *db.insert(ratesDB)* podríamos escribir información en la base de datos (en *MqlRatesDB* hemos formalizado cómo hacerlo correctamente).

La presencia del objeto *DBTransaction* (con la opción «commit» automática activada) dentro del bloque significa que todas las operaciones con el array se «superponen» a una transacción. Para indicar el progreso, durante el procesamiento de cada bloque de barras, la etiqueta de la primera barra se muestra en el registro.

Mientras la función *CopyRates* devuelve los datos y su inserción en la base de datos se realiza con éxito, el bucle en *OnStart* continúa con el desplazamiento de los números de las barras copiadas hacia el fondo del historial. Cuando se alcance el final del historial disponible o el límite de barras establecido en la configuración del terminal, *CopyRates* devolverá el error 4401 (HISTORY_NOT_FOUND) y el script saldrá.

Vamos a ejecutar el script en XAUUSD, gráfico H1. El registro debería mostrar algo como lo siguiente:

```
db.isOpen()=true / ok
db.deleteTable(typename(MqlRatesDB))=true / ok
db.createTable<MqlRatesDB>(true)=true / ok
2022.06.29 20:00:00
2022.05.03 04:00:00
2022.03.04 10:00:00
...
CopyRates failed: 4401 HISTORY_NOT_FOUND
Records added: 100000
```

Ahora tenemos la base *QuotesEURUSDH1.sqlite*, sobre la que puede experimentar para probar diversas hipótesis de trading. Puede abrirlo en MetaEditor para asegurarse de que los datos se transfieren correctamente.

Comprobemos una de las estrategias más sencillas basadas en regularidades del historial. Encontraremos las estadísticas de dos barras consecutivas en la misma dirección, desglosadas por hora intradiaria y día de la semana. Si existe una ventaja tangible para alguna combinación de hora y día de la semana, puede considerarse en el futuro como una señal para entrar en el mercado en la dirección de la primera barra.

En primer lugar, diseñemos una consulta SQL que solicite las cotizaciones de un periodo determinado y calcule el movimiento de precios en cada barra, es decir, la diferencia entre precios de apertura adyacentes.

Dado que la hora de las barras se almacena como un número de segundos (según los estándares de *datetime* en MQL5 y, concurrentemente, la «época Unix» de SQL), es deseable convertir su visualización en una cadena para facilitar su lectura, por lo que vamos a iniciar la consulta SELECT del campo *datetime* basada en la función DATETIME:

```
SELECT
   DATETIME(time, 'unixepoch') as datetime, open, ...
```

Este campo no participará en el análisis y se facilita aquí sólo para el usuario. A continuación, se muestra el precio como referencia, para que podamos comprobar el cálculo de los incrementos de precio mediante la impresión de depuración.

Dado que vamos a seleccionar, en su caso, un período determinado de todo el archivo, la condición requerirá el campo *time* en «forma pura», y también deberá añadirse a la solicitud. Además, según el análisis de cotizaciones previsto, necesitaremos aislar de la etiqueta de la barra su hora intradiaria, así como el día de la semana (su numeración corresponde a la adoptada en MQL5, 0 es domingo). Vamos a llamar a las dos últimas columnas de la consulta *intraday* y *day*, respectivamente, y utilizaremos las funciones TIME y STRFTIME para obtenerlas.

```
SELECT
   DATETIME(time, 'unixepoch') as datetime, open,
   time,
   TIME(time, 'unixepoch') AS intraday,
   STRFTIME('%w', time, 'unixepoch') AS day, ...
```

Para calcular el incremento de precio en SQL, puede utilizar la función LAG. Devuelve el valor de la columna especificada con un desplazamiento del número de filas especificado. Por ejemplo, *LAG(X, 1)* significa obtener el valor *X* de la entrada anterior, con el segundo parámetro 1 que significa el desplazamiento por defecto a 1, es decir, se puede omitir para obtener la entrada equivalente *LAG(X)*. Para obtener el valor de la siguiente entrada, llame a *LAG(X,-1)*. En cualquier caso, cuando se utiliza

LAG, se requiere una construcción sintáctica adicional que especifique el orden de clasificación de los registros, en el caso más sencillo, en forma de *OVER(ORDER BY column)*.

Así, para obtener el incremento de precio entre los precios de apertura de dos barras vecinas, escribimos:

```
...
(LAG(open,-1) OVER (ORDER BY time) - open) AS delta, ...
```

Esta columna es predictiva porque mira hacia el futuro.

Podemos revelar que dos barras se formaron en la misma dirección multiplicando los incrementos por ellas: los valores positivos indican una subida o bajada constante:

```
...
(LAG(open,-1) OVER (ORDER BY time) - open) * (open - LAG(open) OVER (ORDER BY time))
AS product, ...
```

Este indicador se elige como el más sencillo de utilizar en el cálculo: para sistemas de trading reales, puede elegir un criterio más complejo.

Para evaluar el beneficio generado por el sistema en el backtest, es necesario multiplicar la dirección de la barra anterior (que actúa como indicador del movimiento futuro) por el incremento del precio en la barra siguiente. La dirección se calcula en la columna *direction* (utilizando la función SIGNO), sólo como referencia. La estimación de beneficios en la columna *estimate* es el producto del movimiento anterior *direction* y el incremento de la barra siguiente (*delta*): si se conserva la dirección, obtenemos un resultado positivo (en puntos).

```
...
SIGN(open - LAG(open) OVER (ORDER BY time)) AS direction,
(LAG(open,-1) OVER (ORDER BY time) - open) * SIGN(open - LAG(open) OVER (ORDER BY tim
AS estimate ...
```

> En las expresiones de un comando SQL, no se pueden utilizar alias AS definidos en el mismo comando. Por eso no podemos determinar *estimate* como *delta * direction*, y tenemos que repetir el cálculo del producto explícitamente. Sin embargo, recordamos que las columnas *delta* y *direction* no son necesarias para el análisis programático y se añaden aquí sólo para visualizar la tabla ante el usuario.

Al final del comando SQL especificamos la tabla desde la que se realiza la selección y las condiciones de filtrado para el intervalo de fechas del backtest: dos parámetros «desde» y «hasta».

```
...
FROM MqlRatesDB
WHERE (time >= ?1 AND time < ?2)
```

Opcionalmente, podemos añadir una restricción *LIMIT?3* (e introducir algún valor pequeño, por ejemplo, 10) para que la verificación visual de los resultados de la consulta al principio no le obligue a buscar entre decenas de miles de registros.

Puede comprobar el funcionamiento del comando SQL utilizando la función *DatabasePrint*; sin embargo, la función, desafortunadamente, no le permite trabajar con consultas preparadas con parámetros. Por lo tanto, tendremos que sustituir la preparación del parámetro SQL '?n' por el formato de cadena de consulta utilizando *StringFormat* y sustituir allí los valores de los parámetros. Otra posibilidad sería evitar por completo *DatabasePrint* y enviar los resultados al registro de forma independiente, línea por línea (a través de un array *DBRow*).

Así, el fragmento final de la solicitud se convertirá en:

```
...
WHERE (time >= %ld AND time < %ld)
ORDER BY time LIMIT %d;
```

Debe tenerse en cuenta que los valores *datetime* de esta consulta procederán de MQL5 en formato «máquina», es decir, el número de segundos transcurridos desde el comienzo de 1970. Si queremos depurar la misma consulta SQL en MetaEditor, entonces es más conveniente escribir la condición de rango de fechas usando literales de fecha (cadenas) como sigue:

```
WHERE (time >= STRFTIME('%s', '2015-01-01') AND time < STRFTIME('%s', '2021-01-01'
```

De nuevo, aquí necesitamos usar la función STRFTIME (el modificador '%s' en SQL establece la transferencia de la cadena de fecha especificada a la etiqueta «Unix epoch»; el hecho de que '%s' se parezca a una cadena de formato MQL5 es sólo una coincidencia).

Guarde la consulta SQL diseñada en un archivo de texto independiente *DBQuotesIntradayLag.sql* y conéctelo como recurso al script de prueba del mismo nombre, *DBQuotesIntradayLag.mq5*.

```
#resource "DBQuotesIntradayLag.sql" as string sql1
```

El primer parámetro del script le permite establecer un prefijo en el nombre de la base de datos, que ya debería existir después de lanzar *DBquotesImport.mq5* en el gráfico con el mismo símbolo y marco temporal. Las siguientes entradas son para el intervalo de fechas y el límite de longitud de la impresión de depuración en el registro.

```
input string Database = "MQL5Book/DB/Quotes";
input datetime SubsetStart = D'2022.01.01';
input datetime SubsetStop = D'2023.01.01';
input int Limit = 10;
```

La tabla con cotizaciones se conoce de antemano, por el script anterior.

```
const string Table = "MqlRatesDB";
```

En la función *OnStart* abrimos la base de datos y nos aseguramos de que la tabla de cotizaciones está disponible.

```
void OnStart()
{
   Print("");
   DBSQLite db(Database + _Symbol + PeriodToString());
   if(!PRTF(db.isOpen())) return;
   if(!PRTF(db.hasTable(Table))) return;
   ...
```

A continuación, sustituimos los parámetros en la cadena de consulta SQL. Prestamos atención no sólo a la sustitución de los parámetros SQL '?n' por secuencias de formato, sino también al doble de los símbolos de porcentaje '%' primero, porque de lo contrario la función *StringFormat* los percibirá como sus propios comandos, y no los pasará por alto en SQL.

```
   string sqlrep = sql1;
   StringReplace(sqlrep, "%", "%%");
   StringReplace(sqlrep, "?1", "%ld");
   StringReplace(sqlrep, "?2", "%ld");
   StringReplace(sqlrep, "?3", "%d");

   const string sqlfmt = StringFormat(sqlrep, SubsetStart, SubsetStop, Limit);
   Print(sqlfmt);
```

Todas estas manipulaciones eran necesarias únicamente para ejecutar la solicitud en el contexto de la función *DatabasePrint*. En la versión de trabajo del script analítico, leeríamos los resultados de la consulta y los analizaríamos mediante programación, pasando por alto el formateo y llamando a *DatabasePrint*.

Por último, vamos a ejecutar la consulta SQL y mostrar la tabla con los resultados en el registro.

```
   DatabasePrint(db.getHandle(), sqlfmt, 0);
}
```

Esto es lo que veremos para 10 barras EURUSD,H1 a principios de 2022:

```
db.isOpen()=true / ok
db.hasTable(Table)=true / ok
SELECT
DATETIME(time, 'unixepoch') as datetime,
open,
time,
TIME(time, 'unixepoch') AS intraday,
STRFTIME('%w', time, 'unixepoch') AS day,
(LAG(open,-1) OVER (ORDER BY time) - open) AS delta,
SIGN(open - LAG(open) OVER (ORDER BY time)) AS direction,
(LAG(open,-1) OVER (ORDER BY time) - open) * (open - LAG(open) OVER (ORDER BY time))
AS product,
(LAG(open,-1) OVER (ORDER BY time) - open) * SIGN(open - LAG(open) OVER (ORDER BY tim
AS estimate
FROM MqlRatesDB
WHERE (time >= 1640995200 AND time < 1672531200)
ORDER BY time LIMIT 10;
#| datetime open time intraday day delta dir product estimate
--+-------------------------------------------------------------------------------
1| 2022-01-03 00:00:00 1.13693 1641168000 00:00:00 1 0.0003200098
2| 2022-01-03 01:00:00 1.13725 1641171600 01:00:00 1 2.999999e-05 1 9.5999478e-09 2.9
3| 2022-01-03 02:00:00 1.13728 1641175200 02:00:00 1 -0.001060006 1 -3.1799748e-08 -0
4| 2022-01-03 03:00:00 1.13622 1641178800 03:00:00 1 -0.0003400007 -1 3.6040028e-07 0
5| 2022-01-03 04:00:00 1.13588 1641182400 04:00:00 1 -0.001579991 -1 5.3719982e-07 0.
6| 2022-01-03 05:00:00 1.1343 1641186000 05:00:00 1 0.0005299919 -1 -8.3739827e-07 -0
7| 2022-01-03 06:00:00 1.13483 1641189600 06:00:00 1 -0.0007699937 1 -4.0809905e-07 -
8| 2022-01-03 07:00:00 1.13406 1641193200 07:00:00 1 -0.0002600149 -1 2.0020098e-07 0
9| 2022-01-03 08:00:00 1.1338 1641196800 08:00:00 1 0.000510001 -1 -1.3260079e-07 -0.
10| 2022-01-03 09:00:00 1.13431 1641200400 09:00:00 1 0.0004800036 1 2.4480023e-07 0.
...
```

Es fácil asegurarse de que la hora intradiaria de la barra está correctamente asignada, así como el día de la semana - 1, que corresponde al lunes. También puede comprobar el incremento delta. Los valores

product y *estimate* están vacíos en la primera fila porque necesitan que se calcule la fila anterior que falta.

Vamos a complicar nuestra consulta SQL agrupando los registros con las mismas combinaciones de hora del día (*intraday*) y día de la semana (*day*), y calculando un determinado indicador objetivo que caracterice el éxito del trading para cada una de estas combinaciones. Tomemos como indicador el tamaño medio de las celdas *product* dividido por la desviación típica de los mismos productos. Cuanto mayor sea el producto medio de los incrementos de precio de las barras vecinas, mayor será el beneficio esperado, y cuanto menor sea la dispersión de estos productos, más estable será la previsión. El nombre del indicador en la consulta SQL es *objective*.

Además del indicador de objetivos, también calcularemos la estimación de beneficios (*backtest_profit*) y el factor de beneficios (*backtest_PF*). Estimaremos el beneficio como la suma de los incrementos de precio (*estimate*) para todas las barras en el contexto de la hora intradiaria y el día de la semana (el tamaño de la barra de apertura como incremento de precio es un análogo del beneficio futuro en puntos por una barra). El factor de beneficio es tradicionalmente el cociente de los incrementos positivos y negativos.

```
SELECT
AVG(product) / STDDEV(product) AS objective,
SUM(estimate) AS backtest_profit,
SUM(CASE WHEN estimate >= 0 THEN estimate ELSE 0 END) /
SUM(CASE WHEN estimate < 0 THEN -estimate ELSE 0 END) AS backtest_PF,
intraday, day
FROM
(
SELECT
time,
TIME(time, 'unixepoch') AS intraday,
STRFTIME('%w', time, 'unixepoch') AS day,
(LAG(open,-1) OVER (ORDER BY time) - open) AS delta,
SIGN(open - LAG(open) OVER (ORDER BY time)) AS direction,
(LAG(open,-1) OVER (ORDER BY time) - open) * (open - LAG(open) OVER (ORDER BY time))
AS product,
(LAG(open,-1) OVER (ORDER BY time) - open) * SIGN(open - LAG(open) OVER (ORDER BY tim
AS estimate
FROM MqlRatesDB
WHERE (time >= STRFTIME('%s', '2015-01-01') AND time < STRFTIME('%s', '2021-01-01'))
)
GROUP BY intraday, day
ORDER BY objective DESC
```

Se ha anidado la primera consulta SQL, a partir de la cual ahora acumulamos datos con una consulta SQL externa. La agrupación por todas las combinaciones de hora y día de la semana proporciona un «extra» de *GROUP BY intraday, day*. Además, hemos añadido la ordenación por indicador de destino (*ORDER BY objective DESC*) para que las mejores opciones estén en la parte superior de la tabla.

En la consulta anidada, eliminamos el parámetro LIMIT, ya que el número de grupos ha pasado a ser aceptable, mucho menor que el número de barras analizadas. Así, para H1 obtenemos 120 opciones (24 * 5).

La consulta ampliada se coloca en el archivo de texto *DBQuotesIntradayLagGroup.sql*, que a su vez está conectado como recurso al script de prueba del mismo nombre, *DBQuotesIntradayLagGroup.mq5*. Su código fuente difiere poco del anterior, por lo que mostraremos inmediatamente el resultado de su

lanzamiento para el intervalo de fechas por defecto: desde principios de 2015 hasta principios de 2021 (excluyendo 2021 y 2022).

```
db.isOpen()=true / ok
db.hasTable(Table)=true / ok
SELECT
AVG(product) / STDDEV(product) AS objective,
SUM(estimate) AS backtest_profit,
SUM(CASE WHEN estimate >= 0 THEN estimate ELSE 0 END) /
SUM(CASE WHEN estimate < 0 THEN -estimate ELSE 0 END) AS backtest_PF,
intraday, day
FROM
(
SELECT
...
FROM MqlRatesDB
WHERE (time >= 1420070400 AND time < 1609459200)
)
GROUP BY intraday, day
ORDER BY objective DESC
#| objective backtest_profit backtest_PF intraday day
---+-----------------------------------------------------------------------------
1| 0.16713214428916 0.073200000000001 1.46040631486258 16:00:00 5
2| 0.118128291843983 0.0433099999999995 1.33678071539657 20:00:00 3
3| 0.103701251751617 0.00929999999999853 1.14148790506616 05:00:00 2
4| 0.102930330078208 0.0164399999999973 1.1932071923845 08:00:00 4
5| 0.089531492651001 0.0064300000000006 1.10167615433271 07:00:00 2
6| 0.0827628326995007 -8.99999999970369e-05 0.999601152226913 17:00:00 4
7| 0.0823433025146974 0.0159700000000012 1.21665988332657 21:00:00 1
8| 0.0767938336191962 0.00522999999999874 1.04226945769012 13:00:00 1
9| 0.0657741522256548 0.0162299999999986 1.09699976093712 15:00:00 2
10| 0.0635243373432768 0.00932000000000044 1.08294766820933 22:00:00 3
...
110| -0.0814131025461459 -0.0189100000000015 0.820605255668329 21:00:00 5
111| -0.0899571263478305 -0.0321900000000028 0.721250432975386 22:00:00 4
112| -0.0909772560603298 -0.0226100000000016 0.851161872161138 19:00:00 4
113| -0.0961794181717023 -0.00846999999999931 0.936377976414036 12:00:00 5
114| -0.108868074018582 -0.0246099999999998 0.634920634920637 00:00:00 5
115| -0.109368419185336 -0.0250700000000013 0.744496534855268 08:00:00 2
116| -0.121893581607986 -0.0234599999999998 0.610945273631843 00:00:00 3
117| -0.135416609546408 -0.0898899999999971 0.343437294573087 00:00:00 1
118| -0.142128458003631 -0.0255200000000018 0.681835182645536 06:00:00 4
119| -0.142196924506816 -0.0205700000000004 0.629769618430515 00:00:00 2
120| -0.15200009633513 -0.0301499999999988 0.708864426419475 02:00:00 1
```

Por lo tanto, el análisis nos indica que la barra H1 de 16 horas del viernes es la mejor candidata para continuar la tendencia basada en la barra anterior. Le sigue en preferencia la barra de los miércoles a las 20 horas. Y así sucesivamente.

Sin embargo, es conveniente comprobar los ajustes encontrados en el periodo de reenvío.

Para ello, podemos ejecutar la consulta SQL actual no sólo en el intervalo de fechas «pasado» (en nuestra prueba hasta 2021), sino una vez más en el «futuro» (desde principios de 2021). Los resultados de ambas consultas deben unirse (JOIN) por nuestros grupos (*intraday*, *day*). A continuación, manteniendo la ordenación por el indicador objetivo, veremos en las columnas

adyacentes el beneficio y el factor de beneficio para las mismas combinaciones de hora y día de la semana, y cuánto se hundieron.

He aquí la consulta SQL final (abreviada):

```
SELECT * FROM
(
SELECT
AVG(product) / STDDEV(product) AS objective,
SUM(estimate) AS backtest_profit,
SUM(CASE WHEN estimate >= 0 THEN estimate ELSE 0 END) /
SUM(CASE WHEN estimate < 0 THEN -estimate ELSE 0 END) AS backtest_PF,
intraday, day
FROM
(
SELECT ...
FROM MqlRatesDB
WHERE (time >= STRFTIME('%s', '2015-01-01') AND time < STRFTIME('%s', '2021-01-01'))
)
GROUP BY intraday, day
) backtest
JOIN
(
SELECT
SUM(estimate) AS forward_profit,
SUM(CASE WHEN estimate >= 0 THEN estimate ELSE 0 END) /
SUM(CASE WHEN estimate < 0 THEN -estimate ELSE 0 END) AS forward_PF,
intraday, day
FROM
(
SELECT ...
FROM MqlRatesDB
WHERE (time >= STRFTIME('%s', '2021-01-01'))
)
GROUP BY intraday, day
) forward
USING(intraday, day)
ORDER BY objective DESC
```

El texto completo de la solicitud figura en el archivo *DBQuotesIntradayBackAndForward.sql*. Se conecta como recurso en el script *DBQuotesIntradayBackAndForward.mq5*.

Ejecutando el script con la configuración por defecto, obtenemos los siguientes indicadores (con abreviaturas):

```
#| objective backtest_profit backtest_PF intraday day forward_profit forward_PF
--+-------------------------------------------------------------------------------
1| 0.16713214428916 0.073200000001 1.46040631486 16:00:00 5 0.004920000048 1.12852664
2| 0.118128291843983 0.0433099999995 1.33678071539 20:00:00 3 0.007880000055 1.277856
3| 0.103701251751617 0.00929999999853 1.14148790506 05:00:00 2 0.002210000082 1.12149
4| 0.102930330078208 0.0164399999973 1.1932071923 08:00:00 4 0.001409999969 1.0725308
5| 0.089531492651001 0.0064300000006 1.10167615433 07:00:00 2 -0.009119999869 0.56174
6| 0.0827628326995007 -8.99999999970e-05 0.999601152226 17:00:00 4 0.009070000091 1.1
7| 0.0823433025146974 0.0159700000012 1.21665988332 21:00:00 1 0.00250999999 1.121314
8| 0.0767938336191962 0.00522999999874 1.04226945769 13:00:00 1 -0.008490000055 0.753
9| 0.0657741522256548 0.0162299999986 1.09699976093 15:00:00 2 0.01423999997 1.349791
10| 0.0635243373432768 0.00932000000044 1.08294766820 22:00:00 3 -0.00456999993 0.828
...
```

Así pues, el sistema de trading con los mejores calendarios de trading encontrados sigue arrojando beneficios en el periodo «futuro», aunque no tan grandes como en el backtest.

Por supuesto, el ejemplo considerado es sólo un caso particular de un sistema de trading. Podríamos, por ejemplo, encontrar combinaciones de la hora y el día de la semana en que una estrategia de inversión funciona en barras vecinas, o basarnos en otros principios totalmente distintos (análisis de ticks, calendario, cartera de señales de trading, etc.).

La conclusión es que el motor SQLite proporciona muchas herramientas prácticas que tendría que implementar en MQL5 por su cuenta. A decir verdad, aprender SQL lleva su tiempo. La plataforma permite elegir la combinación óptima de dos tecnologías para una programación eficaz.

7.7 Desarrollo y conexión de bibliotecas de formatos binarios

Además de los tipos especializados de programas MQL (*Asesores Expertos*, *indicadores*, scripts y servicios) la plataforma MetaTrader 5 permite crear y conectar módulos binarios independientes con funcionalidad arbitraria, compilados como archivos ex5 o DLL (Dynamic Link Library) de uso común, estándares para Windows. Pueden ser algoritmos analíticos, visualización gráfica, interacción en red con servicios web, control de programas externos o el propio sistema operativo. En cualquier caso, estas librerías funcionan en el terminal no como programas MQL independientes, sino en conjunción con un programa de cualquiera de los 4 tipos anteriores.

La idea de integrar la biblioteca y el programa principal (padre) es que la biblioteca exporte determinadas funciones, es decir, las declare disponibles para su uso desde el exterior, y el programa importe sus prototipos. Es la descripción de prototipos -conjuntos de nombres, listas de parámetros y valores de retorno- lo que permite llamar a estas funciones en el código sin disponer de su implementación.

A continuación, durante el lanzamiento del programa MQL, se realiza la vinculación dinámica temprana. Esto implica cargar la biblioteca después del programa principal y establecer la correspondencia entre los prototipos importados y las funciones exportadas disponibles en la biblioteca. Establecer correspondencias uno a uno por nombres, listas de parámetros y tipos de retorno es un requisito previo para el éxito de la carga. Si no se puede encontrar una implementación exportada correspondiente para la descripción de importación de al menos una función, se cancelará la ejecución del programa MQL (finalizará con un error en la fase de inicio).

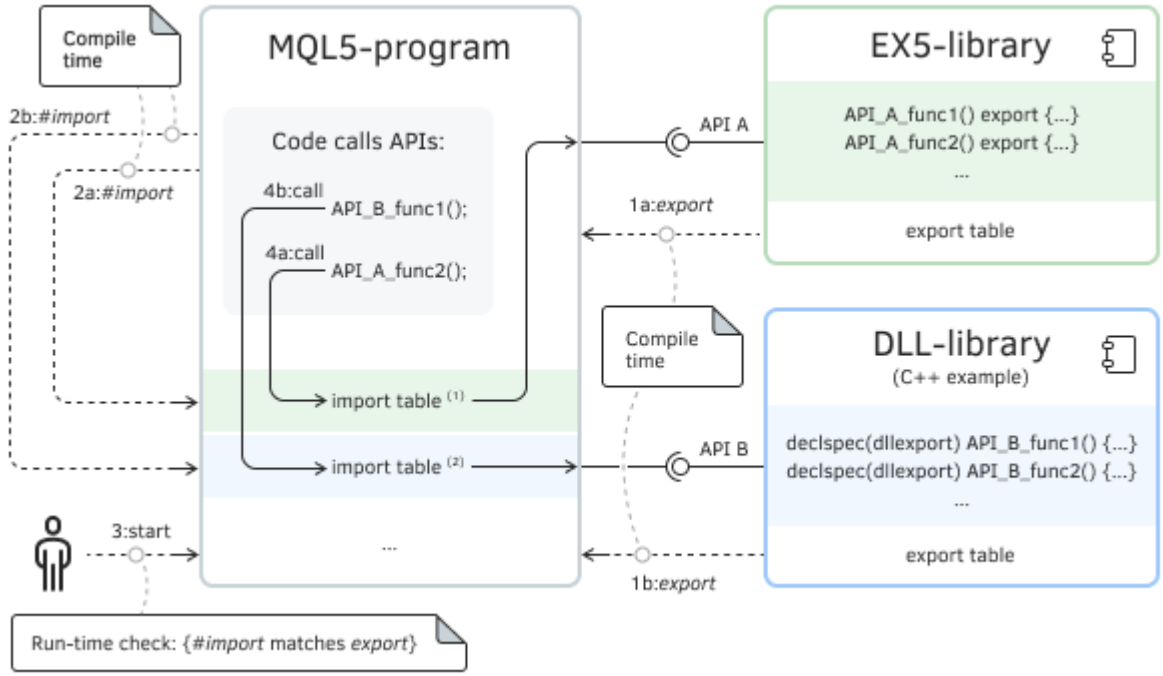


Diagrama de componentes de comunicación de un programa MQL con bibliotecas

No se puede seleccionar una biblioteca incluida al iniciar un programa MQL. Esta vinculación la establece el desarrollador al compilar el programa principal junto con las importaciones de bibliotecas. No obstante, el usuario puede sustituir manualmente un archivo ex5/dll por otro entre inicios del programa (siempre que los prototipos de las funciones exportadas implementadas coincidan en las bibliotecas). Esto puede utilizarse, por ejemplo, para cambiar el idioma de la interfaz de usuario si las bibliotecas contienen recursos de cadenas etiquetadas. Sin embargo, lo más frecuente es que las bibliotecas se utilicen como un producto comercial con ciertos conocimientos técnicos, que el autor no está dispuesto a distribuir en forma de archivos de encabezado abiertos.

Para los programadores que han llegado a MQL5 desde otros entornos y ya están familiarizados con la tecnología DLL, nos gustaría añadir una nota sobre la vinculación dinámica tardía, que es una de las ventajas de las DLL. La conexión dinámica completa de un programa MQL (o módulo DLL) a otro programa MQL durante la ejecución es imposible. La única acción similar que MQL5 le permite hacer «sobre la marcha» es la vinculación de un Asesor Experto y un indicador a través de *iCustom* o *IndicatorCreate*, donde el indicador actúa como una biblioteca vinculada dinámicamente (sin embargo, la interacción programática debe realizarse a través de la API del indicador, lo que supone una mayor sobrecarga para *CopyBuffer*, en comparación con las llamadas directas a funciones a través de *export/#import*).

Tenga en cuenta que, en casos normales, cuando un programa MQL se compila a partir de fuentes sin importar funciones externas, se utiliza el enlazado estático, es decir, el código binario generado hace referencia directa a las funciones llamadas ya que son conocidas en el momento de la compilación.

En sentido estricto, una biblioteca también puede depender de otras bibliotecas, es decir, puede importar algunas de sus funciones. En teoría, la cadena de estas dependencias puede ser incluso más larga: por ejemplo, un programa MQL incluye la biblioteca A, la biblioteca A utiliza la biblioteca B y la biblioteca B, a su vez, utiliza la biblioteca C. Sin embargo, estas cadenas no son deseables porque complican la distribución e instalación del producto, además de dificultar la identificación de las causas de posibles problemas de arranque. Por lo tanto, las bibliotecas suelen estar conectadas directamente al programa MQL padre.

En este capítulo describiremos el proceso de creación de librerías en MQL5, la exportación e importación de funciones (incluyendo las restricciones sobre los tipos de datos utilizados en ellas), así como la conexión de DLLs externas (ya preparadas). El desarrollo de DLL va más allá de lo contemplado en este libro.

7.7.1 Creación de bibliotecas ex5; *export* de funciones

Para describir una biblioteca, añada la directiva *#property library* al código fuente del módulo principal (compilado) (normalmente, al principio del archivo).

```
#property library
```

La especificación de esta directiva en cualquier otro archivo incluido en el proceso de compilación a través de *#include* no tiene ningún efecto.

La propiedad *library* informa al compilador de que el archivo ex5 dado es una biblioteca: una marca sobre esto se almacena en el encabezado del archivo ex5.

Una carpeta separada *MQL5/Libraries* está reservada para las bibliotecas en MetaTrader 5. Puede organizar una jerarquía de carpetas anidadas en ella, al igual que para otros tipos de programas en MQL5.

Las bibliotecas no participan directamente en el manejo de eventos, y por lo tanto el compilador no requiere la presencia de ningún manejador estándar en el código. No obstante, puede llamar a las funciones exportadas de la librería desde los manejadores de eventos del programa MQL al que está conectada la librería.

Para exportar una función de una biblioteca, basta con marcarla con una palabra clave especial *export*. Este modificador debe colocarse al final del encabezado de la función.

```
result_type function_id ( [ parameter_type parameter_id
                          [ = default_value] ...] ) export
{
   ...
}
```

Los parámetros deben ser tipos simples o cadenas, estructuras con campos de dichos tipos o sus arrays. Los punteros y las referencias están permitidos para los tipos de objeto MQL5 (para conocer las restricciones sobre la importación de DLL, consulte la sección correspondiente).

Veamos algunos ejemplos. El parámetro es un número primo:

```
double Algebraic2(const double x) export
{
   return x / sqrt(1 + x * x);
}
```

Los parámetros son un puntero a un objeto y una referencia a un puntero (lo que permite asignar un puntero dentro de la función).

```
class X
{
public:
   X() { Print(__FUNCSIG__); }
};
void setObject(const X *obj) export { ... }
void getObject(X *&obj) export { obj = new X(); }
```

El parámetro es una estructura:

```
struct Data
{
   int value;
   double data[];
   Data(): value(0) { }
   Data(const int i): value(i) { ArrayResize(data, i); }
};

void getRefStruct(const int i, Data &data) export { ... }
```

Sólo se pueden exportar funciones, pero no clases o estructuras enteras. Algunas de estas limitaciones pueden evitarse con la ayuda de punteros y referencias, de los que hablaremos con más detalle más adelante.

Las plantillas de funciones no pueden declararse con la palabra clave *export* y en la directiva *#import*.

El modificador *export* indica al compilador que incluya la función en la tabla de funciones exportadas dentro del ejecutable ex5 dado. Gracias a ello, dichas funciones pasan a estar disponibles («visibles») desde otros programas MQL, donde pueden utilizarse tras importarlas con una directiva especial *#import*.

Todas las funciones que vayan a exportarse deben marcarse con el modificador *export*. Aunque no es necesario que el programa principal los importe todos, ya que sólo puede importar los necesarios.

Si olvida exportar una función pero la incluye en la directiva de importación del programa MQL principal, al iniciar este último se producirá un error:

```
cannot find 'function' in 'library.ex5'
unresolved import function call
```

Un problema similar surgirá si hay discrepancias en la descripción de la función exportada y su prototipo importado. Esto puede ocurrir, por ejemplo, si olvida recompilar una biblioteca o un programa principal después de realizar cambios en la interfaz de programación, que suele describirse en un archivo de encabezado independiente.

> La depuración de bibliotecas no es posible, por lo que si es necesario, debe tener un script auxiliar u otro programa MQL que se construye a partir de los códigos fuente de la biblioteca en modo depurador y se puede ejecutar con puntos de interrupción o paso a paso. Por supuesto, esto requerirá emular llamadas a funciones exportadas utilizando algunos datos reales o artificiales.
>
> En el caso de las DLL, la descripción de las funciones exportadas se realiza de forma diferente, dependiendo del lenguaje de programación en el que se hayan creado. Busque más detalles en la documentación de los entornos de desarrollo que haya elegido.

Consideremos el ejemplo de una biblioteca sencilla *MQL5/Libraries/MQL5Book/LibRand.mq5*, de la que se exportan varias funciones con distintos tipos de parámetros y resultados. La biblioteca está diseñada para generar datos aleatorios:

- de datos numéricos con una distribución pseudonormal;
- de cadenas con caracteres aleatorios de los conjuntos dados (puede ser útil para contraseñas).

En concreto, puede obtener un número aleatorio utilizando la función *PseudoNormalValue*, en la que el valor esperado y la varianza se establecen como parámetros.

```
double PseudoNormalValue(const double mean = 0.0, const double sigma = 1.0,
   const bool rooted = false) export
{
   // use ready-made sqrt for mass generation in a cycle in PseudoNormalArray
   const double s = !rooted ? sqrt(sigma) : sigma;
   const double r = (rand() - 16383.5) / 16384.0; // [-1,+1] excluding borders
   const double x = -(log(1 / ((r + 1) / 2) - 1) * s) / M_PI * M_E + mean;
   return x;
}
```

La función *PseudoNormalArray* rellena el array con valores aleatorios en una cantidad determinada (*n*) y con la distribución requerida.

```
bool PseudoNormalArray(double &array[], const int n,
   const double mean = 0.0, const double sigma = 1.0) export
{
   bool success = true;
   const double s = sqrt(fabs(sigma)); // passing ready sqrt when calling PseudoNorma
   ArrayResize(array, n);
   for(int i = 0; i < n; ++i)
   {
      array[i] = PseudoNormalValue(mean, s, true);
      success = success && MathIsValidNumber(array[i]);
   }
   return success;
}
```

Para generar una cadena aleatoria, escribimos la función *RandomString*, que «selecciona» del conjunto de caracteres suministrado (*pattern*) una cantidad determinada (*length*) de caracteres arbitrarios. Cuando el parámetro *pattern* está en blanco (por defecto), se asume un conjunto completo de letras y números. Para obtenerlo se utilizan las funciones auxiliares *StringPatternAlpha* y *StringPatternDigit*; estas funciones también son exportables (no aparecen en el libro, véase el código fuente).

```
string RandomString(const int length, string pattern = NULL) export
{
   if(StringLen(pattern) == 0)
   {
      pattern = StringPatternAlpha() + StringPatternDigit();
   }
   const int size = StringLen(pattern);
   string result = "";
   for(int i = 0; i < length; ++i)
   {
      result += ShortToString(pattern[rand() % size]);
   }
   return result;
}
```

En general, para trabajar con una biblioteca, es necesario publicar un archivo de encabezado que describa todo lo que debe estar disponible en ella desde el exterior (y los detalles de la implementación interna pueden y deben ocultarse). En nuestro caso, ese archivo se llama *MQL5Book/LibRand.mqh*. En concreto, describe tipos definidos por el usuario (en nuestro caso, la enumeración STRING_PATTERN) y prototipos de funciones.

Aunque aún no conocemos la sintaxis exacta del bloque *#import*, esto no debería afectar a la claridad de las declaraciones que contiene: los encabezados de las funciones exportadas se repiten aquí pero sin la palabra clave *export*.

```
enum STRING_PATTERN
{
   STRING_PATTERN_LOWERCASE = 1, // lowercase letters only
   STRING_PATTERN_UPPERCASE = 2, // capital letters only
   STRING_PATTERN_MIXEDCASE = 3  // both registers
};

#import "MQL5Book/LibRand.ex5"
string StringPatternAlpha(const STRING_PATTERN _case = STRING_PATTERN_MIXEDCASE);
string StringPatternDigit();
string RandomString(const int length, string pattern = NULL);
void RandomStrings(string &array[], const int n, const int minlength,
   const int maxlength, string pattern = NULL);
void PseudoNormalDefaultMean(const double mean = 0.0);
void PseudoNormalDefaultSigma(const double sigma = 1.0);
double PseudoNormalDefaultValue();
double PseudoNormalValue(const double mean = 0.0, const double sigma = 1.0,
   const bool rooted = false);
bool PseudoNormalArray(double &array[], const int n,
   const double mean = 0.0, const double sigma = 1.0);
#import
```

Escribiremos un script de prueba que utilice esta biblioteca en la próxima sección, después de estudiar la directiva *#import*.

7.7.2 Inclusión de bibliotecas; *#import* de funciones

Las funciones se importan de módulos MQL5 compilados (archivos *.ex5) y de módulos de bibliotecas dinámicas de Windows (archivos *.dll). El nombre del módulo se especifica en la directiva *#import*, seguida de las descripciones de los prototipos de las funciones importadas. Un bloque de este tipo debe terminar con otra directiva *#import*, además, puede ser sin nombre y simplemente cerrar el propio bloque, o se puede especificar el nombre de otra biblioteca en la directiva, y así el siguiente bloque de importación comienza al mismo tiempo. Una serie de bloques de importación debe terminar siempre con una directiva sin nombre de biblioteca.

En su forma más simple, la directiva tiene el siguiente aspecto:

```
#import "[path] module_name [.extension]"
  function_type function_name([parameter_list]);
  [function_type function_name([parameter_list]);]
   ...
#import
```

El nombre del archivo de biblioteca puede especificarse sin la extensión: entonces se asume por defecto la DLL. Se necesita la extensión *ex5*.

El nombre puede ir precedido de la ruta de ubicación de la biblioteca. Por defecto, si no hay ruta, las bibliotecas se buscan en la carpeta *MQL5/Libraries* o en la carpeta junto al programa MQL donde está conectada la biblioteca. En caso contrario, se aplican reglas diferentes para buscar las bibliotecas en función de si el tipo es DLL o EX5. Estas normas se tratan en una sección aparte.

He aquí un ejemplo de bloques de importación secuencial de dos bibliotecas:

```
#import "user32.dll"
   int     MessageBoxW(int hWnd, string szText, string szCaption, int nType);
   int     SendMessageW(int hWnd, int Msg, int wParam, int lParam);
#import "lib.ex5"
   double  round(double value);
#import
```

Con estas directivas, las funciones importadas se pueden llamar desde el código fuente de la misma manera que las funciones definidas directamente en el propio programa MQL. Todas las cuestiones técnicas relacionadas con la carga de bibliotecas y la redirección de llamadas a módulos de terceros son gestionadas por el entorno de ejecución del programa MQL.

Para que el compilador emita correctamente la llamada a la función importada y organice el paso de parámetros, se requiere una descripción completa: con el tipo de resultado, con todos los parámetros, modificadores y valores por defecto, si están presentes en el código fuente.

Dado que las funciones importadas están fuera del módulo compilado, el compilador no puede comprobar la corrección de los parámetros pasados y los valores de retorno. Cualquier discrepancia entre el formato de los datos esperados y los recibidos dará lugar a un error durante la ejecución del programa, y esto puede manifestarse como una parada crítica del programa, o un comportamiento inesperado.

Si la biblioteca no se pudo cargar o no se encontró la función importada llamada, el programa MQL termina con un mensaje correspondiente en el registro. El programa no podrá ejecutarse hasta que se resuelva el problema: por ejemplo, modificando y recompilando, colocando la biblioteca necesaria en uno de los lugares de la ruta de búsqueda o permitiendo el uso de la DLL (sólo para DLL).

Cuando compartas varias bibliotecas (no importa si son DLL o EX5), recuerde que deben tener nombres diferentes, independientemente de sus directorios de ubicación. Todas las funciones importadas obtienen un ámbito que coincide con el nombre del archivo de la biblioteca, es decir, es una especie de espacio de nombres asignado implícitamente a cada biblioteca incluida.

Las funciones importadas pueden tener cualquier nombre, incluidos los que coinciden con los nombres de las funciones integradas (aunque no se recomienda). Además, es posible importar simultáneamente funciones con los mismos nombres de distintos módulos. En estos casos, la operación permisos contextuales debe aplicarse para determinar a qué función se debe llamar.

Por ejemplo:

```
#import "kernel32.dll"
   int GetLastError();
#import "lib.ex5"
   int GetLastError();
#import

class Foo
{
public:
   int GetLastError() { return(12345); }
   void func()
   {
      Print(GetLastError());           // call a class method
      Print(::GetLastError());         // calling the built-in (global) MQL5 function
      Print(kernel32::GetLastError()); // function call from kernel32.d
      Print(lib::GetLastError());      // function call from lib.ex5
   }
};

void OnStart()
{
   Foo foo;
   foo.func();
}
```

Veamos un ejemplo sencillo del script *LibRandTest.mq5*, que utiliza funciones de la librería EX5 creada en la sección anterior.

```
#include <MQL5Book/LibRand.mqh>
```

En los parámetros de entrada, puede seleccionar el número de elementos del array de números, los parámetros de distribución, así como el paso del histograma, que calcularemos para asegurarnos de que la distribución corresponde aproximadamente a la ley normal.

```
input int N = 10000;
input double Mean = 0.0;
input double Sigma = 1.0;
input double HistogramStep = 0.5;
input int RandomSeed = 0;
```

La inicialización del generador de números aleatorios integrados en MQL5 (distribución uniforme) se realiza por el valor de *RandomSeed* o, si aquí se deja 0, se elige *GetTickCount* (nuevo en cada inicio).

Para construir un histograma, utilizamos *MapArray* y *QuickSortStructT* (ya hemos trabajado con ellos en las secciones sobre indicadores multidivisa y sobre ordenación de arrays, respectivamente). El mapa acumulará contadores de aciertos de números aleatorios en las celdas del histograma con un paso de *HistogramStep*.

```
#include <MQL5Book/MapArray.mqh>
#include <MQL5Book/QuickSortStructT.mqh>
```

Para mostrar un histograma basado en el mapa, es necesario poder ordenar el mapa en orden clave-valor. Para ello, tuvimos que definir una clase derivada.

```
#define COMMA ,

template<typename K,typename V>
class MyMapArray: public MapArray<K,V>
{
public:
   void sort()
   {
      SORT_STRUCT(Pair<K COMMA V>, array, key);
   }
};
```

Tenga en cuenta que la macro COMMA se convierte en una representación alternativa del carácter coma ',' y se utiliza cuando se llama a otra macro SORT_STRUCT. Si no fuera por esta sustitución, la coma dentro del Par<K,V> sería interpretada por el preprocesador como un separador de parámetros de macro normal, como resultado de lo cual se recibirían 4 parámetros a la entrada de SORT_STRUCT en lugar de los 3 esperados, lo que causaría un error de compilación. El preprocesador no sabe nada acerca de la sintaxis MQL5.

Al principio de *OnStart*, tras la inicialización del generador, comprobamos la recepción de una única cadena aleatoria y un array de cadenas de diferentes longitudes.

```
void OnStart()
{
   const uint seed = RandomSeed ? RandomSeed : GetTickCount();
   Print("Random seed: ", seed);
   MathSrand(seed);

   // call two library functions: StringPatternDigit and RandomString
   Print("Random HEX-string: ", RandomString(30, StringPatternDigit() + "ABCDEF"));
   Print("Random strings:");
   string text[];
   RandomStrings(text, 5, 10, 20);         // 5 lines from 10 to 20 characters long
   ArrayPrint(text);
   ...
```

A continuación, probamos números aleatorios distribuidos normalmente.

```
   // call another library function: PseudoNormalArray
   double x[];
   PseudoNormalArray(x, N, Mean, Sigma);   // filled array x

   Print("Random pseudo-gaussian histogram: ");

   // take 'long' as key type, because 'int' has already been used for index access
   MyMapArray<long,int> map;

   for(int i = 0; i < N; ++i)
   {
// value x[i] determines the cell of the histogram, where we increase the statistics
      map.inc((long)MathRound(x[i] / HistogramStep));
   }
   map.sort();                             // sort by key (i.e. by value)

   int max = 0;                            // searching for maximum for normalization
   for(int i = 0; i < map.getSize(); ++i)
   {
      max = fmax(max, map.getValue(i));
   }

   const double scale = fmax(max / 80, 1); // the histogram has a maximum of 80 symbo

   for(int i = 0; i < map.getSize(); ++i)  // print the histogram
   {
      const int p = (int)MathRound(map.getValue(i) / scale);
      string filler;
      StringInit(filler, p, '*');
      Print(StringFormat("%+.2f (%4d)",
         map.getKey(i) * HistogramStep, map.getValue(i)), " ", filler);
   }
```

He aquí el resultado cuando se ejecuta con la configuración por defecto (temporizador de aleatorización: cada ejecución elegirá un nuevo *seed*).

```
Random seed: 8859858
Random HEX-string: E58B125BCCDA67ABAB2F1C6D6EC677
Random strings:
"K4ZOpdIy5yxq4ble2" "NxTrVRl6q5j3Hr2FY" "6qxRdDzjp3WNA8xV" "UlOPYinnGd36" "6OCmde6rvE
Random pseudo-gaussian histogram:
-9.50 ( 2)
-8.50 ( 1)
-8.00 ( 1)
-7.00 ( 1)
-6.50 ( 5)
-6.00 ( 10) *
-5.50 ( 10) *
-5.00 ( 24) *
-4.50 ( 28) **
-4.00 ( 50) ***
-3.50 ( 100) ******
-3.00 ( 195) ***********
-2.50 ( 272) ***************
-2.00 ( 510) ****************************
-1.50 ( 751) ****************************************
-1.00 (1029) ********************************************************
-0.50 (1288) ************************************************************************
+0.00 (1457) ************************************************************************
+0.50 (1263) ************************************************************************
+1.00 (1060) ********************************************************
+1.50 ( 772) ****************************************
+2.00 ( 480) **************************
+2.50 ( 280) ***************
+3.00 ( 172) **********
+3.50 ( 112) ******
+4.00 ( 52) ***
+4.50 ( 43) **
+5.00 ( 10) *
+5.50 ( 8)
+6.00 ( 8)
+6.50 ( 2)
+7.00 ( 3)
+7.50 ( 1)
```

En esta biblioteca, sólo hemos exportado e importado funciones con tipos integrados. Sin embargo, las interfaces de objetos con estructuras, clases y plantillas son mucho más interesantes y demandadas desde un punto de vista práctico. Hablaremos de los matices de su uso en las bibliotecas en una sección aparte.

Cuando se prueban Asesores Expertos e indicadores en el probador, hay que tener en cuenta un punto importante relacionado con las bibliotecas. Las bibliotecas necesarias para el principal programa MQL probado se determinan automáticamente a partir de las directivas #import. No obstante, si se llama a un indicador personalizado desde el programa principal, al que está conectada alguna biblioteca, entonces es necesario indicar explícitamente en las propiedades del programa que depende indirectamente de una biblioteca concreta. Esto se hace con la directiva

```
#property tester_library«path_library_name.extension».
```

7.7.3 Orden de búsqueda de archivos de biblioteca

Si el nombre de la biblioteca se especifica sin ruta o con una ruta relativa, la búsqueda se realiza según distintas reglas en función del tipo de biblioteca.

Las bibliotecas del sistema (DLL) se cargan según las reglas del sistema operativo. Si la librería ya está cargada (por ejemplo, por otro Asesor Experto, o incluso desde otro terminal de cliente lanzado en paralelo), entonces la llamada va a la librería ya cargada. De lo contrario, la búsqueda sigue la secuencia siguiente:

1. La carpeta desde la que se lanzó el programa EX5 compilado que importó la DLL.
2. La carpeta *MQL5/Libraries*.
3. La carpeta donde se encuentra el terminal MetaTrader 5 en ejecución.
4. La carpeta del sistema (normalmente dentro de Windows)
5. El directorio de Windows
6. La carpeta de trabajo actual del proceso del terminal (puede ser diferente de la carpeta de ubicación del terminal).
7. Carpetas listadas en la variable de sistema PATH.

En las directivas *#import* no se recomienda utilizar un nombre de módulo cargable completamente cualificado de la forma *Drive:/Directory/FileName.dll*.

Si la DLL utiliza otra DLL en su trabajo, en ausencia de la segunda DLL, la primera no podrá cargarse.

La búsqueda de una biblioteca EX5 importada se realiza en la secuencia siguiente:

1. Carpeta para iniciar el programa EX5 de importación.
2. Carpeta *MQL5/Libraries* de instancia de terminal específica.
3. Carpeta *MQL5/Libraries* de la carpeta común de todos los terminales MetaTrader 5 (*Common/MQL5/Libraries*).

Antes de cargar un programa MQL se forma una lista general de todos los módulos de biblioteca EX5, donde los módulos compatibles deben ser utilizados tanto desde el propio programa como desde bibliotecas de esta lista. Se llama lista de dependencias y puede convertirse en un «árbol» muy ramificado.

Para las bibliotecas EX5, el terminal también proporciona una descarga única de módulos reutilizables.

Independientemente del tipo de biblioteca, cada instancia de la misma trabaja con sus propios datos relacionados con el contexto del Asesor Experto, script, servicio o indicador de llamada. Las librerías no son una herramienta de acceso compartido a variables o arrays MQL5.

Las bibliotecas de EX5 y DLL se ejecutan en el subproceso del módulo de llamada.

No hay medios regulares para encontrar en el código de la biblioteca desde donde se cargó.

7.7.4 Particularidades de la conexión DLL

Las siguientes entidades no pueden pasarse como parámetros a funciones importadas de una DLL:

- Clases (objetos y punteros a ellas)
- Estructuras que contienen arrays dinámicas, cadenas, clases y otras estructuras complejas.

- Arrays de cadenas o los objetos complejos anteriores

Todos los parámetros de tipo simple se pasan por valor a menos que se indique explícitamente que se pasan por referencia. Cuando se pasa una cadena, se pasa la dirección del búfer de la cadena copiada; si la cadena se pasa por referencia, entonces la dirección del búfer de esta cadena en particular se pasa a la función importada de la DLL sin copiar.

Cuando se pasa un array a la DLL, siempre se pasa la dirección del principio del búfer de datos (independientemente de la bandera AS_SERIES). La función dentro de la DLL no sabe nada sobre la bandera AS_SERIES, el array pasado es un array de longitud desconocida, y se necesita un parámetro adicional para especificar su tamaño.

Al describir el prototipo de una función importada, puede utilizar parámetros con valores por defecto.

Al importar DLLs, debe dar permiso para usarlas en las propiedades de un programa MQL específico o en la configuración general del terminal. A este respecto, en la sección Permisos presentamos el script *EnvPermissions.mq5*, que, en particular, tiene una función para leer el contenido del portapapeles del sistema Windows utilizando DLLs del sistema. Esta función se proporcionó opcionalmente: su llamada se comentó porque no sabíamos cómo trabajar con bibliotecas. Ahora, lo transferiremos a un script separado *LibClipboard.mq5*.

La ejecución del script puede pedir confirmación al usuario (ya que las DLL están desactivadas por defecto por motivos de seguridad). Si es necesario, active la opción en el cuadro de diálogo, en la pestaña con las dependencias.

Los archivos de encabezado se proporcionan en el directorio *MQL5/Include/WinApi*, que también incluye directivas *#import* para funciones del sistema muy necesarias, como la gestión del portapapeles (*openclipboard*, *GetClipboardData* y *CloseClipboard*), la gestión de la memoria (*GlobalLock* y *GlobalUnlock*), las ventanas de Windows y muchas otras. Incluiremos sólo dos archivos: *winuser.mqh* y *winbase.mqh*, que contienen las directivas de importación necesarias e, indirectamente, a través de la conexión a *windef.mqh*, las macros de términos de Windows (HANDLE y PVOID):

```
#define HANDLE  long
#define PVOID   long

#import "user32.dll"
...
int             OpenClipboard(HANDLE wnd_new_owner);
HANDLE          GetClipboardData(uint format);
int             CloseClipboard(void);
...
#import

#import "kernel32.dll"
...
PVOID           GlobalLock(HANDLE mem);
int             GlobalUnlock(HANDLE mem);
...
#import
```

Además, importamos la función *lstrcatW* de la biblioteca *kernel32.dll* porque no estamos satisfechos con su descripción en *winbase.mqh* que se proporciona de manera predeterminada: esto ofrece a la función un segundo prototipo, adecuado para pasar el valor PVOID en el primer parámetro.

```
#include <WinApi/winuser.mqh>
#include <WinApi/winbase.mqh>

#define CF_UNICODETEXT 13 // one of the standard exchange formats - Unicode text
#import "kernel32.dll"
string lstrcatW(PVOID string1, const string string2);
#import
```

La esencia del trabajo con el portapapeles es «capturar» el acceso al mismo mediante *OpenClipboard*, tras lo cual debe obtener un manejador de datos (*GetClipboardData*), convertirlo en una dirección de memoria (*GlobalLock*) y, por último, copiar los datos de la memoria del sistema a su variable (*lstrcatW*). A continuación, se liberan los recursos ocupados en orden inverso (*GlobalUnlock* y *CloseClipboard*).

```
void ReadClipboard()
{
   if(OpenClipboard(NULL))
   {
      HANDLE h = GetClipboardData(CF_UNICODETEXT);
      PVOID p = GlobalLock(h);
      if(p != 0)
      {
         const string text = lstrcatW(p, "");
         Print("Clipboard: ", text);
         GlobalUnlock(h);
      }
      CloseClipboard();
   }
}
```

Pruebe a copiar el texto en el portapapeles y, a continuación, ejecute el script: el contenido del portapapeles debería registrarse. Si el búfer contiene una imagen u otros datos que no tienen representación textual, el resultado estará vacío.

Las funciones importadas desde una DLL siguen la convención de enlace de ejecutables binarios de las funciones de la API de Windows. Para garantizar esta convención, en el texto fuente de los programas se utilizan palabras clave específicas del compilador, como, por ejemplo, *__stdcall* en C o C++. Estas reglas de enlace implican lo siguiente:

- La función de llamada (en nuestro caso, el programa MQL) debe ver el prototipo de la función llamada (importado de la DLL) con el fin de apilar correctamente los parámetros en la pila.
- La función de llamada (en nuestro caso, el programa MQL) apila los parámetros en orden inverso, de derecha a izquierda: este es el orden en que la función importada lee los parámetros que se le pasan.
- Los parámetros se pasan por valor, excepto los que se pasan explícitamente por referencia (en nuestro caso, cadenas).
- La función importada lee los parámetros que se le pasan y borra la pila.

He aquí otro ejemplo de script que utiliza una DLL: *LibWindowTree.mq5*. Su tarea es recorrer el árbol de todas las ventanas del terminal y obtener sus nombres de clase (según el registro en el sistema usando WinApi) y títulos. Por ventanas entendemos los elementos estándar de la interfaz de Windows, que también incluyen controles. Este procedimiento puede ser útil para automatizar el trabajo con el

terminal: emular la pulsación de botones en ventanas, cambiar modos que no están disponibles a través de MQL5, etc.

Para importar las funciones del sistema necesarias, incluyamos el archivo de encabezado *WinUser.mqh* que utiliza *user32.dll*.

```
#include <WinAPI/WinUser.mqh>
```

Puede obtener el nombre de la clase de ventana y su título utilizando las funciones *GetClassNameW* y *GetWindowTextW*: se llaman en la función *GetWindowData*.

```
void GetWindowData(HANDLE w, string &clazz, string &title)
{
   static ushort receiver[MAX_PATH];
   if(GetWindowTextW(w, receiver, MAX_PATH))
   {
      title = ShortArrayToString(receiver);
   }
   if(GetClassNameW(w, receiver, MAX_PATH))
   {
      clazz = ShortArrayToString(receiver);
   }
}
```

El sufijo «W» en los nombres de las funciones significa que están pensadas para cadenas con formato Unicode (2 bytes por carácter), que son las más utilizadas hoy en día (el sufijo «A» para cadenas ANSI sólo tiene sentido utilizarlo por compatibilidad con bibliotecas antiguas).

Dado un manejador inicial de una ventana de Windows, la función *TraverseUp* permite recorrer la jerarquía de sus ventanas padre: su funcionamiento se basa en la función del sistema *GetParent*. Para cada ventana encontrada, *TraverseUp* llama a *GetWindowData* y muestra el nombre y el título de la clase resultante en el registro.

```
HANDLE TraverseUp(HANDLE w)
{
   HANDLE p = 0;
   while(w != 0)
   {
      p = w;
      string clazz, title;
      GetWindowData(w, clazz, title);
      Print("'", clazz, "' '", title, "'");
      w = GetParent(w);
   }
   return p;
}
```

La función *TraverseDown* se encarga de recorrer la jerarquía en profundidad: la función *FindWindowExW* del sistema se utiliza para enumerar las ventanas secundarias.

```
HANDLE TraverseDown(const HANDLE w, const int level = 0)
{
   // request first child window (if any)
   HANDLE child = FindWindowExW(w, NULL, NULL, NULL);
   while(child)          // oop while there are child windows
   {
      string clazz, title;
      GetWindowData(child, clazz, title);
      Print(StringFormat("%*s", level * 2, ""), "'", clazz, "' '", title, "'");
      TraverseDown(child, level + 1);
      // requesting next child window
      child = FindWindowExW(w, child, NULL, NULL);
   }
   return child;
}
```

En la función *OnStart* encontramos la ventana principal del terminal recorriendo las ventanas hacia arriba desde el manejador del gráfico actual en el que se está ejecutando el script. A continuación, construimos todo el árbol de ventanas de terminal.

```
void OnStart()
{
   HANDLE h = TraverseUp(ChartGetInteger(0, CHART_WINDOW_HANDLE));
   Print("Main window handle: ", h);
   TraverseDown(h, 1);
}
```

También podemos buscar las ventanas requeridas por nombre de clase y/o título, por lo que la ventana principal podría obtenerse inmediatamente llamando a *FindWindowW*, ya que se conocen sus atributos.

```
   h = FindWindowW("MetaQuotes::MetaTrader::5.00", NULL);
```

He aquí un ejemplo de registro (fragmento):

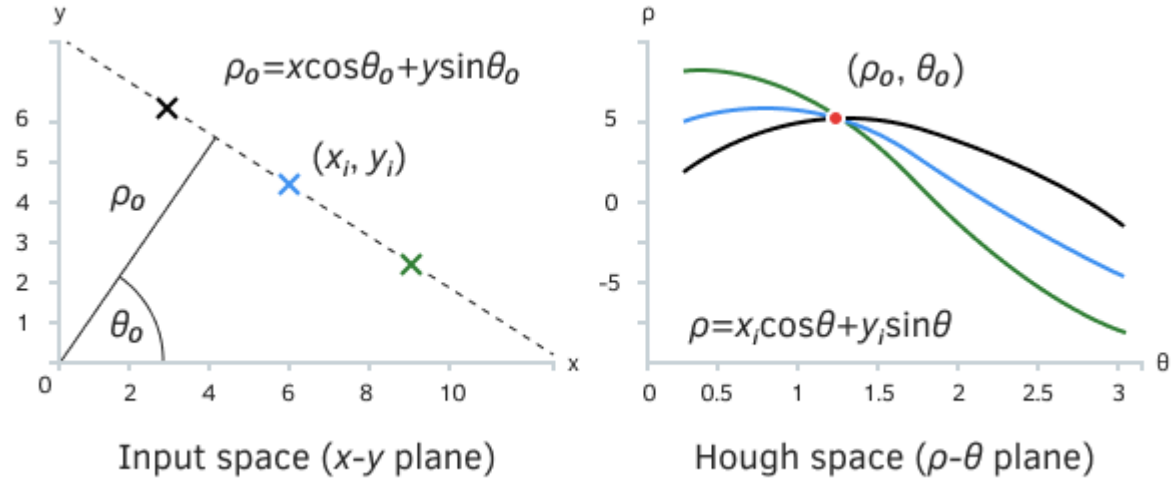
```
'AfxFrameOrView140su' ''
'Afx:000000013F110000:b:0000000000010003:0000000000000006:00000000000306BA' 'EURUSD,H
'MDIClient' ''
'MetaQuotes::MetaTrader::5.00' '12345678 - MetaQuotes-Demo: Demo Account - Hedge - ..
Main window handle: 263576
'msctls_statusbar32' 'For Help, press F1'
'AfxControlBar140su' 'Standard'
'ToolbarWindow32' 'Timeframes'
'ToolbarWindow32' 'Line Studies'
'ToolbarWindow32' 'Standard'
'AfxControlBar140su' 'Toolbox'
'Afx:000000013F110000:b:0000000000010003:0000000000000000:0000000000000000' 'Toolbox'
'AfxWnd140su' ''
'ToolbarWindow32' ''
...
'MDIClient' ''
'Afx:000000013F110000:b:0000000000010003:0000000000000006:00000000000306BA' 'EURUSD,H
'AfxFrameOrView140su' ''
'Edit' '0.00'
'Afx:000000013F110000:b:0000000000010003:0000000000000006:00000000000306BA' 'XAUUSD,D
'AfxFrameOrView140su' ''
'Edit' '0.00'
'Afx:000000013F110000:b:0000000000010003:0000000000000006:00000000000306BA' 'EURUSD,M
'AfxFrameOrView140su' ''
'Edit' '0.00'
```

7.7.5 Clases y plantillas en bibliotecas MQL5

Aunque la exportación e importación de clases y plantillas suelen estar prohibidas, el desarrollador puede eludir estas restricciones trasladando la descripción de las interfaces base abstractas al archivo de encabezado de la biblioteca y pasando punteros. Ilustremos este concepto con un ejemplo de una biblioteca que realiza una transformada de Hough de una imagen.

La transformada de Hough es un algoritmo para extraer características de una imagen comparándola con algún modelo formal (fórmula) descrito por un conjunto de parámetros.

La transformada de Hough más sencilla consiste en seleccionar líneas rectas en la imagen mediante su conversión en coordenadas polares. Con este tratamiento, las secuencias de píxeles «rellenos», dispuestos más o menos en fila, forman picos en el espacio de coordenadas polares en la intersección de un ángulo específico («theta») de la inclinación de la recta y su desplazamiento («ro») respecto al centro de coordenadas.



Transformada de Hough para rectas

Cada uno de los tres puntos en color de la imagen de la izquierda (original) deja un rastro en el espacio de coordenadas polares (derecha) porque se puede trazar un número infinito de líneas rectas a través de un punto en diferentes ángulos y perpendiculares al centro. Cada fragmento de traza se «marca» una sola vez, a excepción de la marca roja: en este punto, las tres trazas se cruzan y dan la respuesta máxima (3). De hecho, como podemos ver en la imagen original, hay una línea recta que pasa por los tres puntos. Así, los dos parámetros de la línea se revelan por el máximo en coordenadas polares.

Podemos utilizar esta transformada de Hough en gráficos de precios para resaltar líneas alternativas de compatibilidad y resistencia. Si estas líneas se suelen trazar en los extremos individuales y, de hecho, realizan un análisis de valores atípicos, entonces las líneas de la transformada de Hough pueden tener en cuenta todos los precios de *High* o de *Low*, o incluso la distribución de los volúmenes de ticks dentro de las barras. Todo esto permite obtener una estimación más razonable de los niveles.

Empecemos con el archivo de encabezado *LibHoughTransform.mqh*. Dado que una imagen abstracta suministra los datos iniciales para el análisis, definamos la plantilla de interfaz *HoughImage*.

```
template<typename T>
interface HoughImage
{
   virtual int getWidth() const;
   virtual int getHeight() const;
   virtual T get(int x, int y) const;
};
```

Todo lo que se necesita saber sobre la imagen a la hora de procesarla son sus dimensiones y el contenido de cada píxel, que, por razones de generalización, se representa mediante el tipo paramétrico T. Es evidente que, en el caso más sencillo, puede ser *int* o *double*.

Llamar al tratamiento analítico de imágenes es un poco más complicado. En la biblioteca, tenemos que describir la clase, cuyos objetos serán devueltos desde una función de fábrica especial (en forma de punteros). Es esta función la que debe exportarse de la biblioteca. Supongamos que es como sigue:

```
template<typename T>
class HoughTransformDraft
{
public:
   virtual int transform(const HoughImage<T> &image, double &result[],
      const int elements = 8) = 0;
};

HoughTransformDraft<?> *createHoughTransform() export { ... } // Problem - template!
```

Sin embargo, los tipos de plantilla y las funciones de plantilla no pueden exportarse. Por lo tanto, haremos una clase intermedia sin plantilla *HoughTransform*, en la que añadiremos un método de plantilla para el parámetro imagen. Por desgracia, los métodos de plantilla no pueden ser virtuales, y por lo tanto despacharemos manualmente las llamadas dentro del método (usando *dynamic_cast*), redirigiendo el procesamiento a una clase derivada con un método virtual.

```
class HoughTransform
{
public:
   template<typename T>
   int transform(const HoughImage<T> &image, double &result[],
      const int elements = 8)
   {
      HoughTransformConcrete<T> *ptr = dynamic_cast<HoughTransformConcrete<T> *>(&thi
      if(ptr) return ptr.extract(image, result, elements);
      return 0;
   }
};

template<typename T>
class HoughTransformConcrete: public HoughTransform
{
public:
   virtual int extract(const HoughImage<T> &image, double &result[],
      const int elements = 8) = 0;
};
```

La implementación interna de la clase *HoughTransformConcrete* se escribirá en el archivo de biblioteca *MQL5/Libraries/MQL5Book/LibHoughTransform.mq5*.

```
#property library

#include <MQL5Book/LibHoughTransform.mqh>

template<typename T>
class LinearHoughTransform: public HoughTransformConcrete<T>
{
protected:
   int size;

public:
   LinearHoughTransform(const int quants): size(quants) { }
   ...
```

Dado que vamos a recalcular los puntos de la imagen en el espacio en nuevas coordenadas polares, debe asignarse un tamaño determinado a la tarea. Aquí hablamos de una transformada de Hough discreta, ya que consideramos la imagen original como un conjunto discreto de puntos (píxeles), y acumularemos los valores de los ángulos con las perpendiculares en celdas (cuantos). Para simplificar, nos centraremos en la variante con un espacio cuadrado, donde el número de lecturas tanto en el ángulo como en la distancia al centro es igual. Este parámetro se pasa al constructor de la clase.

```
template<typename T>
class LinearHoughTransform: public HoughTransformConcrete<T>
{
protected:
   int size;
   Plain2DArray<T> data;
   Plain2DArray<double> trigonometric;

   void init()
   {
      data.allocate(size, size);
      trigonometric.allocate(2, size);
      double t, d = M_PI / size;
      int i;
      for(i = 0, t = 0; i < size; i++, t += d)
      {
         trigonometric.set(0, i, MathCos(t));
         trigonometric.set(1, i, MathSin(t));
      }
   }

public:
   LinearHoughTransform(const int quants): size(quants)
   {
      init();
   }
   ...
```

Para calcular las estadísticas de la «huella» dejada por los píxeles «llenos» en el espacio de tamaño transformado con dimensiones *size* por *size*, describimos el array *data*. La clase de plantilla auxiliar *Plain2DArray* (con parámetro de tipo T) permite la emulación de un array bidimensional de tamaños

arbitrarios. La misma clase pero con un parámetro de tipo *double* se aplica a la tabla *trigonometric* de valores precalculados de senos y cosenos de ángulos. Necesitaremos la tabla para asignar rápidamente píxeles a un nuevo espacio.

El método para detectar los parámetros de las rectas más prominentes se denomina *extract*. Toma una imagen como entrada y debe llenar el array de salida *result* con pares de parámetros encontrados de líneas rectas. En la siguiente ecuación:

```
y = a * x + b
```

el parámetro *a* (pendiente, «theta») se escribirá en los números pares del array *result*, y el parámetro **b** (sangría, «ro») se escribirán en los números impares del array. Por ejemplo, la primera línea recta más notable tras la realización del método se describe mediante la expresión:

```
y = result[0] * x + result[1];
```

Para la segunda línea, los índices aumentarán a 2 y 3, respectivamente, y así sucesivamente, hasta el número máximo de líneas solicitadas (*lines*). El tamaño del array *result* es igual al doble del número de líneas.

```
template<typename T>
class LinearHoughTransform: public HoughTransformConcrete<T>
{
   ...
   virtual int extract(const HoughImage<T> &image, double &result[],
      const int lines = 8) override
   {
      ArrayResize(result, lines * 2);
      ArrayInitialize(result, 0);
      data.zero();

      const int w = image.getWidth();
      const int h = image.getHeight();
      const double d = M_PI / size;     // 180 / 36 = 5 degrees, for example
      const double rstep = MathSqrt(w * w + h * h) / size;
      ...
```

Los bucles anidados sobre los píxeles de la imagen se organizan en el bloque de búsqueda en línea recta. Para cada punto «lleno» (distinto de cero), se realiza un bucle a través de las inclinaciones y se marcan los pares de coordenadas polares correspondientes en el espacio transformado. En este caso, simplemente llamamos al método para aumentar el contenido de la celda en el valor devuelto por el píxel *data.inc((int)r, i, v)*, pero, dependiendo de la aplicación y del tipo T, ello puede requerir un tratamiento más complejo.

```
double r, t;
int i;
for(int x = 0; x < w; x++)
{
   for(int y = 0; y < h; y++)
   {
      T v = image.get(x, y);
      if(v == (T)0) continue;

      for(i = 0, t = 0; i < size; i++, t += d) // t < Math.PI
      {
         r = (x * trigonometric.get(0, i) + y * trigonometric.get(1, i));
         r = MathRound(r / rstep); // range [-range, +range]
         r += size; // [0, +2size]
         r /= 2;

         if((int)r < 0) r = 0;
         if((int)r >= size) r = size - 1;
         if(i < 0) i = 0;
         if(i >= size) i = size - 1;

         data.inc((int)r, i, v);
      }
   }
}
...
```

En la segunda parte del método se realiza la búsqueda de máximos en el nuevo espacio y se rellena el array de salida *result*.

```
        for(i = 0; i < lines; i++)
        {
           int x, y;
           if(!findMax(x, y))
           {
              return i;
           }

           double a = 0, b = 0;
           if(MathSin(y * d) != 0)
           {
              a = -1.0 * MathCos(y * d) / MathSin(y * d);
              b = (x * 2 - size) * rstep / MathSin(y * d);
           }
           if(fabs(a) < DBL_EPSILON && fabs(b) < DBL_EPSILON)
           {
              i--;
              continue;
           }
           result[i * 2 + 0] = a;
           result[i * 2 + 1] = b;
        }

        return i;
     }
```

El método auxiliar *findMax* (véase el código fuente) escribe las coordenadas del valor máximo en el nuevo espacio en las variables *x* y *y*, sobrescribiendo además la vecindad de este lugar para no encontrarlo una y otra vez.

La clase *LinearHoughTransform* está lista, y podemos escribir una función de fábrica exportable para generar objetos.

```
HoughTransform *createHoughTransform(const int quants,
   const ENUM_DATATYPE type = TYPE_INT) export
{
   switch(type)
   {
   case TYPE_INT:
      return new LinearHoughTransform<int>(quants);
   case TYPE_DOUBLE:
      return new LinearHoughTransform<double>(quants);
   ...
   }
   return NULL;
}
```

Dado que las plantillas no están permitidas para la exportación, utilizamos la enumeración ENUM_DATATYPE en el segundo parámetro para variar el tipo de datos durante la conversión y en la representación original de la imagen.

Para probar la exportación/importación de estructuras, también describimos una estructura con metainformación sobre la transformación en una versión determinada de la biblioteca y exportamos una función que devuelve dicha estructura.

```
struct HoughInfo
{
   const int dimension; // number of parameters in the model formula
   const string about;  // verbal description
   HoughInfo(const int n, const string s): dimension(n), about(s) { }
   HoughInfo(const HoughInfo &other): dimension(other.dimension), about(other.about)
};

HoughInfo getHoughInfo() export
{
   return HoughInfo(2, "Line: y = a * x + b; a = p[0]; b = p[1];");
}
```

Diversas modificaciones de las transformadas de Hough pueden revelar no sólo líneas rectas, sino también otras construcciones que corresponden a una fórmula analítica dada (por ejemplo, círculos). Dichas modificaciones revelarán un número diferente de parámetros y tendrán un significado distinto. Disponer de una función autodocumentada puede facilitar la integración de bibliotecas (especialmente cuando hay muchas; tenga en cuenta que nuestro archivo de encabezado contiene sólo información general relacionada con cualquier biblioteca que implemente esta interfaz de transformación de Hough, y no sólo para líneas rectas).

Por supuesto, este ejemplo de exportación de una clase con un único método público es un tanto arbitrario porque sería posible exportar directamente la función de transformación. Sin embargo, en la práctica, las clases suelen contener más funcionalidades. En concreto, es fácil añadir a nuestra clase el ajuste de la sensibilidad del algoritmo, el almacenamiento de patrones ejemplares de líneas para detectar señales comprobadas en el historial, etc.

Utilicemos la biblioteca en un indicador que calcula las líneas de compatibilidad y resistencia mediante los precios *High* y *Low* en un número determinado de barras. Gracias a la transformada de Hough y a la interfaz de programación, la biblioteca permite visualizar varias de las líneas más importantes de este tipo.

El código fuente del indicador se encuentra en el archivo *MQL5/Indicators/MQL5Book/p7/LibHoughChannel.mq5*. También incluye el archivo de encabezado *LibHoughTransform.mqh*, en el que añadimos la directiva de importación.

```
#import "MQL5Book/LibHoughTransform.ex5"
HoughTransform *createHoughTransform(const int quants,
   const ENUM_DATATYPE type = TYPE_INT);
HoughInfo getHoughInfo();
#import
```

En la imagen analizada, indicamos mediante píxeles la posición de determinados tipos de precios (OHLC) en las cotizaciones. Para implementar la imagen, necesitamos describir la clase *HoughQuotes* derivada de *Hough Image<int>*.

Preveremos «pintar» píxeles de varias maneras: dentro del cuerpo de las velas, dentro de todo el rango de las velas, así como directamente en los máximos y mínimos. Todo esto se formaliza en la enumeración PRICE_LINE. Por ahora, el indicador sólo utilizará *HighHigh* y *LowLow*, pero esto puede eliminarse en la configuración.

```
class HoughQuotes: public HoughImage<int>
{
public:
   enum PRICE_LINE
   {
      HighLow = 0,    // Bar Range |High..Low|
      OpenClose = 1, // Bar Body |Open..Close|
      LowLow = 2,     // Bar Lows
      HighHigh = 3,  // Bar Highs
   };
   ...
```

En los parámetros del constructor y las variables internas, especificamos el rango de barras para el análisis. El número de barras *size* determina el tamaño horizontal de la imagen. Para simplificar, utilizaremos el mismo número de lecturas en vertical. Por lo tanto, el paso de discretización de precios (*step*) es igual al rango real de precios (*pp*) de las barras *size* dividido por *size*. Para la variable *base*, calculamos el límite inferior de los precios que son objeto de consideración en las barras indicadas. Esta variable será necesaria para enlazar la construcción de líneas basadas en los parámetros encontrados de la transformada de Hough.

```
protected:
   int size;
   int offset;
   int step;
   double base;
   PRICE_LINE type;

public:
   HoughQuotes(int startbar, int barcount, PRICE_LINE price)
   {
      offset = startbar;
      size = barcount;
      type = price;
      int hh = iHighest(NULL, 0, MODE_HIGH, size, startbar);
      int ll = iLowest(NULL, 0, MODE_LOW, size, startbar);
      int pp = (int)((iHigh(NULL, 0, hh) - iLow(NULL, 0, ll)) / _Point);
      step = pp / size;
      base = iLow(NULL, 0, ll);
   }
   ...
```

Recordemos que la interfaz *HoughImage* requiere la implementación de 3 métodos: *getWidth*, *getHeight* y *get*. Los dos primeros son fáciles.

```
virtual int getWidth() const override
{
   return size;
}

virtual int getHeight() const override
{
   return size;
}
```

El método *get* para obtener «píxeles» basados en cotizaciones devuelve 1 si el punto especificado cae dentro del rango de barras o celdas, según el método de cálculo seleccionado de PRICE_LINE. En caso contrario, se devuelve 0. Este método puede mejorarse significativamente mediante la evaluación de fractales, el aumento constante de los extremos o los precios «redondos» con un mayor peso (grasa de píxel).

```
   virtual int get(int x, int y) const override
   {
      if(offset + x >= iBars(NULL, 0)) return 0;

      const double price = convert(y);
      if(type == HighLow)
      {
         if(price >= iLow(NULL, 0, offset + x) && price <= iHigh(NULL, 0, offset + x)
         {
            return 1;
         }
      }
      else if(type == OpenClose)
      {
         if(price >= fmin(iOpen(NULL, 0, offset + x), iClose(NULL, 0, offset + x))
         && price <= fmax(iOpen(NULL, 0, offset + x), iClose(NULL, 0, offset + x)))
         {
            return 1;
         }
      }
      else if(type == LowLow)
      {
         if(iLow(NULL, 0, offset + x) >= price - step * _Point / 2
         && iLow(NULL, 0, offset + x) <= price + step * _Point / 2)
         {
            return 1;
         }
      }
      else if(type == HighHigh)
      {
         if(iHigh(NULL, 0, offset + x) >= price - step * _Point / 2
         && iHigh(NULL, 0, offset + x) <= price + step * _Point / 2)
         {
            return 1;
         }
      }
      return 0;
   }
```

El método auxiliar *convert* proporciona un recálculo de las coordenadas y del píxel a valores de precio.

```
   double convert(const double y) const
   {
      return base + y * step * _Point;
   }
};
```

Ahora todo está listo para escribir la parte técnica del indicador. En primer lugar, vamos a declarar tres variables de entrada para seleccionar el fragmento que se va a analizar y el número de líneas. Todas las líneas se identificarán con un prefijo común.

```
input int BarOffset = 0;
input int BarCount = 21;
input int MaxLines = 3;

const string Prefix = "HoughChannel-";
```

El objeto que proporciona el servicio de transformación se describirá como global: aquí es donde se llama a la función de fábrica *createHoughTransform* desde la biblioteca.

```
HoughTransform *ht = createHoughTransform(BarCount);
```

En la función *OnInit*, nos limitamos a registrar la descripción de la biblioteca utilizando la segunda función importada *getHoughInfo*.

```
int OnInit()
{
   HoughInfo info = getHoughInfo();
   Print(info.dimension, " per ", info.about);
   return INIT_SUCCEEDED;
}
```

Realizaremos el cálculo en *OnCalculate* una vez, en la apertura de la barra.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
   static datetime now = 0;
   if(now != iTime(NULL, 0, 0))
   {
      ... // see the next block
      now = iTime(NULL, 0, 0);
   }
   return rates_total;
}
```

El cálculo de la transformación propiamente dicho se ejecuta dos veces en un par de imágenes (*highs* y *lows*) formadas por distintos tipos de precios. En este caso, el trabajo lo realiza secuencialmente el mismo objeto *ht*. Si la detección de líneas rectas ha sido satisfactoria, las mostramos en el gráfico mediante la función *DrawLine*. Dado que las líneas se enumeran en el array de resultados en orden descendente de importancia, se les asigna un peso decreciente.

```
      HoughQuotes highs(BarOffset, BarCount, HoughQuotes::HighHigh);
      HoughQuotes lows(BarOffset, BarCount, HoughQuotes::LowLow);
      static double result[];
      int n;
      n = ht.transform(highs, result, fmin(MaxLines, 5));
      if(n)
      {
         for(int i = 0; i < n; ++i)
         {
            DrawLine(highs, Prefix + "Highs-" + (string)i,
               result[i * 2 + 0], result[i * 2 + 1], clrBlue, 5 - i);
         }
      }
      n = ht.transform(lows, result, fmin(MaxLines, 5));
      if(n)
      {
         for(int i = 0; i < n; ++i)
         {
            DrawLine(lows, Prefix + "Lows-" + (string)i,
               result[i * 2 + 0], result[i * 2 + 1], clrRed, 5 - i);
         }
      }
```

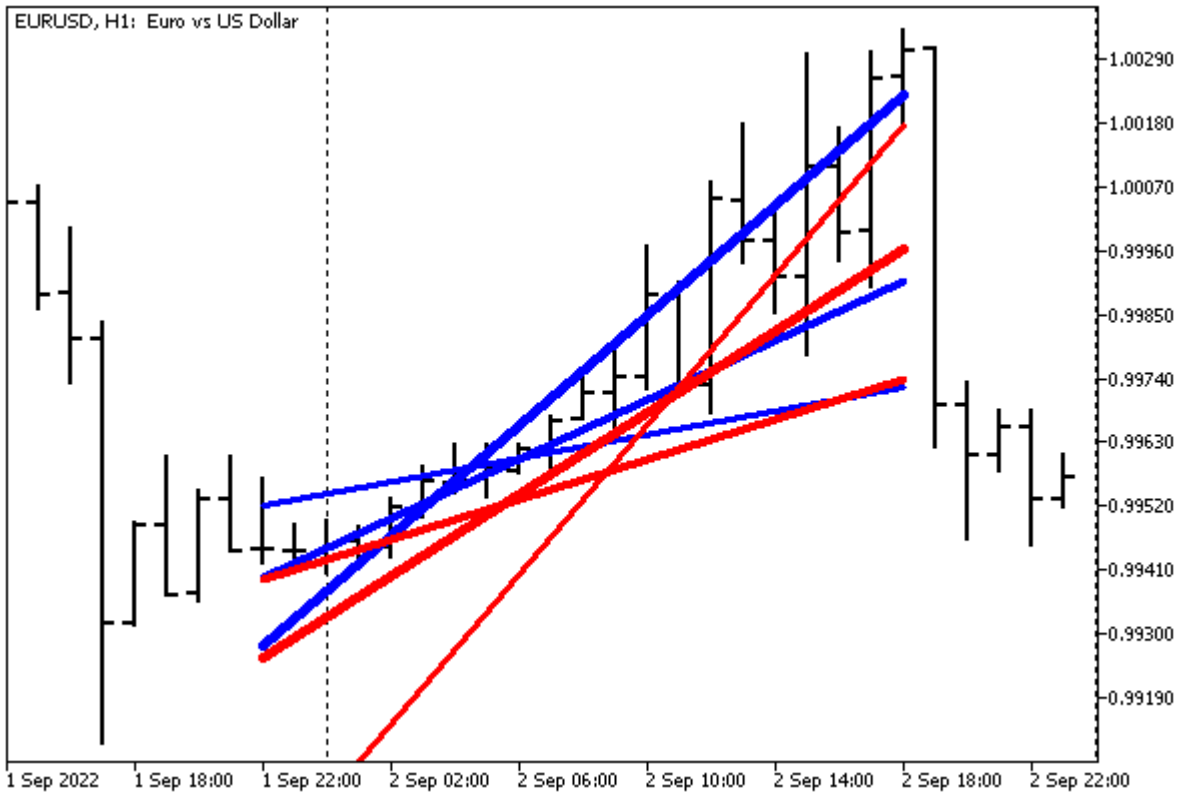
La función *DrawLine* se basa en objetos gráficos de tendencia (OBJ_TREND, véase el código fuente).

Al desinicializar el indicador, borramos las líneas y el objeto analítico.

```
void OnDeinit(const int)
{
   AutoPtr<HoughTransform> destructor(ht);
   ObjectsDeleteAll(0, Prefix);
}
```

Antes de probar un nuevo desarrollo, no olvide compilar tanto la biblioteca como el indicador.

La ejecución del indicador con la configuración predeterminada da como resultado algo como esto:



Indicador con líneas principales para precios altos/bajos basado en la biblioteca de transformadas de Hough

En nuestro caso, la prueba ha sido un éxito. Pero, ¿y si necesita depurar la biblioteca? No existen herramientas integradas para ello, por lo que se puede utilizar el siguiente truco. La prueba del código fuente de la biblioteca se compila condicionalmente en una versión de depuración del producto, y éste se prueba con la biblioteca creada. Consideremos el ejemplo de nuestro indicador.

Vamos a proporcionar la macro LIB_HOUGH_IMPL_DEBUG para permitir la integración de la fuente de la biblioteca directamente en el indicador. La macro debe colocarse antes de incluir el archivo de encabezado.

```
#define LIB_HOUGH_IMPL_DEBUG
#include <MQL5Book/LibHoughTransform.mqh>
```

En el propio archivo de encabezado, superpondremos el bloque de importación de la copia binaria independiente de la biblioteca con instrucciones de compilación condicional del preprocesador. Cuando la macro esté activada, se ejecutará otra rama, con la sentencia *#include*.

```
#ifdef LIB_HOUGH_IMPL_DEBUG
#include "../../Libraries/MQL5Book/LibHoughTransform.mq5"
#else
#import "MQL5Book/LibHoughTransform.ex5"
HoughTransform *createHoughTransform(const int quants,
   const ENUM_DATATYPE type = TYPE_INT);
HoughInfo getHoughInfo();
#import
#endif
```

En el archivo fuente de la biblioteca *LibHoughTransform.mq5*, dentro de la función *getHoughInfo*, añadimos salida al registro de información sobre el método de compilación, dependiendo de si la macro está activada o desactivada.

```
HoughInfo getHoughInfo() export
{
#ifdef LIB_HOUGH_IMPL_DEBUG
   Print("inline library (debug)");
#else
   Print("standalone library (production)");
#endif
   return HoughInfo(2, "Line: y = a * x + b; a = p[0]; b = p[1];");
}
```

Si en el código del indicador, en el archivo *LibHoughChannel.mq5*, elimina el comentario de la instrucción *#define LIB_HOUGH_IMPL_DEBUG*, puede probar el análisis de imagen paso a paso.

7.7.6 Importar funciones de bibliotecas.NET

MQL5 ofrece un servicio especial para trabajar con funciones de bibliotecas.NET: puede simplemente importar la propia DLL sin especificar determinadas funciones. MetaEditor importa automáticamente todas las funciones con las que se puede trabajar:

- Plain Old Data (POD) - estructuras que sólo contienen tipos de datos simples;
- Funciones estáticas públicas cuyos parámetros sólo utilizan tipos y estructuras POD simples o sus arrays.

Lamentablemente, por el momento no es posible ver los prototipos de función tal y como los reconoce MetaEditor.

Por ejemplo, tenemos el siguiente código C# de la función *Inc* de la clase *TestClass* en la biblioteca *TestLib.dll*:

```
public class TestClass
{
   public static void Inc(ref int x)
   {
      x++;
   }
}
```

Luego, para importarlo y llamarlo, basta con escribir:

```
#import "TestLib.dll"

void OnStart()
{
   int x = 1;
   TestClass::Inc(x);
   Print(x);
}
```

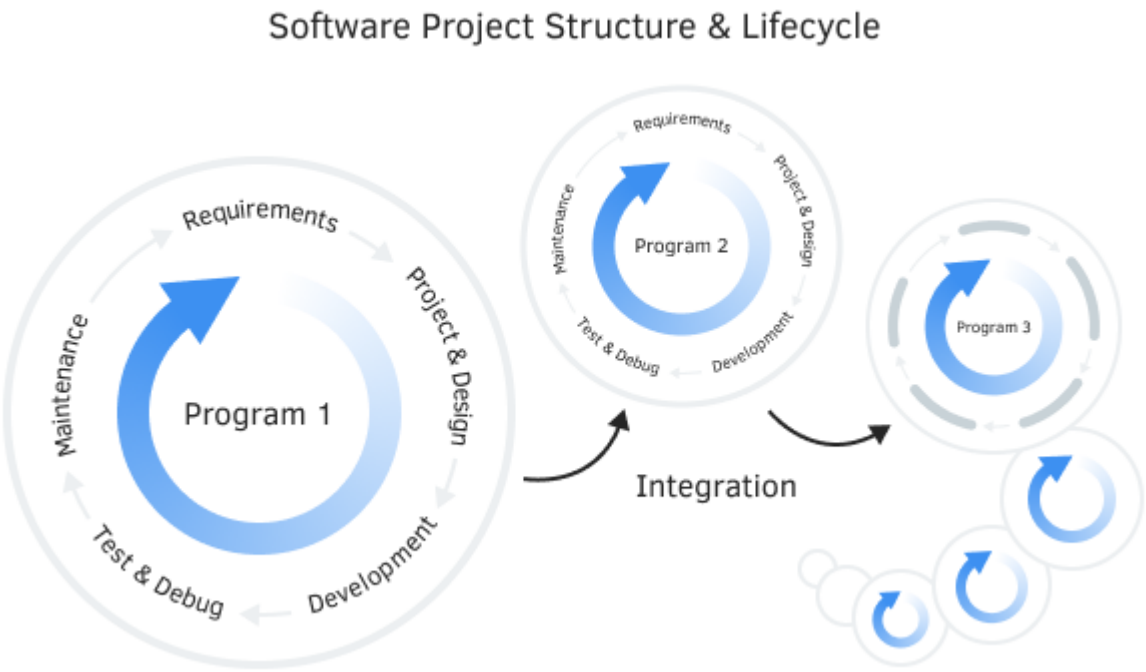
Tras la ejecución, el script devolverá el valor 2.

7.8 Proyectos

Los productos de software, por regla general, se desarrollan dentro del ciclo de vida estándar:

- Recopilación y adición de requisitos
- Diseño
- Desarrollo
- Simulación
- Explotación

Como resultado de la mejora y ampliación constantes de la funcionalidad, suele ser necesario sistematizar los archivos fuente, los recursos y las bibliotecas de terceros (aquí nos referimos no sólo a las bibliotecas en formato binario, sino, en un sentido más general, a cualquier conjunto de archivos, por ejemplo, los encabezados). Es más, los programas individuales se integran en un producto común que encarna una idea aplicada.



Estructura y ciclo de vida del proyecto

Por ejemplo, cuando se desarrolla un robot de trading, a menudo es necesario conectar indicadores ya hechos o personalizados, el uso de algoritmos externos de aprendizaje automático implica escribir un script para exportar datos de cotización y un script para reimportar modelos entrenados, y los programas relacionados con el intercambio de datos a través de Internet (por ejemplo, señales de trading) pueden requerir un servidor web y su configuración en otros lenguajes de programación, al menos para depurar y probar, si no para desplegar un servicio público.

Todo el conjunto de varios productos interrelacionados, junto con sus «dependencias» (es decir, los recursos y bibliotecas utilizados, escritos independientemente o tomados de fuentes de terceros), forman un proyecto de software.

Cuando un programa supera cierto tamaño, su desarrollo cómodo y eficaz resulta difícil sin herramientas especiales de gestión de proyectos. Esto se aplica plenamente a los programas basados en MQL5, ya que muchos operadores utilizan sistemas de trading complejos.

MetaEditor admite el concepto de proyectos de forma similar a otros paquetes de software. Actualmente, esta funcionalidad está al principio de su desarrollo, y para cuando se publique el libro, probablemente cambiará.

Cuando trabaje con proyectos en MQL5, tenga en cuenta que el término «proyecto» en la plataforma se utiliza para dos entidades diferentes:

- Proyecto local en forma de archivo mqproj
- Carpetas en el almacenamiento en la nube MQL5

Un proyecto local permite sistematizar y reunir toda la información sobre códigos fuente, recursos y configuraciones necesarias para construir un determinado programa MQL. Un proyecto de este tipo solo está en su ordenador y puede hacer referencia a archivos de diferentes carpetas.

El archivo con la extensión *mqproj* tiene un formato de texto JSON (JavaScript Object Notation) ampliamente utilizado y universal. Es cómodo, sencillo y muy adecuado para describir datos de cualquier área temática: toda la información se agrupa en objetos o arrays con propiedades nombradas, con compatibilidad para valores de distintos tipos. Todo esto hace que JSON sea conceptualmente muy cercano a los lenguajes de programación orientada a objetos (POO); también proviene de JavaScript orientado a objetos, como se puede adivinar fácilmente por el nombre.

El almacenamiento en la nube funciona sobre la base de un sistema de control de versiones y el trabajo colectivo en un software llamado SVN (Subversion). En este caso, un proyecto es una carpeta de nivel superior dentro del directorio local *MQL5/Shared Projects*, a la que se asigna otra carpeta que tiene el mismo nombre pero que se encuentra en el servidor MQL5 Storage. Dentro de una carpeta de proyecto, puede organizar una jerarquía de subcarpetas. Como su nombre indica, los proyectos en red pueden compartirse con otros desarrolladores y, en general, hacerse públicos (el contenido puede ser descargado por cualquier persona registrada en mql5.com).

El sistema proporciona sincronización bajo demanda (mediante comandos de usuario especiales) entre la imagen de la carpeta en la nube y en la unidad local, y viceversa. Puede tanto «extraer» los cambios de proyectos de otras personas a su ordenador, como «empujar» sus ediciones a la nube. Se pueden sincronizar tanto la imagen completa de la carpeta como archivos selectivos, incluidos, por supuesto, archivos mq5, archivos de encabezado mqh, multimedia, ajustes (archivos set), así como archivos mqproj. Para obtener más información sobre el almacenamiento en la nube, lea la documentación de MetaEditor y los sistemas SVN.

> Es importante señalar que la existencia de un archivo mqproj no implica la creación de ningún proyecto en la nube sobre su base, del mismo modo que la creación de una carpeta compartida no obliga a utilizar un proyecto mqproj.

En el momento de escribir esto, un archivo mqproj sólo puede describir la estructura de un programa, no de varios. Sin embargo, dado que este requisito es habitual cuando se desarrollan proyectos complejos, es probable que esta funcionalidad se añada a MetaEditor en el futuro.

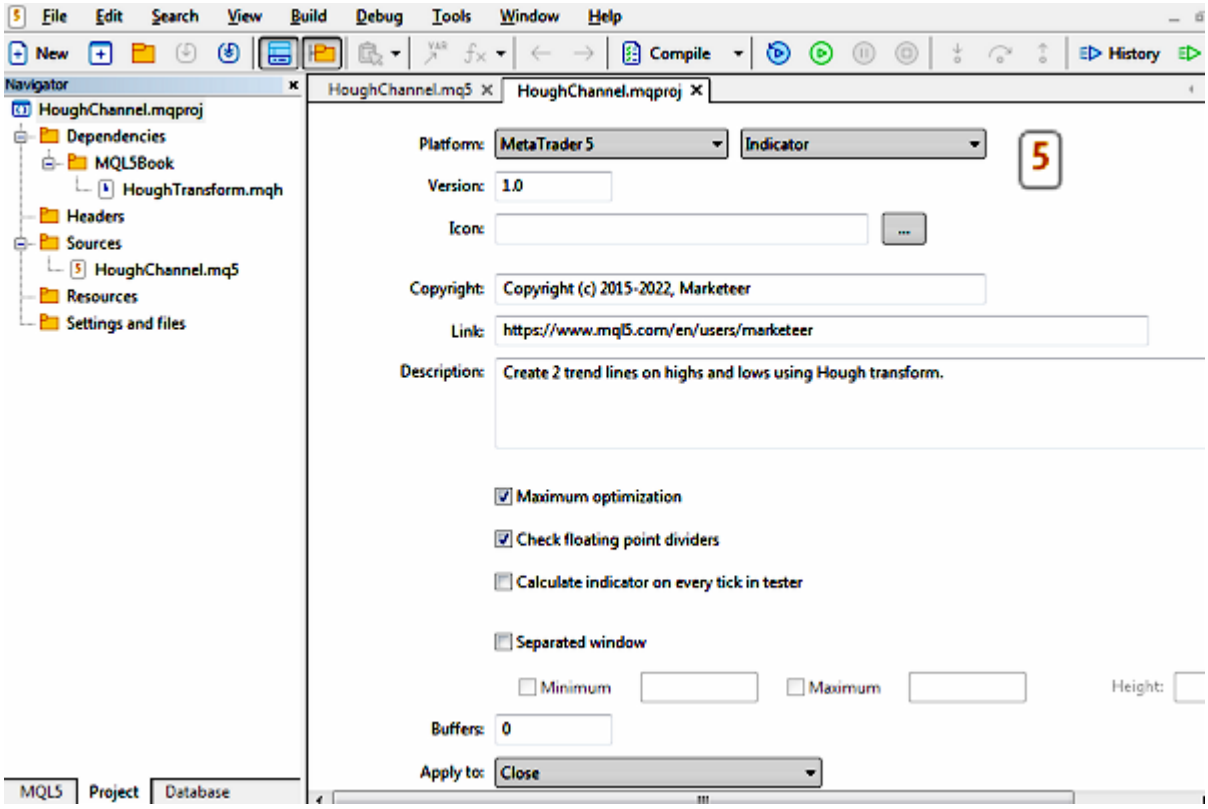
En este capítulo describiremos las principales funciones para crear y organizar proyectos mqproj y daremos una serie de ejemplos.

7.8.1 Normas generales para trabajar con proyectos locales

Se puede crear un proyecto local (archivo mqproj) desde el menú principal de MetaEditor o desde el menú contextual *Navegador* utilizando los comandos *Nuevo proyecto* o *Nuevo proyecto a partir de un archivo fuente*. En este último caso, primero hay que seleccionar el archivo en *Navegador* o elegirlo en el cuadro de diálogo *Abrir*. Como resultado, el archivo mq5 especificado se incluirá inmediatamente en el proyecto. El primero de los comandos mencionados lanza el Asistente MQL, en el que se debe seleccionar el tipo de programa o una opción de proyecto vacío (los archivos fuente se pueden añadir

más tarde). El tipo de programa MQL para un proyecto se elige siguiendo los pasos habituales del Asistente.

El proyecto contiene varias secciones lógicas que se asemejan a un árbol (jerarquía) con todos los componentes. Se muestran en el panel izquierdo de *Navigator*, en una pestaña separada *Proyecto*.



Propiedades del proyecto indicador y navegador

Inmediatamente después de crear el proyecto, o posteriormente haciendo doble clic en la raíz del árbol, se abre en la parte derecha de la ventana un panel para configurar las propiedades del programa MQL. El conjunto de propiedades varía en función del tipo de programa.

La mayoría de las propiedades corresponden a las directivas *#property* del código fuente. Estas propiedades tienen prioridad: si las especifica tanto en el proyecto como en el código fuente, se utilizarán los valores del proyecto.

Algunos desarrolladores prefieren establecer las propiedades de forma interactiva en un cuadro de diálogo en lugar de codificarlas en el código fuente. Además, puede utilizar el mismo archivo mq5 en diferentes proyectos y crear versiones de un programa MQL con diferentes configuraciones (sin cambiar el código fuente).

Algunas propiedades sólo están disponibles en un proyecto. Esto incluye, por ejemplo, activar/desactivar las optimizaciones de compilación y las comprobaciones integradas de división por cero.

Durante la compilación del proyecto, el sistema analiza automáticamente las dependencias, es decir, los archivos de encabezado incluidos, los recursos, etc. Las dependencias aparecen en distintas ramas de la jerarquía del proyecto. En concreto, los archivos de encabezado de las carpetas estándar MQL5/Include incluidos en las directivas *#include* utilizando corchetes angulares (<*filename*>) caen dentro de *Dependencias*, y los archivos de encabezado personalizados incluidos con comillas dobles (*#include "filename"*), en la sección *Encabezados*.

Además, el usuario puede añadir al proyecto archivos relacionados con el producto de software acabado y que pueden ser necesarios para su funcionamiento normal o su demostración (por ejemplo, archivos con modelos de redes neuronales entrenados), pero que no están directamente incrustados en el código fuente. Para ello, puede utilizar la rama *Ajustes y archivos*. Su menú contextual contiene comandos para añadir un único archivo o un directorio completo al proyecto.

En particular, consideraremos más adelante ejemplos de proyectos que incluirán no sólo programas MQL cliente, sino también la parte servidor.

Los comandos *Nuevo archivo* y *Nueva carpeta* añaden un nuevo elemento a la carpeta con el archivo del proyecto: dichos elementos se buscan siempre en relación con el propio proyecto (en el archivo mqproj se marcan con la propiedad *relative_to_project* igual a *true*; véase más adelante).

Los comandos *Añadir un archivo existente* y *Añadir una carpeta existente* seleccionan uno o más elementos de la estructura de directorios existente dentro de la carpeta MQL5, y estos elementos dentro del archivo mqproj son referenciados en relación al MQL5 raíz (la propiedad *relative_to_project* es igual a *false*).

La propiedad *relative_to_project* es solo una de las pocas definidas por los desarrolladores de MetaTrader 5 para representar un proyecto en formato JSON. Recordemos que, como resultado de la edición del proyecto (jerarquía y propiedades), se forma un archivo mqproj con formato JSON.

Este es el aspecto de ese archivo para el proyecto de la imagen anterior:

```
{
  "platform"    :"mt5",
  "program_type":"indicator",
  "copyright"   :"Copyright (c) 2015-2022, Marketeer",
  "link"        :"https:\/\/www.mql5.com\/en\/users\/marketeer",
  "version"     :"1.0",
  "description" :"Create 2 trend lines on highs and lows using Hough transform.",
  "optimize"    :"1",
  "fpzerocheck" :"1",
  "tester_no_cache":"0",
  "tester_everytick_calculate":"0",
  "unicode_character_set":"0",
  "static_libraries":"0",

  "indicator":
  {
    "window":"0"
  },

  "files":
  [
    {
      "path":"HoughChannel.mq5",
      "compile":true,
      "relative_to_project":true
    },
    {
      "path":"MQL5\\Include\\MQL5Book\\HoughTransform.mqh",
      "compile":false,
      "relative_to_project":false
    }
  ]
}
```

Hablaremos de las características técnicas del formato JSON con más detalle en las siguientes secciones, ya que lo aplicaremos en nuestros proyectos de demostración.

> Es importante tener en cuenta que todos los archivos a los que hace referencia el proyecto no se almacenan dentro del archivo mqproj, por lo que copiarlo a una nueva ubicación o mover sólo el archivo del proyecto a otro ordenador no lo restaurará. Para poder migrar un proyecto, configure un proyecto compartido para él y suba todo el contenido del proyecto a la nube. No obstante, esto puede requerir una reorganización de la estructura del sistema de archivos local, ya que todos los componentes deben estar dentro de la carpeta compartida del proyecto, mientras que el formato mqproj no lo requiere.

7.8.2 Plan de proyecto de un servicio web para copiar señales y operaciones de trading

Como proyecto de demostración integral, que desarrollaremos a lo largo de este capítulo, tomaremos un producto sencillo, pero al mismo tiempo bastante avanzado tecnológicamente: un sistema de trading de copias cliente-servidor. La parte cliente serán programas MQL que se comunican con la parte central mediante tecnología de sockets. Teniendo en cuenta que MQL5 solo permite trabajar con

sockets de cliente, tendrá que elegir una plataforma alternativa para el servidor de sockets (se ofrece más información al respecto más adelante). Así pues, el proyecto requerirá la simbiosis de varias tecnologías diferentes y el uso de muchas secciones de la API de MQL5 que ya hemos estudiado, incluidos los códigos de aplicación desarrollados sobre su base.

Gracias a la arquitectura cliente-servidor basada en sockets, el sistema puede utilizarse en distintos escenarios:

- Para facilitar la copia de operaciones de trading entre terminales de un mismo ordenador;
- Para establecer un canal de comunicación privado (personal) entre terminales de distintos ordenadores, incluso no sólo en la red local, sino también a través de Internet;
- Para organizar un servicio público de señales abierto o cerrado que requiera inscripción;
- Para supervisar el trading;
- Para administrar su propia cuenta a distancia.

En todos los casos, los programas cliente actuarán con 2 roles: un publicador (editor, emisor) y un suscriptor (receptor) de datos.

No inventaremos nuestro propio protocolo de red, sino que utilizaremos el popular estándar WebSocket. Su implementación cliente está integrada en todos los navegadores, y tendremos que repetirla (con mayor o menor grado de completitud) en MQL5. Por supuesto, la compatibilidad con WebSocket también está disponible para los servidores web más populares. Por lo tanto, en cualquier caso, nuestros desarrollos no solo pueden adaptarse a otros servidores (si a alguien le conviene), sino que también pueden integrarse con sitios conocidos que prestan servicios web similares. Aquí de lo que se trata es de seguir estrictamente la especificación de su API, construida sobre WebSockets.

A la hora de desarrollar sistemas de software más complejos que un programa independiente, es importante elaborar un plan de acción y, posiblemente, incluso diseñar un proyecto técnico que incluya la estructura de los módulos, su interacción y la secuencia de codificación.

Así que nuestro plan incluye:

1. Análisis teórico del protocolo WebSocket;
2. Selección e instalación de un servidor web con la implementación de un servidor WebSocket;
3. Creación de un servidor eco sencillo (que envía una copia de los mensajes entrantes al cliente) para familiarizarse con la tecnología;
4. Creación de una sencilla página web del lado de cliente para probar la funcionalidad del servidor eco desde un navegador;
5. Creación de un servidor de chat sencillo que envíe mensajes a todos los clientes conectados, y una página web de prueba para el mismo;
6. Creación de un servidor de mensajería entre proveedores y abonados identificables, y de un cliente web de prueba para el mismo;
7. Diseño e implementación de WebSockets en MQL5;
8. Creación de un script sencillo como cliente para un servidor eco;
9. Creación de un Asesor Experto simple como cliente de servidor de chat;
10. Por último, creación de un copiador de operaciones de trading en MQL5 que actuará como proveedor de información (monitor de estado y cambios de la cuenta) y como consumidor de información (reproduciendo operaciones de trading), según la configuración.

Pero antes de empezar a aplicar el plan, tenemos que instalar un servidor web.

7.8.3 Servidor web basado en Nodejs

Para organizar la parte del servidor de nuestros proyectos, necesitamos un servidor web. Utilizaremos el nodejs más ligero y tecnológicamente avanzado. Los scripts del lado del servidor pueden escribirse en JavaScript, que es el mismo lenguaje utilizado en los navegadores para las páginas web interactivas. Esto es práctico desde el punto de vista de la escritura unificada de las partes cliente y servidor del sistema; la parte cliente de cualquier servicio web, por regla general, se requiere tarde o temprano, por ejemplo, para la administración, el registro y la visualización de hermosas estadísticas sobre el uso del servicio.

Quienes conocen MQL5 prácticamente conocen JavaScript, así que tengan fe en sí mismos. Las principales diferencias se comentan en la barra lateral.

MQL5 vs JavaScript

JavaScript es un lenguaje interpretado, a diferencia del MQL5 compilado. Para nosotros, como desarrolladores, esto nos facilita la vida porque no necesitamos una fase de compilación separada para obtener un programa que funcione. No hay que preocuparse por la eficacia de JavaScript: todos los tiempos de ejecución de JavaScript utilizan la compilación JIT (just-in-time) de JavaScript bajo demanda, es decir, la primera vez que se accede a un módulo. Este proceso se produce automáticamente, de forma implícita, una vez por sesión, tras lo cual el script se ejecuta de forma compilada.

MQL5 se refiere a lenguajes con tipado estático, es decir, cuando describimos variables, debemos especificar explícitamente su tipo, y el compilador controla la compatibilidad de tipos. Por el contrario, JavaScript es un lenguaje de tipado dinámico: el tipo de una variable viene determinado por el valor que pongamos en ella y puede cambiar durante la vida de la variable. Esto proporciona flexibilidad, pero exige cautela para evitar errores imprevistos.

JavaScript es, en cierto sentido, un lenguaje más orientado a objetos que MQL5, porque casi todas las entidades en él son objetos. Por ejemplo, una función también es un objeto, y una clase, como un descriptor de las propiedades de los objetos, también es un objeto (de un prototipo).

El propio JavaScript «recoge basura», es decir, libera la memoria asignada por el programa de aplicación para los objetos. En MQL5 tenemos que proporcionar la llamada oportuna de *delete* para objetos dinámicos.

La sintaxis de JavaScript contiene muchas «abreviaturas» prácticas para escribir construcciones que en MQL5 tienen que implementarse de una manera más larga. Por ejemplo, para pasar un parámetro que apunta a otra función a una determinada función en MQL5, necesitamos describir el tipo de dicho puntero utilizando typedef, definir por separado una función que coincida con este prototipo y, solo entonces, pasar su identificador como parámetro. En JavaScript, puede definir la función a la que apunta (¡en su totalidad!) directamente en la lista de argumentos en lugar de un parámetro puntero.

Si es desarrollador web o ya está familiarizado con nodejs, puede saltarse los pasos de instalación y configuración.

Puede descargar nodejs desde el sitio oficial nodejs.org. La instalación está disponible en diferentes versiones, por ejemplo, utilizando un instalador o descomprimiendo un archivo. Como resultado de la instalación, recibirá un archivo ejecutable en el directorio especificado *node.exe* y varios archivos y carpetas de apoyo.

Si el instalador no añadió nodejs a la ruta del sistema, esto se puede hacer para el usuario actual de Windows ejecutando el siguiente comando en la carpeta donde nodejs está instalado (donde se encuentra el archivo *node.exe*):

```
setx PATH "%CD%"
```

Otra alternativa es editar las variables de entorno de Windows desde el cuadro de diálogo de propiedades del sistema (*Computer -> Properties -> Extra options -> Environment Variables*; el tipo de cuadro de diálogo específico depende de la versión del sistema operativo). En cualquier caso, de esta forma nos aseguraremos la capacidad de ejecutar nodejs desde cualquier carpeta del ordenador, lo que nos será útil en el futuro.

Puede comprobar la salud de nodejs ejecutando los siguientes comandos (en la línea de comandos de Windows):

```
node –v
npm version
```

El primer comando muestra la versión de nodejs, y el segundo muestra la versión de un importante servicio integrado en nodejs: el administrador de paquetes *npm*.

Un paquete es un módulo listo para usar que añade una funcionalidad específica a nodejs. Por sí mismo, nodejs es muy pequeño, y sin paquetes, requeriría mucha codificación rutinaria.

Los paquetes más solicitados se almacenan en un repositorio centralizado en la web y pueden descargarse e instalarse en una copia específica de nodejs o globalmente (para todas las copias de nodejs si hay varias en la máquina). La instalación de un paquete en una copia específica se realiza con el siguiente comando:

```
npm install <package name>
```

Ejecútelo en la carpeta donde se instaló nodejs. Este comando colocará el paquete localmente y no afectará a otras copias de nodejs que ya existan o que puedan aparecer en el ordenador más adelante, con ediciones inesperadas.

Nosotros, en concreto, necesitamos el paquete ws, que implementa el protocolo WebSocket. Es decir, hay que ejecutar el comando:

```
npm install ws
```

y espere a que se complete el proceso. Como resultado, la carpeta *<nodejs_install_path>/node_modules/* debería contener una nueva subcarpeta *ws* con el contenido necesario (puede consultar el archivo README.md con la descripción del paquete para asegurarse de que se trata de una librería de protocolo WebSocket).

El paquete contiene implementaciones tanto del servidor como del cliente. Pero en lugar de este último, escribiremos el nuestro propio en MQL5.

Toda la funcionalidad del servidor nodejs se concentra en la carpeta */node_modules*. Se puede comparar en cuanto a propósito con una carpeta estándar *MQL5/Include* de MetaTrader 5. Al escribir programas de aplicación en JavaScript, incluiremos o «importaremos» los módulos necesarios de una manera especial, por analogía con la inclusión de archivos de encabezado mqh utilizando la directiva *#include* en MQL5.

7.8.4 Fundamentos teóricos del protocolo WebSockets

El protocolo WebSocket se basa en las conexiones de red TCP/IP, que se caracterizan por una dirección IP (o un nombre de dominio que la sustituya) y un número de puerto. El protocolo HTTP/HTTPS, con el que ya hemos practicado en el capítulo sobre funciones de red, se basa en el mismo principio. Allí, los números de puerto estándar eran 80 (para conexiones inseguras) y 443 (para conexiones seguras). No existe un número de puerto específico para WebSocket, por lo que los proveedores de servicios web pueden elegir cualquier número disponible. Todos nuestros ejemplos utilizarán el puerto 9000.

Al especificar URL como prefijos de protocolo WebSocket, utilizamos *ws* (para conexiones no seguras) y *wss* (para conexiones seguras).

El formato WebSocket es más eficiente en términos de transferencia de datos que HTTP, ya que utiliza muchos menos datos de control.

El establecimiento de la conexión inicial para un servicio WebSocket repite completamente una solicitud de página web HTTP/HTTPS: es necesario enviar una solicitud GET con encabezados especialmente preparados. Una característica de estos encabezados es la presencia de las líneas

```
Connection: Upgrade
Upgrade: websocket
```

así como algunas líneas adicionales que informan de la versión del protocolo WebSocket y cadenas especiales generadas aleatoriamente. Las claves que intervienen en el procedimiento de «handshaking» entre el cliente y el servidor.

```
Sec-WebSocket-Key: ...
Sec-WebSocket-Version: 13
```

En la práctica, este «handshake» implica que el servidor comprueba la disponibilidad de las opciones solicitadas por el cliente y, en respuesta, con encabezados HTTP estándar, confirma el cambio al modo WebSocket o lo rechaza. La razón más simple para el rechazo puede ser si usted está tratando de conectarse a través de WebSockets a un servidor web simple donde el servidor WebSocket no se proporciona o la versión requerida no es compatible.

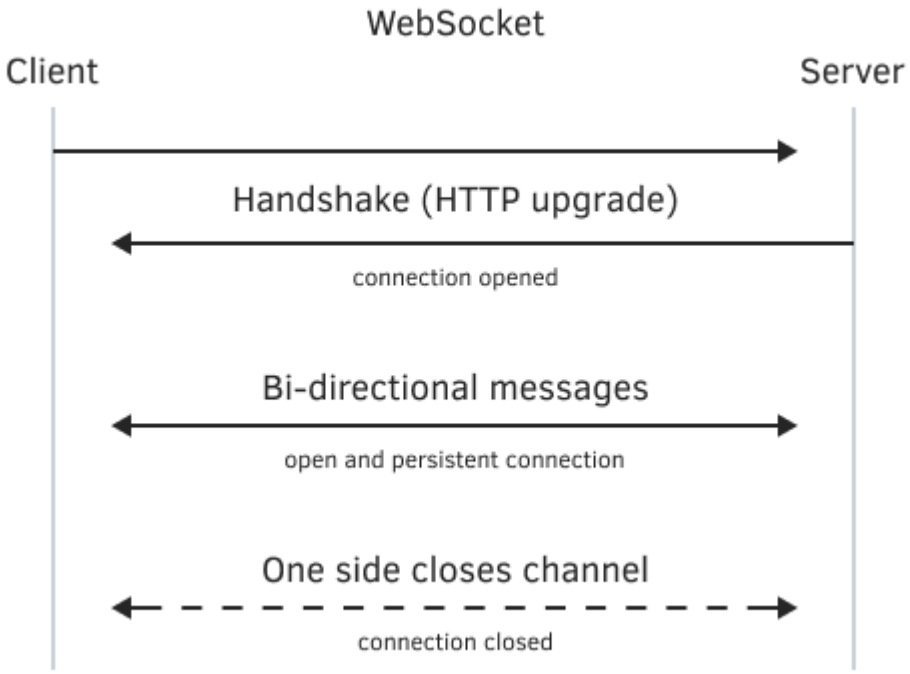
La versión actual del protocolo WebSockets se conoce con el nombre simbólico Hybi y el número 13. Una versión anterior y más sencilla llamada Hixie puede ser útil para la compatibilidad con versiones anteriores. En lo que sigue, utilizaremos solo Hybi, aunque también se incluye una implementación de Hixie.

Una conexión correcta se indica mediante los siguientes encabezados HTTP en la respuesta del servidor:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: ...
```

El servidor calcula y rellena el campo Sec-WebSocket-Accept basándose en la Sec-WebSocket-Key para confirmar el cumplimiento del protocolo. Todo ello está regulado por la especificación RFC6455 y se admitirá también en nuestros programas MQL.

Para mayor claridad, el procedimiento se muestra en la siguiente imagen:



Interacción entre cliente y servidor a través del protocolo WebSocket

Tras establecer una conexión WebSocket, el cliente y el servidor pueden intercambiar información empaquetada en bloques especiales: frames y mensajes. Un mensaje puede constar de uno o varios frames. El tamaño del frame, según la especificación, está limitado a un número astronómico de 2^{63} bytes (9223372036854775807 ~ 9.22 exabytes), pero las implementaciones específicas pueden, por supuesto, tener límites más mundanos, ya que este límite teórico no parece práctico para enviar en un paquete.

En cualquier momento, el cliente o el servidor pueden poner fin a la conexión, previa «despedida cortés» (véase más adelante) o simplemente cerrando el socket de red.

Los frames pueden ser de distintos tipos según se especifique en su encabezado (de 4 a 16 bytes de longitud) que viene al principio de cada frame. Como referencia, vamos a enumerar los códigos operativos (están presentes en el primer byte del encabezado) y la finalidad de los frames de distintos tipos.

- 0 - frame de continuación (hereda las propiedades del frame anterior);
- 1 - frame con información de texto;
- 2 - frame con información binaria;
- 8 - solicitud de frame para cerrar y confirmación de cierre de la conexión (enviada para una «despedida cortés»);
- 9 - frame de ping; puede ser enviado periódicamente por cualquiera de los dos lados para asegurarse de que la conexión está físicamente guardada;
- 10 - frame de pong, enviado en respuesta a un frame de ping.

El último frame de un mensaje se marca con un bit especial en el encabezado. Por supuesto, cuando un mensaje consta de un solo frame, también es el último. La longitud de la carga útil también se indica en el encabezado.

7.8.5 Componente servidor de servicios web basados en el protocolo WebSocket

Para organizar un componente de servidor común a todos los proyectos, crearemos una carpeta independiente *Web* dentro de *MQL5/Experts/MQL5Book/p7/*. Lo ideal sería colocar *Web* como subcarpeta dentro de *Shared Projects*. El hecho es que *MQL5/Shared Projects* está disponible en la distribución estándar de MetaTrader 5 y reservado para proyectos de almacenamiento en la nube. Por lo tanto, más adelante, utilizando la funcionalidad de proyectos compartidos, sería posible subir todos los archivos de nuestros proyectos al servidor (no sólo los archivos web, sino también los programas MQL).

Más adelante, cuando creemos un archivo mqproj con programas cliente MQL5, añadiremos todos los archivos de esta carpeta a la sección del proyecto *Settings and Files*, ya que todos estos archivos forman parte integral del proyecto: la parte del servidor.

Dado que se ha asignado un directorio independiente para el servidor del proyecto, es necesario garantizar la posibilidad de importar módulos de nodejs en este directorio. Por defecto, nodejs busca los módulos en la subcarpeta */node_modules* del directorio actual, y ejecutaremos el servidor desde el proyecto. Por lo tanto, estando en la carpeta donde colocaremos los archivos web del proyecto, ejecute el comando:

```
mklink /j node_modules {drive:/path/to/folder/nodejs}/node_modules
```

Como resultado, aparecerá un enlace «simbólico» de directorio llamado *node_modules*, apuntando a la carpeta original del mismo nombre en el nodejs instalado.

La forma más sencilla de comprobar la funcionalidad de WebSockets es el servicio eco. Su modelo de funcionamiento consiste en devolver al remitente cualquier mensaje recibido. Veamos cómo sería posible organizar un servicio de este tipo con una configuración mínima. Se incluye un ejemplo en el archivo *wsintro.js*.

En primer lugar, conectamos el paquete (módulo) *ws*, que proporciona la funcionalidad WebSocket para nodejs y que instalamos junto con el servidor web.

```
// JavaScript
const WebSocket = require('ws');
```

La función *require* opera de forma similar a la directiva *#include* de MQL5, pero además devuelve un objeto módulo con la API de todos los archivos del paquete *ws*. Gracias a ello, podemos llamar a los métodos y propiedades del objeto *WebSocket*. En este caso, necesitamos crear un servidor WebSocket en el puerto 9000.

```
// JavaScript
const port = 9000;
const wss = new WebSocket.Server({ port: port });
```

Aquí vemos la llamada al constructor habitual de MQL5 mediante el operador *new*, pero se pasa como parámetro un objeto sin nombre (estructura), en el que, como en un mapa, se puede almacenar un conjunto de propiedades con nombre y sus valores. En este caso, solo se utiliza una propiedad *port*, y su valor se establece igual a la variable (más exactamente, una constante) *port* descrita con anterioridad. Básicamente, podemos pasar el número de puerto (y otros ajustes) en la línea de comandos al ejecutar el script.

El objeto servidor se introduce en la variable *wss*. En caso de éxito, indicamos a la ventana de la línea de comandos que el servidor está en ejecución (esperando conexiones).

```
// JavaScript
console.log('listening on port: ' + port);
```

La llamada a *console.log* es similar a la habitual *Print* en MQL5. Tenga en cuenta también que en JavaScript las cadenas pueden encerrarse no sólo entre comillas dobles, sino también entre comillas simples, e incluso entre comillas simples inversas *`this is a ${template}text`*, lo que añade algunas características útiles.

A continuación, para el objeto *wss*, asignamos un manejador de eventos «connection», que hace referencia a la conexión de un nuevo cliente. Obviamente, la lista de eventos de objeto admitidos está definida por los desarrolladores del paquete, en este caso, el paquete *ws* que utilizamos. Todo esto se refleja en la documentación.

El manejador está vinculado por el método *on*, que especifica el nombre del evento y el manejador en sí.

```
// JavaScript
wss.on('connection', function(channel)
{
   ...
});
```

El manejador es una función sin nombre (anónima) definida directamente en el lugar donde se espera un parámetro de referencia para que el código de devolución de llamada se ejecute en una nueva conexión. La función se hace anónima porque sólo se utiliza aquí, y JavaScript permite este tipo de simplificaciones en la sintaxis. La función sólo tiene un parámetro que es el objeto de la nueva conexión. Podemos elegir libremente el nombre del parámetro, que en este caso es *channel*.

Dentro del manejador, se debe establecer otro manejador para el evento «mensaje» relacionado con la llegada de un nuevo mensaje en un canal específico.

```
// JavaScript
   channel.on('message', function(message)
   {
      console.log('message: ' + message);
      channel.send('echo: ' + message);
   });
   ...
```

También utiliza una función anónima con un único parámetro, el objeto mensaje recibido. Lo imprimimos en el registro de la consola para depuración. Pero lo más importante ocurre en la segunda línea: al llamar a *channel.send*, enviamos un mensaje de respuesta al cliente.

Para completar la imagen, vamos a añadir nuestro propio mensaje de bienvenida al manejador de «conexión». Cuando esté completo, tendrá este aspecto:

```
// JavaScript
wss.on('connection', function(channel)
{
   channel.on('message', function(message)
   {
      console.log('message: ' + message);
      channel.send('echo: ' + message);
   });
   console.log('new client connected!');
   channel.send('connected!');
});
```

Es importante entender que, mientras que vincular el manejador de «mensaje» está más arriba en el código que enviar el «hola», el manejador de mensaje será llamado más tarde, y sólo si el cliente envía un mensaje.

Hemos revisado un esquema de script para organizar un servicio de eco. Sin embargo, sería bueno probarlo. Esto puede hacerse de la forma más eficiente utilizando un navegador normal, pero para ello será necesario complicar ligeramente el script: convertirlo en el servidor web más pequeño posible que devuelva una página web con el cliente WebSocket más pequeño posible.

Servicio eco y página web de prueba

El script del servidor eco que vamos a ver ahora se encuentra en el archivo *wsecho.js*. Uno de los puntos principales es que conviene admitir no sólo protocolos abiertos en el servidor *http/ws*, sino también protocolos protegidos *https/wss*. Esta posibilidad se ofrecerá en todos nuestros ejemplos (incluidos los clientes basados en MQL5), pero para ello es necesario realizar algunas acciones en el servidor.

Debería empezar con un par de archivos que contengan claves de encriptación y certificados. Los archivos suelen obtenerse de fuentes autorizadas, es decir, centros certificadores, pero a efectos informativos, puede generarlos usted mismo. Por supuesto, no pueden utilizarse en servidores públicos, y las páginas con dichos certificados provocarán advertencias en cualquier navegador (el icono de la página a la izquierda de la barra de direcciones aparece resaltado en rojo).

La descripción del dispositivo de los certificados y del proceso para generarlos por sí mismos queda fuera del alcance del libro, pero en él se incluyen dos archivos ya preparados: *MQL5Book.crt* y *MQL5Book.key* (existen otras extensiones) con una duración limitada. Estos archivos deben pasarse al constructor del objeto servidor web para que el servidor funcione a través del protocolo HTTPS.

Pasaremos el nombre de los archivos de certificado en la línea de comandos de lanzamiento del script. Por ejemplo, así:

```
node wsecho.js MQL5Book
```

Si ejecuta el script sin un parámetro adicional, el servidor funcionará utilizando el protocolo HTTP.

```
node wsecho.js
```

Dentro del script, los argumentos de la línea de comandos están disponibles a través del objeto integrado *process.argv*, y los dos primeros argumentos siempre contienen, respectivamente, el nombre del servidor *node.exe* y el nombre del script que se va a ejecutar (en este caso, *wsecho.js*), por lo que los descartamos mediante el método *splice*.

```
// JavaScript
const args = process.argv.slice(2);
const secure = args.length > 0 ? 'https' : 'http';
```

Dependiendo de la presencia del nombre del certificado, la variable *secure* obtiene el nombre del paquete que debe cargarse a continuación para crear el servidor: *https* o *http*. En total, tenemos 3 dependencias en el código:

```
// JavaScript
const fs = require('fs');
const http1 = require(secure);
const WebSocket = require('ws');
```

Ya conocemos el paquete *ws*; los paquetes *https* y *http* proporcionan una implementación del servidor web, y el paquete *fs* integrado permite trabajar con el sistema de archivos.

La configuración del servidor web tiene el formato del objeto *options*. Aquí vemos cómo el nombre del certificado de la línea de comandos se sustituye en cadenas con comillas de barra oblicua utilizando la expresión *${args[0]}*. A continuación, se lee el par de archivos correspondiente mediante el método *fs.readFileSync*.

```
// JavaScript
const options = args.length > 0 ?
{
   key : fs.readFileSync(`${args[0]}.key`),
   cert : fs.readFileSync(`${args[0]}.crt`)
} : null;
```

El servidor web se crea llamando al método *createServer*, al que pasamos el objeto de opciones y una función anónima: un manejador de solicitudes HTTP. El manejador tiene dos parámetros: el objeto *req* con una solicitud HTTP y el objeto *res* con el que debemos enviar la respuesta (encabezados HTTP y página web).

```
// JavaScript
http1.createServer(options, function (req, res)
{
   console.log(req.method, req.url);
   console.log(req.headers);

   if(req.url == '/') req.url = "index.htm";

   fs.readFile('./' + req.url, (err, data) =>
   {
      if(!err)
      {
         var dotoffset = req.url.lastIndexOf('.');
         var mimetype = dotoffset == -1 ? 'text/plain' :
         {
            '.htm' : 'text/html',
            '.html' : 'text/html',
            '.css' : 'text/css',
            '.js' : 'text/javascript'
         }[ req.url.substr(dotoffset) ];
         res.setHeader('Content-Type',
            mimetype == undefined ? 'text/plain' : mimetype);
         res.end(data);
      }
      else
      {
         console.log('File not fount: ' + req.url);
         res.writeHead(404, "Not Found");
         res.end();
      }
  });
}).listen(secure == 'https' ? 443 : 80);
```

La página principal del índice (y la única) es *index.htm* (por escribir ahora). Además, el manejador puede enviar archivos js y css, lo que nos será útil en el futuro. Dependiendo de si el modo protegido está activado, el servidor se inicia llamando al método *listen* en los puertos estándar 443 u 80 (cámbielos por otros si ya están ocupados en su ordenador).

Para aceptar conexiones en el puerto 9000 para sockets web, necesitamos desplegar otra instancia de servidor web con las mismas opciones. Pero en este caso, el servidor está ahí con el único propósito de gestionar una solicitud HTTP para «actualizar» la conexión al protocolo Web Sockets.

```
// JavaScript
const server = new http1.createServer(options).listen(9000);
server.on('upgrade', function(req, socket, head)
{
   console.log(req.headers); // TODO: we can add authorization!
});
```

Aquí, en el manejador de eventos «upgrade», aceptamos cualquier conexión que ya haya pasado el «handshake» e imprimimos los encabezados en el registro, pero potencialmente podríamos solicitar autorización al usuario si estuviéramos haciendo un servicio cerrado (de pago).

Por último, creamos un objeto servidor WebSocket, como en el ejemplo introductorio anterior, con la única diferencia de que al constructor se le pasa un servidor web ya preparado. Se cuentan todos los clientes que se conectan y se les da la bienvenida por número de secuencia.

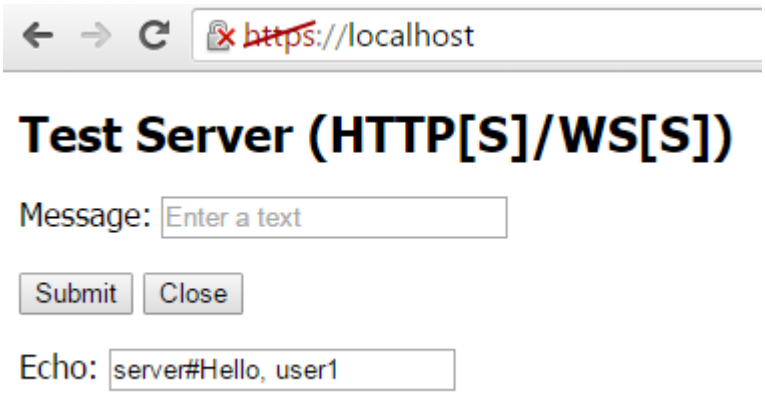
```
// JavaScript
var count = 0;

const wsServer = new WebSocket.Server({ server });
wsServer.on('connection', function onConnect(client)
{
   console.log('New user:', ++count);
   client.id = count;
   client.send('server#Hello, user' + count);

   client.on('message', function(message)
   {
      console.log('%d : %s', client.id, message);
      client.send('user' + client.id + '#' + message);
   });

   client.on('close', function()
   {
      console.log('User disconnected:', client.id);
   });
});
```

Para todos los eventos, incluyendo conexión, desconexión y mensaje, la información de depuración se muestra en la consola.

Bueno, el servidor web con compatibilidad de servidor web socket está listo. Ahora tenemos que crear una página web de cliente *index.htm* para ello.

```
<!DOCTYPE html>
<html>
  <head>
  <title>Test Server (HTTP[S]/WS[S])</title>
  </head>
  <body>
    <div>
      <h1>Test Server (HTTP[S]/WS[S])</h1>
      <p><label>
         Message: <input id="message" name="message" placeholder="Enter a text">
      </label></p>
      <p><button>Submit</button> <button>Close</button></p>
      <p><label>
         Echo: <input id="echo" name="echo" placeholder="Text from server">
      </label></p>
    </div>
  </body>
  <script src="wsecho_client.js"></script>
</html>
```

La página es un formulario con un único campo de entrada y un botón para enviar un mensaje.



Página web del servicio de eco en WebSocket

La página utiliza el script *wsecho_client.js*, que proporciona la respuesta del cliente websocket. En los navegadores, los web sockets están integrados como objetos JavaScript «nativos», por lo que no es necesario conectar nada externo: basta con llamar al constructor *web socket* con el protocolo y el número de puerto deseados.

```
// JavaScript
const proto = window.location.protocol.startsWith('http') ?
              window.location.protocol.replace('http', 'ws') : 'ws:';
const ws = new WebSocket(proto + '//' + window.location.hostname + ':9000');
```

La URL se forma a partir de la dirección de la página web actual (*window.location.hostname*), por lo que la conexión del socket web se realiza al mismo servidor.

A continuación, el objeto *ws* permite reaccionar a eventos y enviar mensajes. En el navegador, el evento de conexión abierto se denomina «open»; se conecta a través de la propiedad *onopen*. La misma sintaxis, ligeramente diferente de la implementación del servidor, también se utiliza para el evento de llegada de un nuevo mensaje: el manejador para él se asigna a la propiedad *onmessage*.

```
// JavaScript
ws.onopen = function()
{
   console.log('Connected');
};

ws.onmessage = function(message)
{
   console.log('Message: %s', message.data);
   document.getElementById('echo').value = message.data;
};
```

El texto del mensaje entrante se muestra en el elemento de formulario con el id «echo». Tenga en cuenta que el objeto del evento del mensaje (parámetro del manejador) no es el mensaje que está disponible en la propiedad *data*. Se trata de una función de implementación de JavaScript.

La reacción a los botones del formulario se asigna utilizando el método *addEventListener* para cada uno de los dos objetos de etiqueta *button*. Aquí vemos otra forma de describir una función anónima en JavaScript: paréntesis con una lista de argumentos que puede estar vacía, y el cuerpo de la función después de la flecha puede ser *(arguments) => { ... }*.

```
// JavaScript
const button = document.querySelectorAll('button'); // request all buttons
// button "Submit"
button[0].addEventListener('click', (event) =>
{
   const x = document.getElementById('message').value;
   if(x) ws.send(x);
});
// button "close"
button[1].addEventListener('click', (event) =>
{
   ws.close();
   document.getElementById('echo').value = 'disconnected';
   Array.from(document.getElementsByTagName('button')).forEach((e) =>
   {
      e.disabled = true;
   });
});
```

Para enviar mensajes, llamamos al método *ws.send*, y para cerrar la conexión llamamos al método *ws.close*.

Esto completa el desarrollo del primer ejemplo de scripts cliente-servidor para demostrar el servicio de eco. Puede ejecutar *wsecho.js* utilizando uno de los comandos mostrados anteriormente y, a continuación, abrir en su navegador la página en *http://localhost* o *https://localhost* (dependiendo de la configuración del servidor). Después de que aparezca el formulario en la pantalla, intente chatear con el servidor y asegúrese de que el servicio está funcionando.

Complicando gradualmente este ejemplo, allanaremos el camino para el servicio web de copia de señales de trading. Pero el siguiente paso será un servicio de chat, cuyo principio es similar al del servicio de señales de trading: los mensajes de un usuario se transmiten a otros usuarios.

Servicio de chat y página web de prueba

El nuevo script del servidor se llama *wschat.js*, y repite mucho de *wsecho.js*. Enumeremos las principales diferencias. En el manejador de solicitudes HTTP del servidor web, cambie la página inicial de *index.htm* a *wschat.htm*.

```
// JavaScript
http1.createServer(options, function (req, res)
{
   if(req.url == '/') req.url = "wschat.htm";
   ...
});
```

Para almacenar la información sobre los usuarios conectados al chat, describiremos el array de map *clients*. *Map* es un contenedor asociativo estándar de JavaScript, en el que se pueden escribir valores arbitrarios utilizando claves de un tipo arbitrario, incluyendo objetos.

```
// JavaScript
const clients = new Map();                    // added this line
var count = 0;
```

En el nuevo manejador de eventos de conexión de usuario, añadiremos el objeto *client*, recibido como parámetro de función, en el mapa bajo el número de secuencia del cliente actual.

```
// JavaScript
wsServer.on('connection', function onConnect(client)
{
   console.log('New user:', ++count);
   client.id = count;
   client.send('server#Hello, user' + count);
   clients.set(count, client);                // added this line
   ...
```

Dentro de la función *onConnect*, establecemos un manejador para el evento sobre la llegada de un nuevo mensaje para un cliente específico, y es dentro del manejador anidado donde enviamos los mensajes. No obstante, esta vez recorremos todos los elementos del mapa (es decir, todos los clientes) y enviamos el texto a cada uno de ellos. El bucle se organiza con las llamadas al método *forEach* para un array del mapa, y la siguiente función anónima que se realizará para cada elemento (*elem*) se pasa al método en su lugar. El ejemplo de este bucle demuestra claramente el paradigma de programación *functional-declarative* que prevalece en JavaScript (en contraste con el enfoque *imperative* de MQL5).

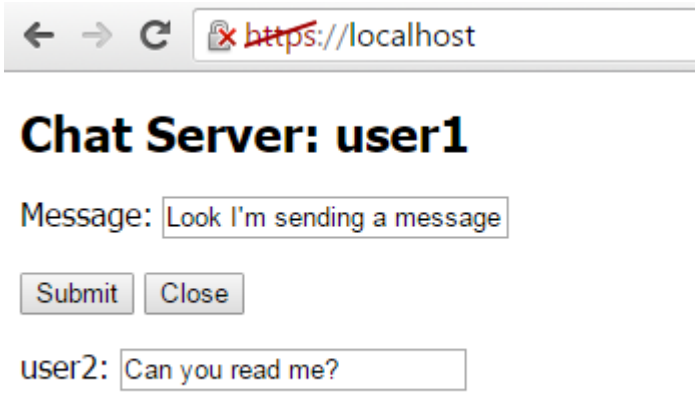
```
// JavaScript
   client.on('message', function(message)
   {
      console.log('%d : %s', client.id, message);
      Array.from(clients.values()).forEach(function(elem) // added a loop
      {
         elem.send('user' + client.id + '#' + message);
      });
   });
```

Es importante señalar que enviamos una copia del mensaje a todos los clientes, incluido el autor original. Podría filtrarse, pero a efectos de depuración, es mejor tener confirmación de que el mensaje se ha enviado.

La última diferencia con el anterior servicio de eco es que cuando un cliente se desconecta, hay que eliminarlo del mapa.

```
// JavaScript
   client.on('close', function()
   {
      console.log('User disconnected:', client.id);
      clients.delete(client.id);                  // added this line
   });
```

En cuanto a la sustitución de la página *index.htm* por *wschat.htm*, aquí añadimos un «campo» para mostrar el autor del mensaje (*origin*) y conectamos un nuevo script de navegador *wschat_client.js*. Analiza los mensajes (utilizamos el símbolo «#» para separar al autor del texto) y rellena los campos del formulario con la información recibida. Dado que nada ha cambiado desde el punto de vista del protocolo WebSocket, no proporcionaremos el código fuente.



Página web del servicio de chat en WebSocket

Puede iniciar nodejs con el servidor de chat *wschat.js* y conectarse a él desde varias pestañas del navegador. Cada conexión recibe un número único que aparece en el encabezado. El texto del campo *Message* se envía a todos los clientes al hacer clic en *Submit*. A continuación, los formularios de cliente muestran tanto el autor del mensaje (etiqueta de la parte inferior izquierda) como el propio texto (campo de la parte inferior central).

Por lo tanto, nos hemos asegurado de que el servidor web con soporte web socket está listo. Pasemos a escribir la parte cliente del protocolo en MQL5.

7.8.6 Protocolo WebSocket en MQL5

Ya hemos examinado con anterioridad los Fundamentos teóricos del protocolo WebSockets. La especificación completa es bastante extensa, y una descripción detallada de su aplicación requeriría mucho espacio y tiempo. Por ello, presentamos la estructura general de las clases ya creadas y sus interfaces de programación. Todos los archivos se encuentran en el directorio *MQL5/Include/MQL5Book/ws/*.

- wsinterfaces.mqh - descripción general abstracta de todas las interfaces, constantes y tipos;
- wstransport.mqh - clase MqlWebSocketTransport que implementa la interfaz de transferencia de datos de red de bajo nivel IWebSocketTransport basada en funciones de socket de MQL5;
- wsframe.mqh - clases WebSocketFrame y WebSocketFrameHixie que implementan la interfaz IWebSocketFrame, que oculta los algoritmos de generación (codificación y decodificación) de frames para los protocolos Hybi e Hixie, respectivamente;
- wsmessage.mqh - clases WebSocketMessage y WebSocketMessageHixie que implementan la interfaz IWebSocketMessage, que formaliza la formación de mensajes a partir de frames para los protocolos Hybi y Hixie, respectivamente;
- wsprotocol.mqh - clases WebSocketConnection, WebSocketConnectionHybi, WebSocketConnectionHixie heredadas de IWebSocketConnection; en ellas tiene lugar la administración coordinada de la formación de frames, mensajes, saludos y desconexión según la especificación, para lo que se utilizan las interfaces anteriores;
- wsclient.mqh - implementación ya hecha de un cliente WebSocket; una clase de plantilla WebSocketClient que admite la interfaz IWebSocketObserver (para el procesamiento de eventos) y espera WebSocketConnectionHybi o WebSocketConnectionHixie como tipo parametrizado;
- wstools.mqh - utilidades prácticas del espacio de nombres WsTools.

Estos archivos de encabezado se incluirán automáticamente en nuestros futuros proyectos mqporj como dependencias de las directivas *#include*.

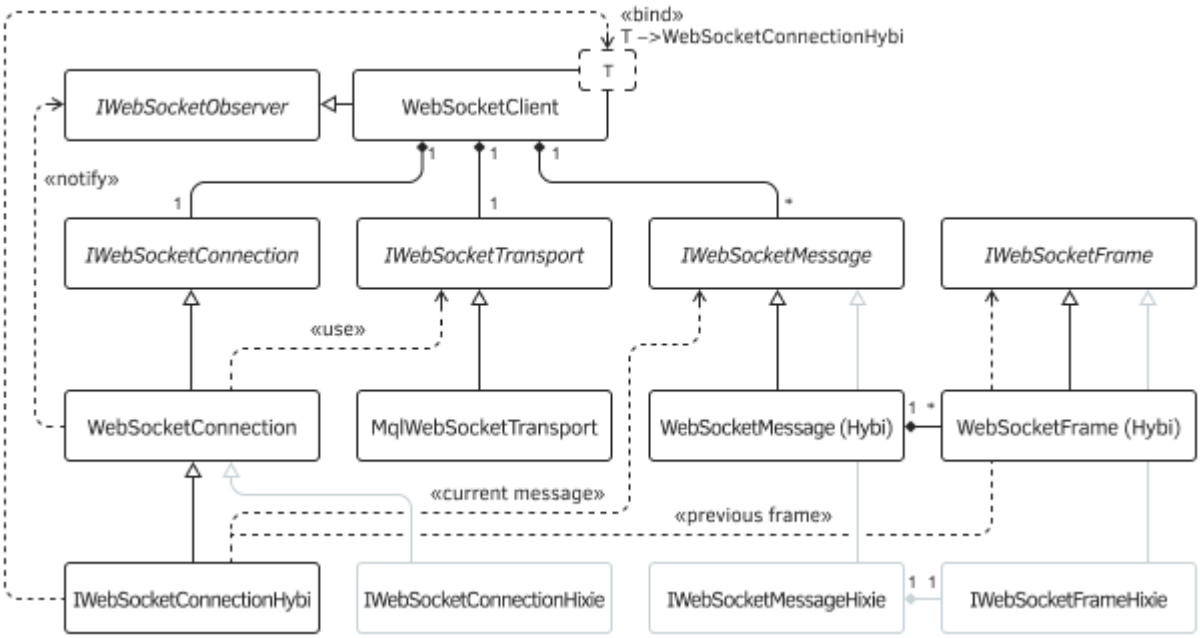


Diagrama de clase WebSocket en MQL5

La interfaz de red de bajo nivel *IWebSocketTransport* dispone de los siguientes métodos.

```
interface IWebSocketTransport
{
   int write(const uchar &data[]); // write the array of bytes to the network
   int read(uchar &buffer[]);      // read data from network into byte array
   bool isConnected(void) const;   // check for connection
   bool isReadable(void) const;    // check for the ability to read from the network
   bool isWritable(void) const;    // check for the possibility of writing to the net
   int getHandle(void) const;      // system socket descriptor
   void close(void);               // close connection
};
```

No es difícil adivinar por los nombres de los métodos qué funciones del socket de la API de MQL5 se utilizarán para construirlos, pero si es necesario, quienes lo deseen pueden implementar esta interfaz por sus propios medios, por ejemplo, a través de una DLL.

La clase *MqlWebSocketTransport* que implementa esta interfaz requiere el protocolo, el nombre de host y el número de puerto al que se realiza la conexión de red cuando se crea una instancia. Además, puede especificar un valor de tiempo de espera.

Los tipos de frame se recopilan en el enum WS_FRAME_OPCODE.

```
enum WS_FRAME_OPCODE
{
   WS_DEFAULT = 0,
   WS_CONTINUATION_FRAME = 0x00,
   WS_TEXT_FRAME = 0x01,
   WS_BINARY_FRAME = 0x02,
   WS_CLOSE_FRAME = 0x08,
   WS_PING_FRAME = 0x09,
   WS_PONG_FRAME = 0x0A
};
```

La interfaz para trabajar con frames contiene métodos estáticos y regulares relacionados con instancias de frame. Los métodos estáticos actúan como fábricas para crear frames del tipo requerido por el lado emisor (*create*) y frames entrantes (*decode*).

```
class IWebSocketFrame
{
public:
   class StaticCreator
   {
   public:
      virtual IWebSocketFrame *decode(uchar &data[], IWebSocketFrame *head = NULL) =
      virtual IWebSocketFrame *create(WS_FRAME_OPCODE type, const string data = NULL,
         const bool deflate = false) = 0;
      virtual IWebSocketFrame *create(WS_FRAME_OPCODE type, const uchar &data[],
         const bool deflate = false) = 0;
   };
   ...
```

La presencia de métodos de fábrica en las clases descendientes se hace obligatoria debido a la presencia de una plantilla *Creator* y una instancia del método *getCreator* que la devuelve (suponiendo la devolución «solitario» («singleton»)).

```
protected:
   template<typename P>
   class Creator: public StaticCreator
   {
   public:
     // decode received binary data in IWebSocketFrame
     // (in case of continuation, previous frame in 'head')
      virtual IWebSocketFrame *decode(uchar &data[],
         IWebSocketFrame *head = NULL) override
      {
         return P::decode(data, head);
      }
      // create a frame of the desired type (text/closing/other) with optional text
      virtual IWebSocketFrame *create(WS_FRAME_OPCODE type, const string data = NULL,
         const bool deflate = false) override
      {
         return P::create(type, data, deflate);
      };
      // create a frame of the desired type (binary/text/closure/other) with data
      virtual IWebSocketFrame *create(WS_FRAME_OPCODE type, const uchar &data[],
         const bool deflate = false) override
      {
         return P::create(type, data, deflate);
      };
   };
public:
   // require a Creator instance
   virtual IWebSocketFrame::StaticCreator *getCreator() = 0;
   ...
```

Los métodos restantes de la interfaz proporcionan todas las manipulaciones necesarias con datos en frames (codificación/decodificación, recepción de datos y diversas banderas).

```
    // encode the "clean" contents of the frame into data for transmission over the ne
    virtual int encode(uchar &encoded[]) = 0;

    // get data as text
    virtual string getData() = 0;

    // get data as bytes, return size
    virtual int getData(uchar &buf[]) = 0;

    // return frame type (opcode)
    virtual WS_FRAME_OPCODE getType() = 0;

    // check if the frame is a control frame or with data:
    // control frames are processed inside classes
    virtual bool isControlFrame()
    {
       return (getType() >= WS_CLOSE_FRAME);
    }

    virtual bool isReady() { return true; }
    virtual bool isFinal() { return true; }
    virtual bool isMasked() { return false; }
    virtual bool isCompressed() { return false; }
};
```

La interfaz *IWebSocketMessage* contiene métodos para realizar acciones similares pero a nivel de mensaje.

```
class IWebSocketMessage
{
public:
   // get an array of frames that make up this message
   virtual void getFrames(IWebSocketFrame *&frames[]) = 0;

   // set text as message content
   virtual bool setString(const string &data) = 0;

   // return message content as text
   virtual string getString() = 0;

   // set binary data as message content
   virtual bool setData(const uchar &data[]) = 0;

   // return the contents of the message in "raw" binary form
   virtual bool getData(uchar &data[]) = 0;

   // sign of completeness of the message (all frames received)
   virtual bool isFinalised() = 0;

   // add a frame to the message
   virtual bool takeFrame(IWebSocketFrame *frame) = 0;
};
```

Teniendo en cuenta las interfaces de frames y mensajes, se define una interfaz común para las conexiones WebSocket *IWebSocketConnection*.

```
interface IWebSocketConnection
{
   // open a connection with the specified URL and its parts,
   // and optional custom headers
   bool handshake(const string url, const string host, const string origin,
      const string custom = NULL);

   // low-level read frames from the server
   int readFrame(IWebSocketFrame *&frames[]);

   // low-level send frame (e.g. close or ping)
   bool sendFrame(IWebSocketFrame *frame);

   // low-level message sending
   bool sendMessage(IWebSocketMessage *msg);

   // custom check for new messages (event generation)
   int checkMessages();

   // custom text submission
   bool sendString(const string msg);

   // custom posting of binary data
   bool sendData(const uchar &data[]);

   // close the connection
   bool disconnect(void);
};
```

Las notificaciones sobre desconexión y nuevos mensajes se reciben a través de los métodos de la interfaz *IWebSocketObserver*.

```
interface IWebSocketObserver
{
  void onConnected();
  void onDisconnect();
  void onMessage(IWebSocketMessage *msg);
};
```

En concreto, la clase *WebSocketClient* se convirtió en sucesora de esta interfaz y, por defecto, simplemente envía información al registro. El constructor de la clase espera una dirección para conectarse al protocolo *ws* o *wss*.

```
template<typename T>
class WebSocketClient: public IWebSocketObserver
{
protected:
   IWebSocketMessage *messages[];

   string scheme;
   string host;
   string port;
   string origin;
   string url;
   int timeOut;
   ...
public:
   WebSocketClient(const string address)
   {
      string parts[];
      URL::parse(address, parts);

      url = address;
      timeOut = 5000;

      scheme = parts[URL_SCHEME];
      if(scheme != "ws" && scheme != "wss")
      {
        Print("WebSocket invalid url scheme: ", scheme);
        scheme = "ws";
      }

      host = parts[URL_HOST];
      port = parts[URL_PORT];

      origin = (scheme == "wss" ? "https://" : "http://") + host;
   }
   ...

   void onDisconnect() override
   {
      Print(" > Disconnected ", url);
   }

   void onConnected() override
   {
      Print(" > Connected ", url);
   }

   void onMessage(IWebSocketMessage *msg) override
   {
      // NB: message can be binary, print it just for notification
      Print(" > Message ", url, " " , msg.getString());
      WsTools::push(messages, msg);
```

```
      }
      ...
  };
```

La clase *WebSocketClient* recoge todos los objetos de mensaje en un array y se encarga de borrarlos si el programa MQL no lo hace.

La conexión se establece en el método *open*.

```
  template<typename T>
  class WebSocketClient: public IWebSocketObserver
  {
  protected:
     IWebSocketTransport *socket;
     IWebSocketConnection *connection;
     ...
  public:
     ...
     bool open(const string custom_headers = NULL)
     {
        uint _port = (uint)StringToInteger(port);
        if(_port == 0)
        {
           if(scheme == "ws") _port = 80;
           else _port = 443;
        }

        socket = MqlWebSocketTransport::create(scheme, host, _port, timeOut);
        if(!socket || !socket.isConnected())
        {
           return false;
        }

        connection = new T(&this, socket);
        return connection.handshake(url, host, origin, custom_headers);
     }
     ...
```

Las formas más cómodas de enviar datos son los métodos sobrecargados *send* para datos de texto y binarios.

```
     bool send(const string str)
     {
        return connection ? connection.sendString(str) : false;
     }

     bool send(const uchar &data[])
     {
        return connection ? connection.sendData(data) : false;
     }
```

Para comprobar si hay nuevos mensajes entrantes, puede llamar al método *checkMessages*. Dependiendo de su parámetro *blocking*, el método esperará un mensaje en un bucle hasta que se agote

el tiempo de espera o devolverá inmediatamente si no hay mensajes. Los mensajes irán al manejador *IWebSocketObserver::onMessage*.

```
void checkMessages(const bool blocking = true)
{
   if(connection == NULL) return;

   uint stop = GetTickCount() + (blocking ? timeOut : 1);
   while(ArraySize(messages) == 0 && GetTickCount() < stop && isConnected())
   {
      // all frames are collected into the appropriate messages, and they become
      // available through event notifications IWebSocketObserver::onMessage,
      // however, control frames have already been internally processed and remove
      if(!connection.checkMessages()) // while no messages, let's make micro-pause
      {
         Sleep(100);
      }
   }
}
```

Una forma alternativa de recibir mensajes se implementa en el método *readMessage*: devuelve un puntero al mensaje al código de llamada (en otras palabras, el manejador de la aplicación *onMessage* no es necesario). Después, el programa MQL se encarga de liberar el objeto.

```
IWebSocketMessage *readMessage(const bool blocking = true)
{
   if(ArraySize(messages) == 0) checkMessages(blocking);

   if(ArraySize(messages) > 0)
   {
      IWebSocketMessage *top = messages[0];
      ArrayRemove(messages, 0, 1);
      return top;
   }
   return NULL;
}
```

La clase también permite cambiar el tiempo de espera, comprobar la conexión y cerrarla.

```
      void setTimeOut(const int ms)
      {
         timeOut = fabs(ms);
      }

      bool isConnected() const
      {
         return socket && socket.isConnected();
      }

      void close()
      {
         if(isConnected())
         {
            if(connection)
            {
               connection.disconnect(); // this will close socket after server acknowled
               delete connection;
               connection = NULL;
            }
            if(socket)
            {
               delete socket;
               socket = NULL;
            }
         }
      }
   };
```

La biblioteca de las clases consideradas permite crear aplicaciones cliente para servicios de eco y chat.

7.8.7 Programas cliente para servicios de eco y chat en MQL5

Vamos a escribir un sencillo script para conectarnos al servicio de eco *MQL5/Experts/MQL5Book/p7/wsEcho/wsecho.mq5* (nótese que se trata de un script, pero lo hemos colocado dentro de la carpeta *MQL5/Experts/MQL5Book/p7/*, convirtiéndola en un contenedor único para programas MQL relacionados con la web, ya que todos los ejemplos posteriores serán Asesores Expertos). Dado que en este capítulo estamos considerando la creación de complejos de software dentro de proyectos, diseñaremos el script como parte de un proyecto mqproj, que también incluirá el componente servidor.

Los parámetros de entrada del script permiten especificar la dirección del servicio y el texto del mensaje. Por defecto, la conexión no es segura. Si va a lanzar el servidor *wsecho.js* con soporte TLS, necesita cambiar el protocolo al seguro *wss*. Tenga en cuenta que establecer una conexión segura lleva más tiempo (un par de segundos) de lo habitual.

```
input string Server = "ws://localhost:9000/";
input string Message = "My outbound message";

#include <MQL5Book/AutoPtr.mqh>
#include <MQL5Book/ws/wsclient.mqh>
```

En la función *OnStart*, creamos una instancia del cliente WebSocket (*wss*) para la dirección dada y llamamos al método *open*. En caso de conexión exitosa, esperamos un mensaje de bienvenida del servicio llamando a *wss.readMessage* en modo de bloqueo (espera de hasta 5 segundos, por defecto). Utilizamos un puntero automático en el objeto resultante para no tener que llamar manualmente a *delete* al final.

```
void OnStart()
{
   Print("\n");
   WebSocketClient<Hybi> wss(Server);
   Print("Opening...");
   if(wss.open())
   {
      Print("Waiting for welcome message (if any)");
      AutoPtr<IWebSocketMessage> welcome(wss.readMessage());
      ...
```

La clase *WebSocketClient* contiene stubs de manejadores de eventos, incluyendo el método simple *onMessage*, que imprimirá el saludo en el log.

A continuación, enviamos nuestro mensaje y esperamos de nuevo una respuesta del servidor. El mensaje de eco también se registrará.

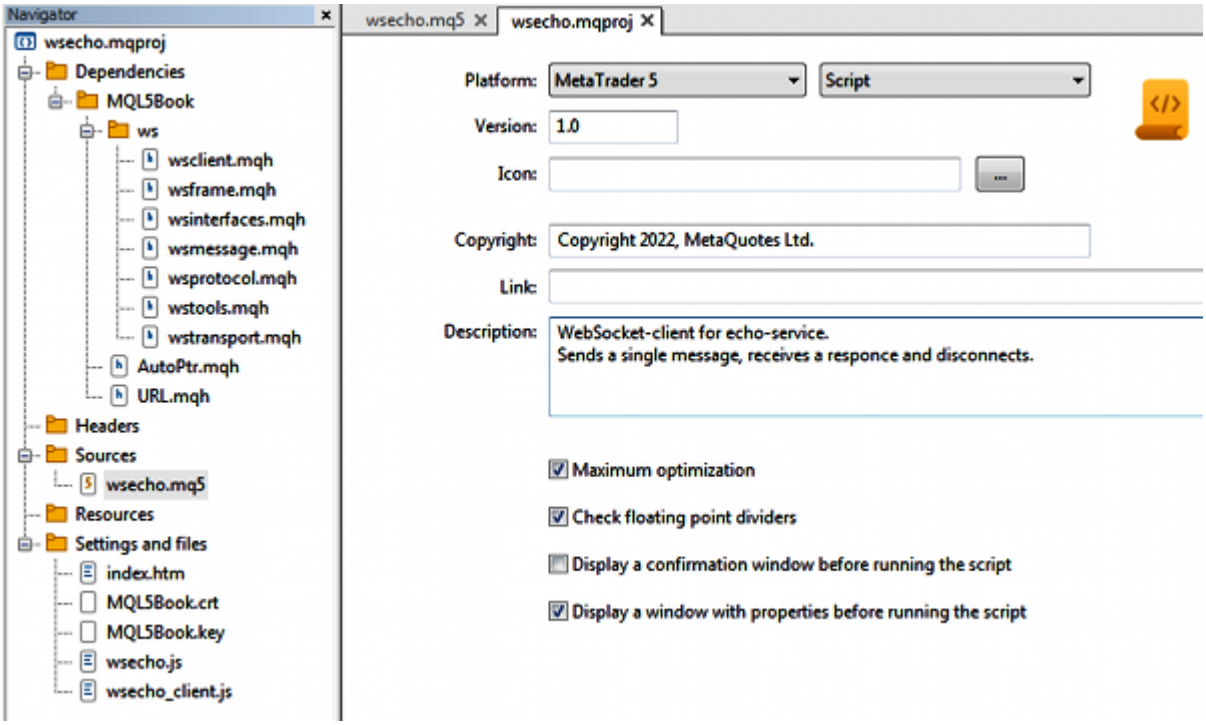
```
      Print("Sending message...");
      wss.send(Message);
      Print("Receiving echo...");
      AutoPtr<IWebSocketMessage> echo(wss.readMessage());
   }
   ...
```

Por último, cerramos la conexión.

```
   if(wss.isConnected())
   {
      Print("Closing...");
      wss.close();
   }
}
```

Basándonos en el archivo de script, vamos a crear un archivo de proyecto (*wsecho.mqproj*). Rellenamos las propiedades del proyecto con el número de versión (1.0), el copyright y la descripción. Añadamos los archivos del servidor del servicio de eco a la rama *Settings and Files* (esto al menos le recordará al desarrollador que existe un servidor de pruebas). Tras la compilación, las dependencias (archivos de encabezado) aparecerán en la jerarquía.

Todo debería tener el aspecto que se ve en la siguiente captura de pantalla:



Proyecto de servicio de eco, servidor y script de cliente

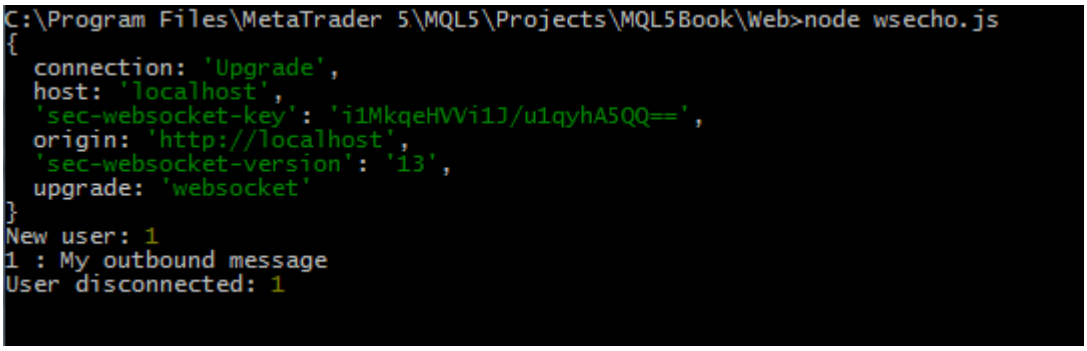
Si el script se encontraba dentro de la carpeta *Shared Projects*, por ejemplo en *MQL5/Shared Projects/MQL5Book/wsEcho/*, después de una compilación correcta, su archivo ex5 se movería automáticamente a la carpeta *MQL5/Scripts/Shared Projects/MQL5Book/wsEcho/*, y se mostraría la entrada correspondiente en el registro de compilación. Este es el comportamiento estándar para compilar cualquier programa MQL en proyectos compartidos.

En todos los ejemplos de este capítulo, no olvide iniciar el servidor antes de probar el script MQL. En este caso, ejecute el comando *node.exe wsecho.js* en la carpeta *web*.

A continuación, vamos a ejecutar el script *wsecho.ex5*. El registro mostrará las acciones que se están llevando a cabo, así como las notificaciones de mensajes.

```
Opening...
Connecting to localhost:9000
Buffer: 'HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: mIpas63g5xGMqJcKtreHKpSbY1w=
'
Headers:
                                [,0]                            [,1]
[0,] "upgrade"                      "websocket"
[1,] "connection"                   "Upgrade"
[2,] "sec-websocket-accept"         "mIpas63g5xGMqJcKtreHKpSbY1w="
 > Connected ws://localhost:9000/
Waiting for welcome message (if any)
 > Message ws://localhost:9000/ server#Hello, user1
Sending message...
Receiving echo...
 > Message ws://localhost:9000/ user1#My outbound message
Closing...
Close requested
Waiting...
SocketRead failed: 5273 Available: 1
 > Disconnected ws://localhost:9000/
Server close ack
```

Los encabezados HTTP anteriores son la respuesta del servidor durante el proceso de «handshake». Si miramos en la ventana de la consola en la que se está ejecutando el servidor, encontraremos los encabezados HTTP recibidos por el servidor desde nuestro cliente.



Registro del servidor del servicio de eco

También se indican aquí la conexión, el mensaje y la desconexión del usuario.

Hagamos un trabajo similar para el servicio de chat: crear un cliente WebSocket en MQL5, un proyecto para el mismo, y probarlo. Esta vez, el tipo de programa cliente será un Asesor Experto porque el chat necesita compatibilidad para eventos interactivos desde el teclado en el gráfico. El Asesor Experto se adjunta al libro en una carpeta *MQL5/MQL5Book/p7/wsChat/wschat.mq5*.

Para demostrar la tecnología de recepción de eventos en métodos manejadores, definamos nuestra propia clase *MyWebSocket*, derivada de *WebSocketClient*.

```
class MyWebSocket: public WebSocketClient<Hybi>
{
public:
   MyWebSocket(const string address, const bool compress = false):
      WebSocketClient(address, compress) { }

   /* void onConnected() override { } */

   void onDisconnect() override
   {
      // we can do something else and call (or not call) the legacy code
      WebSocketClient<Hybi>::onDisconnect();
   }

   void onMessage(IWebSocketMessage *msg) override
   {
     // TODO: we could truncate copies of our own messages,
     // but they are left for debugging
      Alert(msg.getString());
      delete msg;
   }
};
```

Cuando se reciba un mensaje, no lo mostraremos en el registro, sino como una alerta, tras lo cual el objeto deberá ser eliminado.

En el contexto global, describimos el objeto de nuestra clase *wss* y la cadena *message* donde se acumulará la entrada del usuario desde el teclado.

```
MyWebSocket wss(Server);
string message = "";
```

La función *OnInit* contiene la preparación necesaria; en concreto, inicia un temporizador y abre una conexión.

```
int OnInit()
{
  ChartSetInteger(0, CHART_QUICK_NAVIGATION, false);
  EventSetTimer(1);
  wss.setTimeOut(1000);
  Print("Opening...");
  return wss.open() ? INIT_SUCCEEDED : INIT_FAILED;
}
```

El temporizador es necesario para comprobar si hay mensajes nuevos de otros usuarios.

```
void OnTimer()
{
   wss.checkMessages(false); // use a non-blocking check in the timer
}
```

En el manejador *OnChartEvent*, respondemos a las pulsaciones de teclas: todas las teclas alfanuméricas se traducen en caracteres y se adjuntan a la cadena *message*. Si es necesario, puede pulsar *Backspace*

para eliminar el último carácter. Todo el texto escrito se actualiza en el comentario del gráfico. Cuando el mensaje esté completo, pulse *Enter* para enviarlo al servidor.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam,
   const string &sparam)
{
   if(id == CHARTEVENT_KEYDOWN)
   {
      if(lparam == VK_RETURN)
      {
         const static string longmessage = ...
         if(message == "long") wss.send(longmessage);
         else if(message == "bye") wss.close();
         else wss.send(message);
         message = "";
      }
      else if(lparam == VK_BACK)
      {
         StringSetLength(message, StringLen(message) - 1);
      }
      else
      {
         ResetLastError();
         const short c = TranslateKey((int)lparam);
         if(_LastError == 0)
         {
            message += ShortToString(c);
         }
      }
      Comment(message);
   }
}
```

Si introducimos el texto «largo», el programa enviará un texto especialmente preparado bastante largo. Si el texto del mensaje es «adiós», el programa cierra la conexión. Además, la conexión se cerrará cuando salga el programa.

```
void OnDeinit(const int)
{
   if(wss.isConnected())
   {
      Print("Closing...");
      wss.close();
   }
}
```

Vamos a crear un proyecto para el Asesor Experto (archivo *wschat.mqproj*), rellenar sus propiedades y añadir el backend a la rama *Settings and Files*. Esta vez mostraremos el aspecto del archivo de proyecto desde dentro. En el archivo mqproj, la rama *Dependencies* se almacena en la propiedad «files», y la rama *Settings and Files*, en la propiedad «tester».

```
{
"platform" :"mt5",
"program_type":"expert",
"copyright" :"Copyright 2022, MetaQuotes Ltd.",
"version" :"1.0",
"description" :"WebSocket-client for chat-service.\r\nType and send text messages for
"optimize" :"1",
"fpzerocheck" :"1",
"tester_no_cache":"0",
"tester_everytick_calculate":"0",
"unicode_character_set":"0",
"static_libraries":"0",
"files":
[
{
"path":"wschat.mq5",
"compile":true,
"relative_to_project":true
},
{
"path":"MQL5\\Include\\MQL5Book\\ws\\wsclient.mqh",
"compile":false,
"relative_to_project":false
},
{
"path":"MQL5\\Include\\MQL5Book\\URL.mqh",
"compile":false,
"relative_to_project":false
},
{
"path":"MQL5\\Include\\MQL5Book\\ws\\wsframe.mqh",
"compile":false,
"relative_to_project":false
},
{
"path":"MQL5\\Include\\MQL5Book\\ws\\wstools.mqh",
"compile":false,
"relative_to_project":false
},
{
"path":"MQL5\\Include\\MQL5Book\\ws\\wsinterfaces.mqh",
"compile":false,
"relative_to_project":false
},
{
"path":"MQL5\\Include\\MQL5Book\\ws\\wsmessage.mqh",
"compile":false,
"relative_to_project":false
},
{
"path":"MQL5\\Include\\MQL5Book\\ws\\wstransport.mqh",
"compile":false,
"relative_to_project":false
},
```

```
{
"path":"MQL5\\Include\\MQL5Book\\ws\\wsprotocol.mqh",
"compile":false,
"relative_to_project":false
},
{
"path":"MQL5\\Include\\VirtualKeys.mqh",
"compile":false,
"relative_to_project":false
}
],
"tester":
[
{
"type":"file",
"path":"..\\Web\\MQL5Book.crt",
"relative_to_project":true
},
{
"type":"file",
"path":"..\\Web\\MQL5Book.key",
"relative_to_project":true
},
{
"type":"file",
"path":"..\\Web\\wschat.htm",
"relative_to_project":true
},
{
"type":"file",
"path":"..\\Web\\wschat.js",
"relative_to_project":true
},
{
"type":"file",
"path":"..\\Web\\wschat_client.js",
"relative_to_project":true
}
]
}
```

Si el Asesor Experto estuviera dentro de la carpeta *Shared Projects*, por ejemplo en *MQL5/Shared Projects/MQL5Book/wsChat/*, tras una compilación correcta, su archivo ex5 se movería automáticamente a la carpeta *MQL5/Experts/Shared Projects/MQL5Book/wsChat/*.

Inicio del servidor *node.exe wschat.js*. Ahora puede ejecutar un par de copias del Asesor Experto en diferentes gráficos. Básicamente, el servicio implica la «comunicación» entre distintos terminales e incluso distintos ordenadores, pero también puede probarlo desde un solo terminal.

He aquí un ejemplo de comunicación entre los gráficos EURUSD y GBPUSD.

```
(EURUSD,H1)
(EURUSD,H1)   Opening...
(EURUSD,H1)   Connecting to localhost:9000
(EURUSD,H1)   Buffer: 'HTTP/1.1 101 Switching Protocols
(EURUSD,H1)   Upgrade: websocket
(EURUSD,H1)   Connection: Upgrade
(EURUSD,H1)   Sec-WebSocket-Accept: Dg+aQdCBwNExE5mEQsfk5w9J+uE=
(EURUSD,H1)
(EURUSD,H1)   '
(EURUSD,H1)   Headers:
(EURUSD,H1)                                   [,0]                            [,1]
(EURUSD,H1)   [0,] "upgrade"                       "websocket"
(EURUSD,H1)   [1,] "connection"                    "Upgrade"
(EURUSD,H1)   [2,] "sec-websocket-accept"          "Dg+aQdCBwNExE5mEQsfk5w9J+uE="
(EURUSD,H1)    > Connected ws://localhost:9000/
(EURUSD,H1)   Alert: server#Hello, user1
(GBPUSD,H1)
(GBPUSD,H1)   Opening...
(GBPUSD,H1)   Connecting to localhost:9000
(GBPUSD,H1)   Buffer: 'HTTP/1.1 101 Switching Protocols
(GBPUSD,H1)   Upgrade: websocket
(GBPUSD,H1)   Connection: Upgrade
(GBPUSD,H1)   Sec-WebSocket-Accept: NZENnc8p05T4amvngeop/e/+gFw=
(GBPUSD,H1)
(GBPUSD,H1)   '
(GBPUSD,H1)   Headers:
(GBPUSD,H1)                                   [,0]                            [,1]
(GBPUSD,H1)   [0,] "upgrade"                       "websocket"
(GBPUSD,H1)   [1,] "connection"                    "Upgrade"
(GBPUSD,H1)   [2,] "sec-websocket-accept"          "NZENnc8p05T4amvngeop/e/+gFw="
(GBPUSD,H1)    > Connected ws://localhost:9000/
(GBPUSD,H1)   Alert: server#Hello, user2
(EURUSD,H1)   Alert: user1#I'm typing this on EURUSD chart
(GBPUSD,H1)   Alert: user1#I'm typing this on EURUSD chart
(GBPUSD,H1)   Alert: user2#Got it on GBPUSD chart!
(EURUSD,H1)   Alert: user2#Got it on GBPUSD chart!
```

Como nuestros mensajes se envían a todo el mundo, incluido el remitente, se duplican en el registro, pero en gráficos diferentes.

La comunicación también es visible en el lado del servidor.

```
C:\Program Files\MetaTrader 5\MQL5\Projects\MQL5Book\Web>node wschat.js
{
  connection: 'Upgrade',
  host: 'localhost',
  'sec-websocket-key': 'ZBiifDgZZxrYB3HDz5S6Vg==',
  origin: 'http://localhost',
  'sec-websocket-version': '13',
  upgrade: 'websocket'
}
New user: 1
{
  connection: 'Upgrade',
  host: 'localhost',
  'sec-websocket-key': 'a2UZhk69UluW1XblUtIqCw==',
  origin: 'http://localhost',
  'sec-websocket-version': '13',
  upgrade: 'websocket'
}
New user: 2
1 : I'm typing this on EURUSD chart
2 : Got it on GBPUSD chart!
```

Registro del servidor del servicio de chat

Ahora tenemos todos los componentes técnicos para organizar el servicio de señales de trading.

7.8.8 Servicio de señales de trading y página web de prueba

El servicio de señales de trading es técnicamente idéntico al servicio de chat; sin embargo, sus usuarios (o más bien las conexiones de los clientes) deben desempeñar uno de estos dos papeles:

- Proveedor de mensajes
- Consumidor de mensajes

Además, la información no debe estar disponible para todo el mundo, sino funcionar según algún sistema de suscripción.

Para garantizarlo, al conectarse al servicio, se pedirá a los usuarios que faciliten ciertos datos identificativos que difieren en función del papel o rol.

El proveedor debe especificar un identificador de señal pública (PUB_ID) que sea único entre todas las señales. Básicamente, una misma persona podría generar potencialmente más de una señal y, por tanto, debería poder obtener múltiples identificadores. En este sentido, no complicaremos el servicio introduciendo por separado identificadores del proveedor (como una persona concreta) e identificadores de sus señales. En lugar de ello, sólo se admitirán identificadores de señal. Para un verdadero servicio de señalización, hay que resolver este problema, junto con el de la autorización, que dejamos fuera de este libro.

El identificador será necesario para publicitarlo o simplemente transmitirlo a las personas interesadas en abonarse a esta señal. Pero no «todo el mundo» debería poder acceder a la señal conociendo sólo el identificador público. En el caso más sencillo, esto sería aceptable para la supervisión de cuentas abiertas, pero demostraremos la opción de restringir el acceso específicamente en el contexto de las señales.

Para ello, el proveedor debe proporcionarle al servidor una clave secreta (PUB_KEY) conocida sólo por él, pero no por el público en general. Esta clave será necesaria para generar la clave de acceso de un suscriptor concreto.

El consumidor (suscriptor) también debe tener un identificador único (SUB_ID, y aquí también lo haremos sin autorización). Para suscribirse a la señal deseada, el usuario debe indicarle al proveedor de

la señal el identificador (en la práctica, se entiende que en la misma fase es necesario confirmar el pago, y normalmente todo esto está automatizado por el servidor). El proveedor forma una instantánea compuesta por el identificador del proveedor, el identificador del suscriptor y la clave secreta. En nuestro servicio, esto se hará calculando el hash SHA256 a partir de la cadena PUB_ID:PUB_KEY:SUB_ID, tras lo cual los bytes resultantes se convertirán a una cadena en formato hexadecimal. Será la clave de acceso (SUB_KEY o ACCESS_KEY) a la señal de un determinado proveedor para un determinado suscriptor. El proveedor (y en los sistemas reales, el propio servidor automáticamente) reenvía esta clave al suscriptor.

Así, al conectarse al servicio, el suscriptor deberá especificar el identificador del suscriptor (SUB_ID), el identificador de la señal deseada (PUB_ID) y la clave de acceso (SUB_KEY). Como el servidor conoce la clave secreta del proveedor, puede recalcular la clave de acceso para la combinación dada de PUB_ID y SUB_ID, y compararla con la SUB_KEY proporcionada. Una coincidencia significa que el proceso normal de mensajería continúa. La diferencia dará lugar a un mensaje de error y a la desconexión del pseudosuscriptor del servicio.

Es importante señalar que en nuestra demostración, en aras de la simplicidad, no existe un registro normal de usuarios y señales, por lo que la elección de los identificadores es arbitraria. Sólo es importante para nosotros hacer un seguimiento de la unicidad de los identificadores para saber a quién y de quién enviar información en línea. Así pues, nuestro servicio no garantiza que el identificador, por ejemplo, «Super Trend» pertenezca al mismo usuario ayer, hoy y mañana. La reserva de nombres se hace según el principio de que a quien madruga Dios le ayuda. Mientras un proveedor esté conectado continuamente con el identificador dado, la señal se entrega. Si el proveedor se desconecta, el identificador pasa a estar disponible para su selección en cualquier conexión siguiente.

El único identificador que siempre estará ocupado es «Server»: el servidor lo utiliza para enviar sus mensajes de estado de conexión.

Para generar claves de acceso en la carpeta del servidor, existe un sencillo JavaScript *access.js*. Cuando lo ejecute en la línea de comandos, deberá pasar como único parámetro una cadena del tipo PUB_ID:PUB_KEY:SUB_ID (identificadores y la clave secreta entre ellos, conectados por el símbolo ':')

Si no se especifica el parámetro, el script genera una clave de acceso para algunos identificadores de demostración (PUB_ID_001, SUB_ID_100) y un secreto (PUB_KEY_FFF).

```
// JavaScript
const args = process.argv.slice(2);
const input = args.length > 0 ? args[0] : 'PUB_ID_001:PUB_KEY_FFF:SUB_ID_100';
console.log('Hashing "', input, '"');
const crypto = require('crypto');
console.log(crypto.createHash('sha256').update(input).digest('hex'));
```

Al ejecutar el script con el comando

```
node access.js PUB_ID_001:PUB_KEY_FFF:SUB_ID_100
```

obtenemos este resultado:

```
fd3f7a105eae8c2d9afce0a7a4e11bf267a40f04b7c216dd01cf78c7165a2a5a
```

Por cierto: puede comprobar y repetir este algoritmo en MQL5 puro utilizando la función *CryptEncode*.

Una vez analizada la parte conceptual, pasemos a la aplicación práctica.

El script de servidor del servicio de señalización se colocará en el archivo *MQL5/Experts/MQL5Book/p7/Web/wspubsub.js*. Configurar servidores en él es lo mismo que hicimos antes. No obstante, tendrá además que conectar el mismo módulo «crypto» que se utilizó en *access.js*. La página de inicio se llamará *wspubsub.htm*.

```
// JavaScript
const crypto = require('crypto');
...
http1.createServer(options, function (req, res)
{
   ...
   if(req.url == '/')
   {
      req.url = "wspubsub.htm";
   }
   ...
});
```

En lugar de un mapa de clientes conectados, definiremos dos mapas, por separado para proveedores y consumidores de señales.

```
// JavaScript
const publishers = new Map();
const subscribers = new Map();
```

En ambos mapas, la clave es el ID del proveedor, pero el primero almacena los objetos de los proveedores, y el segundo almacena los objetos de los suscriptores suscritos a cada proveedor (arrays de objetos).

Para transferir identificadores y claves durante el «handshake», utilizaremos un encabezado especial permitida por la especificación WebSockets, concretamente Sec-Websocket-Protocol. Acordemos que los identificadores y las claves se pegarán con el símbolo '-': en el caso de un proveedor, se espera una cadena como X-MQL5-publisher-PUB_ID-PUB_KEY, y en el caso de un suscriptor, esperamos X-MQL5-subscriber-SUB_ID-PUB_ID-SUB_KEY.

Cualquier intento de conexión a nuestro servicio sin el encabezado Sec-Websocket-Protocol: X-MQL5-... se detendrá por cierre inmediato.

En el nuevo objeto cliente (en el parámetro del manejador de eventos de «conexión» *onConnect(client)*) este título es fácil de extraer de la propiedad *client.protocol*.

Vamos a mostrar el procedimiento de registro y envío de mensajes del proveedor de señales de forma simplificada, sin tratamiento de errores (se adjunta el código completo). Es importante tener en cuenta que el texto del mensaje se genera en formato JSON (del que hablaremos con más detalle en la siguiente sección). En particular, el remitente del mensaje se pasa en la propiedad «origin» (además, cuando el mensaje es enviado por el propio servicio, este campo contiene la cadena «Server») y los datos de la aplicación del proveedor se colocan en la propiedad «msg», y esto puede ser no solo texto, sino también una estructura anidada de cualquier contenido.

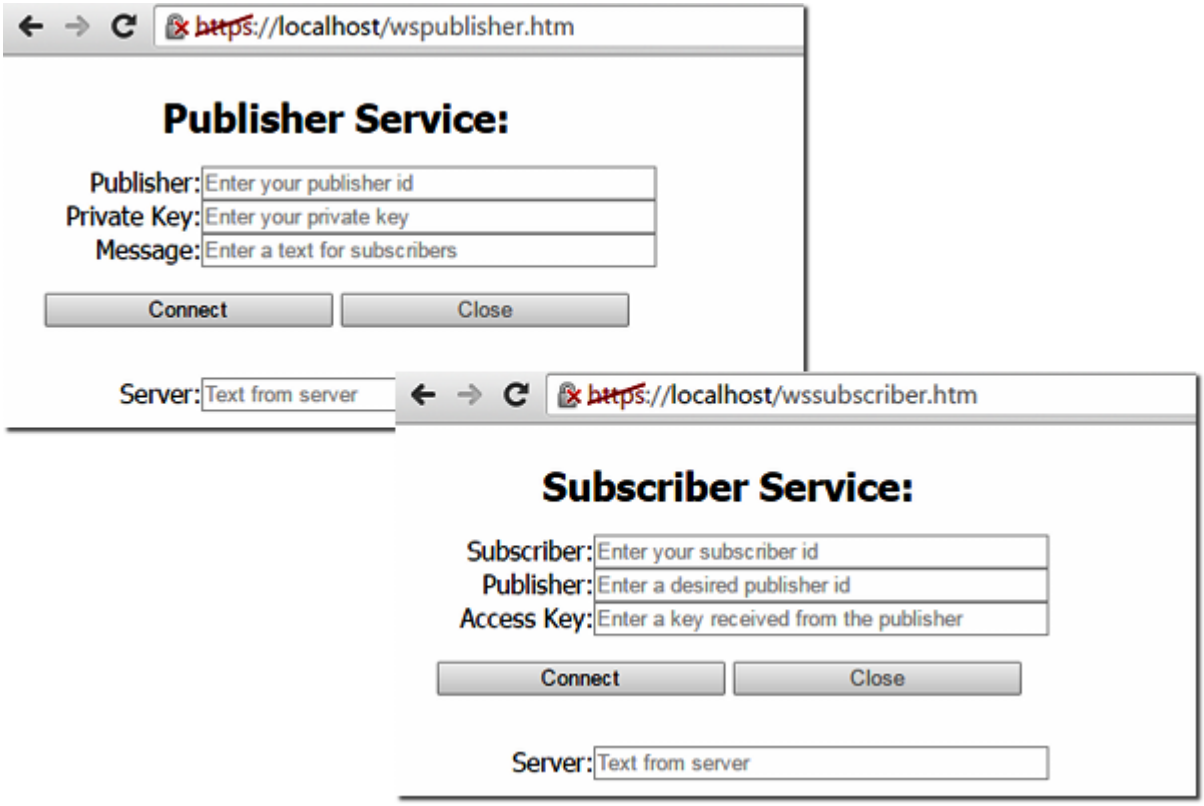
```
// JavaScript
const wsServer = new WebSocket.Server({ server });
wsServer.on('connection', function onConnect(client)
{
   console.log('New user:', ++count, client.protocol);
   if(client.protocol.startsWith('X-MQL5-publisher'))
   {
      const parts = client.protocol.split('-');
      client.id = parts[3];
      client.key = parts[4];
      publishers.set(client.id, client);
      client.send('{"origin":"Server", "msg":"Hello, publisher ' + client.id + '"}');
      client.on('message', function(message)
      {
         console.log('%s : %s', client.id, message);

         if(subscribers.get(client.id))
            subscribers.get(client.id).forEach(function(elem)
         {
            elem.send('{"origin":"publisher ' + client.id + '", "msg":'
               + message + '}');
         });
      });
      client.on('close', function()
      {
         console.log('Publisher disconnected:', client.id);
         if(subscribers.get(client.id))
            subscribers.get(client.id).forEach(function(elem)
         {
            elem.close();
         });
         publishers.delete(client.id);
      });
   }
   ...
```

La mitad del algoritmo para los suscriptores es similar, pero aquí tenemos el cálculo de la clave de acceso y su comparación con lo que transmitió el cliente que se conecta, como añadido.

```
// JavaScript
   else if(client.protocol.startsWith('X-MQL5-subscriber'))
   {
      const parts = client.protocol.split('-');
      client.id = parts[3];
      client.pub_id = parts[4];
      client.access = parts[5];
      const id = client.pub_id;
      var p = publishers.get(id);
      if(p)
      {
         const check = crypto.createHash('sha256').update(id + ':' + p.key + ':'
            + client.id).digest('hex');
         if(check != client.access)
         {
            console.log(`Bad credentials: '${client.access}' vs '${check}'`);
            client.send('{"origin":"Server", "msg":"Bad credentials, subscriber '
               + client.id + '"}');
            client.close();
            return;
         }

         var list = subscribers.get(id);
         if(list == undefined)
         {
            list = [];
         }
         list.push(client);
         subscribers.set(id, list);
         client.send('{"origin":"Server", "msg":"Hello, subscriber '
            + client.id + '"}');
         p.send('{"origin":"Server", "msg":"New subscriber ' + client.id + '"}');
      }

      client.on('close', function()
      {
         console.log('Subscriber disconnected:', client.id);
         const list = subscribers.get(client.pub_id);
         if(list)
         {
            if(list.length > 1)
            {
               const filtered = list.filter(function(el) { return el !== client; });
               subscribers.set(client.pub_id, filtered);
            }
            else
            {
               subscribers.delete(client.pub_id);
            }
         }
      });
```

```
}
```

La interfaz de usuario de la página de cliente *wspubsub.htm* invita simplemente a seguir un enlace a una de las dos páginas con formularios para proveedores (*wspublisher.htm* + *wspublisher_client.js*) o suscriptores (*wssubscriber.htm* + *wssubscriber_client.js*).



Páginas web de clientes de prueba del servicio de señales

Su implementación hereda las características de los clientes JavaScript considerados con anterioridad, pero con respecto a la personalización del encabezado Sec-Websocket-Protocol: X-MQL5- y un matiz más.

Hasta ahora, hemos intercambiado mensajes de texto sencillos. Pero para un servicio de señalización, necesitará transferir mucha información estructurada, y lo más adecuado para ello es JSON. Por lo tanto, los clientes pueden analizar JSON, aunque no lo utilizan para el fin previsto, porque aunque en JSON se encuentre una orden para comprar o vender un ticker específico con una cantidad determinada, el navegador no sabe cómo hacerlo.

Necesitaremos añadir compatibilidad JSON a nuestro cliente de servicio de señales en MQL5. Mientras tanto, puede ejecutar en el servidor *wspubsub.js* y probar la conexión selectiva de los proveedores de señal y los consumidores de acuerdo con los detalles especificados por ellos. Le sugerimos que lo haga usted mismo, por su propio beneficio.

7.8.9 Programa cliente de servicios de señales en MQL5

Así pues, conforme a lo decidido, el texto de los mensajes de servicio estará en formato JSON.

En la versión más común, JSON es una descripción en texto de un objeto, similar a como se hace para las estructuras en MQL5. El objeto se encierra entre llaves, dentro de las cuales se escriben sus propiedades separadas por comas: cada propiedad tiene un identificador entre comillas, seguido de dos

puntos y el valor de la propiedad. Aquí se admiten propiedades de varios tipos primitivos: cadenas, números enteros y reales, booleanos *true/false* y valor vacío *null*. Además, el valor de la propiedad puede ser, a su vez, un objeto o un array. Los arrays se describen mediante corchetes, dentro de los cuales los elementos se separan por comas. Por ejemplo:

```
{
   "string": "this is a text",
   "number": 0.1,
   "integer": 789735095,
   "enabled": true,
   "subobject" :
   {
      "option": null
   },
   "array":
   [
      1, 2, 3, 5, 8
   ]
}
```

Básicamente, el array del nivel superior también es JSON válido. Por ejemplo:

```
[
   {
      "command": "buy",
      "volume": 0.1,
      "symbol": "EURUSD",
      "price": 1.0
   },
   {
      "command": "sell",
      "volume": 0.01,
      "symbol": "GBPUSD",
      "price": 1.5
   }
]
```

Para reducir el tráfico en los protocolos de aplicación que utilizan JSON, es habitual abreviar los nombres de los campos a varias letras (a menudo a una).

Los nombres de las propiedades y los valores de las cadenas van entre comillas dobles. Si desea especificar una cotización dentro de una cadena, debe escaparse con una barra invertida.

El uso de JSON hace que el protocolo sea versátil y extensible. Por ejemplo, para el servicio que se está diseñando (señales de trading y, en un caso más general, copia del estado de la cuenta), puede suponerse la siguiente estructura de mensajes:

```
{
  "origin": "publisher_id",    // message sender ("Server" in technical message)
  "msg" :                      // message (text or JSON) as received from the sender
   {
     "trade" :                 // current trading commands (if there is a signal)
      {
        "operation": ...,      // buy/sell/close
         "symbol": "ticker",
         "volume": 0.1,
        ... // other signal parameters
      },
     "account":                // account status
      {
        "positions":           // positions
         {
           "n": 10,            // number of open positions
           [ { ... },{ ... } ] // array of properties of open positions
         },
        "pending_orders":      // pending orders
         {
            "n": ...
            [ { ... } ]
         }
         "drawdown": 2.56,
         "margin_level": 12345,
        ... // other status parameters
      },
     "hardware":               // remote control of the "health" of the PC
      {
         "memory": ...,
         "ping_to_broker": ...
      }
   }
}
```

Algunas de estas características pueden o no admitir implementaciones específicas de programas cliente (todo lo que no «entiendan», simplemente lo ignorarán). Además, con la condición de que no haya conflictos en los nombres de las propiedades del mismo nivel, cada proveedor de información puede añadir sus propios datos específicos a JSON. El servicio de mensajería se limitará a reenviar esta información. Por supuesto, el programa en el lado receptor debe ser capaz de interpretar estos datos específicos.

El libro viene con un analizador JSON llamado *ToyJson* («toy» JSON, archivo *toyjson.mqh*) que es pequeño e ineficiente y no admite todas las capacidades de la especificación del formato (por ejemplo, en términos de procesamiento de secuencias *escape*). Fue escrito específicamente para este servicio de demostración, ajustado a la estructura esperada, no muy compleja, de la información sobre las señales de trading. No lo describiremos en detalle aquí, y los principios de su uso quedarán claros en el código fuente del cliente MQL del servicio de señales.

Para sus proyectos y para el desarrollo posterior de este proyecto, puede elegir otros analizadores JSON disponibles en la base de código del sitio *mql5.com*.

El objeto de clase *JsValue* describe un elemento (contenedor o propiedad) por *ToyJson*. Se han definido varias sobrecargas del método *put(key, value)*, que se pueden utilizar para la adición de propiedades internas con nombre como en un objeto JSON o *put(value)*, para añadir un valor como en un array JSON. Además, este objeto puede representar un único valor de un tipo primitivo. Para leer las propiedades de un objeto JSON, puede aplicar a *JsValue* una notación del operador [] seguida del nombre de la propiedad requerida entre paréntesis. Obviamente, los índices enteros se admiten para el acceso dentro de un array JSON.

Una vez formada la configuración necesaria de objetos relacionados *JsValue*, puede serializarla en texto JSON utilizando el método *stringify(string&buffer)*.

La segunda clase de *toyjson.mqh -JsParser-* permite realizar la operación inversa: convertir el texto con la descripción JSON en una estructura jerárquica de objetos *JsValue*.

Teniendo en cuenta las clases para trabajar con JSON, vamos a empezar a escribir un Asesor Experto *MQL5/Experts/MQL5Book/p7/wsTradeCopier/wstradecopier.mq5*, que podrá desempeñar ambos papeles en el servicio de copia de operaciones: un proveedor de información sobre las operaciones de trading realizadas en la cuenta, o un receptor de esta información del servicio para reproducir dichas operaciones.

El volumen y contenido de la información enviada queda, desde un punto de vista político, a discreción del proveedor y puede diferir significativamente en función del escenario (propósito) de uso del servicio. En concreto, es posible copiar sólo las operaciones en curso o todo el saldo de la cuenta junto con las órdenes pendientes y los niveles de protección. En nuestro ejemplo, sólo indicaremos la implementación técnica de la transferencia de información, y luego podrá elegir un conjunto específico de objetos y propiedades a su discreción.

En el código, describiremos 3 estructuras que se heredan de las estructuras integradas y que proporcionan «empaquetado» de información en JSON:

- MqlTradeRequestWeb – MqlTradeRequest
- MqlTradeResultWeb – MqlTradeResult
- DealMonitorWeb – DealMonitor*

La última estructura de la lista, en sentido estricto, no está integrada, sino que la definimos nosotros en el archivo *DealMonitor.mqh*, pero se rellena en el conjunto estándar de propiedades de la transacción.

El constructor de cada una de las estructuras derivadas rellena los campos basándose en la fuente primaria transmitida (solicitud de operación, su resultado o transacción). Cada estructura implementa el método *asJsValue*, que devuelve un puntero al objeto *JsValue* que refleja todas las propiedades de la estructura: se añaden al objeto JSON mediante el método *JsValue::put*. Por ejemplo, he aquí cómo se hace en el caso de *MqlTradeRequest*:

```
struct MqlTradeRequestWeb: public MqlTradeRequest
{
   MqlTradeRequestWeb(const MqlTradeRequest &r)
   {
      ZeroMemory(this);
      action = r.action;
      magic = r.magic;
      order = r.order;
      symbol = r.symbol;
      volume = r.volume;
      price = r.price;
      stoplimit = r.stoplimit;
      sl = r.sl;
      tp = r.tp;
      type = r.type;
      type_filling = r.type_filling;
      type_time = r.type_time;
      expiration = r.expiration;
      comment = r.comment;
      position = r.position;
      position_by = r.position_by;
   }

   JsValue *asJsValue() const
   {
      JsValue *req = new JsValue();
      // main block: action, symbol, type
      req.put("a", VerboseJson ? EnumToString(action) : (string)action);
      if(StringLen(symbol) != 0) req.put("s", symbol);
      req.put("t", VerboseJson ? EnumToString(type) : (string)type);

      // volumes
      if(volume != 0) req.put("v", TU::StringOf(volume));
      req.put("f", VerboseJson ? EnumToString(type_filling) : (string)type_filling);

      // block with prices
      if(price != 0) req.put("p", TU::StringOf(price));
      if(stoplimit != 0) req.put("x", TU::StringOf(stoplimit));
      if(sl != 0) req.put("sl", TU::StringOf(sl));
      if(tp != 0) req.put("tp", TU::StringOf(tp));

      // block of pending orders
      if(TU::IsPendingType(type))
      {
         req.put("t", VerboseJson ? EnumToString(type_time) : (string)type_time);
         if(expiration != 0) req.put("d", TimeToString(expiration));
      }

      // modification block
      if(order != 0) req.put("o", order);
      if(position != 0) req.put("q", position);
```

```
      if(position_by != 0) req.put("b", position_by);

      // helper block
      if(magic != 0) req.put("m", magic);
      if(StringLen(comment)) req.put("c", comment);

      return req;
   }
};
```

Transferimos todas las propiedades a JSON (esto es adecuado para el servicio de supervisión de cuentas), pero puede dejar sólo un conjunto limitado.

Para las propiedades que son enumeraciones, hemos proporcionado dos formas de representarlas en JSON: como un número entero y como un nombre de cadena de un elemento de enumeración. La elección del método se realiza mediante el parámetro de entrada *VerboseJson* (lo ideal es que no se escriba en el código de la estructura directamente, sino a través de un parámetro del constructor).

```
input bool VerboseJson = false;
```

Pasar sólo números simplificaría la codificación porque, en el lado receptor, basta con convertirlos al tipo de enumeración deseado para realizar acciones «espejo». Sin embargo, los números dificultan la percepción de la información por parte de una persona, que puede necesitar analizar la situación (mensaje). Por lo tanto, tiene sentido apoyar una opción para la representación de cadena, por ser más «fácil de usar», aunque requiere operaciones adicionales en el algoritmo de recepción.

Los parámetros de entrada también especifican la dirección del servidor, el rol de la aplicación y los detalles de conexión por separado para el proveedor y el suscriptor.

```
enum TRADE_ROLE
{
   TRADE_PUBLISHER,  // Trade Publisher
   TRADE_SUBSCRIBER  // Trade Subscriber
};

input string Server = "ws://localhost:9000/";
input TRADE_ROLE Role = TRADE_PUBLISHER;
input bool VerboseJson = false;
input group "Publisher";
input string PublisherID = "PUB_ID_001";
input string PublisherPrivateKey = "PUB_KEY_FFF";
input string SymbolFilter = ""; // SymbolFilter (empty - current, '*' - any)
input ulong MagicFilter = 0;    // MagicFilter (0 - any)
input group "Subscriber";
input string SubscriberID = "SUB_ID_100";
input string SubscribeToPublisherID = "PUB_ID_001";
input string SubscriberAccessKey = "fd3f7a105eae8c2d9afce0a7a4e11bf267a40f04b7c216dd0
input string SymbolSubstitute = "EURUSD=GBPUSD"; // SymbolSubstitute (<from>=<to>,...
input ulong SubscriberMagic = 0;
```

Los parámetros *SymbolFilter* y *MagicFilter* del grupo de proveedores le permiten limitar la actividad de trading supervisada a un símbolo y un número mágico determinados. Un valor vacío en *SymbolFilter* significa controlar solo el símbolo actual del gráfico; para interceptar cualquier operación de trading,

introduzca el símbolo '*'. El proveedor de señales utilizará para ello la función *FilterMatched*, que acepta el símbolo y el número mágico de la transacción.

```
bool FilterMatched(const string s, const ulong m)
{
   if(MagicFilter != 0 && MagicFilter != m)
   {
      return false;
   }

   if(StringLen(SymbolFilter) == 0)
   {
      if(s != _Symbol)
      {
         return false;
      }
   }
   else if(SymbolFilter != s && SymbolFilter != "*")
   {
      return false;
   }

   return true;
}
```

El parámetro *SymbolSubstitute* del grupo de entrada del suscriptor permite sustituir el símbolo recibido en los mensajes por otro, que se utilizará para la operación de copia. Esta función es útil si los nombres de los tickers del mismo instrumento financiero difieren entre los distintos brókeres. Pero este parámetro también cumple la función de filtro permisivo para señales repetitivas: sólo se negociarán los símbolos aquí especificados. Por ejemplo, para permitir la negociación de señales para el símbolo EURUSD (incluso sin sustitución de ticker), debe establecer la cadena «EURUSD=EURUSD» en el parámetro. El símbolo de los mensajes de señalización se indica a la izquierda del signo «=», y el símbolo de trading, a la derecha.

La lista de sustitución de caracteres es procesada por la función *FillSubstitutes* durante la inicialización y luego utilizada para sustituir y resolver la operación de trading mediante la función *FindSubstitute*.

```
string Substitutes[][2];

void FillSubstitutes()
{
   string list[];
   const int n = StringSplit(SymbolSubstitute, ',', list);
   ArrayResize(Substitutes, n);
   for(int i = 0; i < n; ++i)
   {
      string pair[];
      if(StringSplit(list[i], '=', pair) == 2)
      {
         Substitutes[i][0] = pair[0];
         Substitutes[i][1] = pair[1];
      }
      else
      {
         Print("Wrong substitute: ", list[i]);
      }
   }
}

string FindSubstitute(const string s)
{
   for(int i = 0; i < ArrayRange(Substitutes, 0); ++i)
   {
      if(Substitutes[i][0] == s) return Substitutes[i][1];
   }
   return NULL;
}
```

Para comunicarnos con el servicio, definimos una clase derivada de *WebSocketClient*. Se necesita, en primer lugar, para empezar a el trading con una señal cuando llega un mensaje al manejador *onMessage*. Volveremos sobre esta cuestión un poco más adelante, después de considerar la formación y el envío de señales en el lado del proveedor.

```
class MyWebSocket: public WebSocketClient<Hybi>
{
public:
   MyWebSocket(const string address): WebSocketClient(address) { }

   void onMessage(IWebSocketMessage *msg) override
   {
      ...
   }
};

MyWebSocket wss(Server);
```

La inicialización en *OnInit* activa el temporizador (para una llamada periódica *wss.checkMessages(false)*) y la preparación de encabezados personalizados con detalles del usuario,

dependiendo del rol seleccionado. A continuación, abrimos la conexión con la llamada a *wss.open(custom)*.

```
int OnInit()
{
   FillSubstitutes();
   EventSetTimer(1);
   wss.setTimeOut(1000);
   Print("Opening...");
   string custom;
   if(Role == TRADE_PUBLISHER)
   {
      custom = "Sec-Websocket-Protocol: X-MQL5-publisher-"
         + PublisherID + "-" + PublisherPrivateKey + "\r\n";
   }
   else
   {
      custom = "Sec-Websocket-Protocol: X-MQL5-subscriber-"
         + SubscriberID + "-" + SubscribeToPublisherID
         + "-" + SubscriberAccessKey + "\r\n";
   }
   return wss.open(custom) ? INIT_SUCCEEDED : INIT_FAILED;
}
```

El mecanismo de copia, es decir, interceptar transacciones y enviar información sobre ellas a un servicio web, se pone en marcha en el manejador *OnTradeTransaction*. Como sabemos, esta no es la única forma y sería posible analizar la «instantánea» del estado de la cuenta en *OnTrade*.

```
void OnTradeTransaction(const MqlTradeTransaction &transaction,
   const MqlTradeRequest &request,
   const MqlTradeResult &result)
{
   if(transaction.type == TRADE_TRANSACTION_REQUEST)
   {
      Print(TU::StringOf(request));
      Print(TU::StringOf(result));
      if(result.retcode == TRADE_RETCODE_PLACED           // successful action
         || result.retcode == TRADE_RETCODE_DONE
         || result.retcode == TRADE_RETCODE_DONE_PARTIAL)
      {
         if(FilterMatched(request.symbol, request.magic))
         {
            ... // see next block of code
         }
      }
   }
}
```

Hacemos un seguimiento de los eventos sobre solicitudes de operaciones de trading completadas con éxito que satisfacen las condiciones de los filtros especificados. A continuación, las estructuras de la solicitud, el resultado de la solicitud y la transacción se convierten en objetos JSON. Todos ellos se colocan en un contenedor común *msg* bajo los nombres «req», «res» y «deal», respectivamente. Recuerde que el propio contenedor se incluirá en el mensaje del servicio web como la propiedad «msg».

```
// container to attach to service message will be visible as "msg" property:
// {"origin" : "this_publisher_id", "msg" : { our data is here }}
            JsValue msg;
            MqlTradeRequestWeb req(request);
            msg.put("req", req.asJsValue());

            MqlTradeResultWeb res(result);
            msg.put("res", res.asJsValue());

            if(result.deal != 0)
            {
               DealMonitorWeb deal(result.deal);
               msg.put("deal", deal.asJsValue());
            }
            ulong tickets[];
            Positions.select(tickets);
            JsValue pos;
            pos.put("n", ArraySize(tickets));
            msg.put("pos", &pos);
            string buffer;
            msg.stringify(buffer);

            Print(buffer);

            wss.send(buffer);
```

Una vez lleno, el contenedor se muestra como una cadena en *buffer*, se imprime en el registro y se envía al servidor.

Podemos añadir otra información a este contenedor: el estado de la cuenta (reducción, carga), el número y las propiedades de las órdenes pendientes, etc. Así, sólo para demostrar las posibilidades de ampliar el contenido de los mensajes, hemos añadido el número de posiciones abiertas más arriba. Para seleccionar las posiciones según los filtros, utilizamos el objeto de clase *PositionFilter* (*PositionFilter.mqh*):

```
PositionFilter Positions;

int OnInit()
{
   ...
   if(MagicFilter) Positions.let(POSITION_MAGIC, MagicFilter);
   if(SymbolFilter == "") Positions.let(POSITION_SYMBOL, _Symbol);
   else if(SymbolFilter != "*") Positions.let(POSITION_SYMBOL, SymbolFilter);
   ...
}
```

Básicamente, para aumentar la fiabilidad, tiene sentido que los copiadores analicen el estado de las posiciones, y no sólo intercepten las transacciones.

Esto concluye la consideración de la parte del Asesor Experto que está involucrada en el papel de proveedor de señales.

Como suscriptor, como ya hemos anunciado, el Asesor Experto recibe mensajes en el método *MyWebSocket::onMessage*. Aquí el mensaje entrante es analizado con *JsParser::jsonify*, y el contenedor que se formó por el lado transmisor es recuperado de la propiedad *obj["msg"]*.

```
class MyWebSocket: public WebSocketClient<Hybi>
{
public:
   void onMessage(IWebSocketMessage *msg) override
   {
      Alert(msg.getString());
      JsValue *obj = JsParser::jsonify(msg.getString());
      if(obj && obj["msg"])
      {
         obj["msg"].print();
         if(!RemoteTrade(obj["msg"])) { /* error processing */ }
         delete obj;
      }
      delete msg;
   }
};
```

La función *RemoteTrade* ejecuta las operaciones de trading y análisis de señales. Aquí se da con abreviaturas, sin manejar posibles errores. La función ofrece compatibilidad con ambas formas de representar enumeraciones: como valores enteros o como nombres de elementos de cadena. El objeto JSON entrante se «examina» en busca de las propiedades necesarias (comandos y atributos de señal) aplicando el operador [], incluso varias veces consecutivas (para acceder a objetos JSON anidados).

```
bool RemoteTrade(JsValue *obj)
{
   bool success = false;

   if(obj["req"]["a"] == TRADE_ACTION_DEAL
      || obj["req"]["a"] == "TRADE_ACTION_DEAL")
   {
      const string symbol = FindSubstitute(obj["req"]["s"].s);
      if(symbol == NULL)
      {
         Print("Suitable symbol not found for ", obj["req"]["s"].s);
         return false; // not found or forbidden
      }

      JsValue *pType = obj["req"]["t"];
      if(pType == ORDER_TYPE_BUY || pType == ORDER_TYPE_SELL
         || pType == "ORDER_TYPE_BUY" || pType == "ORDER_TYPE_SELL")
      {
         ENUM_ORDER_TYPE type;
         if(pType.detect() >= JS_STRING)
         {
            if(pType == "ORDER_TYPE_BUY") type = ORDER_TYPE_BUY;
            else type = ORDER_TYPE_SELL;
         }
         else
         {
            type = obj["req"]["t"].get<ENUM_ORDER_TYPE>();
         }

         MqlTradeRequestSync request;
         request.deviation = 10;
         request.magic = SubscriberMagic;
         request.type = type;

         const double lot = obj["req"]["v"].get<double>();
         JsValue *pDir = obj["deal"]["entry"];
         if(pDir == DEAL_ENTRY_IN || pDir == "DEAL_ENTRY_IN")
         {
            success = request._market(symbol, lot) && request.completed();
            Alert(StringFormat("Trade by subscription: market entry %s %s %s - %s",
               EnumToString(type), TU::StringOf(lot), symbol,
               success ? "Successful" : "Failed"));
         }
         else if(pDir == DEAL_ENTRY_OUT || pDir == "DEAL_ENTRY_OUT")
         {
            // closing action assumes the presence of a suitable position, look for i
            PositionFilter filter;
            int props[] = {POSITION_TICKET, POSITION_TYPE, POSITION_VOLUME};
            Tuple3<long,long,double> values[];
            filter.let(POSITION_SYMBOL, symbol).let(POSITION_MAGIC,
               SubscriberMagic).select(props, values);
```

```
            for(int i = 0; i < ArraySize(values); ++i)
            {
              // need a position that is opposite in direction to the deal
               if(!TU::IsSameType((ENUM_ORDER_TYPE)values[i]._2, type))
               {
                  // you need enough volume (exactly equal here!)
                  if(TU::Equal(values[i]._3, lot))
                  {
                     success = request.close(values[i]._1, lot) && request.completed(
                     Alert(StringFormat("Trade by subscription: market exit %s %s %s
                        EnumToString(type), TU::StringOf(lot), symbol,
                        success ? "Successful" : "Failed"));
                  }
               }
            }

            if(!success)
            {
               Print("No suitable position to close");
            }
         }
      }
   }
   return success;
}
```

Esta implementación no analiza el precio de transacción, las posibles restricciones en el lote, los niveles de stop ni otros momentos. Simplemente repetimos la operación al precio local actual. Además, al cerrar una posición, se comprueba la igualdad exacta del volumen, lo que es adecuado para las cuentas de cobertura, pero no para la compensación, donde es posible un cierre parcial si el volumen de la transacción es inferior a la posición (y quizá más, en caso de una inversión, pero la opción DEAL_ENTRY_INOUT no se admite aquí). Todos estos puntos deben ultimarse para su aplicación real.

Vamos a iniciar el servidor *node.exe wspubsub.js* y dos copias del Asesor Experto *wstradecopier.mq5* en diferentes gráficos, en el mismo terminal. El escenario habitual supone que el Asesor Experto necesita ser lanzado en diferentes cuentas, pero una opción «paradójica» también es adecuada para comprobar el rendimiento: copiaremos señales de un símbolo a otro.

En una copia del Asesor Experto, dejaremos la configuración por defecto, con el papel del editor. Debe colocarse en el gráfico EURUSD. En la segunda copia que se ejecuta en el gráfico GBPUSD, cambiamos el rol a suscriptor. La cadena «EURUSD=GBPUSD» en el parámetro de entrada *SymbolSubstitute* permite operar GBPUSD con señales EURUSD.

Los datos de conexión se registrarán, con las cabeceras HTTP y los saludos que ya hemos visto, así que los omitiremos.

Compremos EURUSD y asegurémonos de que se «duplica» en el mismo volumen para GBPUSD.

A continuación se muestran fragmentos del registro (tenga en cuenta que debido al hecho de que ambos Asesores Expertos trabajan en la misma copia del terminal, los mensajes de transacción se enviarán a ambos gráficos y por lo tanto, para facilitar el análisis del registro, puede establecer alternativamente los filtros «EURUSD» y « USDUSD»):

```
(EURUSD,H1) TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_BUY, V=0.01, ORDER_FILLING_FOK, @ 0
(EURUSD,H1) DONE, D=1439023682, #=1461313378, V=0.01, @ 0.99886, Bid=0.99886, Ask=0.9
(EURUSD,H1) {"req" : {"a" : "TRADE_ACTION_DEAL", "s" : "EURUSD", "t" : "ORDER_TYPE_BU
» "f" : "ORDER_FILLING_FOK", "p" : 0.99886, "o" : 1461313378}, "res" : {"code" : 1000
» "o" : 1461313378, "v" : 0.01, "p" : 0.99886, "b" : 0.99886, "a" : 0.99886}, "deal"
» "o" : 1461313378, "t" : "2022.09.19 16:45:50", "tmsc" : 1663605950086, "type" : "DE
» "entry" : "DEAL_ENTRY_IN", "pid" : 1461313378, "r" : "DEAL_REASON_CLIENT", "v" : 0.
» "s" : "EURUSD"}, "pos" : {"n" : 1}}
```

Esto muestra el contenido de la solicitud ejecutada y su resultado, así como un búfer con una cadena JSON enviada al servidor.

Casi instantáneamente, en el lado receptor, en el gráfico GBPUSD, se muestra una alerta con un mensaje del servidor en forma «en bruto» y formateado tras un análisis sintáctico satisfactorio en *JsParser*. En la forma «en bruto», se almacena la propiedad «origen», en la que el servidor nos permite saber quién es la fuente de la señal.

```
(GBPUSD,H1) Alert: {"origin":"publisher PUB_ID_001", "msg":{"req" : {"a" : "TRADE_ACT
» "s" : "EURUSD", "t" : "ORDER_TYPE_BUY", "v" : 0.01, "f" : "ORDER_FILLING_FOK", "p"
» "o" : 1461313378}, "res" : {"code" : 10009, "d" : 1439023682, "o" : 1461313378, "v"
» "p" : 0.99886, "b" : 0.99886, "a" : 0.99886}, "deal" : {"d" : 1439023682, "o" : 146
» "t" : "2022.09.19 16:45:50", "tmsc" : 1663605950086, "type" : "DEAL_TYPE_BUY",
» "entry" : "DEAL_ENTRY_IN", "pid" : 1461313378, "r" : "DEAL_REASON_CLIENT", "v" : 0.
» "p" : 0.99886, "s" : "EURUSD"}, "pos" : {"n" : 1}}}
(GBPUSD,H1)   {
(GBPUSD,H1)    req =
(GBPUSD,H1)    {
(GBPUSD,H1)    a = TRADE_ACTION_DEAL
(GBPUSD,H1)    s = EURUSD
(GBPUSD,H1)    t = ORDER_TYPE_BUY
(GBPUSD,H1)    v = 0.01
(GBPUSD,H1)    f = ORDER_FILLING_FOK
(GBPUSD,H1)    p = 0.99886
(GBPUSD,H1)    o = 1461313378
(GBPUSD,H1)    }
(GBPUSD,H1)    res =
(GBPUSD,H1)    {
(GBPUSD,H1)    code = 10009
(GBPUSD,H1)    d = 1439023682
(GBPUSD,H1)    o = 1461313378
(GBPUSD,H1)    v = 0.01
(GBPUSD,H1)    p = 0.99886
(GBPUSD,H1)    b = 0.99886
(GBPUSD,H1)    a = 0.99886
(GBPUSD,H1)    }
(GBPUSD,H1)    deal =
(GBPUSD,H1)    {
(GBPUSD,H1)    d = 1439023682
(GBPUSD,H1)    o = 1461313378
(GBPUSD,H1)    t = 2022.09.19 16:45:50
(GBPUSD,H1)    tmsc = 1663605950086
(GBPUSD,H1)    type = DEAL_TYPE_BUY
(GBPUSD,H1)    entry = DEAL_ENTRY_IN
(GBPUSD,H1)    pid = 1461313378
(GBPUSD,H1)    r = DEAL_REASON_CLIENT
(GBPUSD,H1)    v = 0.01
(GBPUSD,H1)    p = 0.99886
(GBPUSD,H1)    s = EURUSD
(GBPUSD,H1)    }
(GBPUSD,H1)    pos =
(GBPUSD,H1)    {
(GBPUSD,H1)    n = 1
(GBPUSD,H1)    }
(GBPUSD,H1)   }
(GBPUSD,H1)   Alert: Trade by subscription: market entry ORDER_TYPE_BUY 0.01 GBPUSD -
```

La última de las entradas anteriores indica una transacción exitosa en GBPUSD. En la pestaña de trading de la cuenta, deberían aparecer 2 posiciones.

Transcurrido algún tiempo, cerramos la posición EURUSD, y la posición GBPUSD debería cerrarse automáticamente.

```
(EURUSD,H1) TRADE_ACTION_DEAL, EURUSD, ORDER_TYPE_SELL, V=0.01, ORDER_FILLING_FOK, @
(EURUSD,H1) DONE, D=1439025490, #=1461315206, V=0.01, @ 0.99881, Bid=0.99881, Ask=0.9
(EURUSD,H1) {"req" : {"a" : "TRADE_ACTION_DEAL", "s" : "EURUSD", "t" : "ORDER_TYPE_SE
» "f" : "ORDER_FILLING_FOK", "p" : 0.99881, "o" : 1461315206, "q" : 1461313378}, "res
» "d" : 1439025490, "o" : 1461315206, "v" : 0.01, "p" : 0.99881, "b" : 0.99881, "a" :
» "deal" : {"d" : 1439025490, "o" : 1461315206, "t" : "2022.09.19 16:46:52", "tmsc" :
» "type" : "DEAL_TYPE_SELL", "entry" : "DEAL_ENTRY_OUT", "pid" : 1461313378, "r" : "D
» "v" : 0.01, "p" : 0.99881, "m" : -0.05, "s" : "EURUSD"}, "pos" : {"n" : 0}}
```

Si la transacción tenía un tipo DEAL_ENTRY_IN la primera vez, ahora es DEAL_ENTRY_OUT. La alerta confirma la recepción del mensaje y el cierre con éxito de la posición duplicada.

```
(GBPUSD,H1) Alert: {"origin":"publisher PUB_ID_001", "msg":{"req" : {"a" : "TRADE_ACT
» "s" : "EURUSD", "t" : "ORDER_TYPE_SELL", "v" : 0.01, "f" : "ORDER_FILLING_FOK", "p"
» "o" : 1461315206, "q" : 1461313378}, "res" : {"code" : 10009, "d" : 1439025490, "o"
» "v" : 0.01, "p" : 0.99881, "b" : 0.99881, "a" : 0.99881}, "deal" : {"d" : 143902549
» "o" : 1461315206, "t" : "2022.09.19 16:46:52", "tmsc" : 1663606012990, "type" : "DE
» "entry" : "DEAL_ENTRY_OUT", "pid" : 1461313378, "r" : "DEAL_REASON_CLIENT", "v" : 0
» "p" : 0.99881, "m" : -0.05, "s" : "EURUSD"}, "pos" : {"n" : 0}}}
...
(GBPUSD,H1)   Alert: Trade by subscription: market exit ORDER_TYPE_SELL 0.01 GBPUSD -
```

Por último, junto al Asesor Experto *wstradecopier.mq5*, creamos un archivo de proyecto *wstradecopier.mqproj* para añadirle una descripción y los archivos de servidor necesarios (en el antiguo directorio *MQL5/Experts/p7/MQL5Book/Web/*).

En resumen: hemos organizado un sistema multiusuario técnicamente extensible para intercambiar información de trading a través de un servidor de sockets. Debido a las características técnicas de los web sockets (conexión abierta permanente), esta implementación del servicio de señales es más adecuada para el trading a corto plazo y de alta frecuencia, así como para el control de situaciones de arbitraje en las cotizaciones.

Resolver el problema exigía combinar varios programas en distintas plataformas y conectar un gran número de dependencias, que es lo que suele caracterizar la transición al nivel de proyecto. También se amplía el entorno de desarrollo, que va más allá del compilador y el editor de código fuente. En concreto, la presencia en el proyecto de las partes cliente o servidor suele implicar el trabajo de distintos programadores responsables de ellas. En este caso, los proyectos compartidos en la nube y con control de versiones se hacen indispensables.

Tenga en cuenta que al desarrollar un proyecto en la carpeta *MQL5/Shared Projects* a través de MetaEditor, los archivos de encabezado del directorio estándar *MQL5/Include* no se incluyen en el almacenamiento compartido. Por otro lado, crear una carpeta dedicada *Include* dentro de su proyecto y transferir a ella los archivos mqh estándar necesarios dará lugar a la duplicación de información y a posibles discrepancias en las versiones de los archivos de encabezado. Es probable que este comportamiento se mejore en MetaEditor.

Otro punto para los proyectos públicos es la necesidad de administrar usuarios y autorizarlos. En nuestro último ejemplo, este problema sólo se identificó, pero no se aplicó. No obstante, el sitio mql5.com ofrece una solución ya preparada basada en el conocido protocolo OAuth. Cualquiera que tenga una cuenta en mql5.com puede familiarizarse con el principio de OAuth y configurarlo para su servicio web: sólo tiene que encontrar la sección *Applications* (enlace parecido a *https://www.mql5.com/es/users/<login> /apps*) en su perfil. Al registrar un servicio web en las aplicaciones de mql5.com, podrá autorizar usuarios a través del sitio web de mql5.com.

7.9 Compatibilidad nativa con python

El éxito potencial del trading automatizado depende en gran medida de la amplitud de la tecnología disponible para poner en práctica la idea. Como ya hemos visto en las secciones anteriores, MQL5 permite ir más allá de las tareas de trading estrictamente aplicadas y ofrece oportunidades de integración con servicios externos (por ejemplo, basados en funciones de red y símbolos personalizados), procesamiento y almacenamiento de datos mediante bases de datos relacionales, así como conexión de bibliotecas arbitrarias.

El último punto le permite garantizar la interacción con cualquier software que proporcione API en formato DLL. Algunos desarrolladores utilizan este método para conectarse a SGBD industriales distribuidos (en lugar del SQLite integrado), paquetes matemáticos como R o MATLAB y otros lenguajes de programación.

Python se ha convertido en uno de los lenguajes de programación más populares. Su característica es un núcleo compacto, que se complementa con paquetes que son colecciones de scripts listos para construir soluciones de aplicación. Los operadores de trading se benefician de la amplia selección y funcionalidad de los paquetes para el análisis fundamental del mercado (cálculos estadísticos, visualización de datos) y la simulación de hipótesis de trading, incluido el aprendizaje automático.

Siguiendo esta tendencia, MQ introdujo la compatibilidad con Python en MQL5 en 2019. Esta integración más ajustada y «lista para usar» permite transferir completamente los algoritmos de trading y de análisis técnico al entorno Python.

Desde un punto de vista técnico, la integración se consigue instalando el paquete «MetaTrader5» en Python, que organiza la interacción interproceso con el terminal (en el momento de escribir esto, a través del mecanismo ipykernel/RPC).

Entre las funciones del paquete, hay análogos completos de las funciones MQL5 integradas para obtener información sobre el terminal, cuenta de trading, símbolos en *Observación de Mercado*, cotizaciones, ticks, Profundidad de Mercado, órdenes, posiciones y transacciones. Además, el paquete le permite cambiar de cuenta de trading, enviar órdenes de trading, comprobar los requisitos de margen y evaluar los posibles beneficios/pérdidas en tiempo real.

Sin embargo, la integración con Python tiene algunas limitaciones. En concreto, no es posible en Python implementar el manejo de eventos como *OnTick*, *OnBookEvent* y otros. Por ello, es necesario utilizar un bucle infinito para comprobar los nuevos precios, de una forma muy parecida a como nos vimos obligados a hacer en los scripts de MQL5. El análisis de la ejecución de las órdenes de trading es igual de difícil: en ausencia de *OnTradeTransaction*, se necesitaría más código para saber si una posición se cerró total o parcialmente. Para saltarse estas restricciones, puede organizar la interacción del script de Python y MQL5, por ejemplo, a través de sockets. En el sitio mql5.com encontrará artículos con ejemplos de la aplicación de un puente de este tipo.

Por lo tanto, parece que es natural utilizar Python en conjunción con MetaTrader 5 para tareas de aprendizaje automático que se ocupan de las cotizaciones, los ticks o el historial de la cuenta de trading. Por desgracia, no se pueden obtener lecturas de indicadores en Python.

7.9.1 Instalar Python y el paquete MetaTrader5

Para estudiar los materiales de este capítulo, Python debe estar instalado en su ordenador. Si aún no lo ha instalado, descargue la última versión de Python (por ejemplo, la 3.10 en el momento de escribir estas líneas) de *https://www.python.org/downloads/windows*.

Al instalar Python, se recomienda marcar la bandera «Add Python to PATH» para poder ejecutar scripts de Python desde la línea de comandos desde cualquier carpeta.

Una vez descargado y ejecutado Python, instale el módulo MetaTrader5 desde la línea de comandos (aquí *pip* es un programa estándar de administración de paquetes de Python):

```
pip install MetaTrader5
```

Posteriormente, puede comprobar la actualización del paquete con la siguiente línea de comandos:

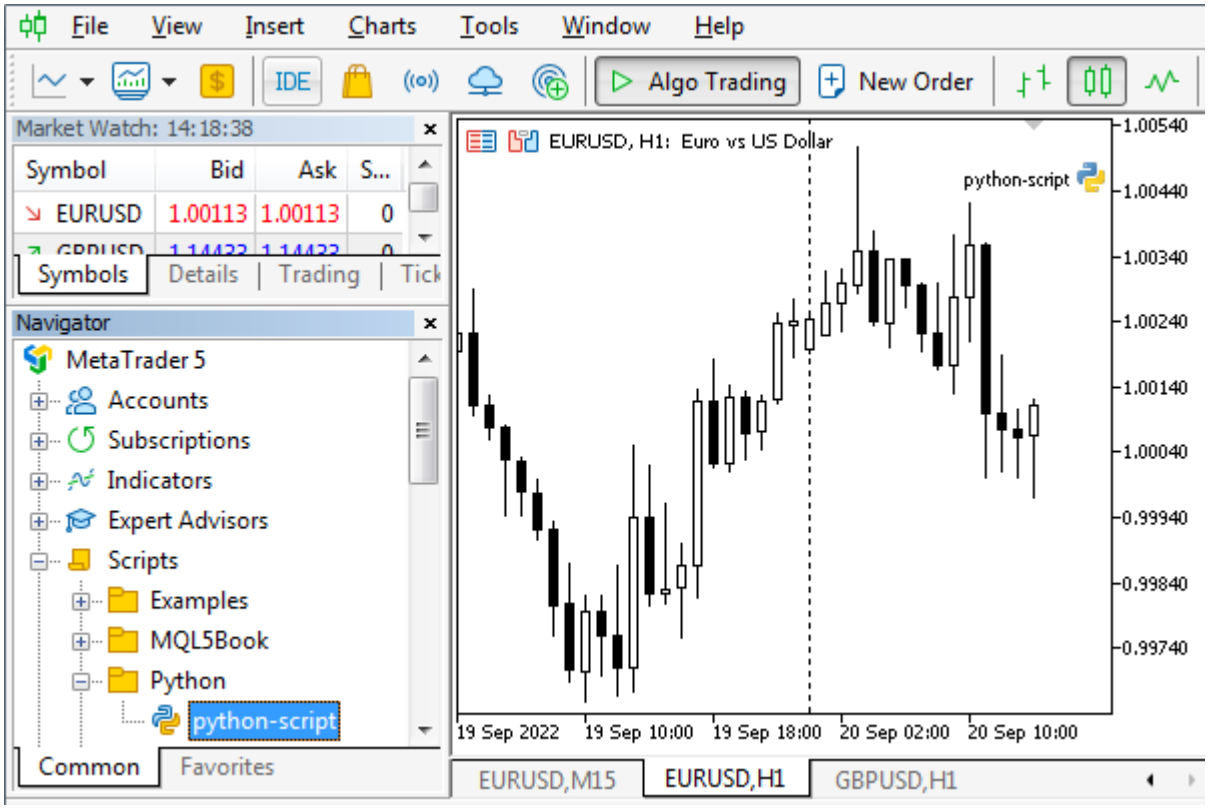
```
pip install --upgrade MetaTrader5
```

La sintaxis para añadir otros paquetes de uso común es similar. En concreto, muchos scripts requieren paquetes de análisis y visualización de datos: *pandas* y *matplotlib*, respectivamente.

```
pip install matplotlib
pip install pandas
```

Puede crear un nuevo script de Python directamente desde el Asistente MQL5 en MetaEditor. Además del nombre del script, el usuario puede seleccionar opciones para importar varios paquetes, como *TensorFlow*, *NumPy* o *Datetime*.

Por defecto, se sugiere colocar los scripts en la carpeta *MQL5/Scripts*. Los scripts de Python recién creados y los ya existentes se muestran en el navegador de MetaTrader 5, marcados con un icono especial, y se pueden lanzar desde el navegador de la forma habitual. Los scripts de Python pueden ejecutarse en el gráfico en paralelo con otros Asesores Expertos y scripts de MQL5. Para detener un script si su ejecución está en bucle, basta con eliminarlo del gráfico.



Ejecución de un script de Python en el terminal

El script de Python lanzado desde el terminal recibe el nombre del símbolo y el marco temporal del gráfico a través de parámetros de línea de comandos. Por ejemplo, podemos ejecutar el siguiente script en el gráfico EURUSD, H1, en el que los argumentos están disponibles como el array *sys.argv*:

```
import sys

print('The command line arguments are:')
for i in sys.argv:
   print(i)
```

Se enviará al registro del experto:

```
The command line arguments are:
C:\Program Files\MetaTrader 5\MQL5\Scripts\MQL5Book\Python\python-args.py
EURUSD
60
```

Además, se puede ejecutar un script de Python directamente desde MetaEditor especificando la ubicación de instalación de Python en el cuadro de diálogo del editor *Ajustes*, pestaña *Compiladores*: a continuación, el comando de compilación para archivos con la extensión **.py* se convierte en un comando de ejecución.

Por último, los scripts de Python también pueden ejecutarse en su entorno nativo pasándolos como parámetros en llamadas a *python.exe* desde la línea de comandos o desde otro IDE (Entorno de Desarrollo Integrado) adaptado para Python, como Jupyter Notebook.

Si el trading algorítmico está activado en el terminal, el trading desde Python también está activado por defecto. Para proteger aún más las cuentas cuando se utilizan bibliotecas Python de terceros, la configuración de la plataforma ofrece la opción «Desactivar el trading automático a través de la API de Python externa». Así, los scripts Python pueden bloquear selectivamente el trading, dejándolo disponible para los programas MQL. Cuando esta opción está activada, las llamadas a funciones de trading en un script Python devolverán el error 10027 (TRADE_RETCODE_CLIENT_DISABLES_AT), que indica que el trading algorítmico ha sido desactivado por el terminal de cliente.

MQL5 frente a Python

Python es un lenguaje interpretado, a diferencia de MQL5, que es compilado. Para nosotros, como desarrolladores, esto nos facilita la vida porque no necesitamos una fase de compilación separada para obtener un programa que funcione. No obstante, la velocidad de ejecución de los scripts en Python es notablemente inferior a la de los compilados en MQL5.

Python es un lenguaje de tipado dinámico: el tipo de una variable viene determinado por el valor que pongamos en ella. Por un lado, esto otorga flexibilidad, pero también exige cautela para evitar errores imprevistos. MQL5 utiliza tipado estático, es decir, a la hora de describir variables, debemos especificar de forma explícita su tipo, y el compilador controla la compatibilidad de tipos.

El propio Python «limpia la basura», es decir, libera la memoria asignada por el programa de aplicación para los objetos. En MQL5 tenemos que seguir la llamada oportuna de *delete* para objetos dinámicos.

En la sintaxis de Python, la separación o sangría del código fuente desempeña un importante papel. Si necesita escribir una sentencia compuesta (por ejemplo, un bucle o condicional) con un bloque de varias sentencias anidadas, entonces Python utiliza espacios o tabuladores para este propósito (deben ser del mismo tamaño dentro del bloque). No está permitido mezclar tabuladores y

espacios. Una sangría incorrecta provocará un error. En MQL5, formamos bloques de sentencias compuestas encerrándolas entre llaves {... }, pero el formato no desempeña ningún papel, y puede aplicar el estilo que quiera sin romper el rendimiento del programa.

Las funciones de Python admiten dos tipos de parámetros: con nombre y posicionales. El segundo tipo corresponde a lo que estamos acostumbrados en MQL5: el valor de cada parámetro debe pasarse estrictamente en su orden en la lista de argumentos (según el prototipo de la función). Por el contrario, los parámetros con nombre se pasan como una combinación de nombre y valor (con '=' entre ellos), y por tanto pueden especificarse en cualquier orden, por ejemplo, *func(param2 = value2, param1 = value1)*.

7.9.2 Visión general de las funciones del paquete MetaTrader5 para Python

Las funciones de la API disponibles en Python se pueden dividir condicionalmente en 2 grupos: funciones que tienen análogos completos en la API de MQL5 y funciones disponibles solo en Python. La presencia del segundo grupo se debe en parte al hecho de que la conexión entre Python y MetaTrader 5 debe organizarse técnicamente antes de poder utilizar las funciones de la aplicación. Esto explica la presencia y el propósito de un par de funciones *initialize* y *shutdown*: la primera establece una conexión con el terminal, y la segunda la termina.

Es importante que durante el proceso de inicialización se pueda lanzar la copia necesaria del terminal (si aún no se ha ejecutado) y se pueda seleccionar una cuenta de trading específica. Además, es posible cambiar la cuenta de trading en el contexto de una conexión ya abierta con el terminal: esto se hace mediante la función *inicio de sesión*.

Después de conectarse al terminal, un script de Python puede obtener un resumen de la versión del terminal utilizando la función *version*. Toda la información sobre el terminal está disponible en *terminal_info*, que es un análogo completo de tres funciones *TerminalInfo*, como si estuvieran unidas en una sola llamada.

En la siguiente tabla se enumeran las funciones de aplicación de Python y sus equivalentes en la API de MQL5.

Python	MQL5
last_error	*GetLastError* (¡Atención! Python tiene sus códigos de error nativos)
account_info	*AccountInfoInteger*, *AccountInfoDouble*, *AccountInfoString*
terminal_info	*TerminalInfoInteger*, *TerminalInfoDouble*, *TerminalInfoDouble*
symbols_total	*SymbolsTotal* (todos los símbolos, incluidos los personalizados y los desactivados)
symbols_get	*SymbolsTotal* + *SymbolInfo* (funciones)
symbol_info	*SymbolInfoInteger*, *SymbolInfoDouble*, *SymbolInfoString*
symbol_info_tick	*SymbolInfoTick*
symbol_select	*SymbolSelect*
market_book_add	*MarketBookAdd*

Python	MQL5
market_book_get	*MarketBookGet*
market_book_release	*MarketBookRelease*
copy_rates_from	*CopyRates* (por el número de barras, a partir de la fecha/hora)
copy_rates_from_pos	*CopyRates* (por el número de barras, empezando por el número de barra)
copy_rates_range	*CopyRates* (en el intervalo fecha/hora)
copy_ticks_from	*CopyTicks* (por el número de ticks, a partir de la hora especificada)
copy_ticks_range	*CopyTicksRange* (en el intervalo de tiempo especificado)
orders_total	*OrdersTotal*
orders_get	*OrdersTotal* + *OrderGet* (funciones)
order_calc_margin	*OrderCalcMargin*
order_calc_profit	*OrderCalcProfit*
order_check	*OrderCheck*
order_send	*OrderSend*
positions_total	*PositionsTotal*
positions_get	*PositionsTotal* + *PositionGet* (funciones)
history_orders_total	*HistoryOrdersTotal*
history_orders_get	*HistoryOrdersTotal* + *HistoryOrderGet* (funciones)
history_deals_total	*HistoryDealsTotal*
history_deals_get	*HistoryDealsTotal* + *HistoryDealGet* (funciones)

Las funciones de la API de Python tienen varias características.

Como ya se ha indicado, las funciones pueden tener parámetros con nombre: cuando se llama a una función, dichos parámetros se especifican junto con un nombre y un valor, en cada par de nombre y valor se combinan con el signo igual '='. El orden de especificación de los parámetros con nombre no es importante (a diferencia de los parámetros posicionales, que se utilizan en MQL5 y deben seguir el orden estricto especificado por el prototipo de la función).

Las funciones de Python operan sobre tipos de datos nativos de Python. Esto incluye no sólo las cadenas y números habituales, sino también varios tipos compuestos, algo similar a las estructuras y arrays de MQL5.

Por ejemplo, muchas funciones devuelven estructuras de datos especiales de Python: *tuple* y *namedtuple*.

Una tupla es una secuencia de elementos de un tipo arbitrario. Puede considerarse como un array, pero a diferencia de ésta, los elementos de una tupla pueden ser de distintos tipos. También puede considerar una tupla como un conjunto de campos de estructura.

Un parecido aún mayor con la estructura puede encontrarse con las tuplas con nombre, en las que a cada elemento se le asigna un ID. Solo se puede utilizar un índice para acceder a un elemento de una tupla común (entre corchetes, como en MQL5, es decir, [i]). No obstante, podemos aplicar el operador de desreferenciación (punto '.') a una tupla con nombre para obtener su «propiedad» igual que en la estructura de MQL5 (*tuple.field*).

Además, las tuplas y las tuplas con nombre no pueden editarse en código (es decir, son constantes).

Otro tipo popular es un diccionario: un array asociativo que almacena pares de claves y valores, y los tipos de ambos pueden variar. Se accede al valor del diccionario mediante el operador [], y la clave (sea del tipo que sea, por ejemplo, una cadena) se indica entre corchetes, lo que asemeja los diccionarios a los arrays. Un diccionario no puede tener dos pares con la misma clave, es decir, las claves son siempre únicas. En concreto, una tupla con nombre puede convertirse fácilmente en un diccionario con el método *namedtuple._asdict()*.

7.9.3 Conectar un script de Python al terminal y la cuenta

La función *initialize* establece una conexión con el terminal de MetaTrader 5 y tiene 2 formas: corta (sin parámetros) y completa (con varios parámetros opcionales, el primero de ellos es *path* y es posicional, y todos los demás tienen nombre).

```
bool initialize()
bool initialize(path, account = <ACCOUNT>, password = <"PASSWORD">,
 server = <"SERVER">, timeout = 60000, portable = False)
```

El parámetro *path* establece la ruta al archivo de terminal (*metatrader64.exe*) (tenga en cuenta que se trata de un parámetro sin nombre, a diferencia de todos los demás, por lo que si se especifica, debe ser el primero de la lista).

Si no se especifica la ruta, el módulo intentará encontrar el archivo ejecutable por sí mismo (los desarrolladores no revelan el algoritmo exacto). Para eliminar ambigüedades, utilice la segunda forma de la función con parámetros.

En el parámetro *account* puede especificar el número de la cuenta de trading. Si no se especifica, se utilizará la última cuenta de trading de la instancia seleccionada del terminal.

La contraseña de la cuenta de trading se especifica en el parámetro *password* y también puede omitirse: en este caso, la contraseña almacenada en la base de datos del terminal para la cuenta de trading especificada se sustituye automáticamente.

El parámetro *server* se procesa de forma similar con el nombre del servidor de trading (tal y como se especifica en el terminal); si no se especifica, se sustituye automáticamente por el servidor guardado en la base de datos del terminal para la cuenta de trading especificada.

El parámetro *timeout* indica el tiempo de espera en milisegundos que se da para la conexión (si se supera, se producirá un error). El valor predeterminado es 60000 (60 segundos).

El parámetro *portable* contiene una bandera para lanzar el terminal en modo portátil (de manera predeterminada es *False*).

La función devuelve *True* en caso de conexión exitosa con el terminal de MetaTrader 5, y *False* en caso contrario.

Si es necesario, al realizar una llamada a *initialize*, se puede iniciar el terminal de MetaTrader 5.

Por ejemplo, la conexión a una cuenta de trading específica se realiza del siguiente modo:

```
import MetaTrader5 as mt5
if not mt5.initialize(login = 562175752, server = "MetaQuotes-Demo", password = "abc"
   print("initialize() failed, error code =", mt5.last_error())
   quit()
...
```

La función *login* también se conecta a la cuenta de trading con los parámetros especificados, pero esto implica que ya se ha establecido la conexión con el terminal, es decir, la función se suele utilizar para cambiar la cuenta.

bool login(account, password = <"PASSWORD">, server = <"SERVER">, timeout = 60000)

El número de cuenta de trading se facilita en el parámetro *account*. Se trata de un parámetro obligatorio sin nombre, lo que significa que debe ser el primero de la lista.

Los parámetros *password*, *server* y *timeout* son idénticos a los parámetros correspondientes de la función *initialize*.

La función devuelve *True* en caso de conexión exitosa a la cuenta de trading, y *False* en caso contrario.

shutdown()

La función *shutdown* cierra la conexión previamente establecida con el terminal de MetaTrader 5.

El ejemplo para las funciones anteriores se proporcionará en la sección siguiente.

Cuando se establece la conexión, el script puede encontrar la versión del terminal.

tuple version()

La función *version* devuelve una breve información sobre la versión del terminal de MetaTrader 5 como una tupla de tres valores: número de versión, número de compilación y fecha de compilación.

Tipo de campo	Descripción
número entero	Versión del terminal de MetaTrader 5 (actual, 500)
número entero	Número de compilación (por ejemplo, 3456)
cadena	Fecha de compilación (por ejemplo, «25 feb 2022»)

En caso de error, la función devuelve *None*, y el código de error puede obtenerse utilizando *last_error*.

Se puede obtener información más completa sobre el terminal utilizando la función *terminal_info*.

7.9.4 Comprobación de errores: last_error

La función *last_error* devuelve información sobre el último error de Python.

int last_error()

Los códigos de error de números enteros difieren de los códigos asignados a los errores MQL5 y devueltos por la función *GetLastError* estándar. En la siguiente tabla, la abreviatura IPC significa «Comunicación entre procesos».

Constante	Significado	Descripción
RES_S_OK	1	Éxito
RES_E_FAIL	-1	Error común
RES_E_INVALID_PARAMS	-2	Argumentos/parámetros no válidos
RES_E_NO_MEMORY	-3	Error de asignación de memoria
RES_E_NOT_FOUND	-4	No se ha encontrado el historial solicitado
RES_E_INVALID_VERSION	-5	Versión no compatible
RES_E_AUTH_FAILED	-6	Error de autorización
RES_E_UNSUPPORTED	-7	Método no admitido
RES_E_AUTO_TRADING_DISABLED	-8	La negociación Algo está desactivada
RES_E_INTERNAL_FAIL	-10000	Error general interno de IPC
RES_E_INTERNAL_FAIL_SEND	-10001	Error interno al enviar datos de IPC
RES_E_INTERNAL_FAIL_RECEIVE	-10002	Error interno al enviar datos de IPC
RES_E_INTERNAL_FAIL_INIT	-10003	Error de inicialización interna de IPC
RES_E_INTERNAL_FAIL_CONNECT	-10003	No IPC
RES_E_INTERNAL_FAIL_TIMEOUT	-10005	Tiempo de espera de IPC

En el siguiente script (*MQL5/Scripts/MQL5Book/Python/init.py*), en caso de error al conectar con el terminal, mostramos el código de error y salimos.

```
import MetaTrader5 as mt5
# show MetaTrader5 package version
print("MetaTrader5 package version: ", mt5.__version__)  #  5.0.37

# let's try to establish a connection or launch the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()
... # the working part of the script will be here
# terminate the connection to the terminal
mt5.shutdown()
```

7.9.5 Obtener información sobre una cuenta de trading

La función *account_info* obtiene información completa sobre la cuenta de trading actual.

namedtuple account_info()

La función devuelve información como una estructura de tuplas con nombre (*namedtuple*). En caso de error, el resultado es *None*.

Con esta función, puede utilizar una llamada para obtener toda la información proporcionada por *AccountInfoInteger*, *AccountInfoDouble* y *AccountInfoString* en MQL5, con todas las variantes de propiedades compatibles. Los nombres de los campos de la tupla se corresponden con los nombres de los elementos de la enumeración sin el prefijo «ACCOUNT_», reducidos a minúsculas.

El siguiente script *MQL5/Scripts/MQL5Book/Python/accountinfo.py* se incluye con el libro.

```
import MetaTrader5 as mt5

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

account_info = mt5.account_info()
if account_info != None:
   # display trading account data as is
   print(account_info)
   # display data about the trading account in the form of a dictionary
   print("Show account_info()._asdict():")
   account_info_dict = mt5.account_info()._asdict()
   for prop in account_info_dict:
      print("  {}={}".format(prop, account_info_dict[prop]))

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()
```

El resultado debería ser algo parecido a lo siguiente:

```
AccountInfo(login=25115284, trade_mode=0, leverage=100, limit_orders=200, margin_so_m
Show account_info()._asdict():
login=25115284
trade_mode=0
leverage=100
limit_orders=200
margin_so_mode=0
trade_allowed=True
trade_expert=True
margin_mode=2
currency_digits=2
fifo_close=False
balance=99511.4
credit=0.0
profit=41.82
equity=99553.22
margin=98.18
margin_free=99455.04
margin_level=101398.67590140559
margin_so_call=50.0
margin_so_so=30.0
margin_initial=0.0
margin_maintenance=0.0
assets=0.0
liabilities=0.0
commission_blocked=0.0
name=MetaQuotes Dev Demo
server=MetaQuotes-Demo
currency=USD
company=MetaQuotes Software Corp.
```

7.9.6 Obtener información sobre el terminal

La función *terminal_info* permite obtener el estado y los parámetros del terminal de MetaTrader 5 conectado.

namedtuple terminal_info()

En caso de éxito, la función devuelve la información como una estructura de tuplas con nombre (*namedtuple*), y en caso de error devuelve *None*.

En una llamada a esta función, puede obtener toda la información proporcionada por *TerminalInfoInteger*, *TerminalInfoDouble*, y *TerminalInfoDouble* en MQL5, con todas las variantes de propiedades compatibles. Los nombres de los campos de la tupla corresponden a los nombres de los elementos de la enumeración sin el prefijo «TERMINAL_», reducidos a minúsculas.

Por ejemplo (véase *MQL5/Scripts/MQL5Book/Python/terminalinfo.py*):

```
import MetaTrader5 as mt5

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# display brief information about the MetaTrader 5 version
print(mt5.version())
# display full information about the settings and the state of the terminal
terminal_info = mt5.terminal_info()
if terminal_info != None:
   # display terminal data as is
   print(terminal_info)
   # display the data as a dictionary
   print("Show terminal_info()._asdict():")
   terminal_info_dict = mt5.terminal_info()._asdict()
   for prop in terminal_info_dict:
      print("  {}={}".format(prop, terminal_info_dict[prop]))

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()
```

Deberíamos tener algo como lo siguiente:

```
[500, 3428, '14 Sep 2022']
TerminalInfo(community_account=True, community_connection=True, connected=True,....
Show terminal_info()._asdict():
community_account=True
community_connection=True
connected=True
dlls_allowed=False
trade_allowed=False
tradeapi_disabled=False
email_enabled=False
ftp_enabled=False
notifications_enabled=False
mqid=False
build=2366
maxbars=5000
codepage=1251
ping_last=77850
community_balance=707.10668201585
retransmission=0.0
company=MetaQuotes Software Corp.
name=MetaTrader 5
language=Russian
path=E:\ProgramFiles\MetaTrader 5
data_path=E:\ProgramFiles\MetaTrader 5
commondata_path=C:\Users\User\AppData\Roaming\MetaQuotes\Terminal\Common
```

7.9.7 Obtener información sobre instrumentos financieros

El grupo de funciones del paquete MetaTrader5 proporciona información sobre instrumentos financieros.

La función *symbol_info* devuelve información sobre un instrumento financiero como una estructura de tupla con nombre.

namedtuple symbol_info(symbol)

El nombre del instrumento financiero deseado se especifica en el parámetro *symbol*.

Una llamada proporciona toda la información que puede obtenerse utilizando tres funciones MQL5 *SymbolInfoInteger*, *SymbolInfoDouble*, y *SymbolInfoString* con todas las propiedades. Los nombres de los campos de la tupla con nombre son los mismos que los nombres de los elementos de enumeración utilizados en las funciones especificadas, pero sin el prefijo «SYMBOL_» y en minúsculas.

En caso de error, la función devuelve *None*.

> ¡Atención! Para garantizar el éxito de la ejecución de la función, el símbolo solicitado debe seleccionarse en *Observación de Mercado*. Esto puede hacerse desde Python llamando a *symbol_select* (ver más adelante).

Ejemplo (*MQL5/Scripts/MQL5Book/Python/eurjpy.py*):

```
import MetaTrader5 as mt5

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# make sure EURJPY is present in the Market Watch, or abort the algorithm
selected = mt5.symbol_select("EURJPY", True)
if not selected:
   print("Failed to select EURJPY")
   mt5.shutdown()
   quit()

# display the properties of the EURJPY symbol
symbol_info = mt5.symbol_info("EURJPY")
if symbol_info != None:
   # display the data as is (as a tuple)
   print(symbol_info)
   # output a couple of specific properties
   print("EURJPY: spread =", symbol_info.spread, ", digits =", symbol_info.digits)
   # output symbol properties as a dictionary
   print("Show symbol_info(\"EURJPY\")._asdict():")
   symbol_info_dict = mt5.symbol_info("EURJPY")._asdict()
   for prop in symbol_info_dict:
      print("  {}={}".format(prop, symbol_info_dict[prop]))

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()
```

Resultado:

```
SymbolInfo(custom=False, chart_mode=0, select=True, visible=True, session_deals=0, se
EURJPY: spread = 17, digits = 3
Show symbol_info()._asdict():
custom=False
chart_mode=0
select=True
visible=True
...
time=1585069682
digits=3
spread=17
spread_float=True
ticks_bookdepth=10
trade_calc_mode=0
trade_mode=4
...
trade_exemode=1
swap_mode=1
swap_rollover3days=3
margin_hedged_use_leg=False
expiration_mode=7
filling_mode=1
order_mode=127
order_gtc_mode=0
...
bid=120.024
ask=120.041
last=0.0
...
point=0.001
trade_tick_value=0.8977708350166538
trade_tick_value_profit=0.8977708350166538
trade_tick_value_loss=0.8978272580355541
trade_tick_size=0.001
trade_contract_size=100000.0
...
volume_min=0.01
volume_max=500.0
volume_step=0.01
volume_limit=0.0
swap_long=-0.2
swap_short=-1.2
margin_initial=0.0
margin_maintenance=0.0
margin_hedged=100000.0
...
currency_base=EUR
currency_profit=JPY
currency_margin=EUR
...
```

bool symbol_select(symbol, enable = None)

La función *symbol_select* añade el símbolo especificado a *Observación de Mercado* o lo elimina. El símbolo se especifica en el primer parámetro. El segundo parámetro se pasa como *True* o *False*, lo que significa mostrar u ocultar el símbolo, respectivamente.

Si se omite el segundo parámetro opcional sin nombre, según las reglas de conversión de tipos de Python, *bool(none)* es equivalente a *False*.

La función es un análogo de *SymbolSelect*.

int symbols_total()

La función *symbols_total* devuelve el número de todos los instrumentos en el terminal de MetaTrader 5, teniendo en cuenta los símbolos personalizados y los que no se muestran actualmente en la ventana *Observación de Mercado*. Esto es el análogo de la función *SymbolsTotal(false)*.

A continuación, la función *symbols_get* devuelve un array de tuplas con información sobre todos los instrumentos o instrumentos favoritos cuyos nombres coinciden con el filtro especificado en el parámetro opcional con nombre *group*.

tuple[] symbols_get(group = "PATTERN")

Cada elemento de la tupla del array es una tupla con nombre con un conjunto completo de propiedades de símbolo (vimos una tupla similar con anterioridad en el contexto de la descripción de la función *symbol_info*).

Como sólo hay un parámetro, su nombre puede omitirse al llamar a la función.

En caso de error, la función devolverá un valor especial de *None*.

El parámetro *group* permite seleccionar símbolos por su nombre, utilizando opcionalmente el carácter de sustitución (comodín) '*' al principio y/o al final de la cadena buscada. '*' significa 0 o cualquier número de caracteres. Así, puede organizar una búsqueda de una subcadena que aparezca en el nombre con un número arbitrario de otros caracteres antes o después del fragmento especificado. Por ejemplo, «EUR*» significa símbolos que empiezan por «EUR» y tienen cualquier extensión de nombre (o simplemente «EUR»). El filtro «*EUR» devolverá los símbolos cuyos nombres contengan la subcadena «EUR» en cualquier lugar.

Además, el parámetro *group* puede contener varias condiciones separadas por comas. Cada condición puede especificarse como una máscara utilizando '*'. Para excluir símbolos, puede utilizar el signo lógico de negación '!'. En este caso, todas las condiciones se aplican secuencialmente, es decir, primero hay que especificar las condiciones de inclusión y luego, las de exclusión. Por ejemplo, group= «*, !*EUR*» significa que tenemos que seleccionar primero todos los símbolos y luego excluir los que contengan «EUR» en el nombre (en cualquier lugar).

Por ejemplo, para mostrar información sobre los tipos de cambio entre divisas, excepto para las 4 divisas principales de Forex, puede ejecutar la siguiente consulta:

```
crosses = mt5.symbols_get(group = "*,!*USD*,!*EUR*,!*JPY*,!*GBP*")
print('len(*,!*USD*,!*EUR*,!*JPY*,!*GBP*):', len(crosses)) # the size of the resultin
for s in crosses:
   print(s.name, ":", s)
```

He aquí un ejemplo del resultado:

```
len(*,!*USD*,!*EUR*,!*JPY*,!*GBP*): 10
AUDCAD : SymbolInfo(custom=False, chart_mode=0, select=True, visible=True, session_de
AUDCHF: SymbolInfo(custom=False, chart_mode=0, select=True, visible=True, session_dea
AUDNZD : SymbolInfo(custom=False, chart_mode=0, select=True, visible=True, session_de
CADCHF : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
NZDCAD : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
NZDCHF : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
NZDSGD : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
CADMXN : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
CHFMXN : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
NZDMXN : SymbolInfo(custom=False, chart_mode=0, select=False, visible=False, session_
```

La función *symbol_info_tick* permite obtener el último tick del instrumento financiero especificado.

tuple symbol_info_tick(symbol)

El único parámetro obligatorio especifica el nombre del instrumento financiero.

La información se devuelve como una tupla con los mismos campos que en la estructura *MqlTick*. La función es un análogo de *SymbolInfoTick*.

None se devuelve si se produce un error.

Para que la función funcione correctamente, el símbolo debe estar activado en *Observación de Mercado*. Vamos a demostrarlo en el script *MQL5/Scripts/MQL5Book/Python/gbpusdtick.py*.

```
import MetaTrader5 as mt5

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# try to include the GBPUSD symbol in the Market Watch
selected=mt5.symbol_select("GBPUSD", True)
if not selected:
   print("Failed to select GBPUSD")
   mt5.shutdown()
   quit()

# display the last tick of the GBPUSD symbol as a tuple
lasttick = mt5.symbol_info_tick("GBPUSD")
print(lasttick)
# display the values of the tick fields in the form of a dictionary
print("Show symbol_info_tick(\"GBPUSD\")._asdict():")
symbol_info_tick_dict = lasttick._asdict()
for prop in symbol_info_tick_dict:
   print("  {}={}".format(prop, symbol_info_tick_dict[prop]))

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()
```

El resultado debería ser el siguiente:

```
Tick(time=1585070338, bid=1.17264, ask=1.17279, last=0.0, volume=0, time_msc=15850703
Show symbol_info_tick._asdict():
time=1585070338
bid=1.17264
ask=1.17279
last=0.0
volume=0
time_msc=1585070338728
flags=2
volume_real=0.0
```

7.9.8 Suscripción a los cambios en el libro de órdenes

La API de Python incluye tres funciones para trabajar con el libro de órdenes.

bool market_book_add(symbol)

La función *market_book_add* se suscribe para recibir eventos sobre cambios en el libro de órdenes para el símbolo especificado. El nombre del instrumento financiero requerido se indica en un único parámetro sin nombre.

La función devuelve una indicación booleana de éxito.

La función es un análogo de *MarketBookAdd*. Una vez finalizado el trabajo con el libro de órdenes, la suscripción debe cancelarse llamando a *market_book_release* (véase más adelante).

tuple[] market_book_get(symbol)

La función *market_book_get* solicita el contenido actual del libro de órdenes para el símbolo especificado. El resultado se devuelve como una tupla (array) de registros *BookInfo*. Cada entrada es un análogo de la estructura *MqlBookInfo*, y desde el punto de vista de Python, se trata de una tupla con nombre con los campos «type», «price», «volume», «volume_real». En caso de error, se devuelve el valor *None*.

> Tenga en cuenta que, por alguna razón, en Python el campo se llama *volume_dbl*, aunque en MQL5 el campo correspondiente se llama *volume_real*.

Para trabajar con esta función, primero debe suscribirse para recibir eventos del libro de órdenes mediante la función *market_book_add*.

La función es un análogo de *MarketBookGet*. Tenga en cuenta que un script de Python no puede recibir eventos de *OnBookEvent* directamente y debe sondear el contenido del cristal en un bucle.

bool market_book_release(symbol)

La función *market_book_release* cancela la suscripción a eventos de cambio de libro de órdenes para el símbolo especificado. Si tiene éxito, la función devuelve *True*. La función es un análogo de *MarketBookRelease*.

Tomemos un ejemplo sencillo (véase *MQL5/Scripts/MQL5Book/Python/eurusdbook.py*).

```
import MetaTrader5 as mt5
import time               # connect a pack for the pause

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   mt5.shutdown()
   quit()

# subscribe to receive DOM updates for the EURUSD symbol
if mt5.market_book_add('EURUSD'):
   # run 10 times a loop to read data from the order book
   for i in range(10):
      # get the contents of the order book
      items = mt5.market_book_get('EURUSD')
      # display the entire order book in one line as is
      print(items)
      # now display each price level separately in the form of a dictionary, for clar
      for it in items or []:
         print(it._asdict())
      # let's pause for 5 seconds before the next request for data from the order boo
      time.sleep(5)
   # unsubscribe to order book changes
   mt5.market_book_release('EURUSD')
else:
   print("mt5.market_book_add('EURUSD') failed, error code =", mt5.last_error())

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()
```

He aquí un ejemplo del resultado:

```
(BookInfo(type=1, price=1.20036, volume=250, volume_dbl=250.0), BookInfo(type=1, pric
{'type': 1, 'price': 1.20036, 'volume': 250, 'volume_dbl': 250.0}
{'type': 1, 'price': 1.20029, 'volume': 100, 'volume_dbl': 100.0}
{'type': 1, 'price': 1.20028, 'volume': 50, 'volume_dbl': 50.0}
{'type': 1, 'price': 1.20026, 'volume': 36, 'volume_dbl': 36.0}
{'type': 2, 'price': 1.20023, 'volume': 36, 'volume_dbl': 36.0}
{'type': 2, 'price': 1.20022, 'volume': 50, 'volume_dbl': 50.0}
{'type': 2, 'price': 1.20021, 'volume': 100, 'volume_dbl': 100.0}
{'type': 2, 'price': 1.20014, 'volume': 250, 'volume_dbl': 250.0}
(BookInfo(type=1, price=1.20035, volume=250, volume_dbl=250.0), BookInfo(type=1, pric
{'type': 1, 'price': 1.20035, 'volume': 250, 'volume_dbl': 250.0}
{'type': 1, 'price': 1.20029, 'volume': 100, 'volume_dbl': 100.0}
{'type': 1, 'price': 1.20027, 'volume': 50, 'volume_dbl': 50.0}
{'type': 1, 'price': 1.20025, 'volume': 36, 'volume_dbl': 36.0}
{'type': 2, 'price': 1.20023, 'volume': 36, 'volume_dbl': 36.0}
{'type': 2, 'price': 1.20022, 'volume': 50, 'volume_dbl': 50.0}
{'type': 2, 'price': 1.20021, 'volume': 100, 'volume_dbl': 100.0}
{'type': 2, 'price': 1.20014, 'volume': 250, 'volume_dbl': 250.0}
(BookInfo(type=1, price=1.20037, volume=250, volume_dbl=250.0), BookInfo(type=1, pric
{'type': 1, 'price': 1.20037, 'volume': 250, 'volume_dbl': 250.0}
{'type': 1, 'price': 1.20031, 'volume': 100, 'volume_dbl': 100.0}
{'type': 1, 'price': 1.2003, 'volume': 50, 'volume_dbl': 50.0}
{'type': 1, 'price': 1.20028, 'volume': 36, 'volume_dbl': 36.0}
{'type': 2, 'price': 1.20025, 'volume': 36, 'volume_dbl': 36.0}
{'type': 2, 'price': 1.20023, 'volume': 50, 'volume_dbl': 50.0}
{'type': 2, 'price': 1.20022, 'volume': 100, 'volume_dbl': 100.0}
{'type': 2, 'price': 1.20016, 'volume': 250, 'volume_dbl': 250.0}
...
```

7.9.9 Leer cotizaciones

La API de Python permite obtener arrays de precios (barras) mediante tres funciones que difieren en la forma de especificar el rango de datos solicitados: por números de barra o por tiempo. Todas las funciones son similares a diferentes formas de *CopyRates*.

En todas las funciones, los dos primeros parámetros se utilizan para especificar el nombre del símbolo y el marco temporal. Los marcos temporales se recogen en la enumeración TIMEFRAME, que es similar a la enumeración ENUM_TIMEFRAMES de MQL5.

> Tenga en cuenta lo siguiente: en Python, los elementos de esta enumeración llevan el prefijo TIMEFRAME_, mientras que los elementos de una enumeración similar en MQL5 llevan el prefijo PERIOD_.

Identificador	Descripción
TIMEFRAME_M1	1 minuto
TIMEFRAME_M2	2 minutos
TIMEFRAME_M3	3 minutos
TIMEFRAME_M4	4 minutos
TIMEFRAME_M5	5 minutos

Identificador	Descripción
TIMEFRAME_M6	6 minutos
TIMEFRAME_M10	10 minutos
TIMEFRAME_M12	12 minutos
TIMEFRAME_M12	15 minutos
TIMEFRAME_M20	20 minutos
TIMEFRAME_M30	30 minutos
TIMEFRAME_H1	1 hora
TIMEFRAME_H2	2 horas
TIMEFRAME_H3	3 horas
TIMEFRAME_H4	4 horas
TIMEFRAME_H6	6 horas
TIMEFRAME_H8	8 horas
TIMEFRAME_H12	12 horas
TIMEFRAME_D1	1 día
TIMEFRAME_W1	1 semana
TIMEFRAME_MN1	1 mes

Las tres funciones devuelven barras como un array de lotes *numpy* con las columnas con nombre *time*, *open*, *high*, *low*, *close*, *tick_volume*, *spread* y *real_volume*. El array *numpy.ndarray* es un análogo más eficiente de las tuplas con nombre. Para acceder a las columnas, utilice la notación de corchetes, *array['column']*.

None se devuelve si se produce un error.

Todos los parámetros de la función son obligatorios y no llevan nombre.

numpy.ndarray copy_rates_from(symbol, timeframe, date_from, count)

La función *copy_rates_from* solicita barras a partir de la fecha especificada (*date_from*) en el número de barras *count*. La fecha puede establecerse mediante el objeto *datetime*, o como el número de segundos transcurridos desde 1970.01.01.

A la hora de crear el objeto *datetime*, Python utiliza la zona horaria local, mientras que el terminal de MetaTrader 5 almacena las horas de apertura de barras y ticks en UTC (GMT, sin desplazamiento). Por lo tanto, para ejecutar funciones que utilicen hora, es necesario crear variables *datetime* en UTC. Para configurar las zonas horarias, puede utilizar el paquete *pytz*. Por ejemplo (véase *MQL5/Scripts/MQL5Book/Python/eurusdrates.py*):

```
from datetime import datetime
import MetaTrader5 as mt5
import pytz                    # import the pytz module to work with the timezone
# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   mt5.shutdown()
   quit()

# set the timezone to UTC
timezone = pytz.timezone("Etc/UTC")

# create a datetime object in the UTC timezone so that the local timezone offset is n
utc_from = datetime(2022, 1, 10, tzinfo = timezone)

# get 10 bars from EURUSD H1 starting from 10/01/2022 in the UTC timezone
rates = mt5.copy_rates_from("EURUSD", mt5.TIMEFRAME_H1, utc_from, 10)

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()

# display each element of the received data (tuple)
for rate in rates:
   print(rate)
```

He aquí una muestra de los datos recibidos:

```
(1641567600, 1.12975, 1.13226, 1.12922, 1.13017, 8325, 0, 0)
(1641571200, 1.13017, 1.13343, 1.1299, 1.13302, 7073, 0, 0)
(1641574800, 1.13302, 1.13491, 1.13293, 1.13468, 5920, 0, 0)
(1641578400, 1.13469, 1.13571, 1.13375, 1.13564, 3723, 0, 0)
(1641582000, 1.13564, 1.13582, 1.13494, 1.13564, 1990, 0, 0)
(1641585600, 1.1356, 1.13622, 1.13547, 1.13574, 1269, 0, 0)
(1641589200, 1.13572, 1.13647, 1.13568, 1.13627, 1031, 0, 0)
(1641592800, 1.13627, 1.13639, 1.13573, 1.13613, 982, 0, 0)
(1641596400, 1.1361, 1.13613, 1.1358, 1.1359, 692, 1, 0)
(1641772800, 1.1355, 1.13597, 1.13524, 1.1356, 1795, 10, 0)
```

numpy.ndarray copy_rates_from_pos(symbol, timeframe, start, count)

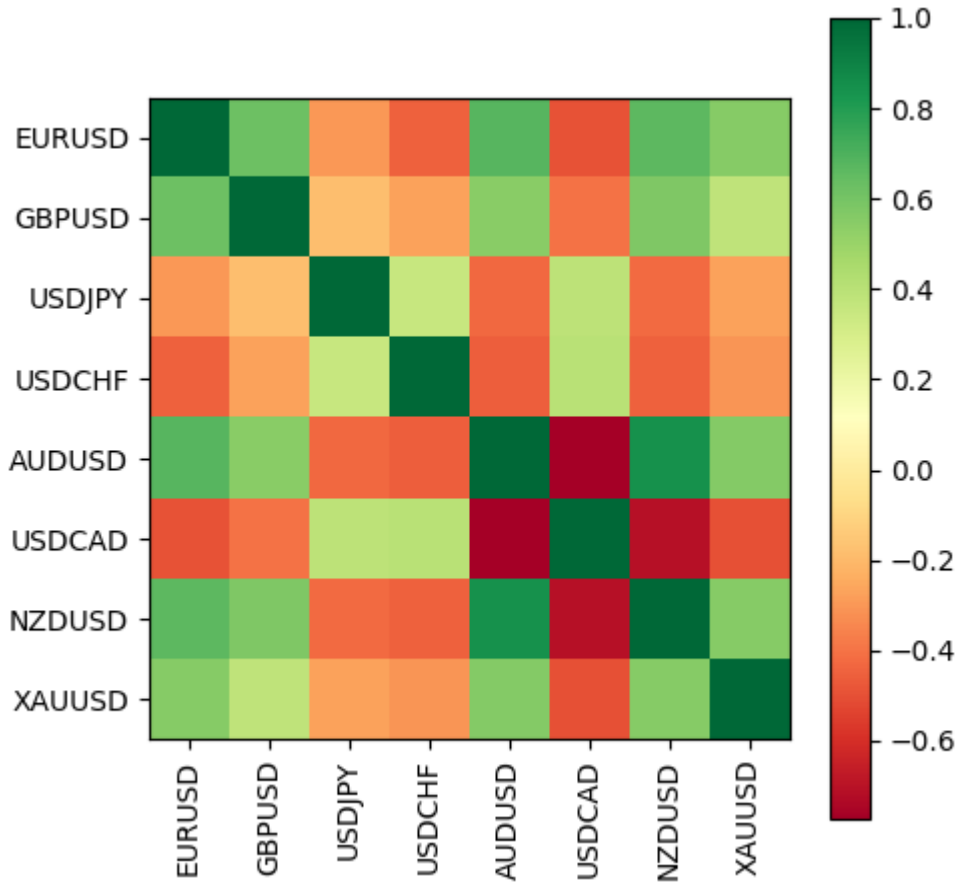
La función *copy_rates_from_pos* solicita barras a partir del índice *start* especificado, en la cantidad de *count*.

> El terminal de MetaTrader 5 representa las barras sólo dentro de los límites del historial disponible para el usuario en los gráficos. El número de barras disponibles para el usuario se fija en los ajustes mediante el parámetro «Máximo de barras en la ventana».

En el siguiente ejemplo (*MQL5/Scripts/MQL5Book/Python/ratescorr.py*) se muestra una representación gráfica del array de correlaciones de varias divisas basada en cotizaciones.

```
import MetaTrader5 as mt5
import pandas as pd              # connect the pandas module to output data
import matplotlib.pyplot as plt  # connect the matplotlib module for drawing

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   mt5.shutdown()
   quit()

# create a path in the sandbox for the image with the result
image = mt5.terminal_info().data_path + r'\MQL5\Files\MQL5Book\ratescorr'

# the list of working currencies for calculating correlation
sym = ['EURUSD','GBPUSD','USDJPY','USDCHF','AUDUSD','USDCAD','NZDUSD','XAUUSD']

# copy the closing prices of bars into DataFrame structures
d = pd.DataFrame()
for i in sym:        # last 1000 M1 bars for each symbol
   rates = mt5.copy_rates_from_pos(i, mt5.TIMEFRAME_M1, 0, 1000)
   d[i] = [y['close'] for y in rates]

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()

# calculate the price change as a percentage
rets = d.pct_change()

# compute correlations
corr = rets.corr()

# draw the correlation matrix
fig = plt.figure(figsize = (5, 5))
fig.add_axes([0.15, 0.1, 0.8, 0.8])
plt.imshow(corr, cmap = 'RdYlGn', interpolation = 'none', aspect = 'equal')
plt.colorbar()
plt.xticks(range(len(corr)), corr.columns, rotation = 'vertical')
plt.yticks(range(len(corr)), corr.columns)
plt.show()
plt.savefig(image)
```

El archivo de imagen *ratescorr.png* se forma en el sandbox de la copia de trabajo actual de MetaTrader 5. La visualización interactiva de una imagen en una ventana independiente mediante una llamada a *plt.show()* puede no funcionar si su instalación de Python no incluye las características opcionales «tcl/tk e IDLE», o si no añade el paquete *pip install.tk*.



Array de correlación de divisas

numpy.ndarray copy_rates_range(symbol, timeframe, date_from, date_to)

La función *copy_rates_range* le permite obtener barras en el rango de fecha y hora especificado, entre *date_from* y *date_to*: ambos valores se dan como el número de segundos desde el comienzo de 1970, en la zona horaria UTC (ya que Python utiliza la zona horaria *datetime* local, debe convertirla utilizando el módulo *pytz*). El resultado incluye barras con horarios de apertura, *time* >= *date_from* y *time* <= *date_to*.

En el siguiente script, solicitaremos barras en un rango de tiempo específico.

```
from datetime import datetime
import MetaTrader5 as mt5
import pytz             # connect the pytz module to work with the timezone
import pandas as pd     # connect the pandas module to display data in a tabular form

pd.set_option('display.max_columns', 500) # how many columns to show
pd.set_option('display.width', 1500)      # max. table width to display

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# set the timezone to UTC
timezone = pytz.timezone("Etc/UTC")
# create datetime objects in UTC timezone so that local timezone offset is not applie
utc_from = datetime(2020, 1, 10, tzinfo=timezone)
utc_to = datetime(2020, 1, 10, minute = 30, tzinfo=timezone)

# get bars for USDJPY M5 for period 2020.01.10 00:00 - 2020.01.10 00:30 in UTC timezo
rates = mt5.copy_rates_range("USDJPY", mt5.TIMEFRAME_M5, utc_from, utc_to)

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()

# create a DataFrame from the received data
rates_frame = pd.DataFrame(rates)
# convert time from number of seconds to datetime format
rates_frame['time'] = pd.to_datetime(rates_frame['time'], unit = 's')

# output data
print(rates_frame)
```

He aquí un ejemplo del resultado:

```
time open high low close tick_volume spread real_volume
0 2020-01-10 00:00:00 109.513 109.527 109.505 109.521 43 2 0
1 2020-01-10 00:05:00 109.521 109.549 109.518 109.543 215 8 0
2 2020-01-10 00:10:00 109.543 109.543 109.466 109.505 98 10 0
3 2020-01-10 00:15:00 109.504 109.534 109.502 109.517 155 8 0
4 2020-01-10 00:20:00 109.517 109.539 109.513 109.527 71 4 0
5 2020-01-10 00:25:00 109.526 109.537 109.484 109.520 106 9 0
6 2020-01-10 00:30:00 109.520 109.524 109.508 109.510 205 7 0
```

7.9.10 Leer historial de ticks

La API de Python incluye dos funciones para leer el historial de ticks reales: *copy_ticks_from* con una indicación del número de ticks a partir de la fecha especificada, y *copy_ticks_range* para todos los ticks del período especificado.

Ambas funciones tienen cuatro parámetros necesarios sin nombre, el primero de los cuales especifica el símbolo. El segundo parámetro especifica el tiempo inicial de los ticks solicitados. El tercer parámetro indica si se pasa el número requerido de ticks (en la función *copy_ticks_from*) o el tiempo final de ticks (en la función *copy_ticks_range*).

El último parámetro determina qué tipo de ticks se devolverán. Puede contener uno de los siguientes indicadores (COPY_TICKS):

Identificador	Descripción
COPY_TICKS_ALL	Todos los ticks
COPY_TICKS_INFO	Ticks que contienen cambios en los precios de compra y/o venta
COPY_TICKS_TRADE	Ticks que contienen cambios en el último precio y/o volumen (Volume)

Ambas funciones devuelven los ticks como un array *numpy.ndarray* (del paquete *numpy*) con columnas nombradas *time*, *bid*, *ask*, *last* y *flags*. El valor del campo *flags* es una combinación de banderas de bits de la enumeración TICK_FLAG: cada bit significa un cambio en el campo correspondiente con la propiedad tick.

Identificador	Propiedad tick modificada
TICK_FLAG_BID	Precio de compra (Bid)
TICK_FLAG_ASK	Precio de venta (Ask)
TICK_FLAG_LAST	Último precio
TICK_FLAG_VOLUME	Volumen
TICK_FLAG_BUY	Último precio de compra
TICK_FLAG_SELL	Último precio de venta

numpy.ndarray copy_ticks_from(symbol, date_from, count, flags)

La función *copy_ticks_from* solicita ticks a partir de la hora especificada (*date_from*) en la cantidad dada (*count*).

La función es un análogo de *CopyTicks*.

numpy.array copy_ticks_range(symbol, date_from, date_to, flags)

La función *copy_ticks_range* permite obtener los ticks para el intervalo de tiempo especificado.

La función es un análogo de *CopyTicksRange*.

En el siguiente ejemplo (*MQL5/Scripts/MQL5Book/Python/copyticks.py*), generamos una página web interactiva con un gráfico de tick (nota: aquí se utiliza el paquete *plotly*; para instalarlo en Python, ejecute el comando *pip install plotly*).

```
import MetaTrader5 as mt5
import pandas as pd
import pytz
from datetime import datetime

# connect to terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# set the name of the file to save to the sandbox
path = mt5.terminal_info().data_path + r'\MQL5\Files\MQL5Book\copyticks.html'

# copy 1000 EURUSD ticks from a specific moment in history
utc = pytz.timezone("Etc/UTC")
rates = mt5.copy_ticks_from("EURUSD", \
datetime(2022, 5, 25, 1, 15, tzinfo = utc), 1000, mt5.COPY_TICKS_ALL)
bid = [x['bid'] for x in rates]
ask = [x['ask'] for x in rates]
time = [x['time'] for x in rates]
time = pd.to_datetime(time, unit = 's')

# terminate the connection to the terminal
mt5.shutdown()

# connect the graphics package and draw 2 rows of ask and bid prices on the web page
import plotly.graph_objs as go
from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot
data = [go.Scatter(x = time, y = bid), go.Scatter(x = time, y = ask)]
plot(data, filename = path)
```

He aquí cuál podría ser el resultado:

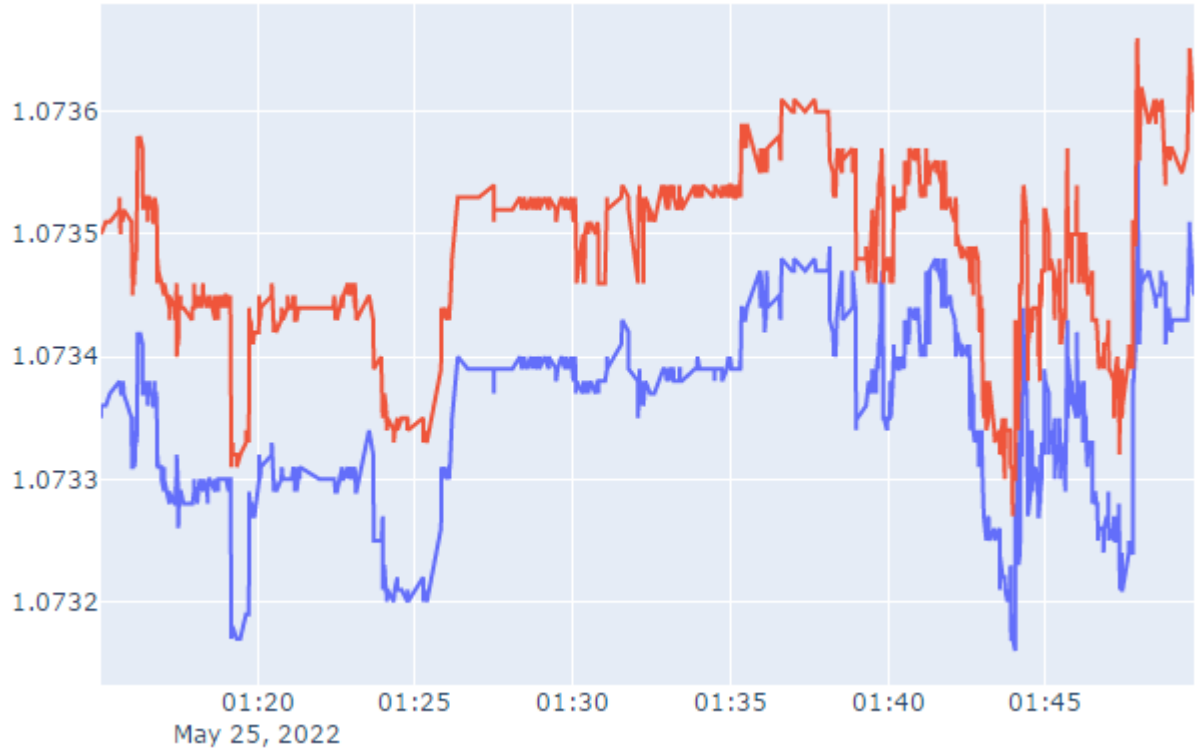


Gráfico con ticks recibidos en un script Python

La página web *copyticks.html* se genera en el subdirectorio *MQL5/Files/MQL5Book*.

7.9.11 Calcular requisitos de margen y evaluar beneficios

Un desarrollador de Python puede calcular directamente el margen y el beneficio o pérdida potencial de la operación de trading propuesta en el script utilizando las funciones *order_calc_margin* y *order_calc_profit*. En caso de ejecución correcta, el resultado de cualquier función es un número real; en caso contrario, es *None*.

float order_calc_margin(action, symbol, volume, price)

La función *order_calc_margin* devuelve el importe del margen (en la divisa de la cuenta) necesario para completar la operación de trading especificada *action*, que puede ser uno de los dos elementos de la enumeración ENUM_ORDER_TYPE: ORDER_TYPE_BUY u ORDER_TYPE_SELL. Los siguientes parámetros especifican el nombre del instrumento financiero, el volumen de la operación de trading y el precio de apertura.

La función es un análogo de *OrderCalcMargin*.

float order_calc_profit(action, symbol, volume, price_open, price_close)

La función *order_calc_profit* devuelve el importe de las ganancias o pérdidas (en la divisa de la cuenta) para el tipo de operación, símbolo y volumen especificados, así como la diferencia entre los precios de entrada y salida del mercado.

La función es un análogo de *OrderCalcProfit*.

Se recomienda comprobar el margen y el resultado previsto de la operación antes de enviar una orden.

7.9.12 Comprobación y envío de una orden de trading

Si es necesario, puede operar directamente desde un script de Python. El par de funciones *order_check* y *order_send* permite comprobar previamente y ejecutar a continuación una operación de trading.

Para ambas funciones, el único parámetro es la estructura de solicitud *TradeRequest* (puede inicializarse como un diccionario en Python, véase un ejemplo). Los campos de estructura son exactamente los mismos que para *MqlTradeRequest*.

OrderCheckResult order_check(request)

La función *order_check* comprueba la corrección de los campos de la solicitud de trading y la suficiencia de fondos para completar la operación de trading requerida.

El resultado de la función se devuelve como la estructura *OrderCheckResult*. Repite la estructura de *MqlTradeCheckResult* pero contiene además el campo *request* con una copia de la solicitud original.

La función *order_check* es un análogo de *OrderCheck*.

Ejemplo (*MQL5/Scripts/MQL5Book/python/ordercheck.py*):

```
import MetaTrader5 as mt5

# let's establish a connection to the MetaTrader 5 terminal
...
# get account currency for information
account_currency=mt5.account_info().currency
print("Account currency:", account_currency)

# get the necessary properties of the deal symbol
symbol = "USDJPY"
symbol_info = mt5.symbol_info(symbol)
if symbol_info is None:
   print(symbol, "not found, can not call order_check()")
   mt5.shutdown()
   quit()

point = mt5.symbol_info(symbol).point
# if the symbol is not available in the Market Watch, add it
if not symbol_info.visible:
   print(symbol, "is not visible, trying to switch on")
   if not mt5.symbol_select(symbol, True):
      print("symbol_select({}) failed, exit", symbol)
      mt5.shutdown()
      quit()

# prepare the query structure as a dictionary
request = \
{
   "action": mt5.TRADE_ACTION_DEAL,
   "symbol": symbol,
   "volume": 1.0,
   "type": mt5.ORDER_TYPE_BUY,
   "price": mt5.symbol_info_tick(symbol).ask,
   "sl": mt5.symbol_info_tick(symbol).ask - 100 * point,
   "tp": mt5.symbol_info_tick(symbol).ask + 100 * point,
   "deviation": 10,
   "magic": 234000,
   "comment": "python script",
   "type_time": mt5.ORDER_TIME_GTC,
   "type_filling": mt5.ORDER_FILLING_RETURN,
}

# run the test and display the result as is
result = mt5.order_check(request)
print(result)                        # [?this is not in the help log?]

# convert the result to a dictionary and output element by element
result_dict = result._asdict()
for field in result_dict.keys():
   print("   {}={}".format(field, result_dict[field]))
   # if this is the structure of a trade request, then output it element by element a
```

```
      if field == "request":
         traderequest_dict = result_dict[field]._asdict()
         for tradereq_filed in traderequest_dict:
            print("       traderequest: {}={}".format(tradereq_filed,
            traderequest_dict[tradereq_filed]))

   # terminate the connection to the terminal
   mt5.shutdown()
```

Resultado:

```
Divisa de la cuenta: USD
OrderCheckResult(retcode=0, balance=10000.17, equity=10000.17, profit=0.0, margin=100
retcode=0
balance=10000.17
equity=10000.17
profit=0.0
margin=1000.0
margin_free=9000.17
margin_level=1000.017
comment=Done
request=TradeRequest(action=1, magic=234000, order=0, symbol='USDJPY', volume=1.0, pr
traderequest: action=1
traderequest: magic=234000
traderequest: order=0
traderequest: symbol=USDJPY
traderequest: volume=1.0
traderequest: price=144.128
traderequest: stoplimit=0.0
traderequest: sl=144.028
traderequest: tp=144.228
traderequest: deviation=10
traderequest: type=0
traderequest: type_filling=2
traderequest: type_time=0
traderequest: expiration=0
traderequest: comment=python script
traderequest: position=0
traderequest: position_by=0
```

OrderSendResult order_send(request)

La función *order_send* envía una solicitud desde el terminal al servidor de trading para realizar una operación.

El resultado de la función se devuelve como la estructura *OrderSendResult*. Repite la estructura de *MqlTradeResult* pero contiene además el campo *request* con una copia de la solicitud original.

La función es un análogo de *OrderSend*.

Ejemplo (*MQL5/Scripts/MQL5Book/python/ordersend.py*):

```
import time
import MetaTrader5 as mt5

# let's establish a connection to the MetaTrader 5 terminal
...
# assign the properties of the working symbol
symbol = "USDJPY"
symbol_info = mt5.symbol_info(symbol)
if symbol_info is None:
   print(symbol, "not found, can not trade")
   mt5.shutdown()
   quit()

# if the symbol is not available in the Market Watch, add it
if not symbol_info.visible:
   print(symbol, "is not visible, trying to switch on")
   if not mt5.symbol_select(symbol, True):
      print("symbol_select({}) failed, exit", symbol)
      mt5.shutdown()
      quit()

# let's prepare the request structure for the purchase
lot = 0.1
point = mt5.symbol_info(symbol).point
price = mt5.symbol_info_tick(symbol).ask
deviation = 20
request = \
{
   "action": mt5.TRADE_ACTION_DEAL,
   "symbol": symbol,
   "volume": lot,
   "type": mt5.ORDER_TYPE_BUY,
   "price": price,
   "sl": price - 100 * point,
   "tp": price + 100 * point,
   "deviation": deviation,
   "magic": 234000,
   "comment": "python script open",
   "type_time": mt5.ORDER_TIME_GTC,
   "type_filling": mt5.ORDER_FILLING_RETURN,
}

# send a trade request to open a position
result = mt5.order_send(request)
# check the execution result
print("1. order_send(): by {} {} lots at {}".format(symbol, lot, price));
if result.retcode != mt5.TRADE_RETCODE_DONE:
   print("2. order_send failed, retcode={}".format(result.retcode))
   # request the result as a dictionary and display it element by element
   result_dict = result._asdict()
   for field in result_dict.keys():
```

```
        print("   {}={}".format(field, result_dict[field]))
        # if this is the structure of a trade request, then output it element by elemen
        if field == "request":
            traderequest_dict = result_dict[field]._asdict()
            for tradereq_filed in traderequest_dict:
                print("       traderequest: {}={}".format(tradereq_filed,
                traderequest_dict[tradereq_filed]))
    print("shutdown() and quit")
    mt5.shutdown()
    quit()

print("2. order_send done, ", result)
print("   opened position with POSITION_TICKET={}".format(result.order))
print("   sleep 2 seconds before closing position #{}".format(result.order))
time.sleep(2)
# create a request to close
position_id = result.order
price = mt5.symbol_info_tick(symbol).bid
request = \
{
    "action": mt5.TRADE_ACTION_DEAL,
    "symbol": symbol,
    "volume": lot,
    "type": mt5.ORDER_TYPE_SELL,
    "position": position_id,
    "price": price,
    "deviation": deviation,
    "magic": 234000,
    "comment": "python script close",
    "type_time": mt5.ORDER_TIME_GTC,
    "type_filling": mt5.ORDER_FILLING_RETURN,
}
# send a trade request to close the position
result = mt5.order_send(request)
# check the execution result
print("3. close position #{}: sell {} {} lots at {}".format(position_id,
symbol, lot, price));
if result.retcode != mt5.TRADE_RETCODE_DONE:
    print("4. order_send failed, retcode={}".format(result.retcode))
    print("   result", result)
else:
    print("4. position #{} closed, {}".format(position_id, result))
    # request the result as a dictionary and display it element by element
    result_dict = result._asdict()
    for field in result_dict.keys():
        print("   {}={}".format(field, result_dict[field]))
        # if this is the structure of a trade request, then output it element by elemen
        if field == "request":
            traderequest_dict = result_dict[field]._asdict()
            for tradereq_filed in traderequest_dict:
                print("       traderequest: {}={}".format(tradereq_filed,
```

```
            traderequest_dict[tradereq_filed]))

# terminate the connection to the terminal
mt5.shutdown()
```

Resultado:

```
1. order_send(): by USDJPY 0.1 lots at 144.132
2. order_send done, OrderSendResult(retcode=10009, deal=1445796125, order=1468026008,
opened position with POSITION_TICKET=1468026008
sleep 2 seconds before closing position #1468026008
3. close position #1468026008: sell USDJPY 0.1 lots at 144.124
4. position #1468026008 closed, OrderSendResult(retcode=10009, deal=1445796155, order
retcode=10009
deal=1445796155
order=1468026041
volume=0.1
price=144.124
bid=144.124
ask=144.132
comment=Request executed
request_id=2
retcode_external=0
request=TradeRequest(action=1, magic=234000, order=0, symbol='USDJPY', volume=0.1, pr
traderequest: action=1
traderequest: magic=234000
traderequest: order=0
traderequest: symbol=USDJPY
traderequest: volume=0.1
traderequest: price=144.124
traderequest: stoplimit=0.0
traderequest: sl=0.0
traderequest: tp=0.0
traderequest: deviation=20
traderequest: type=1
traderequest: type_filling=2
traderequest: type_time=0
traderequest: expiration=0
traderequest: comment=python script close
traderequest: position=1468026008
traderequest: position_by=0
```

7.9.13 Obtener el número y la lista de órdenes activas

La API de Python proporciona las siguientes funciones para trabajar con órdenes activas.

int orders_total()

La función *orders_total* devuelve el número de órdenes activas.

La función es un análogo de *Orders Total*.

Se puede obtener información detallada sobre cada orden utilizando la función *orders_get*, que dispone de varias opciones con la posibilidad de filtrar por símbolo o ticket. En cualquier caso, la función devuelve el array de tuplas con nombre *TradeOrder* (los nombres de los campos coinciden con

ENUM_ORDER_PROPERTY_enumerations sin el prefijo «ORDER_» y reducido a minúsculas). En caso de error, el resultado es *None*.

```
namedtuple[] orders_get()
namedtuple[] orders_get(symbol = <"SYMBOL">)
namedtuple[] orders_get(group = <"PATTERN">)
namedtuple[] orders_get(ticket = <TICKET>)
```

La función *orders_get* sin parámetros devuelve órdenes para todos los símbolos.

El parámetro opcional con nombre *symbol* permite especificar un nombre de símbolo concreto para la selección de órdenes.

El parámetro opcional con nombre *group* sirve para especificar un patrón de búsqueda utilizando el carácter comodín '*' (como sustituto de un número arbitrario de caracteres cualesquiera, incluido el carácter cero en el lugar dado del patrón) y el carácter de negación lógica de la condición '!'. El principio de funcionamiento de la plantilla de filtros se describió en la sección Obtener información sobre instrumentos financieros.

Si se especifica el parámetro *ticket*, se busca en un orden determinado.

En una sola llamada a la función, puede obtener todas las órdenes activas. Se trata de un análogo del uso combinado de las funciones *OrdersTotal*, *OrderSelect*, y *OrderGet*.

En el siguiente ejemplo (*MQL5/Scripts/MQL5Book/Python/ordersget.py*), solicitamos información sobre las órdenes de diferentes maneras.

```
import MetaTrader5 as mt5
import pandas as pd
pd.set_option('display.max_columns', 500) # how many columns to show
pd.set_option('display.width', 1500)      # max. table width to display

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# display information about active orders on the GBPUSD symbol
orders = mt5.orders_get(symbol = "GBPUSD")
if orders is None:
   print("No orders on GBPUSD, error code={}".format(mt5.last_error()))
else:
   print("Total orders on GBPUSD:", len(orders))
   # display all active orders
   for order in orders:
      print(order)
print()

# getting a list of orders on symbols whose names contain "*GBP*"
gbp_orders = mt5.orders_get(group="*GBP*")
if gbp_orders is None:
   print("No orders with group=\"*GBP*\", error code={}".format(mt5.last_error()))
else:
   print("orders_get(group=\"*GBP*\")={}".format(len(gbp_orders)))
   # display orders as a table using pandas.DataFrame
   df = pd.DataFrame(list(gbp_orders), columns = gbp_orders[0]._asdict().keys())
   df.drop(['time_done', 'time_done_msc', 'position_id', 'position_by_id',
   'reason', 'volume_initial', 'price_stoplimit'], axis = 1, inplace = True)
   df['time_setup'] = pd.to_datetime(df['time_setup'], unit = 's')
   print(df)

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()
```

A continuación se muestra el resultado de muestra:

```
Total orders on GBPUSD: 2
TradeOrder(ticket=554733548, time_setup=1585153667, time_setup_msc=1585153667718, tim
TradeOrder(ticket=554733621, time_setup=1585153671, time_setup_msc=1585153671419, tim

orders_get(group="*GBP*")=4
ticket time_setup time_setup_msc type ... volume_current price_open sl tp price_curre
0 554733548 2020-03-25 16:27:47 1585153667718 3 ... 0.2 1.25379 0.0 0.0 1.16803 GBPUS
1 554733621 2020-03-25 16:27:51 1585153671419 2 ... 0.2 1.14370 0.0 0.0 1.16815 GBPUS
2 554746664 2020-03-25 16:38:14 1585154294401 3 ... 0.2 0.93851 0.0 0.0 0.92428 EURGB
3 554746710 2020-03-25 16:38:17 1585154297022 2 ... 0.2 0.90527 0.0 0.0 0.92449 EURGB
```

7.9.14 Obtener el número y la lista de posiciones abiertas

La función *positions_total* devuelve el número de posiciones abiertas.

int positions_total()

La función es un análogo de *PositionsTotal*.

Para obtener información detallada sobre cada posición, utilice la función *positions_get*, que dispone de múltiples opciones. Todas las variantes devuelven un array de tuplas de nombre *TradePosition* con las claves correspondientes a las propiedades de posición (véanse los elementos de ENUM_POSITION_PROPERTY_enumerations, sin el prefijo «POSITION_», en minúsculas). En caso de error, el resultado es *None*.

namedtuple[] positions_get()
namedtuple[] positions_get(symbol = <"SYMBOL">)
namedtuple[] positions_get(group = <"PATTERN">)
namedtuple[] positions_get(ticket = <TICKET>)

La función sin parámetros devuelve todas las posiciones abiertas.

La función con el parámetro *symbol* permite la selección de posiciones para el símbolo especificado.

La función con el parámetro *group* proporciona filtrado por máscara de búsqueda con comodines '*' (se sustituye cualquier carácter) y negación lógica de la condición '!'. Para obtener más detalles, consulte la sección Obtener información sobre instrumentos financieros.

Una versión con los parámetros *ticket* selecciona una posición con un ticket específico (propiedad POSITION_TICKET).

La función *positions_get* se puede utilizar para obtener todas las posiciones y sus propiedades en una sola llamada, lo que lo hace similar a un montón de funciones *PositionsTotal*, *PositionSelect* y *PositionGet*.

En el script *MQL5/Scripts/MQL5Book/Python/positionsget.py*, solicitamos posiciones para un símbolo específico y una máscara de búsqueda.

```
import MetaTrader5 as mt5
import pandas as pd
pd.set_option('display.max_columns', 500) # how many columns to show
pd.set_option('display.width', 1500)      # max. table width to display

# let's establish a connection to the MetaTrader 5 terminal
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# get open positions on USDCHF
positions = mt5.positions_get(symbol = "USDCHF")
if positions == None:
   print("No positions on USDCHF, error code={}".format(mt5.last_error()))
elif len(positions) > 0:
   print("Total positions on USDCHF =", len(positions))
 # display all open positions
   for position in positions:
      print(position)

# get a list of positions on symbols whose names contain "*USD*"
usd_positions = mt5.positions_get(group = "*USD*")
if usd_positions == None:
    print("No positions with group=\"*USD*\", error code={}".format(mt5.last_error())
elif len(usd_positions) > 0:
   print("positions_get(group=\"*USD*\") = {}".format(len(usd_positions)))
   # display the positions as a table using pandas.DataFrame
   df=pd.DataFrame(list(usd_positions), columns = usd_positions[0]._asdict().keys())
   df['time'] = pd.to_datetime(df['time'], unit='s')
   df.drop(['time_update', 'time_msc', 'time_update_msc', 'external_id'],
   axis=1, inplace=True)
   print(df)

# complete the connection to the MetaTrader 5 terminal
mt5.shutdown()
```

He aquí cuál podría ser el resultado:

```
Total positions on USDCHF = 1
TradePosition(ticket=1468454363, time=1664217233, time_msc=1664217233239, time_update
time_update_msc=1664217233239, type=1, magic=0, identifier=1468454363, reason=0, volu
sl=0.0, tp=0.0, price_current=0.9853, swap=-0.01, profit=6.24, symbol='USDCHF', comme
positions_get(group="*USD*") = 2
ticket time type ... identifier volume price_open ... _current swap profit symbol com
0 1468454363 2022-09-26 18:33:53 1 ... 1468454363 0.01 0.99145 ... 0.98530 -0.01 6.24
1 1468475849 2022-09-26 18:44:00 0 ... 1468475849 0.01 1.06740 ... 1.08113 0.00 13.73
```

7.9.15 Leer el historial de órdenes y transacciones

También es posible trabajar con órdenes y transacciones en el historial de la cuenta utilizando scripts de Python. Para ello existen las funciones *history_orders_total*, *history_orders_get*, *history_deals_total* y *history_deals_get*.

int history_orders_total(date_from, date_to)

La función *history_orders_total* devuelve el número de órdenes del historial de trading en el intervalo de tiempo especificado. Cada uno de los parámetros se establece mediante el objeto *datetime* o como el número de segundos transcurridos desde 1970.01.01.

La función es un análogo de *HistoryOrdersTotal*.

La función *history_orders_get* está disponible en varias versiones y admite el filtrado de órdenes por subcadena en el nombre del símbolo, ticket o ID de posición. Todas las variantes devuelven un array de tuplas con nombre *TradeOrder* (los nombres de campo coinciden con ENUM_ORDER_PROPERTY_enumerations sin el prefijo «ORDER_» y en minúsculas). Si no hay órdenes coincidentes, el array estará vacío. En caso de error, la función devolverá *None*.

namedtuple[] history_orders_get(date_from, date_to, group = <"PATTERN">)
namedtuple[] history_orders_get(ticket = <ORDER_TICKET>)
namedtuple[] history_orders_get(position = <POSITION_ID>)

La primera versión selecciona las órdenes dentro del intervalo de tiempo especificado (similar a *history_orders_total*). En el parámetro opcional con nombre *group*, puede especificar un patrón de búsqueda para una subcadena del nombre del símbolo (puede utilizar los caracteres comodín '*' y la negación '!' en él, consulte la sección Obtener información sobre instrumentos financieros).

La segunda versión está diseñada para buscar una orden específica por su ticket.

La última versión selecciona las órdenes por ID de posición (propiedad ORDER_POSITION_ID).

Cualquiera de las dos opciones equivale a llamar a varias funciones - *HistoryOrdersTotal*, *HistoryOrderSelect* y *HistoryOrderGet* - de MQL5.

Veamos en un ejemplo del script *historyordersget.py* sobre cómo obtener el número y la lista de órdenes históricas para diferentes condiciones.

```
from datetime import datetime
import MetaTrader5 as mt5
import pandas as pd
pd.set_option('display.max_columns', 500) # how many columns to show
pd.set_option('display.width', 1500)      # max. table width for display
...
# get the number of orders in the history for the period (total and *GBP*)
from_date = datetime(2022, 9, 1)
to_date = datetime.now()
total = mt5.history_orders_total(from_date, to_date)
history_orders=mt5.history_orders_get(from_date, to_date, group="*GBP*")
# print(history_orders)
if history_orders == None:
   print("No history orders with group=\"*GBP*\", error code={}".format(mt5.last_erro
else :
   print("history_orders_get({}, {}, group=\"*GBP*\")={} of total {}".format(from_dat
   to_date, len(history_orders), total))

# display all canceled historical orders for ticket position 0
position_id = 0
position_history_orders = mt5.history_orders_get(position = position_id)
if position_history_orders == None:
   print("No orders with position #{}".format(position_id))
   print("error code =", mt5.last_error())
elif len(position_history_orders) > 0:
   print("Total history orders on position #{}: {}".format(position_id,
   len(position_history_orders)))
   # display received orders as is
   for position_order in position_history_orders:
      print(position_order)
   # display these orders as a table using pandas.DataFrame
   df = pd.DataFrame(list(position_history_orders),
   columns = position_history_orders[0]._asdict().keys())
   df.drop(['time_expiration', 'type_time', 'state', 'position_by_id', 'reason', 'vol
   'price_stoplimit','sl','tp', 'time_setup_msc', 'time_done_msc', 'type_filling', 'e
   axis = 1, inplace = True)
   df['time_setup'] = pd.to_datetime(df['time_setup'], unit='s')
   df['time_done'] = pd.to_datetime(df['time_done'], unit='s')
   print(df)
...
```

Este es el resultado del script (indicado con abreviaturas):

```
history_orders_get(2022-09-01 00:00:00, 2022-09-26 21:50:04, group="*GBP*")=15 of total 44

Total history orders on position #0: 14
TradeOrder(ticket=1437318706, time_setup=1661348065, time_setup_msc=1661348065049, time_done=166134
time_done_msc=1661348083632, time_expiration=0, type=2, type_time=0, type_filling=2, state=2, magic
position_id=0, position_by_id=0, reason=3, volume_initial=0.01, volume_current=0.01, price_open=0.9
sl=0.0, tp=0.0, price_current=0.99311, price_stoplimit=0.0, symbol='EURUSD', comment='', external_i
TradeOrder(ticket=1437331579, time_setup=1661348545, time_setup_msc=1661348545750, time_done=166134
time_done_msc=1661348551354, time_expiration=0, type=2, type_time=0, type_filling=2, state=2, magic
position_id=0, position_by_id=0, reason=3, volume_initial=0.01, volume_current=0.01, price_open=0.9
sl=0.0, tp=0.0, price_current=0.99284, price_stoplimit=0.0, symbol='EURUSD', comment='', external_i
TradeOrder(ticket=1437331739, time_setup=1661348553, time_setup_msc=1661348553935, time_done=166134
time_done_msc=1661348563412, time_expiration=0, type=2, type_time=0, type_filling=2, state=2, magic
position_id=0, position_by_id=0, reason=3, volume_initial=0.01, volume_current=0.01, price_open=0.9
sl=0.0, tp=0.0, price_current=0.99286, price_stoplimit=0.0, symbol='EURUSD', comment='', external_i
...

ticket time_setup time_done type ... _initial price_open price_current symbol comment
0 1437318706 2022-08-24 13:34:25 2022-08-24 13:34:43 2 0.01 0.99301 0.99311 EURUSD
1 1437331579 2022-08-24 13:42:25 2022-08-24 13:42:31 2 0.01 0.99281 0.99284 EURUSD
2 1437331739 2022-08-24 13:42:33 2022-08-24 13:42:43 2 0.01 0.99285 0.99286 EURUSD
...
```

Podemos ver que en septiembre sólo hubo 44 órdenes, 15 de las cuales incluían la divisa GBP (un número impar debido a la posición abierta). El historial contiene 14 órdenes canceladas.

int history_deals_total(date_from, date_to)

La función *history_deals_total* devuelve el número de transacciones en el historial para el periodo especificado.

La función es un análogo de *HistoryDealsTotal*.

La función *history_deals_get* tiene varias formas y ha sido diseñada para seleccionar operaciones con la posibilidad de filtrar por ticket de orden o ID de posición. Todas las formas de la función devuelven un array de tuplas con nombre *TradeDeal*, cuyos campos reflejan las propiedades de la función ENUM_DEAL_PROPERTY_enumerations (se ha eliminado el prefijo «DEAL_» de los nombres de campo y se ha aplicado la minúscula). En caso de error, obtenemos *None*.

namedtuple[] history_deals_get(date_from, date_to, group = <"PATTERN">)
namedtuple[] history_deals_get(ticket = <ORDER_TICKET>)
namedtuple[] history_deals_get(position = <POSITION_ID>)

La primera forma de la función es similar a la solicitud de órdenes históricas mediante *history_orders_get*.

El segundo formulario permite la selección de transacciones generadas por una orden específica por su ticket (la propiedad DEAL_ORDER).

Por último, el tercer formulario solicita las transacciones que han formado una posición con un ID determinado (la propiedad DEAL_POSITION_ID).

La función permite obtener todas las transacciones junto con sus propiedades en una sola llamada, lo que es análogo al montón de funciones *HistoryDealsTotal*, *HistoriaDealSelect* y *HistoriaDealGet*.

Aquí está la parte principal del script de prueba *historydealsget.py*.

```
# set the time range
from_date = datetime(2020, 1, 1)
to_date = datetime.now()

# get trades for symbols whose names do not contain either "EUR" or "GBP"
deals = mt5.history_deals_get(from_date, to_date, group="*,!*EUR*,!*GBP*")
if deals == None:
   print("No deals, error code={}".format(mt5.last_error()))
elif len(deals) > 0:
   print("history_deals_get(from_date, to_date, group=\"*,!*EUR*,!*GBP*\") =",
   len(deals))
   # display all received deals as they are
   for deal in deals:
      print("  ",deal)
   # display these trades as a table using pandas.DataFrame
   df = pd.DataFrame(list(deals), columns = deals[0]._asdict().keys())
   df['time'] = pd.to_datetime(df['time'], unit='s')
   df.drop(['time_msc','commission','fee'], axis = 1, inplace = True)
   print(df)
```

Un ejemplo de resultado:

```
history_deals_get(from_date, to_date, group="*,!*EUR*,!*GBP*") = 12
TradeDeal(ticket=1109160642, order=0, time=1632188460, time_msc=1632188460852, type=2
TradeDeal(ticket=1250629232, order=1268074569, time=1645709385, time_msc=164570938581
TradeDeal(ticket=1250639814, order=1268085019, time=1645709950, time_msc=164570995061
TradeDeal(ticket=1250639928, order=1268085129, time=1645709955, time_msc=164570995550
TradeDeal(ticket=1250640111, order=1268085315, time=1645709965, time_msc=164570996514
TradeDeal(ticket=1250640309, order=1268085512, time=1645709973, time_msc=164570997362
TradeDeal(ticket=1250640400, order=1268085611, time=1645709978, time_msc=164570997870
TradeDeal(ticket=1250640616, order=1268085826, time=1645709988, time_msc=164570998827
TradeDeal(ticket=1250640810, order=1268086019, time=1645709996, time_msc=164570999699
TradeDeal(ticket=1445796125, order=1468026008, time=1664199450, time_msc=166419945048
TradeDeal(ticket=1445796155, order=1468026041, time=1664199452, time_msc=166419945256
TradeDeal(ticket=1446217804, order=1468454363, time=1664217233, time_msc=166421723323

ticket order time t… e… … position_id volume price profit symbol comment external_id
0 1109160642 0 2021-09-21 01:41:00 2 0 0 0.00 0.00000 10000.00
1 1250629232 1268074569 2022-02-24 13:29:45 0 0 1268074569 0.01 1970.98000 0.00 XAUUS
2 1250639814 1268085019 2022-02-24 13:39:10 1 1 1268074569 0.01 1970.09000 -0.89 XAUU
3 1250639928 1268085129 2022-02-24 13:39:15 1 0 1268085129 0.01 1969.98000 0.00 XAUUS
4 1250640111 1268085315 2022-02-24 13:39:25 0 1 1268085129 0.01 1970.17000 -0.19 XAUU
5 1250640309 1268085512 2022-02-24 13:39:33 1 0 1268085512 0.10 1970.09000 0.00 XAUUS
6 1250640400 1268085611 2022-02-24 13:39:38 0 1 1268085512 0.10 1970.22000 -1.30 XAUU
7 1250640616 1268085826 2022-02-24 13:39:48 1 0 1268085826 1.10 1969.95000 0.00 XAUUS
8 1250640810 1268086019 2022-02-24 13:39:56 0 1 1268085826 1.10 1969.88000 7.70 XAUUS
9 1445796125 1468026008 2022-09-26 13:37:30 0 0 1468026008 0.10 144.13200 0.00 USDJPY
10 1445796155 1468026041 2022-09-26 13:37:32 1 1 1468026008 0.10 144.12400 -0.56 USDJ
11 1446217804 1468454363 2022-09-26 18:33:53 1 0 1468454363 0.01 0.99145 0.00 USDCHF
```

7.10 Soporte integrado para computación paralela: OpenCL

OpenCL es un estándar abierto de programación paralela que permite crear aplicaciones para su ejecución simultánea en muchos núcleos de procesadores modernos, diferentes en arquitectura, en particular, gráfica (GPU) o central (CPU).

En otras palabras: OpenCL permite utilizar todos los núcleos del procesador central o toda la potencia de cálculo de la tarjeta de vídeo para computar una tarea, lo que en última instancia reduce el tiempo de ejecución del programa. Por lo tanto, OpenCL es muy útil para tareas de cálculo intensivo, pero es importante tener en cuenta que los algoritmos para resolver estas tareas deben ser divisibles en hilos paralelos. Por ejemplo, el entrenamiento de redes neuronales, la transformada de Fourier o la resolución de sistemas de ecuaciones de grandes dimensiones.

Por ejemplo, en relación con los aspectos específicos del trading, se puede lograr un aumento del rendimiento con un script, indicador o Asesor Experto que realice un análisis complejo y prolongado de los datos históricos de varios símbolos y plazos, y cuyo cálculo no dependa de otros.

> Al mismo tiempo, los principiantes suelen preguntarse si es posible acelerar las pruebas y la optimización de los Asesores Expertos utilizando OpenCL. La respuesta a ambas preguntas es no. Las pruebas reproducen el proceso real de trading secuencial y, por tanto, cada barra o tick siguiente depende de los resultados de los anteriores, lo que hace imposible paralelizar los cálculos de una pasada. En cuanto a la optimización, los agentes del probador sólo admiten núcleos de CPU. Esto se debe a la complejidad de un análisis completo de las cotizaciones o ticks, el seguimiento de las posiciones y el cálculo del saldo y el valor negociable. Sin embargo, si la complejidad no le asusta, puede implementar su propio motor de optimización en los núcleos de la tarjeta gráfica transfiriendo a OpenCL todos los cálculos que emulan el entorno de trading con la fiabilidad necesaria.

OpenCL significa Open Computing Language, o lenguaje de computación abierto. Es similar a los lenguajes C y C++ y, por tanto, a MQL5. Sin embargo, para preparar («compilar») un programa OpenCL, pasarle datos de entrada, ejecutarlo en paralelo en varios núcleos y obtener resultados de cálculo, se utiliza una interfaz de programación (un conjunto de funciones) especial. Esta API de OpenCL también está disponible para programas MQL que deseen implementar la ejecución en paralelo.

Para utilizar OpenCL no es necesario disponer de una tarjeta de vídeo en el PC, ya que basta con la presencia de un procesador central. En cualquier caso, se necesitan controladores especiales del fabricante (se requiere OpenCL versión 1.1 y superior). Si su ordenador tiene juegos u otros programas (por ejemplo, científicos, de edición de vídeo, etc.) que funcionan directamente con tarjetas de vídeo, lo más probable es que ya disponga de la capa de software necesaria. Esto se puede comprobar probando a ejecutar un programa MQL en el terminal con una llamada a OpenCL (al menos un ejemplo sencillo de la entrega del terminal; véase más adelante).

Si no hay compatibilidad con OpenCL verá un error en el registro.

```
OpenCL OpenCL not found, please install OpenCL drivers
```

Si hay un dispositivo adecuado en su ordenador y se ha habilitado la compatibilidad con OpenCL para él, el terminal mostrará un mensaje con el nombre y el tipo de este dispositivo (puede haber varios dispositivos). Por ejemplo:

```
OpenCL Device #0: CPU GenuineIntel Intel(R) Core(TM) i7-2700K CPU @ 3.50GHz with Open
OpenCL Device #1: GPU Intel(R) Corporation Intel(R) UHD Graphics 630 with OpenCL 2.1
```

El procedimiento de instalación de los controladores de los distintos dispositivos se describe en el artículo en mql5.com. La compatibilidad se extiende a los dispositivos más populares de Intel, AMD, ATI y Nvidia.

En cuanto al número de núcleos y la velocidad de computación distribuida, los procesadores centrales son significativamente inferiores a las tarjetas gráficas, pero un buen procesador central multinúcleo bastará para aumentar considerablemente el rendimiento.

> Importante: si su ordenador dispone de una tarjeta de vídeo compatible con OpenCL, no es necesario que instale la emulación de software OpenCL en la CPU.

Los controladores de dispositivos OpenCL automatizan la distribución de los cálculos entre los núcleos. Por ejemplo, si necesita realizar un millón de cálculos del mismo tipo con vectores diferentes y solo tiene mil núcleos a su disposición, los controladores iniciarán automáticamente cada tarea siguiente a medida que las anteriores estén listas y se liberen los núcleos.

Las operaciones preparatorias para configurar el entorno de ejecución OpenCL en un programa MQL se realizan una sola vez utilizando las funciones de la API de OpenCL anterior.

1. Crear un contexto para un programa OpenCL (selección de un dispositivo, como una tarjeta de vídeo, una CPU o cualquier otro disponible): *CLContextCreate(CL_USE_ANY)*. La función devolverá un descriptor de contexto (un número entero, denotémoslo condicionalmente *ContextHandle*).

2. Crear un programa OpenCL en el contexto recibido: se compila a partir del código fuente en el lenguaje OpenCL mediante la llamada a la función *CLProgramCreate*, a la que se pasa el texto del código a través del parámetro *Source*:*CLProgramCreate(ContextHandle, Source, BuildLog)*. La función devolverá el manejador del programa (número entero *ProgramHandle*). Es importante señalar aquí que, dentro del código fuente de este programa, debe haber funciones (al menos una) marcadas con una palabra clave especial *__kernel* (o simplemente *kernel*): contienen las partes del algoritmo que se van a paralelizar (véase el ejemplo más abajo). Por supuesto, para simplificar (descomponer el código fuente), el programador puede dividir las subtareas lógicas de la función del kernel en otras funciones auxiliares y llamarlas desde el kernel: al mismo tiempo, no es necesario marcar las funciones auxiliares con la palabra *kernel*.

3. Registrar un kernel que se va a ejecutar por el nombre de una de esas funciones que están marcadas en el código del programa OpenCL como formadoras de kernel: *CLKernelCreate(ProgramHandle, KernelName)*. La llamada a esta función devolverá un manejador al kernel (un entero, digamos, *KernelHandle*). Puede preparar muchas funciones diferentes en código OpenCL y registrarlas como distintos kernels.

4. Si es necesario, crear búferes para los arrays de datos pasados por referencia al kernel y para los valores/arrays devueltos: *CLBufferCreate(ContextHandle, Size * sizeof(double), CL_MEM_READ_WRITE),* etc. Los búferes también se identifican y gestionan con descriptores.

A continuación, una o varias veces, si es necesario, (por ejemplo, en manejadores de eventos de indicadores o Asesores Expertos), los cálculos se realizan directamente según el siguiente esquema:

I. Pasar datos de entrada y/o vincular búferes de entrada/salida con *CLSetKernelArg(KernelHandle,...)* y/o *CLSetKernelArgMem(KernelHandle,..., BufferHandle)*. La primera función permite fijar un valor escalar, y la segunda equivale a pasar o recibir un valor (o un array de valores) por referencia. En esta fase, los datos se trasladan de MQL5 al núcleo de ejecución de OpenCL.

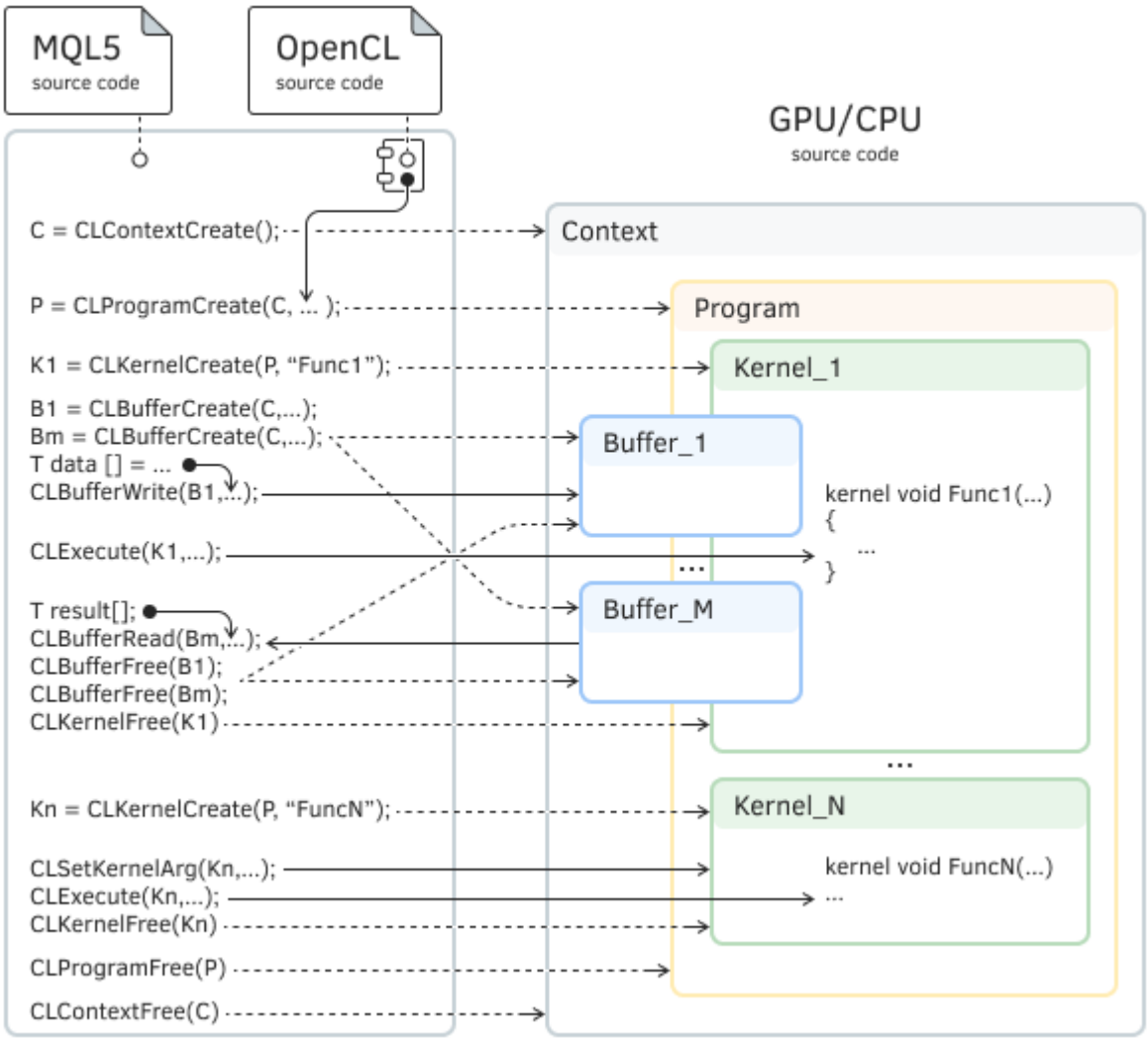
CLBufferWrite(BufferHandle,...) escribe datos en el búfer. Los parámetros y búferes estarán disponibles para el programa OpenCL durante la ejecución del kernel.

II. Realizar cálculos paralelos llamando a un kernel específico *CLExecute(KernelHandle,...)*. La función del kernel podrá escribir los resultados de su trabajo en el búfer de salida.

III. Obtener resultados con *CLBufferRead(BufferHandle)*. En esta fase, los datos vuelven de OpenCL a MQL5.

Una vez finalizados los cálculos, deben liberarse todos los descriptores: *CLBufferFree(BufferHandle)*,*CLKernelFree(KernelHandle)*, *CLProgramFree(ProgramHandle)* y *CLContextFree(ContextHandle)*.

Esta secuencia se indica convencionalmente en el siguiente diagrama.



Esquema de interacción entre un programa MQL y un archivo adjunto OpenCL

Se recomienda escribir el código fuente de OpenCL en archivos de texto separados, que luego se pueden conectar al programa MQL5 utilizando variables de recursos.

La biblioteca de cabecera estándar suministrada con el terminal contiene una clase envoltorio para trabajar con OpenCL: *MQL5/Include/OpenCL/OpenCL.mqh*.

Encontrará ejemplos de uso de OpenCL en la carpeta *MQL5/Scripts/Examples/OpenCL/*. En particular, el script *MQL5/Scripts/Examples/OpenCL/Double/Wavelet.mq5* produce una transformada de ondícula (wavelet) de las series temporales (puede tomar una curva artificial según el modelo estocástico de Weierstrass o el incremento de los precios del instrumento financiero actual). En cualquier caso, los datos iniciales para el algoritmo son un array que es una imagen bidimensional de una serie.

Cuando se ejecuta este script, igual que cuando se ejecuta cualquier otro programa MQL con código OpenCL, el terminal seleccionará el dispositivo más rápido (si hay varios, y el dispositivo específico no se seleccionó en el propio programa o no estaba ya definido anteriormente). La información al respecto se muestra en la pestaña *Journal* (registro del terminal, no expertos).

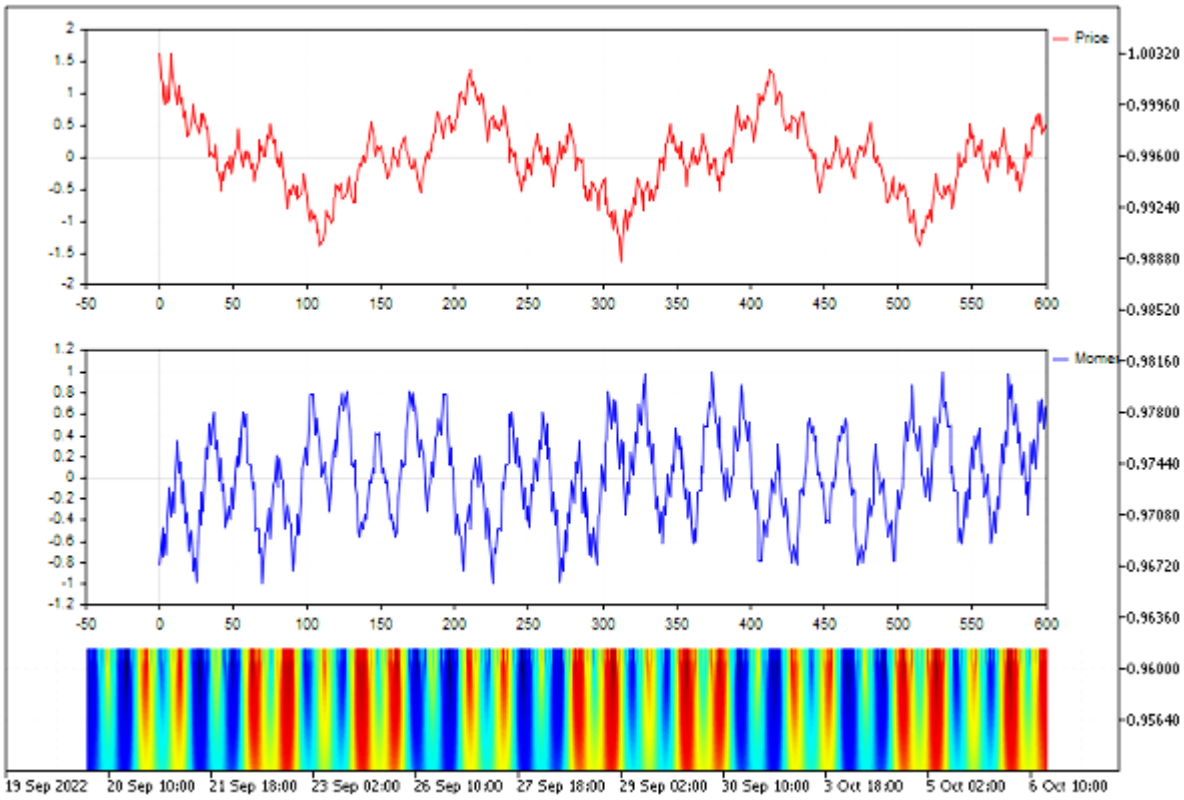
```
Scripts script Wavelet (EURUSD,H1) loaded successfully
OpenCL  device #0: GPU NVIDIA Corporation NVIDIA GeForce GTX 1650 with OpenCL 3.0 (16
OpenCL  device #1: GPU Intel(R) Corporation Intel(R) UHD Graphics 630 with OpenCL 3.0
OpenCL  device performance test started
OpenCL  device performance test successfully finished
OpenCL  device #0: GPU NVIDIA Corporation NVIDIA GeForce GTX 1650 with OpenCL 3.0 (16
OpenCL  device #1: GPU Intel(R) Corporation Intel(R) UHD Graphics 630 with OpenCL 3.0
Scripts script Wavelet (EURUSD,H1) removed
```

Como resultado de la ejecución, el script muestra en la pestaña *Experts* registros con mediciones de la velocidad de cálculo de la forma habitual (en serie, en la CPU) y en paralelo (en núcleos OpenCL).

```
OpenCL: GPU device 'Intel(R) UHD Graphics 630' selected
time CPU=5235 ms, time GPU=125 ms, CPU/GPU ratio: 41.880000
```

La relación de velocidades, dependiendo de las particularidades de la tarea, puede llegar a decenas.

El script muestra en el gráfico la imagen original, su derivada en forma de incrementos y el resultado de la transformada wavelet.



La serie simulada original, sus incrementos y la transformada wavelet

Tenga en cuenta que los objetos gráficos permanecen en el gráfico después de que el script haya terminado de funcionar. Deberán eliminarse manualmente.

Este es el aspecto del código fuente de OpenCL de la transformada wavelet, implementado en un archivo independiente *MQL5/Scripts/Examples/OpenCL/Double/Kernels/wavelet.cl*.

```
// increased calculation accuracy double is required
// (by default, without this directive we get float)
#pragma OPENCL EXTENSION cl_khr_fp64 : enable

// Helper function Morlet
double Morlet(const double t)
{
   return exp(-t * t * 0.5) * cos(M_2_PI * t);
}

// OpenCL kernel function
__kernel void Wavelet_GPU(__global double *data, int datacount,
   int x_size, int y_size, __global double *result)
{
   size_t i = get_global_id(0);
   size_t j = get_global_id(1);
   double a1 = (double)10e-10;
   double a2 = (double)15.0;
   double da = (a2 - a1) / (double)y_size;
   double db = ((double)datacount - (double)0.0) / x_size;
   double a = a1 + j * da;
   double b = 0 + i * db;
   double B = (double)1.0;
   double B_inv = (double)1.0 / B;
   double a_inv = (double)1.0 / a;
   double dt = (double)1.0;
   double coef = (double)0.0;

   for(int k = 0; k < datacount; k++)
   {
      double arg = (dt * k - b) * a_inv;
      arg = -B_inv * arg * arg;
      coef = coef + exp(arg);
   }

   double sum = (float)0.0;
   for(int k = 0; k < datacount; k++)
   {
      double arg = (dt * k - b) * a_inv;
      sum += data[k] * Morlet(arg);
   }
   sum = sum / coef;
   uint pos = (int)(j * x_size + i);
   result[pos] = sum;
```

```
}
```

Encontrará información completa sobre la sintaxis, las funciones integradas y los principios de funcionamiento de OpenCL en el sitio web oficial de Khronos Group.

En particular, es interesante observar que OpenCL admite no solo los tipos de datos numéricos escalares habituales (empezando por *char* y terminando por *double*), sino también el vector *(u)charN, (u)shortN, (u)intN, (u)longN, floatN, doubleN*, donde N = {2|3|4|8|16} y denota la longitud del vector. En este ejemplo no se utiliza.

Además de la mencionada palabra clave *kernel*, un papel importante en la organización de la computación paralela lo desempeña la función *get_global_id*: permite encontrar en el código el número de la subtarea computacional que se está ejecutando en ese momento. Obviamente, los cálculos en diferentes subtareas deben ser diferentes (de lo contrario, no tendría sentido utilizar muchos núcleos). En este ejemplo, dado que la tarea implica el análisis de una imagen bidimensional, es más conveniente identificar sus fragmentos utilizando dos coordenadas ortogonales. En el código anterior, los obtenemos mediante dos llamadas, *get_global_id(0)* y *get_global_id(1)*.

De hecho, nosotros mismos establecemos la dimensión de los datos para la tarea cuando llamamos a la función MQL5 *CLExecute* (véase más adelante).

En el archivo *Wavelet.mq5* se incluye el código fuente OpenCL utilizando la directiva:

```
#resource "Kernels/wavelet.cl" as string cl_program
```

El tamaño de la imagen se establece mediante macros:

```
#define SIZE_X 600
#define SIZE_Y 200
```

Para gestionar OpenCL se utiliza la biblioteca estándar con la clase *COpenCL*. Sus métodos tienen nombres similares y utilizan internamente las correspondientes funciones OpenCL integradas de la API de MQL5. Le sugerimos que se familiarice con ella.

```
#include <OpenCL/OpenCL.mqh>
```

De forma simplificada (sin comprobación de errores ni visualización) se muestra a continuación el código MQL que lanza la transformación. Las acciones relacionadas con la transformada wavelet se resumen en la clase *CWavelet*.

```
class CWavelet
{
protected:
   ...
   int        m_xsize;              // image dimensions along the axes
   int        m_ysize;
   double     m_wavelet_data_GPU[]; // result goes here
   COpenCL    m_OpenCL;             // wrapper object
   ...
};
```

La computación paralela principal está organizada por su método *CalculateWavelet_GPU*.

```
bool CWavelet::CalculateWavelet_GPU(double &data[], uint &time)
{
   int datacount = ArraySize(data); // image size (number of dots)

   // compile the cl-program according to its source code
   m_OpenCL.Initialize(cl_program, true);

   // register a single kernel function from the cl file
   m_OpenCL.SetKernelsCount(1);
   m_OpenCL.KernelCreate(0, "Wavelet_GPU");

   // register 2 buffers for input and output data, write the input array
   m_OpenCL.SetBuffersCount(2);
   m_OpenCL.BufferFromArray(0, data, 0, datacount, CL_MEM_READ_ONLY);
   m_OpenCL.BufferCreate(1, m_xsize * m_ysize * sizeof(double), CL_MEM_READ_WRITE);
   m_OpenCL.SetArgumentBuffer(0, 0, 0);
   m_OpenCL.SetArgumentBuffer(0, 4, 1);

   ArrayResize(m_wavelet_data_GPU, m_xsize * m_ysize);
   uint work[2];              // task of analyzing a two-dimensional image - hence th
   uint offset[2] = {0, 0};   // start from the very beginning (or you can skip somet
   work[0] = m_xsize;
   work[1] = m_ysize;

   // set input data
   m_OpenCL.SetArgument(0, 1, datacount);
   m_OpenCL.SetArgument(0, 2, m_xsize);
   m_OpenCL.SetArgument(0, 3, m_ysize);

   time = GetTickCount();     // cutoff time for speed measurement
   // start computing on the GPU, two-dimensional task
   m_OpenCL.Execute(0, 2, offset, work);

   // get results into output buffer
   m_OpenCL.BufferRead(1, m_wavelet_data_GPU, 0, 0, m_xsize * m_ysize);

   time = GetTickCount() - time;

   m_OpenCL.Shutdown(); // free all resources - call all necessary functions CL***Fre
   return true;
}
```

En el código fuente del ejemplo hay una línea comentada que llama a *PreparePriceData* para preparar un array de entrada basado en precios reales: puede activarlo en lugar de la línea anterior con la llamada a *PrepareModelData* (que genera un número artificial).

```
void OnStart()
{
   int momentum_period = 8;
   double price_data[];
   double momentum_data[];
   PrepareModelData(price_data, SIZE_X + momentum_period);

   // PreparePriceData("EURUSD", PERIOD_M1, price_data, SIZE_X + momentum_period);

   PrepareMomentumData(price_data, momentum_data, momentum_period);
   ... // visualization of the series and increments
   CWavelet wavelet;
   uint time_gpu = 0;
   wavelet.CalculateWavelet_GPU(momentum_data, time_gpu);
   ... // visualization of the result of the wavelet transform
}
```

Se ha asignado un conjunto especial de códigos de error (con el prefijo ERR_OPENCL_, empezando por el código 5100, ERR_OPENCL_NOT_SUPPORTED) para las operaciones con OpenCL. Los códigos se describen en la ayuda. Si hay problemas con la ejecución de programas OpenCL, el terminal emite diagnósticos detallados en el registro, indicando los códigos de error.

Conclusión

Con esta sección concluye el libro. A lo largo de siete partes y numerosos capítulos, hemos explorado diversos aspectos de la programación MQL5, partiendo de los fundamentos del lenguaje y avanzando hasta las sofisticadas tecnologías asociadas que hacen posible una transición gradual desde la creación de herramientas individuales específicas para el operador de trading a complejos sistemas y productos de trading.

Los conocimientos que adquiera le ayudarán a dar vida a diversas ideas y a alcanzar el éxito en el mundo del trading algorítmico profesional.

- Desarrolle aplicaciones y véndalas en el Mercado, el mayor almacén de programas para MetaTrader con una infraestructura a disposición de los autores. El Mercado proporciona acceso a una enorme audiencia, ofreciendo protección de productos y licencias junto con un sistema integrado para aceptar pagos.
- Desarrolle aplicaciones personalizadas mediante Freelance. Acceda a toda la gama de órdenes de desarrollo y benefíciese de un cómodo sistema de trabajo y protección de pagos.
- Comparta su experiencia publicando su código en la librería de códigos fuente. Presente sus programas a miles de traders de la comunidad MQL5.

Y, por supuesto, siga aprendiendo. En el sitio web www.mql5.com se ofrece abundante información y algoritmos listos para usar:

- Artículos de programaciónen los que autores profesionales abordan problemas prácticos.
- Foro donde podrá intercambiar experiencias y pedir consejo a otros desarrolladores.
- librería de códigos fuente con códigos fuente de programas que le ayudarán a descubrir las capacidades de los lenguajes MQL5 y a crear sus propios programas.

Por último, me gustaría recordarle que el desarrollo de software no sólo tiene que ver con programar, sino también con muchas otras tareas igualmente importantes: redactar especificaciones técnicas (aunque sólo sea para Ud. mismo), diseñar, crear prototipos, crear el diseño de la interfaz de usuario y proporcionar documentación y asistencia posterior. Todos estos aspectos influyen significativamente en la eficacia de su trabajo como programador y en la calidad del resultado final.

En concreto, la mayoría de las tareas prácticas pueden descomponerse en principios y algoritmos estándar que los programadores de distintos lenguajes llevan mucho tiempo utilizando. Ello incluye patrones de diseño, conjuntos de estructuras de datos optimizadas para tareas específicas y herramientas para automatizar el desarrollo. Todo esto se debe aplicar en la plataforma MetaTrader 5 con la ayuda de MQL5 y de forma adicional al mismo. El libro es sólo el primer paso en el camino hacia el crecimiento profesional.

¡Únase a www.mql5.com, la comunidad de desarrolladores de robots de trading!