ПРОГРАММИРОВАНИЕ НА MQL5 ДЛЯ ТРЕЙДЕРОВ

Станислав Короткий

MetaQuotes

© 2000-2024, MetaQuotes Ltd

Содержание

Программирование на MQL5 для трейдеров 13

Часть 1. Знакомство с языком и средой разработки 16
- 1.1 Редактирование, компиляция и запуск программ 18
- 1.2 Мастер MQL и эскиз программы 20
- 1.3 Инструкции, блоки кода и функции 23
- 1.4 Первая программа 25
- 1.5 Типы данных и значения 28
- 1.6 Переменные и идентификаторы 29
- 1.7 Присваивание и инициализация, выражения и массивы 30
- 1.8 Ввод данных 33
- 1.9 Работа над ошибками и отладка 35
- 1.10 Вывод данных 39
- 1.11 Форматирование, отступы и пробелы 40
- 1.12 Подводим промежуточный итог 42

Часть 2. Основы программирования 44
- 2.1 Идентификаторы 45
- 2.2 Встроенные типы данных 46
  - 2.2.1 Целые числа 49
  - 2.2.2 Вещественные числа 52
  - 2.2.3 Символьные типы 56
  - 2.2.4 Строковый тип 57
  - 2.2.5 Логический тип 58
  - 2.2.6 Дата и время 59
  - 2.2.7 Цвет 60
  - 2.2.8 Перечисления 61
  - 2.2.9 Пользовательские перечисления 64
  - 2.2.10 Тип void 67
- 2.3 Переменные 67
  - 2.3.1 Объявление и определение переменных 68
  - 2.3.2 Контекст, область видимости и время жизни переменных 69
  - 2.3.3 Инициализация 72
  - 2.3.4 Статические переменные 77
  - 2.3.5 Переменные-константы 79
  - 2.3.6 Входные переменные 80
  - 2.3.7 Внешние переменные 82
- 2.4 Массивы 86
  - 2.4.1 Характеристики массивов 86
  - 2.4.2 Описание массивов 88
  - 2.4.3 Использование массивов 90
- 2.5 Выражения 93
  - 2.5.1 Базовые понятия 94
  - 2.5.2 Операция присваивания 96
  - 2.5.3 Арифметические операции 98
  - 2.5.4 Инкремент и декремент 100
  - 2.5.5 Операции сравнения 101
  - 2.5.6 Логические операции 103
  - 2.5.7 Побитовые операции 104
  - 2.5.8 Операции модификации 106
  - 2.5.9 Условный тернарный оператор 108
  - 2.5.10 Запятая 110
  - 2.5.11 Специальные операторы sizeof и typename 111

2.5.12 Группировка с помощью круглых скобок 112
2.5.13 Приоритеты операций 113
2.6 Приведение типов 115
2.6.1. Неявное приведение типов 115
2.6.2. Арифметические преобразования типов 117
2.6.3. Явное приведение типов 119
2.7 Инструкции 120
2.7.1 Составные инструкции (блоки кода) 121
2.7.2 Инструкции объявления/определения 121
2.7.3 Простые инструкции (выражения) 126
2.7.4 Обзор управляющих инструкций 127
2.7.5 Цикл for 128
2.7.6 Цикл while 132
2.7.7 Цикл do 134
2.7.8 Выбор if 134
2.7.9 Выбор switch 137
2.7.10 Переход break 140
2.7.11 Переход continue 143
2.7.12 Переход return 144
2.7.13 Пустая инструкция 145
2.8 Функции 146
2.8.1 Определение функции 146
2.8.2 Вызов функции 148
2.8.3 Параметры и аргументы 149
2.8.4 Параметры-значения и параметры-ссылки 150
2.8.5 Необязательные параметры 153
2.8.6 Возврат значений 154
2.8.7 Объявление функции 155
2.8.8 Рекурсия 156
2.8.9 Перегрузка функций 157
2.8.10 Указатели на функции (typedef) 160
2.8.11 Инлайнинг 163
2.9 Препроцессор 163
2.9.1 Включение исходных файлов (#include) 164
2.9.2 Обзор директив макроподстановки 165
2.9.3 Простая форма #define 166
2.9.4 Форма #define в виде псевдо-функции 169
2.9.5 Специальные операторы '#' и '##' внутри определений #define 171
2.9.6 Отмена макроподстановки (#undef) 172
2.9.7 Предопределенные константы препроцессора 172
2.9.8 Условная компиляция (#ifdef/#ifndef/#else/#endif) 173
2.9.9 Общие свойства программ (#property) 174
Часть 3. Объектно-Ориентированное Программирование 176
3.1 Структуры и объединения 176
3.1.1 Определение структур 177
3.1.2 Функции (методы) в структурах 179
3.1.3 Копирование структур 180
3.1.4 Конструкторы и деструкторы 181
3.1.5 Упаковка структур в памяти и взаимодействие с DLL 184
3.1.6 Компоновка и наследование структур 185
3.1.7 Права доступа 187
3.1.8 Объединения 188
3.2 Классы и интерфейсы 190
3.2.1 Основы ООП: абстракция 191
3.2.2 Основы ООП: инкапсуляция 192
3.2.3 Основы ООП: наследование 192

3.2.4 Основы ООП: полиморфизм ... 193
3.2.5 Основы ООП: композиция (дизайн) ... 194
3.2.6 Определение класса ... 195
3.2.7 Права доступа ... 198
3.2.8 Конструкторы: по умолчанию, параметрический, копирования ... 199
3.2.9 Деструкторы ... 205
3.2.10 Ссылка на себя: this ... 206
3.2.11 Наследование ... 209
3.2.12 Динамическое создание объектов: new и delete ... 214
3.2.13 Указатели ... 216
3.2.14 Виртуальные методы (virtual и override) ... 223
3.2.15 Статические члены ... 232
3.2.16 Вложенные типы, пространства имен и оператор контекста '::' ... 234
3.2.17 Разнесение объявления и определения класса ... 238
3.2.18 Абстрактные классы и интерфейсы ... 241
3.2.19 Перегрузка операторов ... 243
3.2.20 Приведение объектных типов: dynamic_cast и указатель void * ... 255
3.2.21 Указатели, ссылки и const ... 259
3.2.22 Управление наследованием: final и delete ... 263
3.3 Шаблоны ... 265
3.3.1 Заголовок шаблона ... 266
3.3.2 Общие принципы работы шаблонов ... 267
3.3.3 Шаблоны vs макросы препроцессора ... 269
3.3.4 Особенности встроенных и объектных типов в шаблонах ... 270
3.3.5 Шаблоны функций ... 274
3.3.6 Шаблоны объектных типов ... 280
3.3.7 Шаблоны методов ... 285
3.3.8 Вложенные шаблоны ... 291
3.3.9 Специализация шаблонов, которой нет ... 293
Часть 4. Общеупотребительные функции ... 297
4.1 Преобразование данных встроенных типов ... 297
4.1.1 Числа в строки и обратно ... 299
4.1.2 Нормализация чисел double ... 302
4.1.3 Дата и время ... 303
4.1.4 Цвет ... 312
4.1.5 Структуры ... 315
4.1.6 Перечисления ... 317
4.1.7 Тип complex ... 319
4.2 Работа со строками и символами ... 321
4.2.1 Инициализация и измерение строк ... 321
4.2.2 Суммирование (соединение) строк ... 325
4.2.3 Сравнение строк ... 327
4.2.4 Изменение регистра символов и обрезка пробелов ... 333
4.2.5 Поиск, замена и извлечение фрагментов строк ... 335
4.2.6 Работа с символами и кодовыми страницами ... 339
4.2.7 Универсальный форматированный вывод данных в строку ... 346
4.3 Работа с массивами ... 353
4.3.1 Вывод массивов в журнал ... 353
4.3.2 Динамические массивы ... 357
4.3.3 Измерение массивов ... 364
4.3.4 Инициализация и заполнение массивов ... 366
4.3.5 Копирование и редактирование массивов ... 368
4.3.6 Перемещение (обмен) массивов ... 379
4.3.7 Сравнение, сортировка и поиск в массивах ... 382
4.3.8 Направление индексации массивов как в таймсерии ... 397
4.3.9 Обнуление объектов и массивов ... 401

4.4 Математические функции 406
4.4.1 Абсолютное значение числа 407
4.4.2 Максимальное и минимальное из двух чисел 409
4.4.3 Функции округления 409
4.4.4 Деление чисел по модулю 410
4.4.5 Степени и корни 411
4.4.6 Показательные и логарифмические функции 411
4.4.7 Тригонометрические функции 413
4.4.8 Гиперболические функции 416
4.4.9 Проверка вещественных чисел на нормальность 417
4.4.10 Генерация случайных чисел 420
4.4.11 Управление порядком байтов в целых числах 421
4.5 Работа с файлами 423
4.5.1 Способы хранения информации: текстовый и двоичный 426
4.5.2 Запись и чтение файлов в упрощенном режиме 428
4.5.3 Открытие и закрытие файлов 432
4.5.4 Управление дескрипторами файлов 439
4.5.5 Выбор кодировки для текстового режима 446
4.5.6 Запись и чтение массивов 449
4.5.7 Запись и чтение структур (бинарные файлы) 455
4.5.8 Запись и чтение переменных (бинарные файлы) 460
4.5.9 Запись и чтение переменных (текстовые файлы) 469
4.5.10 Управление позицией внутри файла 478
4.5.11 Получение свойств файла 485
4.5.12 Принудительная запись кэша на диск 488
4.5.13 Удаление и проверка на существование файла 494
4.5.14 Копирование и перемещение файлов 495
4.5.15 Поиск файлов и папок 497
4.5.16 Работа с папками 501
4.5.17 Диалог выбора файла или папки 502
4.6 Глобальные переменные терминала 506
4.6.1 Запись и чтение глобальной переменной 508
4.6.2 Проверка существования и времени последнего действия 510
4.6.3 Получение списка глобальных переменных 511
4.6.4 Удаление глобальных переменных 512
4.6.5 Временные глобальные переменные 513
4.6.6 Синхронизация программ с помощью глобальных переменных 514
4.6.7 Сброс глобальных переменных на диск 524
4.7 Функции для работы со временем 525
4.7.1 Время локальное и серверное 528
4.7.2 Переход на летнее время (локальное) 531
4.7.3 Универсальное время 537
4.7.4 Приостановка выполнения программы 538
4.7.5 Счетчики временных интервалов 539
4.8 Взаимодействие с пользователем 540
4.8.1 Вывод сообщений в журнал 541
4.8.2 Предупреждающие сигналы (алерты) 546
4.8.3 Вывод сообщений в окно графика 546
4.8.4 Диалоговое окно сообщений 551
4.8.5 Звуковые оповещения 556
4.9 Среда исполнения MQL-программ 557
4.9.1 Получение общего списка свойств терминала и программы 558
4.9.2 Номер сборки терминала 562
4.9.3 Тип и лицензия программы 563
4.9.4 Режимы работы терминала и программы 565
4.9.5 Разрешения 567

4.9.6 Проверка сетевых подключений ... 571
4.9.7 Вычислительные ресурсы: память, диск, процессор ... 572
4.9.8 Характеристики экрана ... 574
4.9.9 Строковые свойства терминала и программы ... 577
4.9.10 Настраиваемые свойства: лимит баров и язык интерфейса ... 578
4.9.11 Привязка программы к свойствам среды исполнения ... 578
4.9.12 Проверка состояния клавиатуры ... 588
4.9.13 Проверка статуса и причины остановки MQL-программы ... 590
4.9.14 Программное закрытие терминала и код возврата ... 592
4.9.15 Обработка ошибок времени исполнения программы ... 594
4.9.16 Пользовательские ошибки ... 596
4.9.17 Управление отладкой ... 602
4.9.18 Предопределенные переменные ... 602
4.9.19 Предопределенные константы языка MQL5 ... 603
4.10 Матрицы и векторы ... 604
4.10.1 Типы матриц и векторов ... 605
4.10.2 Создание и инициализация матриц и векторов ... 607
4.10.3 Копирование матриц, векторов и массивов ... 610
4.10.4 Копирование таймсерий в матрицу или вектор ... 612
4.10.5 Копирование истории тиков в матрицу или вектор ... 613
4.10.6 Вычисление выражений с матрицами и векторами ... 614
4.10.7 Манипуляции над матрицами и векторами ... 615
4.10.8 Произведения матриц и векторов ... 619
4.10.9 Преобразования (разложение) матриц ... 621
4.10.10 Получение статистики ... 623
4.10.11 Характеристики матриц и векторов ... 624
4.10.12 Решение уравнений ... 626
4.10.13 Методы машинного обучения ... 633
Часть 5. Создание прикладных программ ... 644
5.1 Общие принципы выполнения MQL-программ ... 645
5.1.1 Конструирование MQL-программ различных типов ... 646
5.1.2 Потоки ... 649
5.1.3 Обзор функций обработки событий ... 650
5.1.4 Особенности запуска и остановки программ разных типов ... 657
5.1.5 Опорные события индикаторов и советников: OnInit и OnDeinit ... 660
5.1.6 Главная функция скриптов и сервисов: OnStart ... 662
5.1.7 Программное удаление советников и скриптов: ExpertRemove ... 663
5.2 Скрипты и сервисы ... 665
5.2.1 Скрипты ... 665
5.2.2 Сервисы ... 666
5.2.3 Ограничения для скриптов и сервисов ... 671
5.3 Временные ряды (таймсерии) ... 672
5.3.1 Символы и таймфреймы ... 674
5.3.2 Технические особенности организации и хранения таймсерий ... 677
5.3.3 Получение характеристик массивов котировок ... 679
5.3.4 Количество доступных баров (Bars/iBars) ... 680
5.3.5 Поиск индекса бара по времени (iBarShift) ... 681
5.3.6 Обзор Copy-функций для получения массивов котировок ... 683
5.3.7 Получение котировок в виде массива структур MqlRates ... 688
5.3.8 Раздельный запрос массивов цен, объемов, спредов, времени ... 691
5.3.9 Чтение цены, объема, спреда и времени по индексу бара ... 693
5.3.10 Поиск максимального и минимального значения в таймсерии ... 696
5.3.11 Работа с массивами реальных тиков в структурах MqlTick ... 700
5.4 Создание пользовательских индикаторов ... 711
5.4.1 Основные характеристики индикаторов ... 712
5.4.2 Главное событие индикаторов: OnCalculate ... 713

5.4.3 Два типа индикаторов: для главного и отдельного окна ... 717
5.4.4 Настройка количества буферов и графических построений ... 718
5.4.5 Назначение массива в качестве буфера: SetIndexBuffer ... 720
5.4.6 Настройка графических построений: PlotIndexSetInteger ... 723
5.4.7 Правила сопоставления буферов и диаграмм ... 730
5.4.8 Применение директив для настройки графических построений ... 733
5.4.9 Установка названий для графических построений ... 735
5.4.10 Визуализация пропусков данных (пустых элементов) ... 736
5.4.11 Индикаторы с собственным подокном: размер и уровни ... 742
5.4.12 Общие свойства индикаторов: заголовок и точность значений ... 748
5.4.13 Поэлементное раскрашивание диаграмм ... 749
5.4.14 Пропуск отрисовки на начальных барах ... 752
5.4.15 Ожидание данных и управление видимостью (DRAW_NONE) ... 758
5.4.16 Мультивалютные и мультитаймфреймовые индикаторы ... 770
5.4.17 Отслеживание формирования баров ... 792
5.4.18 Тестирование индикаторов ... 795
5.4.19 Ограничения и преимущества индикаторов ... 797
5.4.20 Создание заготовки индикатора в Мастере MQL ... 798
5.5 Использование готовых индикаторов из MQL-программ ... 800
5.5.1 Дескрипторы и счетчики владельцев индикаторов ... 801
5.5.2 Простой способ создания экземпляров индикаторов: iCustom ... 803
5.5.3 Проверка количества просчитанных баров: BarsCalculated ... 807
5.5.4 Получение данных таймсерии из индикатора: CopyBuffer ... 809
5.5.5 Поддержка множества символов и таймфреймов ... 819
5.5.6 Обзор встроенных индикаторов ... 826
5.5.7 Использование встроенных индикаторов ... 832
5.5.8 Расширенный способ создания индикаторов: IndicatorCreate ... 841
5.5.9 Гибкое создание индикаторов с помощью IndicatorCreate ... 852
5.5.10 Обзор функций управления индикаторами на графике ... 861
5.5.11 Комбинирование вывода в главное окно и вспомогательное ... 861
5.5.12 Чтение данных из диаграмм, имеющих сдвиг ... 864
5.5.13 Удаление экземпляров индикаторов: IndicatorRelease ... 867
5.5.14 Получение настроек индикатора по его дескриптору ... 873
5.5.15 Определение источника данных для индикатора ... 876
5.6 Работа с таймером ... 877
5.6.1 Включение и отключение таймера ... 878
5.6.2 Событие таймера: OnTimer ... 879
5.6.3 Таймер повышенной точности: EventSetMillisecondTimer ... 888
5.7 Работа с графиками ... 890
5.7.1 Функции для получения основных свойств текущего графика ... 891
5.7.2 Идентификация графиков ... 892
5.7.3 Получение списка графиков ... 893
5.7.4 Получение символа и таймфрейма произвольного графика ... 894
5.7.5 Обзор функций для работы с полным набором свойств ... 896
5.7.6 Описательные свойства графика ... 898
5.7.7 Проверка состояния основного окна ... 900
5.7.8 Получение количества и признака видимости окон/подокон ... 900
5.7.9 Режимы отображения графика ... 902
5.7.10 Управление видимостью элементов графика ... 910
5.7.11 Отступы по горизонтали ... 914
5.7.12 Горизонтальный масштаб (по времени) ... 915
5.7.13 Вертикальный масштаб (по цене и показаниям индикаторов) ... 917
5.7.14 Цвета ... 920
5.7.15 Управление мышью и клавиатурой ... 923
5.7.16 Открепление окна графика ... 926
5.7.17 Получение координат буксировки MQL-программы на график ... 927

5.7.18 Перевод экранных координат во время/цену и обратно 929
5.7.19 Прокрутка графика по оси времени 932
5.7.20 Запрос перерисовки графика 935
5.7.21 Переключение символа и таймфрейма 936
5.7.22 Управление индикаторами на графике 936
5.7.23 Открытие и закрытие графиков 942
5.7.24 Работа с tpl-шаблонами графика 945
5.7.25 Сохранение изображения графика 960
5.8 Графические объекты 963
5.8.1 Типы объектов и особенности указания их координат 964
5.8.2 Объекты с привязкой ко времени и цене 965
5.8.3 Объекты с привязкой к экранным координатам 967
5.8.4 Создание объектов 968
5.8.5 Удаление объектов 970
5.8.6 Поиск объектов 973
5.8.7 Обзор функций доступа к свойствам объектов 976
5.8.8 Основные свойства объектов 996
5.8.9 Координаты времени и цены 998
5.8.10 Угол окна привязки и экранные координаты 1000
5.8.11 Определение точки привязки на объекте 1004
5.8.12 Управление состоянием объекта 1006
5.8.13 Приоритет объектов (Z-порядок) 1007
5.8.14 Настройка отображения объекта: цвет, стиль и рамка 1010
5.8.15 Настройки шрифта 1023
5.8.16 Поворот текста на произвольный угол 1026
5.8.17 Определение ширины и высоты объектов 1028
5.8.18 Видимость объектов в разрезе таймфреймов 1035
5.8.19 Назначение кода символа для метки 1038
5.8.20 Свойства лучей для объектов с прямыми линиями 1039
5.8.21 Управление нажатым состоянием объекта 1042
5.8.22 Настройка изображений в объектах-картинках 1044
5.8.23 Кадрирование (вывод части) изображения 1046
5.8.24 Свойства поля ввода: выравнивание и "только чтение" 1048
5.8.25 Ширина канала стандартного отклонения 1050
5.8.26 Настройка уровней в объектах с их поддержкой 1050
5.8.27 Дополнительные свойства Ганна, Фибоначчи и Эллиота 1054
5.8.28 Объект-график 1055
5.8.29 Перемещение объектов 1059
5.8.30 Получение времени или цены в заданных точках линий 1060
5.9 Интерактивные события на графиках 1064
5.9.1 Функция обработки событий OnChartEvent 1065
5.9.2 Связанные с событиями свойства графика 1067
5.9.3 Событие изменений графика 1069
5.9.4 События клавиатуры 1071
5.9.5 События мыши 1080
5.9.6 События с графическими объектами 1083
5.9.7 Генерация пользовательских событий 1087
Часть 6. Автоматизация торговли 1093
6.1 Финансовые инструменты и обзор рынка 1093
6.1.1 Получение списков доступных символов и Обзора рынка 1094
6.1.2 Редактирование списка Обзора рынка 1096
6.1.3 Проверка символа на существование 1098
6.1.4 Проверка актуальности данных по символу 1099
6.1.5 Получение последнего тика по символу 1101
6.1.6 Расписания торговых и котировочных сессий 1105
6.1.7 Маржинальные коэффициенты по символу 1111

6.1.8 Обзор функций получения свойств символа ... 1112
6.1.9 Проверка состояния символа ... 1122
6.1.10 Тип цены для построения графиков по символу ... 1124
6.1.11 Базовая, котировочная и маржинальная валюты инструмента ... 1129
6.1.12 Точность представления и шаг изменения цен ... 1136
6.1.13 Разрешенные объемы торговых операций ... 1139
6.1.14 Разрешения на торговлю ... 1142
6.1.15 Торговые условия и режимы исполнения приказов по символу ... 1146
6.1.16 Маржинальные требования ... 1150
6.1.17 Правила истечения сроков отложенных ордеров ... 1159
6.1.18 Спреды и отступы приказов от текущей цены ... 1164
6.1.19 Получение величины свопов ... 1168
6.1.20 Текущая рыночная информация (тик) ... 1173
6.1.21 Описательные свойства символов ... 1175
6.1.22 Глубина стакана цен ... 1178
6.1.23 Свойства пользовательских символов ... 1180
6.1.24 Специфические свойства (биржа, срочный рынок, облигации) ... 1181
6.2 Стакан цен ... 1182
6.2.1 Управление подпиской на события о стакане цен ... 1183
6.2.2 Получение событий об изменении стакана цен ... 1186
6.2.3 Чтение данных текущего стакана цен ... 1188
6.2.4 Использование данных стакана в прикладных алгоритмах ... 1195
6.3 Информация о торговом счете ... 1203
6.3.1 Обзор функций получения свойств счета ... 1204
6.3.2 Идентификация счета, клиента, сервера и брокера ... 1207
6.3.3 Вид счета: реальный, демо или конкурсный ... 1208
6.3.4 Валюта счета ... 1209
6.3.5 Тип счета: неттинг или хедж ... 1209
6.3.6 Ограничения и разрешения для операций по счету ... 1210
6.3.7 Маржинальные настройки счета ... 1214
6.3.8 Текущие финансовые показатели счета ... 1217
6.4 Создание экспертов ... 1218
6.4.1 Главное событие экспертов: OnTick ... 1219
6.4.2 Основные принципы и понятия: ордер, сделка, позиция ... 1221
6.4.3 Типы торговых операций ... 1223
6.4.4 Типы ордеров ... 1223
6.4.5 Режимы исполнения ордеров по цене и объемам ... 1225
6.4.6 Сроки действия отложенных ордеров ... 1227
6.4.7 Расчет залога для будущего ордера: OrderCalcMargin ... 1227
6.4.8 Оценка прибыли торговой операции: OrderCalcProfit ... 1240
6.4.9 Структура торгового запроса MqlTradeRequest ... 1246
6.4.10 Структура проверки запроса MqlTradeCheckResult ... 1249
6.4.11 Проверка корректности запроса: OrderCheck ... 1251
6.4.12 Результат отправки запроса: структура MqlTradeResult ... 1256
6.4.13 Отправка торгового запроса: OrderSend и OrderSendAsync ... 1257
6.4.14 Совершение покупки или продажи ... 1264
6.4.15 Модификация уровней Stop Loss и/или Take Profit позиции ... 1280
6.4.16 Трейлинг стоп ... 1288
6.4.17 Полное и частичное закрытие позиции ... 1298
6.4.18 Полное и частичное закрытие встречных позиций (хедж) ... 1307
6.4.19 Установка отложенного ордера ... 1316
6.4.20 Модификация отложенного ордера ... 1327
6.4.21 Удаление отложенного ордера ... 1339
6.4.22 Получение списка действующих ордеров ... 1342
6.4.23 Свойства ордеров (действующих и в истории) ... 1344
6.4.24 Функции для чтения свойств действующих ордеров ... 1348

6.4.25 Отбор ордеров по свойствам 1357
6.4.26 Получение списка позиций 1374
6.4.27 Свойства позиций 1377
6.4.28 Функции для чтения свойств позиций 1379
6.4.29 Свойства сделок 1388
6.4.30 Выборка ордеров и сделок из истории 1392
6.4.31 Функции для чтения свойств ордеров из истории 1394
6.4.32 Функции для чтения свойств сделок из истории 1397
6.4.33 Типы торговых транзакций 1410
6.4.34 Событие OnTradeTransaction 1413
6.4.35 Синхронные и асинхронные запросы 1431
6.4.36 Событие OnTrade 1445
6.4.37 Контроль за изменениями торгового окружения 1454
6.4.38 Особенности создания мультисимвольных экспертов 1484
6.4.39 Ограничения и преимущества экспертов 1499
6.4.40 Создание заготовки эксперта в Мастере MQL 1500
6.5 Тестирование и оптимизация экспертов 1504
6.5.1 Генерация тиков в тестере 1505
6.5.2 Управление ходом времени в тестере: таймер, Sleep, GMT 1513
6.5.3 Визуализация тестирования: график, объекты, индикаторы 1514
6.5.4 Мультивалютное тестирование 1515
6.5.5 Критерии оптимизации 1520
6.5.6 Получение финансовых показателей теста: TesterStatistics 1522
6.5.7 Событие OnTester 1533
6.5.8 Авто-настройка: ParameterGetRange и ParameterSetRange 1544
6.5.9 Группа OnTester-событий для контроля оптимизации 1551
6.5.10 Отправка фреймов данных с агентов в терминал 1553
6.5.11 Получение фреймов данных в терминале 1553
6.5.12 Директивы препроцессора для тестера 1561
6.5.13 Управление видимостью индикаторов: TesterHideIndicators 1566
6.5.14 Эмуляция пополнения депозита и снятия средств 1567
6.5.15 Принудительная остановка тестирования: TesterStop 1571
6.5.16 Большой пример эксперта 1572
6.5.17 Математические вычисления 1615
6.5.18 Отладка и профилирование 1617
6.5.19 Ограничения работы функций в тестере 1618
Часть 7. Расширенные средства языка 1620
7.1 Ресурсы 1620
7.1.1 Описание ресурсов с помощью директивы #resource 1621
7.1.2 Разделяемое использование ресурсов разных MQL-программ 1623
7.1.3 Ресурсные переменные 1624
7.1.4 Подключение пользовательских индикаторов как ресурсов 1628
7.1.5 Динамическое создание ресурсов: ResourceCreate 1635
7.1.6 Удаление динамических ресурсов: ResourceFree 1641
7.1.7 Чтение и модификация данных ресурса: ResourceReadImage 1641
7.1.8 Сохранение изображений в файл: ResourceSave 1651
7.1.9 Шрифты и вывод текста в графические ресурсы 1662
7.1.10 Прикладное применение графических ресурсов в трейдинге 1675
7.2 Пользовательские символы 1683
7.2.1 Создание и удаление пользовательских символов 1684
7.2.2 Свойства пользовательских символов 1687
7.2.3 Установка маржинальных коэффициентов 1689
7.2.4 Настройка котировочных и торговых сессий 1690
7.2.5 Добавление, замена и удаление котировок 1690
7.2.6 Добавление, замена и удаление тиков 1699
7.2.7 Трансляция изменений стакана заявок 1728

7.2.8 Особенности торговли с пользовательскими символами 1734
7.3 Экономический календарь 1751
7.3.1 Основные понятия календаря 1752
7.3.2 Получение списка и описаний доступных стран 1760
7.3.3 Запрос видов событий по странам и валютам 1761
7.3.4 Получение описания вида события по идентификатору 1766
7.3.5 Получение записей о событиях по странам или валютам 1766
7.3.6 Получение записей о событиях конкретного вида 1771
7.3.7 Чтение записи о событии по идентификатору 1774
7.3.8 Отслеживание изменений событий по стране или валюте 1778
7.3.9 Отслеживание изменений событий по типу 1788
7.3.10 Фильтрация событий по множеству условий 1788
7.3.11 Перенос базы календаря в тестер 1807
7.3.12 Торговля по календарю 1830
7.4 Криптография 1839
7.4.1 Обзор доступных методов преобразования информации 1840
7.4.2 Шифрование, хэширование и упаковка данных: CryptEncode 1843
7.4.3 Дешифрование и распаковка данных: CryptDecode 1855
7.5 Сетевые функции 1861
7.5.1 Отправка Push-уведомлений 1862
7.5.2 Отправка уведомлений по электронной почте 1863
7.5.3 Отправка файлов на сервер FTP 1863
7.5.4 Обмен данными с веб-сервером по протоколу HTTP/HTTPS 1864
7.5.5 Установление и разрыв соединения сетевого сокета 1884
7.5.6 Проверка состояния сокета 1886
7.5.7 Настройка таймаутов передачи и приема данных сокетами 1888
7.5.8 Чтение, запись данных по незащищенному сокет-соединению 1888
7.5.9 Подготовка защищенного сокет-соединения 1893
7.5.10 Чтение и запись данных по защищенному сокет-соединению 1895
7.6 База данных SQLite 1905
7.6.0 Знакомство с принципами работы с базой данных в MQL5 1907
7.6.1 Основы SQL 1911
7.6.2 Структура (схема) таблиц: типы данных и ограничения 1916
7.6.3 Интеграция ООП (MQL5) и SQL: концепция ORM 1919
7.6.4 Создание, открытие и закрытие базы данных 1921
7.6.5 Выполнение запросов без привязки к данным MQL5 1923
7.6.6 Проверка существования таблицы в базе данных 1933
7.6.7 Подготовка запросов с привязкой: DatabasePrepare 1934
7.6.8 Удаление и сброс подготовленных запросов 1936
7.6.9 Привязка данных к параметрам запроса:DatabaseBind/Array 1938
7.6.10 Выполнение подготовленных запросов: DatabaseRead/Bind 1940
7.6.11 Раздельное чтение полей: DatabaseColumn-функции 1942
7.6.12 Примеры CRUD-операций в SQLite через объекты ORM 1943
7.6.13 Транзакции 1961
7.6.14 Импорт и экспорт таблицы базы данных 1965
7.6.15 Печать таблиц и SQL-запросов в журнал 1966
7.6.16 Пример поиска торговой стратегии средствами SQLite 1967
7.7 Разработка и подключение библиотек двоичных форматов 1978
7.7.1 Создание ex5-библиотек и export функций 1980
7.7.2 Подключение библиотек и #import функций 1984
7.7.3 Порядок поиска файлов библиотек 1989
7.7.4 Особенности подключения DLL-библиотек 1989
7.7.5 Классы и шаблоны в библиотеках MQL5 1995
7.7.6 Импорт функций из .NET библиотек 2009
7.8 Проекты 2010
7.8.1 Общие принципы работы с локальными проектами 2012

7.8.2 План проекта веб-сервиса копирования сделок и сигналов ... 2014
7.8.3 Веб-сервер на основе nodejs ... 2016
7.8.4 Теоретические основы протокола WebSockets ... 2018
7.8.5 Серверная часть веб-сервисов на базе WebSocket-протокола ... 2020
7.8.6 Протокол WebSocket-ов на MQL5 ... 2029
7.8.7 Клиентские программы эхо и чат-сервисов на MQL5 ... 2039
7.8.8 Сервис торговых сигналов и тестовая веб-страница ... 2048
7.8.9 Клиентская программа сигнального сервиса на MQL5 ... 2053
7.9 Встроенная поддержка Python ... 2069
7.9.1 Установка Python и пакета MetaTrader5 ... 2070
7.9.2 Обзор функций пакета MetaTrader5 для Python ... 2072
7.9.3 Подключение скрипта Python к терминалу и счету ... 2074
7.9.4 Проверка ошибок: last_error ... 2076
7.9.5 Получение информации о торговом счете ... 2077
7.9.6 Получение информации о терминале ... 2079
7.9.7 Получение информации о финансовых инструментах ... 2081
7.9.8 Подписка на стакан цен ... 2085
7.9.9 Чтение котировок ... 2087
7.9.10 Чтение истории тиков ... 2092
7.9.11 Вычисление маржинальных требований и оценка прибыли ... 2095
7.9.12 Проверка и отправка торгового приказа ... 2096
7.9.13 Получение количества и списка действующих ордеров ... 2101
7.9.14 Получение количества и списка открытых позиций ... 2104
7.9.15 Чтение истории ордеров и сделок ... 2106
7.10 Встроенная поддержка параллельных вычислений: OpenCL ... 2110
Заключение ... 2119

Программирование на MQL5 для трейдеров

Современный трейдинг невозможно представить без компьютера. Автоматизация рабочего места трейдера уже давно перешагнула границы бирж и офисов брокеров и стала доступной рядовому пользователю в виде специализированного программного обеспечения. Одна из первых таких программ — MetaTrader, ведущий свою историю с начала 2000-х годов. Данная торговая платформа, представленная сегодня версией MetaTrader 5, продолжает пополняться все новыми и новыми возможностями.

Среди прочего постоянно совершенствуется и встроенный язык программирования MQL5. Его наличие выводит трейдеров на качественно новый уровень автоматической торговли, известный как алготрейдинг. Благодаря MQL5 трейдер может реализовать свои идеи в виде прикладной программы: самостоятельно написать пользовательский индикатор, скрипт для выполнения разовых операций или создать советник — автоматическую торговую систему (торговый робот). Советник может работать круглосуточно без постороннего вмешательства, отслеживая изменения цен финансовых инструментов, отправляя уведомления по электронной почте или на мобильный телефон, а также выполняя множество других полезных действий.

MQL5 позволяет реализовать в прикладной программе практически любую идею, будь то стратегия на пересечение двух скользящих средних, три экрана Элдера или фрактальный анализ Петерса, цифровая обработка сигналов, нейронная сеть или геометрические построения. В принципе, MQL5 объединяет в себе черты предметно-ориентированного языка со специализацией для алготрейдинга и языка общего назначения. Это тем более верно, если учесть, что за последние годы в терминал и в язык MQL5 были встроены различные расширения для работы с 3D-графикой, параллельными расчетами на OpenCL, интеграции с Python, сетевыми функциями и базами данных SQLite.

Чтобы раскрыть потенциал всех или только некоторых из этих инструментов, разумеется, потребуется изучить язык MQL5, и эта книга поможет Вам выполнить данную задачу.

Предполагается, что читатель данной книги уже знаком с клиентским терминалом MetaTrader 5 и умеет работать с ним в качестве пользователя. Также подразумевается, что он понимает основные принципы работы терминала в составе общей распределенной информационной системы, обеспечивающей торговлю. В частности, напомним, что терминал в "боевом" состоянии всегда подключен к серверу, установленному в дилинговом центре, и через него получает данные от других участников рынка — банков, бирж и других финансовых организаций. Через этот же сервер выполняются и торговые операции клиента. За счет постоянного сетевого соединения в терминале поддерживается актуальная информация о состоянии рынка и торговом окружении: ценах, спецификациях инструментов, настройках торгового счета и пр.

Как известно, в графическом интерфейсе терминала имеется богатый спектр инструментов для проведения визуального технического анализа рынка и ручной торговли. Все приемы работы и возможности для интерактивного управления подробно описаны в Руководстве пользователя терминала и не будут здесь дублироваться.

Вместе с тем важно отметить, что встроенные в язык MQL5 программные интерфейсы (API), или по-простому — наборы функций, доступных для MQL-программ, предоставляют практически те же возможности, что и пользовательский интерфейс. Причем за счет гибко настраиваемых автоматизированных действий, таких как проверки различных условий, комбинирование и зацикливание последовательностей действий, алгоритмы с использованием функций MQL5 позволяют реализовывать гораздо более сложные сценарии использования MetaTrader, чем

ручное управление. Кроме того, через MQL5 API доступно много инструментов, которые отсутствуют в пользовательском интерфейсе. В частности, чтение и запись данных с помощью сетевых функций, работа с базой данных SQLite, генерирование 3D-изображений — все это примеры расширенных, в некотором смысле низкоуровневых API, которые "видны" только в MQL5.

Но, разумеется, прикладные программы и ручные средства управления могут использоваться в клиентском терминале одновременно, взаимно дополняя друг друга.

MQL5 позволяет автоматизировать множество различных аспектов работы с торговым терминалом, повысив их скорость и удобство, а также исключив рутинные ручные операции. В некоторых случаях, например при ведении высокочастотной торговли, в принципе невозможно обеспечить эффективность стратегии без программной поддержки со стороны MQL5. В других, таких как балансировка корзины валют или биржевых инструментов, обойтись без MQL5 хоть и можно теоретически, но на практике это неприемлемо.

Изложение материала разделено на 7 частей.

- Часть 1 позволяет с нуля познакомиться с базовыми принципами программирования на MQL5 в обзорном режиме и попробовать в действии стандартную среду разработки для MQL5 — редактор и компилятор MetaEditor. Пользователям, имеющим опыт в программировании на других языках, стоит обратить внимание на особенности среды.
- Часть 2 рассказывает об основных понятиях, таких как типы, инструкции, операторы, выражения, переменные, блоки кода, структура программ и их применении для написания MQL-программ в процедурном стиле. Пользователи, которые хорошо знают MQL4, могут пропустить эту часть и приступить к 3-ей части.
- Часть 3 посвящена объектно-ориентированному программированию (ООП) на MQL5. Несмотря на схожесть с другими языками с поддержкой парадигмы ООП (в особенности с C++), MQL5 все же имеет отличительные черты, знания о которых пригодятся даже тем читателям, кто с ООП "на ты". Если хотите, MQL5 — это, своего рода C±±.
- Часть 4 знакомит читателя с общеупотребительными встроенными функциями, которые пригодятся в любой программе.
- Часть 5 описывает архитектурные особенности MQL-программ, их специализацию по типам для выполнения различных трейдерских задач, таких как технический анализ с помощью индикаторов, управление графиками и их разметка с наложением графических объектов, реакция на интерактивные действия и события с MQL-программами.
- Часть 6 посвящена анализу торгового окружения и автоматизации торговли с помощью роботов. Здесь же представлено программное взаимодействие с тестером в различных режимах, включая оптимизацию стратегий.
- Часть 7 содержит информацию о расширенном наборе специализированных API, облегчающих интеграцию MQL5 со смежными технологиями, такими как базы данных, обмен данными по сети, OpenCL, Python и других.

Программирование основывается на многих сущностях, которые в равной степени важны и тесно взаимосвязаны, поэтому порядок изложения может иногда потребовать перехода от одной концепции к другой, и затем возврата к первой с большими подробностями. В некотором роде, это проблема "курицы и яйца". Невозможно научиться сразу всем принципам, методикам и языковым конструкциям, которые требуются для написания произвольной MQL-программы. Мы будем стараться взвешенно совмещать описания общих подходов и понятий, демонстрации частных примеров и переходы в формальную техническую плоскость — на уровень синтаксиса, порядка исполнения кода, и проектирования структуры программ.

Большинство примеров MQL-программ из этой книги доступно в виде исходных кодов для установки в среду MetaEditor/MetaTrader 5 и проверки изложенного материала на практике.

Часть 1. Знакомство с MQL5 и средой разработки

Одним из важнейших изменений в языке MQL5 при переходе к его новой инкарнации в MetaTrader 5 явилась поддержка концепции объектно-ориентированного программирования (ООП). Если предыдущий MQL4 (язык MetaTrader 4) было принято сравнивать в момент его появления с языком программирования C, то для MQL5 уместно проводить параллели с C++. Справедливости ради отметим, что сегодня все инструменты ООП, присутствовавшие изначально только в MQL5, уже перенесены и в MQL4, но, тем не менее, для пользователей, мало знакомых с программированием, ООП по-прежнему кажется чем-то чересчур сложным.

Эта книга призвана, в некотором смысле, сделать сложное простым. Она является не заменой, а дополнением к справочному руководству по MQL5, которое поставляется вместе с терминалом и также доступно на сайте mql5.com.

В данной книге мы последовательно расскажем о всех составных частях и приемах программирования на MQL5, двигаясь небольшими шагами — так, чтобы каждая итерация была понятной, а технология ООП постепенно раскрывала свой потенциал, наиболее заметный, как и в случае с любым мощным инструментом, только при правильном, обоснованном применении. В результате разработчики MQL-программ смогут выбирать наиболее подходящий для себя и для конкретной задачи стиль программирования — не только объектно-ориентированный, но и "старый" процедурный, а также комбинировать их в произвольных сочетаниях.

Пользователей торгового терминала можно условно разделить на категории "программистов" (тех, кто имеет опыт программирования хотя бы на одном языке) и "непрограммистов" ("чистых" трейдеров, которые интересуются возможностью кастомизации терминала с помощью MQL5). Первые могут, при желании, пропустить первую и вторую части книги, описывающую базовые понятия языка, и сразу приступить к изучению специфических API (Application Programming Interface — программный интерфейс), встроенных в MetaTrader 5. Для вторых рекомендуется поступательное чтение.

Из представителей категории "программистов" наибольшую фору в освоении MQL5 получат знатоки C++ — в силу похожести синтаксисов MQL5 и C++. Однако эта "медаль" имеет и обратную сторону. Дело в том, что MQL5 все же не полностью соответствует C++ (особенно, если сравнивать со свежими стандартами) и потому зачастую попытки написать ту или иную конструкцию по привычке "как на плюсах" будут прерываться неожиданными ошибками компилятора. Мы постараемся обращать внимание на данные отличия при рассмотрении конкретных элементов языка.

Технический анализ, выполнение торговых приказов или интеграция со внешними источниками данных — все эти функции доступны пользователям терминала как из пользовательского интерфейса, так и через программные средства, встроенные в MQL5.

И поскольку MQL-программы должны выполнять столь разные функции, в MetaTrader 5 поддерживается несколько специализированных типов программ. Это является стандартным приемом во многих программных системах. Например, в системе Windows помимо наиболее привычных нам программ с оконным интерфейсом существуют консольные программы с управлением из командной строки и службы.

В MQL5 же доступны следующие типы программ:

- индикаторы — программы для графического отображения массивов данных, рассчитываемых по заданной формуле, обычно на основе рядов котировок;
- эксперты — программы для полной или частичной автоматизации торговли;

- скрипты — программы для выполнения одноразовых действий;
- сервисы — программы для выполнения постоянных фоновых действий.

Более подробно о назначении и особенностях каждого типа мы поговорим позднее. Сейчас важно отметить, что все они создаются на языке MQL5 и имеют много общего. Поэтому мы начнем изучение с общих черт и постепенно будем знакомиться со спецификой каждого типа.

Основополагающая техническая особенность торговой системы MetaTrader заключается в том, что всё управление осуществляется в клиентском терминале, и инициированные в нем команды отправляются на сервер. Иными словами, прикладные MQL-программы могут работать только внутри клиентского терминала, причем большинство из них требует "живого" подключения к серверу для правильного функционирования. На сервере никакие прикладные программы не устанавливаются. Сервер лишь обрабатывает приказы, поступающие со стороны клиентского терминала, и передает обратно на него изменения торгового окружения. Эти изменения становятся также доступными и для MQL-программ.

Большинство типов MQL-программ выполняется в контексте графика, то есть программу нужно "набросить" на требуемый график, чтобы запустить. Исключение составляет только особый тип — сервисы: они предназначены для работы в фоне, без привязки к графику.

Напомним, что все MQL-программы находятся внутри рабочей папки MetaTrader 5, во вложенной папке с именем вида */MQL5/<тип>*, где *<тип>* — это, соответственно:

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

В зависимости от способа установки MetaTrader 5 путь к рабочей папке может различаться (в частности, при ограниченных правах пользователя в системе Windows, в обычном режиме или portable). Например, он может быть:

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

или

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

Пользователь может узнать фактическое размещение данной папки, выполнив команду *Файл -> Открыть каталог данных* (она имеется как в терминале, так и в редакторе). Кроме того, при создании новой программы Вы можете не думать о поиске правильной папки за счет использования встроенного в редактор Мастера MQL. Он вызывается командой *Файл -> Новый* и позволяет выбрать требуемый тип MQL-программы. Соответствующий текстовый файл с заготовкой исходного кода будет автоматически создан в нужном месте после завершения Мастера и открыт для редактирования.

В папке MQL5 есть и другие вложенные папки помимо упомянутых, и они тоже имеют непосредственное отношение к MQL-программированию, но мы обратимся к ним чуть позже.

Программирование на MQL5 для трейдеров — исходные коды из книги: Часть 1

Примеры из книги также доступны в публичном проекте \MQL5\Shared Projects\MQL5Book

1.1 Редактирование, компиляция и запуск программ

Все программы в MetaTrader 5 являются компилируемыми. Это означает, что исходный код, который пишется на языке MQL5, должен быть откомпилирован для получения бинарного (или двоичного) представления — именно оно и будет выполняться в терминале.

Редактирование и компиляцию программ обеспечивает MetaEditor.

MetaEditor - [GoodTime2.mq5]

Файл Правка Поиск Вид Сборка Отладка Сервис Окно Справка

Создать Компилировать

GoodTime2.mq5

```
10 #property script_show_inputs
11
12 input uint GreetingHour = 0; // Greeting Hour (0-23)
13
14 //+------------------------------------------------------------------+
15 //| Return greeting for given time of day (hour)                     |
16 //+------------------------------------------------------------------+
17 string Greeting(int hour)
18 {
19    string messages[3] = {"Good morning", "Good day", "Good evening"};
20    return messages[hour % 24 / 8];
21 }
22
23 //+------------------------------------------------------------------+
24 //| Script program start function                                    |
25 //+------------------------------------------------------------------+
26 void OnStart()
27 {
28    Print(Greeting(GreetingHour), ", ", Symbol());
29 }
```

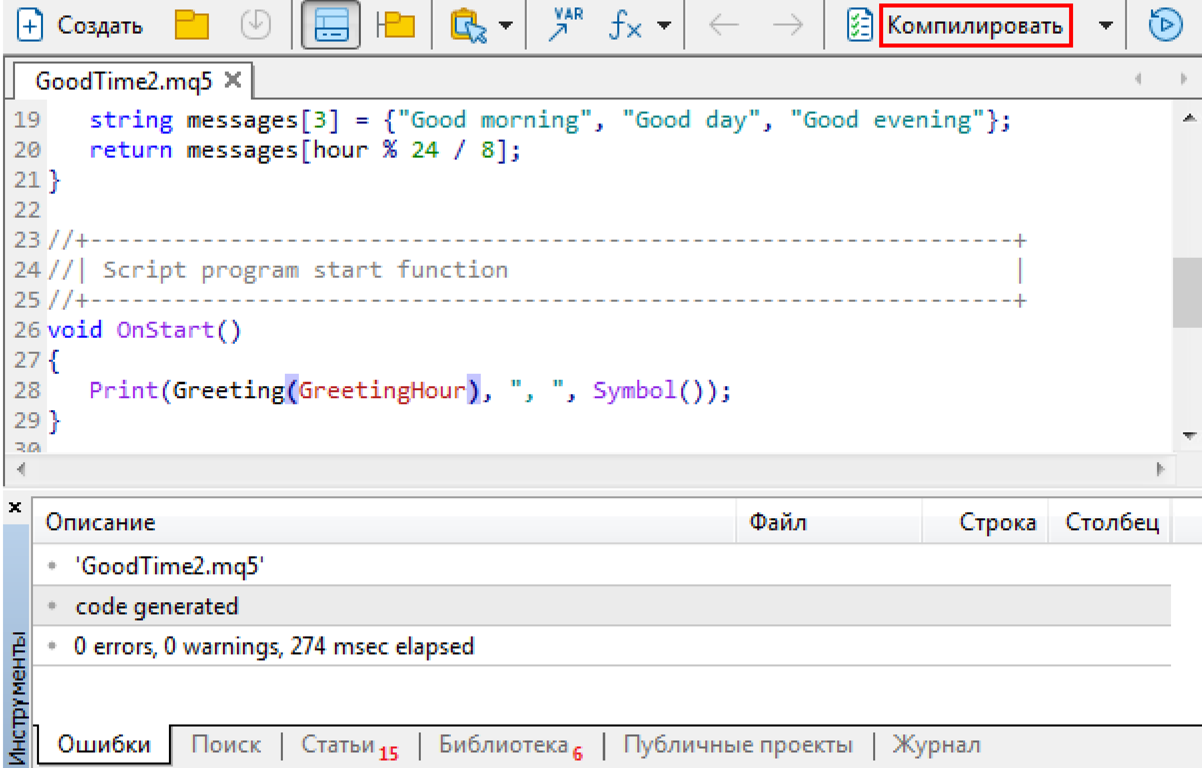
Редактирование MQL-программы в редакторе MetaEditor

Исходный код — это текст, написанный по правилам MQL5 и сохраненный в файл с расширением *mq5*. Файл с откомпилированной программой получит то же имя, но расширение *ex5*.

В простейшем случае одному исполняемому файлу соответствует один файл с исходным кодом, но, как мы увидим в дальнейшем, для написания сложных программ часто приходится делить исходный код на несколько файлов — один главный и несколько вспомогательных, специальным образом подключаемых из главного. В этом случае главный файл по-прежнему должен иметь расширение mq5, а подключенные из него — расширение mqh. В этом случае в генерируемый исполняемый файл фактически попадут инструкции из всех исходных файлов. Таким образом, источником для создания одного исполняемого файла (программы) могут быть несколько файлов с исходным кодом. Все это, упомянутое здесь для полноты картины, мы начнем осваивать лишь во второй части книги.

Набор всех правил, по которым разрешено конструировать программы на MQL5, будем далее называть синтаксисом MQL5. Только строгое следование синтаксису позволяет писать программы, понятные компилятору. По сути, обучение программированию заключается в последовательном знакомстве со всеми правилами конкретного языка, в нашем случае — MQL5. И это — основная задача данной книги.

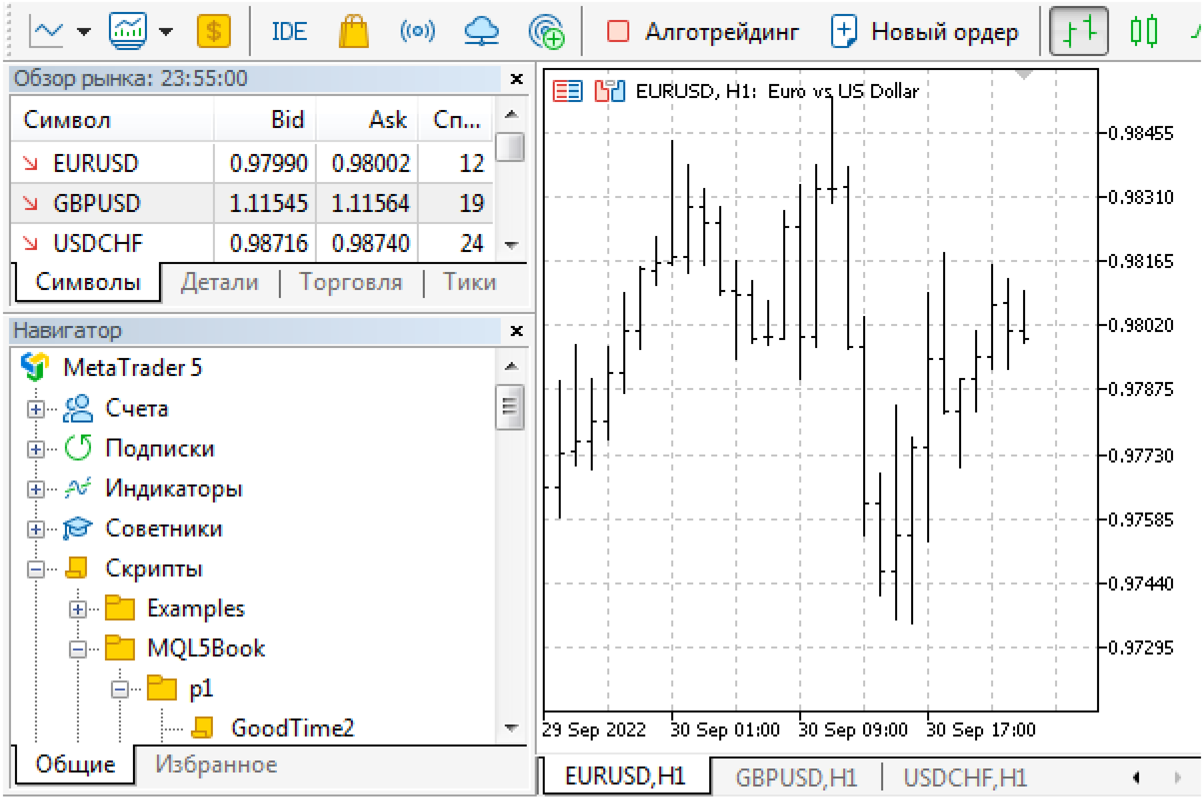
Обычно для компиляции исходного кода используется команда MetaEditor *Файл -> Компилировать*, или достаточно просто нажать *F7*. Но существуют и другие, специальные способы компиляции — мы поговорим о них позднее. Процесс компиляции сопровождается выводом изменяющегося статуса в лог редактора (когда MQL-программа состоит из нескольких файлов с исходным кодом, подключение каждого файла отмечается на отдельной строке лога).



Компиляция MQL-программы в редакторе MetaEditor

Признаком успешной компиляции является отсутствие ошибок ("0 errors"). Наличие предупреждений не сказывается на результате компиляции, но сигнализирует о потенциальных проблемах, поэтому их рекомендуется исправлять наравне с ошибками (о том, как это делать, мы расскажем позже). В идеале предупреждений быть не должно ("0 warnings").

В результате успешной компиляции mq5-файла мы получаем одноименный файл с расширением *ex5*. *Навигатор* в MetaTrader 5 отображает в виде дерева все исполняемые ex5-файлы, находящиеся в папке MQL5 и её подпапках, в том числе и только что откомпилированный.



Навигатор MetaTrader 5 с откомпилированной MQL-программой

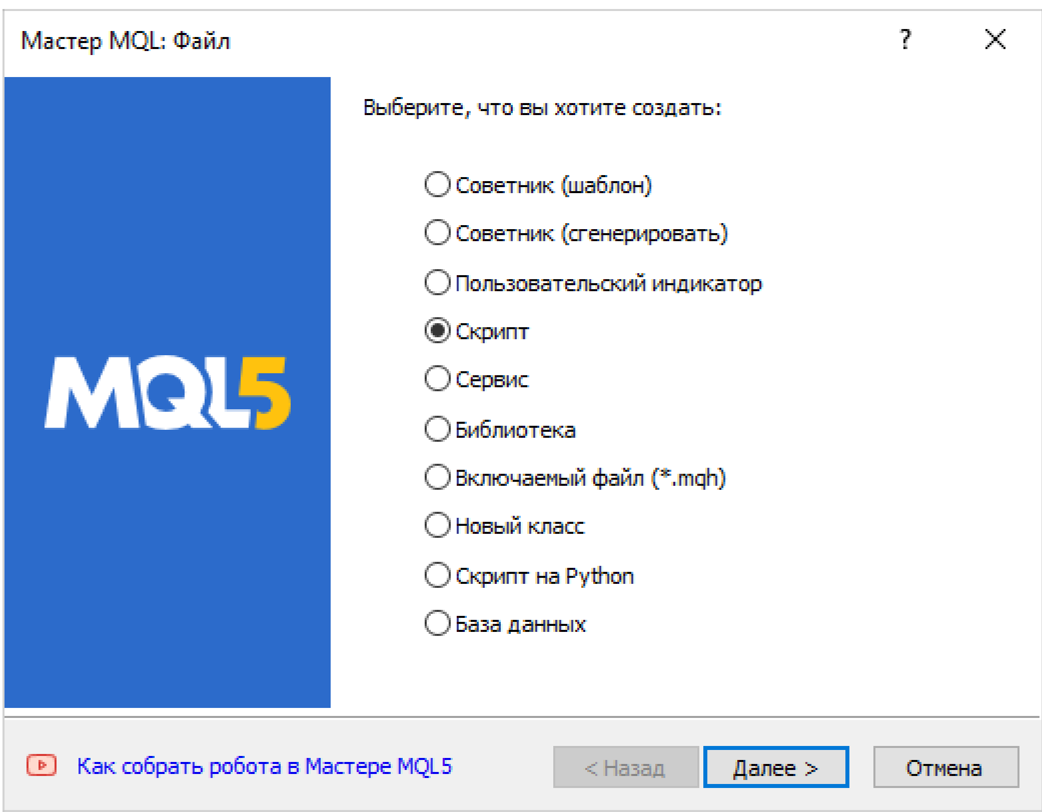
Готовые программы запускаются в терминале привычными для пользователя способами. Например, любую программу, за исключением сервиса, можно перетащить из *Навигатора* на график с помощью мыши. Про особенности сервисов будет рассказано отдельно.

Кроме того, разработчикам часто требуется выполнить программу в режиме отладки — для поиска причин ошибок. Для этой цели существует даже не одна специальная команда — мы обратимся к ним в разделе Работа над ошибками и отладка.

1.2 Мастер MQL и эскиз программы

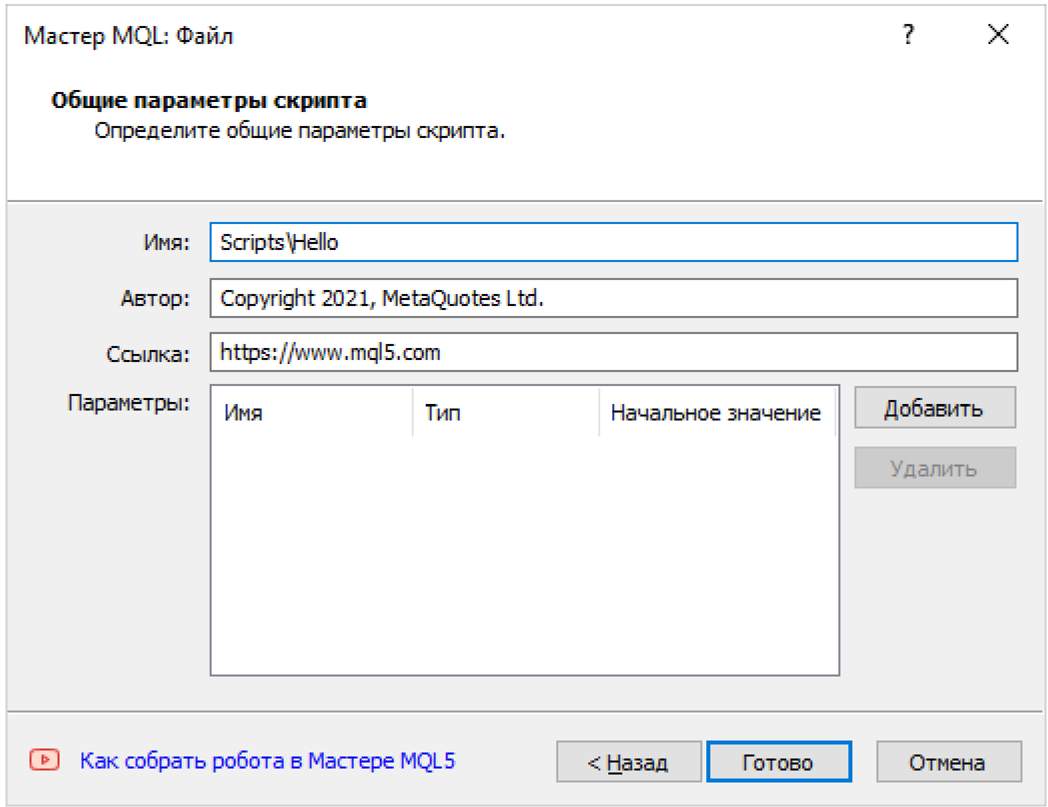
В данной главе мы рассмотрим простейшую MQL5-программу, которая по сути ничего не делает. Её цель в том, чтобы можно было познакомиться с процессом написания исходного кода в редакторе, его компиляции и запуска в терминале. Самостоятельно повторив указанные шаги, вы удостоверитесь в доступности программирования для обычных пользователей и начнете осваиваться в интегрированной среде разработки MQL-программ. Она будет постоянно нужна для практического закрепления материала.

Самым простым типом MQL-программ являются скрипты, поэтому попробуем создать именно скрипт. Для этого запустим MQL-Мастер (*Файл -> Новый*). На первом шаге выберем из перечня типов *Скрипт* и нажмем кнопку *Далее*:



Создание скрипта с помощью Мастера MQL. 1-й шаг

На втором шаге введем название скрипта в поле Имя, дописав его после уже указанной папки по умолчанию и разделителя — "Scripts\". Например, назовем скрипт "Hello" (т.е. в поле Имя будет строка "Scripts\Hello") и, больше ничего не меняя, нажмем кнопку *Готово*.



Создание скрипта с помощью Мастера MQL. 2-й шаг

В результате Мастер создаст для нас файл *Hello.mq5* и откроет его на редактирование. Файл размещается в папке *MQL5/Scripts* (стандартной для скриптов), потому что мы оставили предложенную по умолчанию папку, но мы могли бы добавить любую вложенную папку или целую иерархию вложенных папок. Например, если на первом шаге Мастера в поле *Имя* написать "Scripts\Exercise\Hello", то в папке *Scripts* будет автоматически создана вложенная папка *Exercise*, и уже в ней будет находиться файл *Hello.mq5*.

Все примеры из данной книги будут располагаться в папках MQL5Book внутри каталогов, отведенных под MQL-программы соответствующих типов. Это нужно для упрощения установки примеров в вашу рабочую копию терминала и исключения конфликтов имен с другими MQL-программами, которые у вас уже установлены.

Например, файл *Hello.mq5*, поставляемый вместе с книгой, находится по пути *MQL5\Scripts\MQL5Book\p1*, где *p1* обозначает номер главы (от "part 1"), к которой относится пример.

Получившаяся заготовка скрипта *Hello.mq5* содержит следующий текст.

```
//+------------------------------------------------------------------+
//|                                                    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()
{
}

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

Все строки, начинающиеся с символов "//", являются комментариями и на суть программы не влияют. Они не обрабатываются компилятором и не исполняются затем терминалом. Они нужны только для обмена пояснительной информацией между разработчиками или визуального выделения отдельных частей в целях улучшения читабельности текста. Например, в данной заготовке в начале файла идет блок с комментарием, где предполагается указать название скрипта и права автора. Второй блок комментариев является "шапкой" для главной функции скрипта — о том, что это такое, речь пойдет чуть ниже. Наконец последняя строка с комментарием визуально выделяет конец файла.

Три строки, начинающиеся со специальной директивы *#property*, сообщают компилятору кое-какие атрибуты, которые он особым образом встраивает в программу. В нашем случае они пока не важны и даже могут быть удалены. Для каждого типа MQL-программ доступны свои специфические директивы — о них мы узнаем, когда перейдем к изучению конкретных типов программ.

Главная часть скрипта, где предполагается описать суть действий программы, представлена функцией *OnStart*. Здесь нам потребуется познакомиться с понятиями блока кода и функции.

1.3 Инструкции, блоки кода и функции

Итак, функция *OnStart* выглядит в скрипте, сгенерированном Мастером, следующим образом:

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

Именно она является нашим первым предметом изучения в контексте программирования на MQL5. И здесь нам сразу встречаются неизвестные понятия и последовательности символов, для пояснения которых необходимо сделать отступление.

Обычно программа должна реализовать следующие характерные этапы в своей работе:

- определить переменные — именованные ячейки в памяти компьютера для хранения данных;
- организовать ввод исходных данных;
- выполнить обработку данных — прикладной алгоритм;
- организовать вывод результатов.

Все эти этапы являются необязательными, если говорить о сохранении синтаксической правильности программы. Например, если мы создадим программу, которая считает результат умножения "2*2", то для неё, очевидно, не нужно никаких входных данных, потому что числа для умножения вшиты в текст программы. Кроме того, поскольку 2 и 2 в этом выражении являются постоянными значениями, то в программе не нужны и дополнительные именованные ячейки (переменные). А поскольку мы и так знаем, сколько будет дважды два, то, в принципе, можем не сообщать результат умножения. Конечно, такая программа не имела бы практического смысла, но была бы абсолютно корректной с технической точки зрения.

Еще более интересно, что программа может не содержать никаких инструкций по обработке. Наша заготовка скрипта как раз представляет собой образец пустой программы. Но что же тогда означает вышеприведенный фрагмент текста?

Один из столпов программирования Никлаус Вирт дал в свое время простое, обобщенное определение программирования как симбиоза алгоритмов и структур данных.

Под словом алгоритм понимается последовательность инструкций конкретного языка программирования. Инструкция — это своего рода предложение, законченная мысль на языке программирования, сформулированное по правилам синтаксиса. Само название "инструкция" говорит о том, что она воспринимается компьютером как руководство к действию. Иными словами, инструкции описывают, когда и каким образом следует обработать требуемые прикладные структуры данных. Именно поэтому взаимное проникновение алгоритмов и структур данных позволяет воплотить идею автора (разработчика) в виде программы.

К сожалению, в большинстве практических задач количество инструкций столь велико, что их нужно как-то систематизировать, чтобы человек мог осознавать и контролировать поведение программы.

И здесь на помощь приходит принцип "разделяй и властвуй", который используется в программировании практически повсеместно и под разными "соусами". Мы изучим их все по мере продвижения по разделам книги, а пока отметим лишь суть.

Как известно, принцип сводится к дроблению большой, сложной задачи на более мелкие и простые. Здесь можно провести аналогию со строительством дома или сборкой космической ракеты. Оба этих "изделия" состоят из множества различных модулей, которые в свою очередь состоят из компонентов, а те — из еще более мелких деталей и так далее.

Продолжая ту же параллель на область алгоритмов, можно сказать, что инструкции являются мелкими деталями, а вся программа — это дом или ракета. Следовательно, нам требуются строительные блоки промежуточных размеров.

Поэтому при реализации алгоритмов принято объединять логически связанные инструкции в более крупные именованные фрагменты — функции. В нужных местах программы мы можем обратиться к функции по её имени (вызвать её), и тем самым попросить компьютер выполнить все инструкции, содержащиеся внутри функции. Вся программа целиком является, по сути, самым крупным, внешним блоком и потому также может быть представлена функцией, из которой вызываются более мелкие функции или непосредственно выполняются инструкции, если их мало. Тут мы и подходим к функции *OnStart*.

Имя *OnStart* зарезервировано в скриптах для обозначения самой главной функции — эту функцию терминал вызывает сам в ответ на действия пользователя, когда тот запускает скрипт через команду контекстного меню или перетаскивая мышью на график. Таким образом, предыдущий фрагмент кода определяет функцию *OnStart*, задающую поведение всего нашего скрипта.

Для тех, кто знаком с программированием на других языках (например, C, C++, Rust, Kotlin), очевидна аналогия данной функции с функцией *main* — основной точкой входа в программу.

Любой скрипт обязательно должен содержать функцию *OnStart*. В противном случае компиляция закончится с ошибкой.

Пустая функция *OnStart*, как у нас, начинает выполняться терминалом (как только скрипт запускается тем или иными способом) и тут же завершает работу. Строго говоря, прикладного алгоритма в нашем скрипте еще нет, но функция-заглушка для его добавления уже есть.

В других типах MQL-программ также есть специальные функции, которые программист обязан определить в своем коде — мы затронем их специфику в соответствующих разделах.

Синтаксис определения функций мы подробно рассмотрим во второй части книги. А для первого знакомства с ним, чтобы понять описание *OnStart*, достаточно упомянуть следующие основные моменты.

Поскольку функции, как правило, предназначены для получения прикладного результата, в их определении специальным образом описывают характеристики ожидаемой величины: какого рода данные должны получиться, да и нужны ли они вообще. Некоторые функции могут выполнять действия, не требующие возврата значения. Например, функция может быть предназначена для изменения настроек текущего графика или для отправки push-уведомления при достижении заданного уровня просадки на счету. Все это можно запрограммировать инструкциями внутри функции, и она не создает каких-либо новых данных (которые имело бы смысл возвращать в другие части программы).

В нашем случае ситуация аналогичная: как главная функция скрипта, *OnStart* могла бы возвратить свой результат только во внешнюю среду (непосредственно в терминал) при своем завершении, но это никак не повлияло бы на работу самого скрипта (потому что он уже отработал).

Именно поэтому перед именем функции *OnStart* стоит слово *void*, которое сообщает компилятору, что результат нам не важен (*void* — пустота). *void* — это одно из множества служебных слов, зарезервированных в MQL5. Компилятору известны смыслы всех служебных слова, и он руководствуется ими при разборе исходного кода. В частности, с помощью служебных слов программист может определять новые понятия для компилятора, такие как сама функция *OnStart*.

Круглые скобки после имени — обязательная часть описания любой функции: в них может содержаться список параметров функции. Например, если бы мы писали функцию, возводящую число в квадрат, то в ней нужно было бы предусмотреть один параметр — под то самое число. Затем мы могли бы вызвать эту функцию из любой части программы, передав в неё один аргумент — конкретное значение для параметра. Как описать список параметров, мы увидим позднее, а в текущем примере он отсутствует. Это требование наложено на функцию *OnStart* из-за того, что её вызывает сам терминал, и он никогда ничего в неё не передает в качестве параметров.

Наконец фигурные скобки используются для маркировки начала и конца блока с инструкциями. Идущий непосредственно после строки с именем функции, такой блок будет содержать набор действий, выполняемых данной функцией, — его еще называют телом функции. В данном случае внутри фигурных скобок ничего нет, и потому заготовка скрипта пока ничего не делает.

Приведенная последовательность слова *void*, имени *OnStart*, пустого списка параметров и пустого блока кода определяет для компилятора минимально возможную, пустую реализацию функции *OnStart*. Позднее, добавляя инструкции в тело функции, мы будем расширять определение функции *OnStart*.

Выполнив команду *Компилировать*, мы убедимся, что скрипт успешно компилируется, и готовая программа появляется в *Навигаторе* терминала в папке *Scripts/MQL5Book/p1*. Это результат того, что на диске в соответствующей папке теперь есть файл *Hello.ex5*, это легко проверить в любом файловом менеджере.

Мы можем запустить скрипт на чарте, но единственным подтверждением его выполнения будут записи в журнале терминала (вкладка *Журнал* в окне *Инструменты*, не путать с панелью инструментов):

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

То есть скрипт загрузился, управление передалось функции *OnStart*, но тут же вернулось терминалу, потому что функция ничего не делает, и после этого терминал выгрузил скрипт с чарта.

1.4 Первая программа

Попробуем добавить в скрипт что-нибудь простое, но вместе с тем наглядное для демонстрации его работы. Модифицированный скрипт сохраним под новым именем *HelloChart.mq5*.

Во многих учебниках по программированию начальный пример выводит сакраментальную фразу "Hello, world". На MQL5 похожее приветствие могло бы выглядеть так:

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

Но мы сделаем его слегка более информативным:

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

Итак, мы добавили всего одну строку с некоторыми конструкциями языка.

Здесь *Print* — имя встроенной в терминал функции, предназначенной для вывода сообщений в журнал *Экспертов* (вкладка *Эксперты* в окне *Инструменты*; несмотря на название *Эксперты* вкладка собирает сообщения от MQL-программ всех типов). В отличие от функции *OnStart*, которую мы определяем сами, функция *Print* определена для нас заранее и навсегда. *Print* — одна из множества встроенных функций, составляющих MQL5 API (программный интерфейс).

Новая строка в нашем коде обозначает инструкцию вызова функции *Print* с передачей ей списка аргументов (внутри круглых скобок), которые и будут выведены в журнал. Аргументы в списке разделяются запятыми. В данном случае аргументов два: строка "Hello, " и вызов еще одной встроенной функции — *Symbol*, которая возвращает название рабочего инструмента текущего графика (полученное из неё значение сразу попадет в список аргументов функции *Print*, на то место, откуда был произведен вызов функции *Symbol*).

Функция *Symbol* не имеет параметров и потому внутри круглых скобок в неё ничего не передается.

Например, если скрипт помещен на график "EURUSD", то вызов функции *Symbol()* вернет "EURUSD", и инструкция с обращением к функции *Print* приобретет с точки зрения выполняющейся программы обновленный вид: *Print("Hello, ", "EURUSD")*. Разумеется, с точки зрения пользователя все эти вызовы функций и динамическая подстановка промежуточных результатов происходят незаметно и моментально. Для программиста же важно представлять, как программа выполняется шаг за шагом, чтобы избежать логических ошибок и добиться работы в строгом соответствии с задуманным планом.

Строка "Hello, " в двойных кавычках — это так называемый литерал, неизменяемая последовательность символов, которые понимаются компьютером в виде текста, как есть (как он введен в исходный код программы).

Таким образом, приведенная инструкция печати должна вывести в журнал два переданных аргумента, один за другим, в результате чего две строки фактически состыкуются, и получится "Hello, EURUSD".

Важно отметить, что запятая внутри кавычек будет выведена в журнал как часть строки и специальным образом не обрабатывается. В отличие от этого, запятая, которая стоит после кавычек и перед вызовом *Symbol()*, является символом-разделителем в списке аргументов, то есть влияет на поведение программы. Если первую запятую пропустить, программа от этого не потеряет свою корректность, хотя и станет выводить слово "Hello" без запятой после него. Если же пропустить вторую запятую, программа перестанет компилироваться, поскольку синтаксис списка аргументов функции будет нарушен: все значения в нем (а у нас это — две строки) должны быть разделены запятыми.

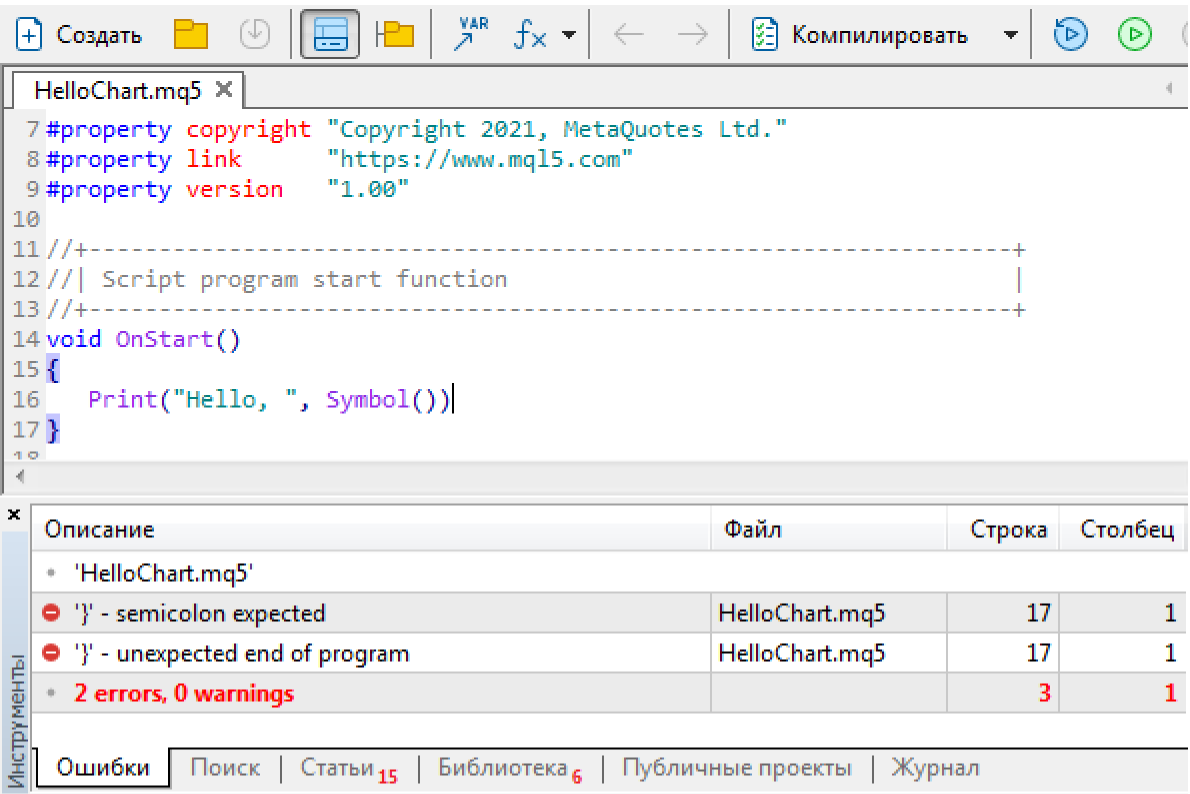
Ошибка компилятора будет выглядеть следующим образом:

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

Компилятор "жалуется", что перед упоминанием *Symbol* чего-то не хватает. Компиляция при этом обрывается и исполняемый файл программы не создается. Поэтому вернем запятую на место.

На этом примере мы видим, насколько важно четко следовать синтаксису языка. Зачастую одни и те же символы могут по-разному работать, находясь в различных местах программы. И даже одно маленькое упущение может оказаться критическим. Например, обратите внимание на точку с запятой в конце строки с вызовом *Print*. Точка с запятой обозначает здесь конец инструкции, и если её забыть поставить, то могут возникать странные ошибки компилятора.

Чтобы убедиться в этом, попробуем удалить эту точку с запятой и откомпилировать скрипт снова. В результате получим новые ошибки с описанием сути и места проблемы в исходном коде.



Ошибки компиляции в журнале редактора MetaEditor

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

Первая ошибка явным образом указывает на отсутствие ожидаемой компилятором точки с запятой. Вторая ошибка является наведенной: закрывающая фигурная скобка, сигнализирующая конец программы, была обнаружена раньше, чем кончилась текущая инструкция, а продолжается она по мнению компилятора потому, что ему пока так и не встретилась точка с запятой. Исправление ошибок очевидно — необходимо вернуть точку с запятой на крайнюю правую позицию в инструкции.

Откомпилируем и запустим исправленный скрипт. Хотя он выполняется очень быстро и почти моментально удаляется с графика, в журнале *Экспертов* теперь появляется запись, подтверждающая работу скрипта.

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

1.5 Типы данных и значения

Помимо вызова встроенной функции *Symbol*, мы могли бы использовать и какую-нибудь свою функцию, определенную нами в исходном коде. Допустим, мы хотим выводить в журнал не просто "Hello", а различное приветствие в зависимости от времени суток. Время суток будем определять с точностью до часа: с 0 до 8 — утро, с 8 до 16 — день, и с 16 до 24 — вечер.

Логично предположить, что определение новой функции должно своей структурой походить на уже знакомое нам определение функции *OnStart*. Однако имя должно быть уникальным — не дублировать названия других функций и отличаться от зарезервированных слов. Список этих слов мы изучим далее по ходу книги, а пока удачно предположим, что в качестве имени можно использовать слово *Greeting*.

Функция должна возвращать, как и функция *Symbol*, строку, но на этот раз строка должна быть одной из фраз "Good morning", "Good day" или "Good evening" в зависимости от часа.

Руководствуясь здравым смыслом, мы здесь оперируем общеупотребительным понятием строки. И оно, видимо, знакомо компилятору, потому что мы видели, как он сгенерировал программу, выводящую в журнал заданный текст. Таким образом, мы плавно подошли к концепции типов в языке программирования, причем одним из типов является строка — последовательность символов.

В MQL5 этот тип описывается ключевым словом *string*. Это уже второй тип, который мы знаем, после *void*. И мы уже даже встречали значение данного типа, но не знали, что это он: речь о литерале "Hello, ". Просто, когда мы вставляем в исходный код константу (в частности, вроде текста в кавычках), описание её типа не требуется — компилятор автоматически определяет правильный тип.

Используя определение функции *OnStart* как образец, мы можем предположить, как в первом приближении должна выглядеть функция *Greeting*.

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

Этот текст означает наше намерение создать функцию *Greeting*, способную вернуть произвольное значение типа *string* (строка). Но для того, чтобы функция действительно что-то вернула, необходимо воспользоваться специальной инструкцией с оператором *return*. Это один из множества операторов MQL5: мы изучим их все позднее. Когда функция имеет тип возвращаемого значения отличный от *void*, в ней обязательно должен быть оператор *return*.

В частности, для того чтобы вернуть из функции прежнюю строку приветствия "Hello, ", следует написать:

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

```
}
```

Оператор *return* прекращает выполнение функции и передает "наружу" в качестве результата то, что справа от него. За словом "наружу" скрывается тот фрагмент исходного кода, откуда функция была вызвана.

Мы еще не знаем всех возможностей по написанию выражений, которые могли бы сформировать произвольную строку, но самый простой случай с текстом в кавычках переносится сюда без изменений. Важно, чтобы тип возвращаемого значения совпадал с типом функции, как и в данном случае. В конце инструкции ставим точку с запятой.

Однако мы хотели генерировать различное приветствие в зависимости от времени суток. Следовательно, у функции должен быть параметр, задающий час, который может принимать значения от 0 до 23 (включительно). Очевидно, что номер часа — это целое число, то есть число, у которого нет дробной части. Понятно, что внутри часа время тоже не стоит, и в нем отсчитываются минуты, причем количество минут — тоже целое число. Впрочем, вряд ли имеет смысл определять время суток с точностью до минуты, и потому мы ограничимся выбором приветствия только по номеру часа.

Для целочисленных значений в MQL5 имеется специальный тип *int*. И поскольку это значение нужно передать в функцию *Greeting* из другого места в программе, откуда эта функция будет вызываться, мы впервые сталкиваемся с необходимостью описать именованную ячейку памяти, то есть переменную.

1.6 Переменные и идентификаторы

Переменная — это ячейка памяти с уникальным именем (чтобы можно было на неё ссылаться без ошибок), способная хранить значения определенного типа. Данная способность обеспечивается тем, что компилятор резервирует под переменную ровно столько памяти, сколько под неё требуется в специальном внутреннем формате: каждый тип имеет размер и соответствующий формат хранения в памяти. Подробнее об этом — во второй части книги.

В принципе, для имен переменных, функций и многих других сущностей (которые предстоит изучить позднее) в программе имеется более строгое понятие — идентификатор. Идентификатор подчиняется нескольким правилам. В частности, он должен содержать только буквы латинского алфавита, цифры и символ подчеркивания, и не начинаться с цифры. Вот почему слово Greeting, выбранное в предыдущем параграфе для функции, этим требованиям удовлетворяет.

Сами значения переменной могут быть разными, меняться по ходу выполнения программы с помощью специальных инструкций.

Кроме типа и имени переменная характеризуется контекстом — областью в программе, где она определена и может использоваться без ошибок компилятора. Понять эту концепцию на первых порах и без подробных технических рассуждений, вероятно, будет проще на нашем примере.

Дело в том, что частным случаем переменной является параметр функции. Назначение параметра — передать внутрь функции некое значение. Отсюда становится очевидно, что фрагмент кода, где такая переменная существует, должен ограничиваться телом самой функции. Иными словами, параметр можно использовать во всех инструкциях внутри блока функции, но нельзя вовне. Если бы язык программирования позволял такие вольности, это стало бы источником множества ошибок из-за потенциальной возможности испортить содержимое внутренностей функции из произвольного, никак не связанного с ней, участка программы.

В любом случае, это слегка упрощенное определение переменной, достаточное для данной вводной части. Более тонкие нюансы мы рассмотрим позднее.

Итак, обобщим наши знания о переменных и параметрах: они должны иметь имя, тип и контекст. Первые две характеристики мы пишем в коде явным образом, последняя — вытекает из места определения.

Посмотрим, как можно определить параметр с номером часа в функции *Greeting*. Мы уже знаем нужный нам тип — *int*, а имя можно выбрать по смыслу — *hour*.

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

Такая функция по-прежнему вернет "Hello, " вне зависимости от часа. Теперь необходимо добавить некие инструкции, которые выбирали бы различную возвращаемую строку на основе значения параметра *hour*. Напомним, вариантов "ответа" функции три: "Good morning", "Good day", "Good evening". Можно было бы предположить, что нам нужны 3 переменные для описания этих строк. Однако в подобных случаях гораздо удобнее воспользоваться массивом, который обеспечивает единообразный способ кодирования алгоритмов с доступом к элементам.

1.7 Присваивание и инициализация, выражения и массивы

Массив — это именованный набор однотипных ячеек, расположенных в памяти следом друг за другом, с возможностью доступа к каждой из них по индексу. В некотором роде, это составная переменная, характеризующаяся общим идентификатором, типом хранимых значений и количеством пронумерованных элементов.

Например, целочисленный массив из 5 элементов можно описать так:

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

Размер массива указывается в квадратных скобках после имени. Элементы нумеруются от 0 до N-1, где N — размер массива. Доступ к ним, то есть чтение значений, производится с помощью похожего синтаксиса. Например, чтобы вывести в журнал первый элемент указанного массива, можно было бы написать инструкцию:

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

Обратите внимание, что индекс 0 соответствует самому первому элементу. А для вывода последнего элемента инструкция поменялась бы на такую:

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

Конечно, предполагается, что перед печатью элемента массива в него когда-то было записано полезное значение. Такая запись выполняется с помощью специальной инструкции — оператора присваивания. Отличительной чертой этого оператора является использованием символа '=', слева от которого указывается элемент массива (или переменная), куда производится запись, а справа — записываемое значение или его "эквивалент". Под "эквивалентом" здесь скрывается возможность языка вычислять арифметические, логические и прочие типы выражений (мы изучим их во второй части книги). Синтаксис выражений во многом походит на правила записи уравнений из школьных курсов арифметики и алгебры. Например, в выражении можно использовать операции сложения ('+'), вычитания ('-'), умножения ('*'), деления ('/').

Вот, например, как могли бы выглядеть операторы для заполнения некоторых элементов указанного массива.

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

Эти инструкции демонстрируют разные способы присваивания и конструирования выражений: в первой строке в элемент *array[0]* записывается литерал 10, во второй и третьей — используются выражения, вычисление которых приводит к получению результатов, указанных для наглядности в комментариях.

Когда элементы массивов (или переменные, в общем случае) участвуют в выражении, компьютер в процессе выполнения программы считывает из памяти их значения и выполняет над ними указанные операции.

Следует различать использование переменных и элементов массивов слева и справа от знака '=' в инструкции присваивания: слева стоит "получатель" обработанных данных (он всегда один), а справа — "источники" исходных данных для вычислений ("источников" может быть много в выражении, как в этом примере в последней строке, где перемножаются значения элементов *array[0]* и *array[1]*).

В приведенных примерах знак '=' был применен для присвоения значений элементам массива, который определен заранее. Но зачастую бывает удобно присваивать начальные значения переменным и массивам непосредственно в момент их определения. Это называется инициализацией. Для неё также используется знак "равно". Рассмотрим этот синтаксис в контексте нашей прикладной задачи.

Опишем массив строк с вариантами приветствия внутри функции *Greeting*:

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

В добавленной инструкции производится не только определение массива *messages* с 3 элементами, но и его инициализация, то есть заполнение нужными начальными значениями. Инициализацию выделяет знак равно '=' после описания типа и имени переменной или массива. В случае переменной после '=' необходимо указать одно единственное значение (без фигурных скобок), а в случае массива, как мы видим, можно написать несколько значений, разделенных запятыми и заключенных в фигурные скобки.

Не путайте инициализацию и присваивание. Первая указывается при определении переменной/массива (и производится единожды), а второе встречается в отдельных инструкциях (причем одной и той же переменной или элементу массива могут многократно присваиваться разные значения). Присваивать элементы массива можно только по отдельности: MQL5 не поддерживает присваивание сразу всех элементов по аналогии с инициализацией.

Массив *messages*, будучи определен внутри функции, доступен только внутри неё, как и параметр *hour*. Далее мы увидим, как можно описать переменные, доступные во всем коде программы.

Каким же образом, преобразовать входящее значение *hour* с номером часа в один из трех элементов?

Напомним, что по задумке *hour* может быть равен числу от 0 до 23 (включительно). Если разделить его на 8 без остатка, получим значения от 0 до 2 (включительно). Например, деление 1 на 8 даст 0 и 7 на 8 даст 0 (дробная часть при делении без остатка отбрасывается). А вот деление 8 на 8 — это уже 1, и вплоть до 15 все числа будут давать 1 при делении на 8. Числа с 16 по 23 будут соответствовать результату деления 2. Полученные целые значения 0, 1, 2 следует использовать как индексы для чтения элемента массива *messages*.

В MQL5 деление без остатка для целых чисел позволяет вычислять операция '/'.

Выражение для получения результата деления похоже на те, что мы недавно рассмотрели для массива *array*, только в вычислении должен использоваться параметр *hour* и операция '/'. В качестве демонстрации возможной реализации преобразования *hour* в индекс элемента приведем такую инструкцию:

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

Здесь определяется некая новая целочисленная переменная *index* и инициализируется значением указанного выражения.

Однако, мы можем не сохранять промежуточное значение в переменной *index*, а сразу перенести данное выражение (справа от знака '=') внутрь квадратных скобок, где указывается номер элемента массива.

Тогда в инструкции с оператором *return* мы можем извлекать подходящее приветствие следующим образом:

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

Функция в общих чертах готова. Кое-какие правки мы внесем через пару разделов. А пока сохраним наработки в файле под новым именем *GoodTime0.mq5* и попробуем вызвать свою функцию. Для этого в *OnStart* используем обращение к *Greeting* внутри вызова *Print*.

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

Мы оставили разделяющую запятую (стоявшую внутри литерала "Hello, ") между приветствием и названием инструмента. Теперь в вызове функции *Print* три аргумента: первый и последний будут вычислены "на лету" с помощью вызовов, соответственно, функций *Greeting* и *Symbol*, а запятая отправится на печать как есть.

В функцию *Greeting* мы пока передаем константу 0. Именно её значение попадет в параметр *hour*. Откомпилировав и запустив программу, мы можем убедиться, что она выводит в журнал требуемый текст.

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

Однако, на практике выбор приветствия должен осуществляться динамически, в зависимости от времени, указанного пользователем.

Таким образом, мы подошли к необходимости организовать ввод данных.

1.8 Ввод данных

Основным способом передачи данных в MQL-программу являются входные параметры. Они похожи на параметры функций и просто переменные во многих смыслах, в частности, по синтаксису описания и по принципам дальнейшего использования в коде.

Вместе с тем, описание входного параметра имеет несколько существенных отличий:

- Оно размещается в тексте снаружи всех блоков (мы пока познакомились только с блоками, составляющими тело функций, но позднее узнаем и о других), или иными словами, вне любых пар фигурных скобок;
- Оно начинается с ключевого слова *input*;
- Оно имеет инициализацию значением по умолчанию.

Как правило, входные параметры рекомендуется располагать в начале исходного кода.

Например, чтобы определить входной параметр для ввода номера часа в нашем скрипте, следует добавить следующую строку сразу после тройки директив *#property*:

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

Такая запись означает, несколько вещей.

- Во-первых, в скрипте появилась переменная *GreetingHour*, доступная из любого места исходного кода, в том числе внутри любой функции. Подобное определение называют определением на глобальном уровне, причем оно становится таковым благодаря выполнению пункта 1 из вышеприведенного списка.
- Во-вторых, использование ключевого слова *input* делает такую переменную видимой не только внутри программы, но и в пользовательском интерфейсе, в диалоге настройки свойств MQL-программы, который открывается при её запуске. Таким образом, при старте программы пользователь задает необходимое значение параметров (в нашем случае, один параметр *GreetingHour*), и они становятся значениями соответствующих переменных во время выполнения программы.

Еще раз обратим внимание, что значение по умолчанию, которое мы указали в коде, будет показано пользователю в диалоге, но он сможет его изменить, и в этом случае именно это новое, введенное вручную значение попадет в программу (а не значение инициализации).

На начальное значение входных параметров влияет не только инициализация в коде и интерактивный выбор пользователя при их запуске, но и тип MQL-программы, и способ её запуска. Дело в том, что разные типы MQL-программ имеют разный жизненный цикл после запуска на графиках. Так, индикаторы и эксперты после однократного размещения на графике "прописываются" на нем навсегда, вплоть до момента, пока пользователь не удалит их явным образом. Поэтому терминал запоминает для них последние выбранные настройки и автоматически применяет их, например, после рестарта терминала. Однако скрипты не

сохраняются на графиках между сессиями терминала, и потому при запуске скрипта нам может показываться только значение по умолчанию.

К сожалению, для скриптов (как для отдельного типа MQL-программ) описание входного параметра еще не гарантирует вызов диалога настроек при запуске скрипта. Чтобы это произошло, необходимо добавить в код еще одну директиву *#property*, специфическую для скриптов:

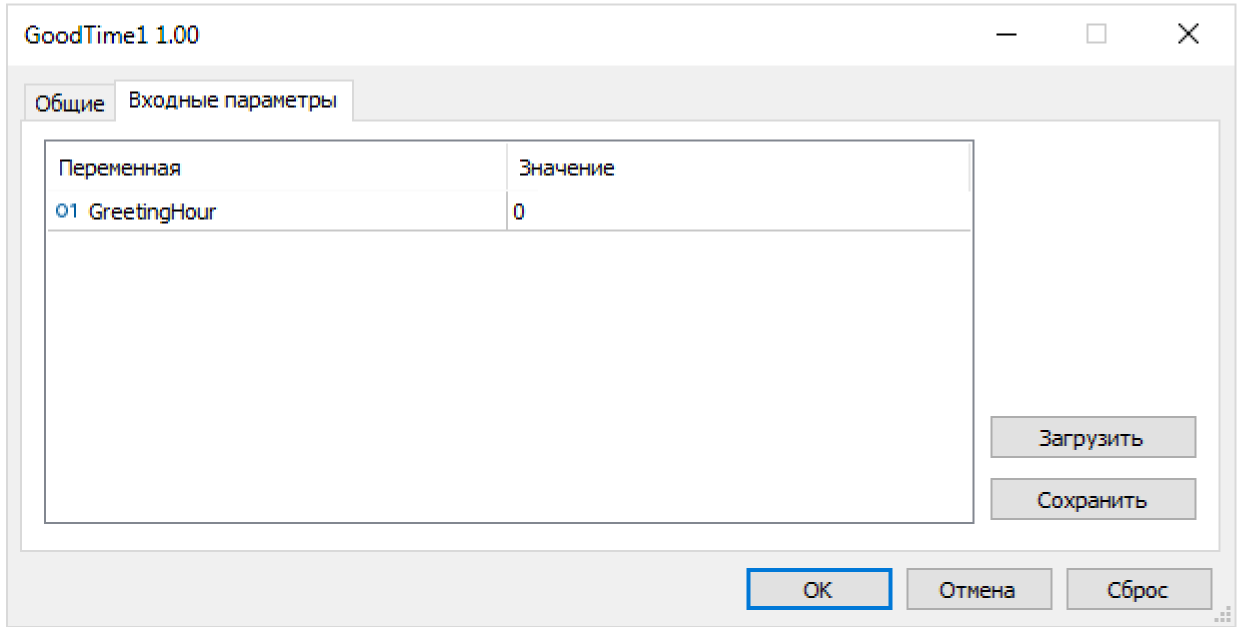
```
#property script_show_inputs
```

Как мы увидим далее, для других типов MQL-программ эта директива не нужна.

Входной параметр *GreetingHour* требовался нам для передачи его значения в функцию *Greeting*. Для этого достаточно вставить его в вызов функции *Greeting*, вместо 0:

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

С учетом изменений, которые мы внесли для описания входного параметра, сохраним новую версию скрипта в файл *GoodTime1.mq5*. Если его откомпилировать и запустить, мы увидим диалог ввода данных:



Диалог ввода параметров скрипта GoodTime1.mq5

Например, если мы отредактируем значение *GreetingHour*, сделав его 10, то скрипт выдаст следующее приветствие:

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

Это правильный, ожидаемый результат.

Ради интереса запустим скрипт еще раз и введем теперь значение 100. Вместо какого-либо вразумительного ответа мы получим:

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

Мы столкнулись с новым для себя явлением — ошибкой времени исполнения. В данном случае терминал сообщает, что в строке 19, в позиции 18 наш скрипт попытался прочитать значение элемента массива с несуществующим индексом (за пределами размера массива).

Поскольку ошибки являются постоянным, неизбежным спутником программиста, и нам нужно научиться их исправлять, расскажем о них чуть подробнее.

1.9 Работа над ошибками и отладка

Искусство программирования базируется не только на умении вложить в программу, что и как она должна делать, но и на том, чтобы оградить её от потенциальной возможности сделать что-то не так. Второе, к сожалению, гораздо труднее выполнить из-за множества не столько очевидных заранее факторов, влияющих на поведение программы: некорректные данные, недостаток ресурсов, чужие и собственные ошибки кодирования — вот лишь некоторые из проблем.

При написании программ никто не застрахован от ошибок. Ошибки могут проявляться на разных этапах и условно делятся на:

- Ошибки компиляции, которые выдает компилятор при обнаружении исходного кода, не отвечающего требуемому синтаксису (с такими ошибками мы уже познакомились ранее); их проще всего исправлять, поскольку их поиск выполняет компилятор;
- Ошибки времени исполнения программы, которые выдает терминал при возникновении в программе некорректного условия, такого как деление на ноль, взятие квадратного корня из отрицательного числа или попытка обратиться к несуществующему элементу массива, как случилось в нашем случае; их сложнее обнаружить, потому что они, как правило, возникают не при любых значениях входных параметров, а только при конкретных специфических условиях;
- Ошибки проектирования программы, которые приводят к её полной неработоспособности без каких-либо подсказок со стороны терминала, например, зависание в бесконечном цикле; такие ошибки могут оказаться самыми сложными в плане их локализации и воспроизведения, а ведь возможность воспроизвести проблемное состояние программы — необходимое условие для последующего исправления;
- Скрытые ошибки, когда программа вроде бы работает гладко, но выдаваемый результат не соответствует правильному — это легко обнаружить, если 2*2 не равно 4, а на практике заметить расхождения бывает намного сложнее.

Но вернемся к разбору конкретной ситуации со скриптом. Согласно сообщению об ошибке, которую нам выдала среда исполнения MQL-программ, неверно написана следующая инструкция:

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

При вычислении индекса элемента из массива, в зависимости от значения переменной *hour*, может быть получена величина, выходящая за размер массива, равный трем.

Убедиться в том, что именно так и происходит, позволяет встроенный в MetaEditor отладчик. Все его команды собраны в меню Отладка. Они предоставляют много полезных функций, но мы здесь остановимся только на двух: *Отладка -> Начать на реальных данных* (F5) и *Отладка -> Начать на исторических данных* (Ctrl+F5). Про остальные можно почитать в Справке по MetaEditor.

Обе команды компилируют программу особым образом — с отладочной информацией. Такая версия программы получается не оптимизированной, как при стандартной компиляции (подробнее про оптимизацию см. Документацию), зато позволяет за счет отладочной информации "заглянуть" внутрь программы в процессе выполнения: увидеть состояние переменных и стек вызовов функций.

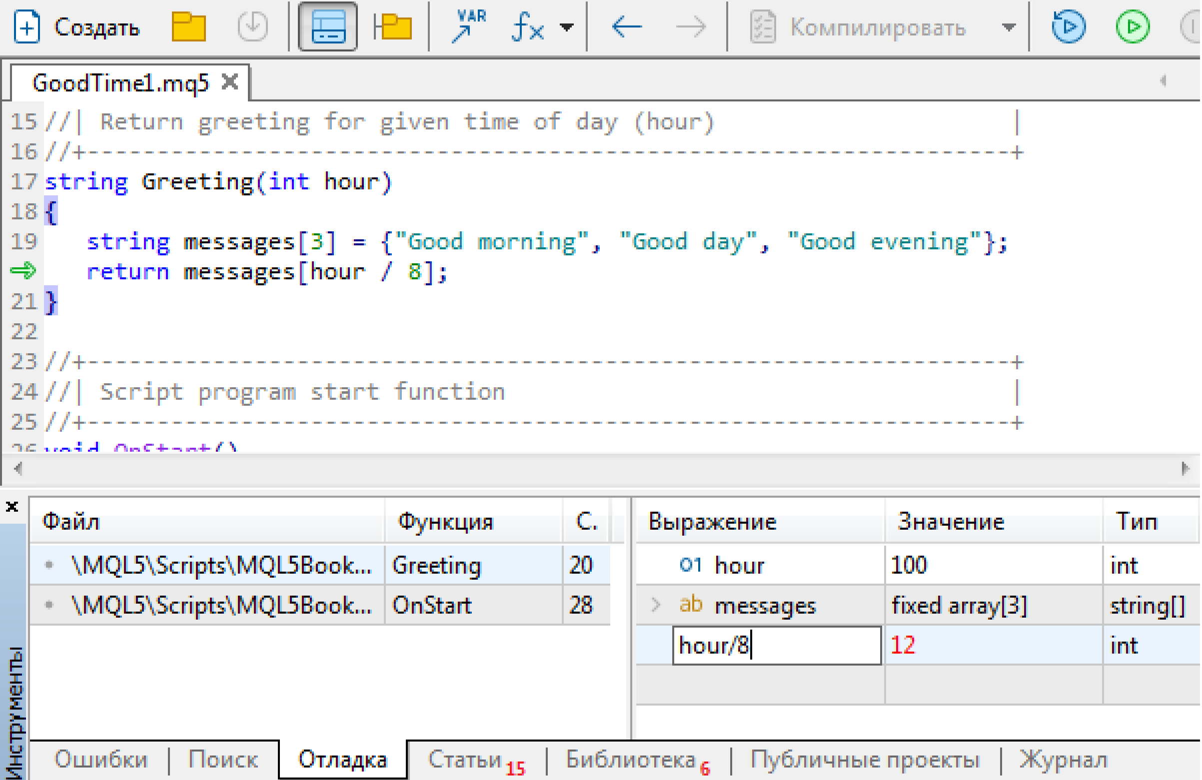
Разница между отладкой на реальных данных и исторических заключается в том, что в первом случае программа запускается на онлайн-графике, а во втором — на графике тестера в визуальном режиме. Для указания редактору, какой именно график и с какими настройками использовать (символ, таймфрейм, диапазон дат и прочее), следует предварительно открыть диалог *Настройки -> Отладка*, и заполнить в нем требуемые поля. Опция *Использовать указанные настройки* должна быть включена. Если данный флаг сброшен, при онлайн-отладке будет использован первый символ из *Обзора рынка* и таймфрейм H1, а при отладке на истории берутся настройки тестера.

Обратите внимание, что в тестере можно отлаживать только индикаторы и эксперты. Для скриптов доступна только отладка онлайн.

Запустим наш скрипт с помощью F5 и введем 100 в параметр *GreetingHour*, чтобы воспроизвести прошлую проблемную ситуацию. Скрипт начнет выполняться и почти сразу терминал выведет сообщение об ошибке и запрос на открытие отладчика.

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

Ответив утвердительно, мы попадем в MetaEditor, где в исходном коде будет подсвечена текущая строка, где возникла ошибка (обратите внимание на зеленую стрелочку на левом поле).



MetaEditor в режиме отладки при возникновении ошибки

В нижней левой части окна выводится текущий стек вызовов: в нем (в порядке снизу вверх) перечислены все функции, которые были вызваны, прежде чем выполнение кода остановилось на текущей строке. В частности, в нашем скрипте была вызвана функция *OnStart* (её вызвал сам терминал), а из неё вызвана функция *Greeting* (её мы вызвали из своего кода). В правой нижней части окна располагается панель наблюдения. В неё можно вводить названия переменных или целые выражения в колонку *Выражение* и наблюдать их значения в колонке *Значение* в той же строке.

Например, мы сейчас можем по команде *Добавить* контекстного меню или по двойному щелчку мыши на первой свободной строке ввести выражение "hour / 8" и убедиться, что оно равно 12.

Поскольку отладка приостановилась в результате ошибки, продолжать работу программы бессмысленно и можно выполнить команду *Отладка -> Завершить* (Shift+F5).

В более сложных случаях, когда источник проблем не столь очевиден, отладчик дает возможность отследить по шагам, строка за строкой, последовательность выполнения инструкций и содержимое переменных.

Для решения проблемы необходимо обеспечить в коде, чтобы индекс элемента всегда попадал в диапазон 0-2 (т.е. соответствовал размеру массива). Строго говоря, нужно было бы дописать несколько инструкций, проверяющих введенные данные на корректность (в нашем случае значение *GreetingHour* может принимать значение только в диапазоне от 0 по 23), и при нарушении условий — либо выводить подсказку, либо автоматически исправлять.

В рамках текущего вводного проекта мы ограничимся более простым исправлением: усовершенствуем выражение, вычисляющее индекс элемента таким образом, чтобы его результат всегда попадал в требуемый диапазон. Для этого познакомимся с еще одним оператором — делением по модулю, которое работает только для целых чисел. Для обозначения этой операции используется символ '%'. Результатом деления по модулю является остаток от целочисленного деления делимого на делитель. Например:

```
11 % 5 = 1
```

Здесь при целочисленном делении 11 на 5 получилось бы 2, что соответствует максимальной кратной 5 величине в составе 11, а это 10. А остаток между 11 и 10 как раз и дает 1.

Для исправления ошибки в функции *Greeting* достаточно предварительно выполнить деление *hour* по модулю на 24 — таким образом будет обеспечено, что номер часа окажется в диапазоне 0 — 23. Функция *Greeting* станет выглядеть следующим образом:

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

Хотя данная правка, несомненно, сработает хорошо (мы проверим это через минуту), она не затрагивает другую проблему, которая осталась вне нашего внимания. Дело в том, что параметр *GreetingHour* имеет тип *int*, то есть может принимать не только положительные, но и отрицательные значения. Если бы мы попытались ввести, например, -8 (или большее отрицательное число), то получили бы ту же ошибку времени выполнения — выход за пределы массива, только на этот раз индекс превышает не максимальное значение (размер массива), а становится меньше минимального (в частности, -8 приводит к обращению к -1-му элементу,

причем, что интересно, значения от -7 до -1 отображаются на 0-й элемент и ошибки не вызывают).

Для исправления данной проблемы мы заменим тип параметра *GreetingHour* на тип беззнакового целого: вместо *int* будем использовать *uint* (про все доступные типы мы расскажем во второй части, а здесь нам пригодится именно *uint*). Руководствуясь ограничением на неотрицательность значений, встроенном на уровне компилятора для *uint*, MQL5 самостоятельно обеспечит, чтобы ни пользователь (в диалоге свойств), ни программа (в своих расчетах) не "уходили в минус".

Сохраним новую версию скрипта под именем *GoodTime2*, откомпилируем и запустим. Введем значение 100 для параметра *GreetingHour* и убедимся, что на этот раз скрипт выполняется без ошибок, а в журнал терминала выводится приветствие "Good morning". Это ожидаемое (правильное) поведение, так как мы можем с помощью калькулятора проверить, что остаток от деления 100 на 24 по модулю дает 4, а деление 4 на 8 без остатка равно 0 (а это обозначает у нас утро). Конечно, такое поведение можно посчитать неожиданным с точки зрения пользователя, но с другой стороны, он тоже действовал неожиданно, вводя в качестве номера часа значение 100. Может быть он полагал, что наша программа упадет? Но этого не произошло, и это положительный момент. Разумеется, в реальных программах нужно проверять вводимые значения на допустимость и сообщать пользователю о нестыковках.

Здесь же в качестве дополнительной меры по предупреждению ввода неверного числа воспользуемся специальной возможностью MQL5 указать для входного параметра более подробное "дружелюбное" название. Достигается это с помощью комментария, указываемого после описания входного параметра, в той же строке. Например, так:

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

Обратите внимание, что в комментарии мы написали слова из имени переменной раздельно (потому что это уже не идентификатор в коде, а подсказка о нем для пользователя). Кроме того, мы добавили в круглых скобках диапазон допустимых значений. При запуске такого скрипта в диалоге ввода параметров прежний *GreetingHour* будет показан как:

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

Теперь можно быть уверенным, что если кто-то и введет 100 в качестве часа, это не наша вина.

Внимательный читатель может поинтересоваться, зачем мы определили функцию *Greeting* с параметром *hour* и передаем в неё *GreetingHour*, если можно было использовать входной параметр непосредственно в ней. Функция, как обособленный логический фрагмент кода, формируется не только для дробления программы на обозримые, простые для понимания части, но и для последующего повторного использования. Функции, как правило, вызываются из нескольких частей программы или входят в состав библиотеки, которую подключают ко многим разным программам. Поэтому правильно написанная функция должна быть независимой от внешнего контекста и переносима между программами.

Например, если потребуется перенести нашу функцию *Greeting* в другой скрипт, то там она перестанет компилироваться, так как в нём не будет параметра *GreetingHour*. И требовать добавить его не совсем корректно, потому что другой скрипт может рассчитывать время каким-то другим способом. Иными словами, при написании функции нужно стараться избавиться от необязательных внешних зависимостей. Вместо них следует декларировать параметры функции, которые могут быть заполнены вызывающим кодом.

1.10 Вывод данных

В случае нашего скрипта вывод данных производится простой записью приветствия в журнал с помощью функции *Print*. При необходимости MQL5 позволяет сохранять результаты в файлы, базы данных, отправлять в Интернет, отображать в виде графических серий (в индикаторах) или объектов на графиках.

Самый простой способ сообщить пользователю какую-то простую сиюминутную информацию, не вынуждая его заглядывать в журнал (который все-таки является служебным средством мониторинга работы программ и может не отображаться на экране), предоставляет функция MQL5 API — *Comment*. Её можно использовать точно так же, как функцию Print, но в результате её выполнения текст выводится не в журнал, а на текущий график, в его верхнем левом углу.

Например, заменив *Print* на *Comment* в тестовом скрипте, мы получим такую функцию *Greeting*:

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

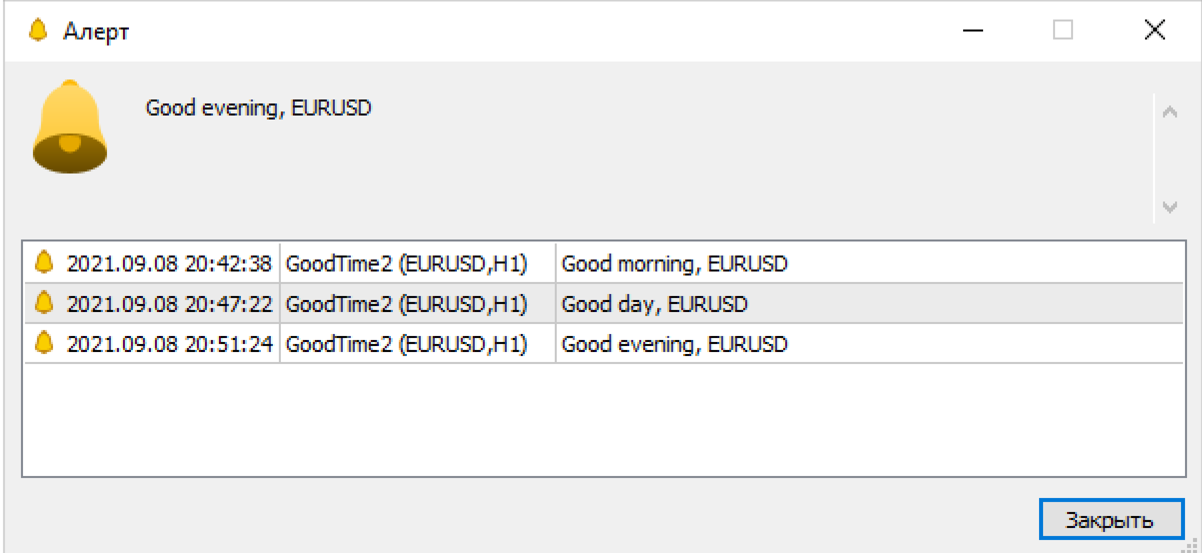
Запустив измененный скрипт в терминале, увидим такую картину:

Вывод текстовой информации на график с помощью функции Comment

Если требуется не просто отобразить текст для пользователя, но и обратить его внимание на какое-то изменение в среде, сопряженное с новой информацией, то лучше использовать функцию *Alert*. Она отправляет сообщение в отдельное окно терминала, которое всплывает поверх главного окна, и сопровождает это звуковым сигналом. Это бывает полезно, например, при возникновении торгового сигнала или в нештатных ситуациях, требующих вмешательства пользователя.

Синтаксис *Alert* точно такой же как у *Print* и *Comment*.

На изображении ниже показан результат работы функции *Alert*.



Вывод уведомления с помощью функции Alert

Варианты скрипта с функциями *Comment* и *Alert* не прикладываются к книге, чтобы читатель самостоятельно попробовал отредактировать *GoodTime2.mq5* и воспроизвести приведенные скриншоты.

1.11 Форматирование, отступы и пробелы

Язык MQL5 относится к числу языков со свободным форматированием (free-form), как C-подобные и многие другие языки. Это означает, что размещение служебных символов (например, скобок, операторов) и ключевых слов может быть произвольным, при условии соблюдения синтаксических правил. Синтаксис ограничивает только взаимную последовательность этих символов и слов, а размер отступов в начале каждой строки исходного текста или количество пробелов между составными элементами инструкции не имеют для компилятора никакого значения. В любом месте текста, где нужно вставить пробел, чтобы разделить элементы языка (например, ключевое слово с названием типа переменной и идентификатор переменной), можно использовать большее количество пробелов. Более того, вместо пробелов разрешено применять другие символы, обозначающие пустое пространство, такие как табуляция и перевод строки.

Если между некоторыми элементами инструкции уже есть какой-либо символ-разделитель (мы узнаем подробности о них во второй части), такой как запятая ',' между параметрами функции, то использовать пробел вовсе не обязательно.

Исполняемый код при изменении форматирования исходного текста не меняется.

В принципе, существует множество языков, не являющихся свободно-форматируемыми. В некоторых из них формирование блока кода, которое производится в MQL5 с помощью парных фигурных скобок, основывается на равных отступах от левого края.

Благодаря свободному форматированию, MQL5 позволяет программистам использовать множество различных способов оформления исходного кода, в целях улучшения его читабельности, наглядности, простоты внутренней навигации.

Рассмотрим несколько примеров того, как можно записать исходный текст функции *Greeting* из нашего скрипта, не меняя её сути.

Вот максимально "упакованная" версия вообще без лишних пробелов и переводов строк (перевод строки, обозначенный здесь символом '\', сделан только для соблюдения ограничений, накладываемых на публикацию исходных текстов в данной книге).

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

Вот версия, где наоборот вставлены лишние пробелы и переводы строк.

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

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

MetaEditor имеет встроенный стилизатор кода, который позволяет автоматически отформатировать исходный код текущего файла в соответствии с одним из поддерживаемых стилей. Выбор конкретного стиля осуществляется в диалоге *Сервис -> Настройки -> Стилизатор*. Применение стиля выполняется с помощью команды *Сервис -> Стилизатор*.

Следует иметь в виду, что свобода расстановки пробелов имеет ограничения. В частности, нельзя вставлять пробелы внутрь идентификаторов, ключевых слов или чисел — так компилятор не сможет их распознать. Например, если вставить хоть один пробел между цифрами 2 и 4 в числе 24, компилятор выдаст кучу ошибок при попытке откомпилировать скрипт.

Вот заведомо неправильно модифицированная строка:

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

А вот журнал с ошибками:

```
'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
```

Сообщения компилятора могут выглядеть не всегда понятно. Следует учитывать, что уже после первой (по порядку) ошибки велика вероятность, что внутреннее представление программы (как её "на полуслове" воспринял компилятор) существенно отличается от предполагаемого программистом. В частности, в данном случае только первая и вторая ошибки содержат ключ к пониманию проблемы, а все остальные являются наведенными.

Согласно первой ошибке, компилятор ожидал найти между 2 и 4 символ какой-нибудь операции (т.к. 2 и 4 расцениваются им, как два разных числа, а не разделенное пробелом 24). Альтернативная логика заключается в том, что в данном месте пропущена закрывающая квадратная скобка, и компилятор выдал вторую ошибку — "не хватает парной скобки для '['". После этого разбор выражения окончательно рушится, в результате чего последующие число 8 и закрывающая скобка ']' кажутся компилятору также неуместными. А на самом деле, достаточно убрать лишний пробел между 2 и 4, и ситуация нормализуется.

Разумеется, такой анализ ошибок гораздо проще делать, когда мы сами намеренно внесли проблему. На практике не всегда удается понять, как именно следует исправлять ту или иную ситуацию. Даже в рассмотренном выше случае, если предположить, что Вы получили данный испорченный исходный код от другого программиста и элементы массива содержат не столь тривиальную информацию, легко заподозрить другой вариант исправления: нужно оставить либо 2, либо 4, потому что, вероятно, автор хотел заменить одно число на другое и не "подчистил следы".

1.12 Подводим промежуточный итог

В Части 1 мы познакомились со средой разработки MetaEditor, создали заготовку скрипта с помощью Мастера MQL и постепенно наполнили скрипт кодом для решения простой задачи. Для этого мы применили несколько базовых принципов и синтаксических конструкций MQL5. Затем мы попробовали в деле отладчик, исправили несколько проблем и добились стабильной работы программы.

Эволюция наших примеров со скриптом выглядит так:

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

Hello.mq5

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

HelloChart.mq5

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

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

HelloChartFunc.mq5

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

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

GoodTime0.mq5

```
#property script_show_inputs

input int GreetingHour = 0;

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

GoodTime1.mq5

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

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

GoodTime2.mq5

В следующих разделах книги мы приступим к доскональному изучению этих и многих других возможностей языка MQL5, технической стороны программирования и прикладных аспектов для области трейдинга.

Часть 2. Основы программирования на MQL5

Как и любой другой язык программирования, MQL5 основывается на нескольких базовых понятиях, которые применяются для создания более сложных конструкций и в конечном счете — программ в целом. В этой части мы изучим большинство таких понятий: типы данных, идентификаторы, переменные, выражения, операторы, а также способы комбинирования различных инструкций в коде для выстраивания требуемой логики работы программы.

Освоение материала позволит читателям самостоятельно использовать на практике процедурное программирование. Это одно из самых первых появившихся направлений программирования для решения произвольных задач. Его суть сводится к формированию программы из мелких шагов (инструкций), выполняемых в требуемой последовательности для обработки данных. Примером такого стиля является и тестовый скрипт, который мы видели в первой части книги.

Этот раздел охватывает широкий спектр основных концепций и инструментов, необходимых для успешного программирования в MQL5, включая следующие подразделы:

Идентификаторы:

- Идентификаторы являются основой любого программного кода. В этом подразделе рассматривается назначение и правила именования идентификаторов в MQL5.

Встроенные типы данных:

- В MQL5 существует разнообразие встроенных типов данных, каждый из которых предназначен для хранения и обработки определенного вида информации. Этот раздел предоставляет полное понимание базовых типов данных.

Переменные:

Переменные служат для хранения и управления данными в программе. Раздел "Переменные" обучает основам работы с переменными, их объявлению, инициализации и присваиванию значений.

Массивы:

- Массивы предоставляют структурированный способ хранения данных. Здесь рассматриваются основы создания и использования массивов в MQL5.

Выражения:

- Выражения формируют основу вычислений и логики программы. В этом подразделе разбираются основные принципы построения и оценки выражений в MQL5.

Приведение типов:

- Преобразование типов данных является неотъемлемой частью программирования. Раздел "Приведение типов" предоставляет понимание процесса преобразования данных между различными типами в MQL5.

Инструкции:

- Инструкции представляют собой команды, которые управляют выполнением программы. Здесь подробно рассматриваются различные виды инструкций и их применение.

Функции:

- Функции позволяют структурировать код и повторно использовать его части. Этот раздел погружает в основы создания и вызова функций в MQL5.

Препроцессор:

- Препроцессор MQL5 выполняет обработку исходного кода перед компиляцией. Раздел "Препроцессор" описывает принципы использования препроцессорных директив и их влияние на код.

Принципы процедурного программирования послужат фундаментом для последующего изучения более мощной парадигмы — Объектно-ориентированного программирования (ООП). К ней мы обратимся в Части 3.

Программирование на MQL5 для трейдеров — исходные коды из книги: Часть 2

Примеры из книги также доступны в публичном проекте \MQL5\Shared Projects\MQL5Book

2.1 Идентификаторы

Как мы скоро увидим, программы строятся из множества элементов, к которым необходимо обращаться по уникальным именам для избегания путаницы. Эти имена и принято называть идентификаторами.

Идентификатор — это составленное по определенным правилам слово: в нем разрешено использовать только символы латинского алфавита, символ подчеркивания ('_') и цифры, причем цифра не может быть первым символом. Буквы могут быть как строчными (нижний регистр), так и прописными (верхний регистр).

Максимальная длина идентификатора — 63 символа. Идентификатор не должен совпадать со служебными словами MQL5, в частности, с названиями типов. Полный список служебных слов можно найти в Справке. Нарушение любого из правил составления идентификатора приведет к ошибке компиляции.

Вот примеры правильных идентификаторов:

```
i              // один символ
abc            // строчные буквы
ABC            // прописные буквы
Abc            // смешанный регистр
_abc           // подчеркивание в начале
_a_b_c_        // подчеркивание в любом месте
step1          // цифра
_1step         // подчеркивание и цифра
```

В скрипте *HelloChart* мы уже видели использование идентификаторов в качестве имен переменных и названий функций.

Рекомендуется давать идентификаторам осмысленные названия, из которых становится ясно назначение или содержимое соответствующего элемента. В некоторых случаях принято использовать однобуквенные идентификаторы — о них мы поговорим в разделе про циклы.

Существует несколько общепринятых практик по составлению идентификаторов. Например, если мы выбираем имя для переменной, хранящей значение профит-фактора, хорошими кандидатами считаются следующие:

```
ProfitFactor   // "горбатый" стиль, все слова с заглавной буквы
profitFactor   // "горбатый" стиль, все слова кроме первого с заглавной буквы
profit_factor  // "змеиный" стиль, между всеми словами вставляется подчеркивание
```

Во многих языках программирования принято использовать разные стили для именования разных сущностей. Например, можно придерживаться практики, что переменные начинаются только с маленькой буквы, а названия классов (см. Часть 3) — только с большой. Это помогает программисту анализировать исходный код при работе в команде или если он возвращается к собственному фрагменту кода после долгого перерыва.

Помимо упомянутых выше, есть и другие стили, некоторые из них применяются в особых случаях:

```
profitfactor   // "ровный" стиль, все буквы строчные
PROFITFACTOR   // "ровный" стиль, все буквы заглавные
PROFIT_FACTOR  // "макро" стиль, все буквы заглавные с подчеркиванием между слов
```

Все заглавные буквы иногда используются в именах констант.

"Макро" стиль обычно применяется в именах макро-определений препроцессора.

2.2 Встроенные типы данных

Тип данных — это фундаментальное понятие, которым мы органично пользуемся в быту, даже особо не задумываясь о его существовании. Он подразумевается, исходя из значения информации, которой мы обмениваемся, и допустимых для неё способов обработки. Например, контролируя домашний бюджет, мы складываем и вычитаем числа, представляющие собой наши доходы и траты. Здесь "число" является описателем типа, для которого мы хорошо представляем возможные значения и арифметические операции над ними. В контексте торговли в среде MetaTrader 5 имеется аналогичная величина — текущий баланс счета, и MQL5, разумеется, предоставляет некий механизм по созданию чисел и манипулированию ими.

В отличие от чисел, текстовая информация, такая как название торгового инструмента, подчиняется другим правилам. Здесь мы можем сложить слово из букв или предложение из слов, но подсчитать нарастающий итог или среднее от нескольких строк невозможно. Таким образом, "строка" относится к другому тип данных, нечисловому.

Кроме назначения и характерного набора операций, имеющих смысл для каждого типа, существует еще одна важная вещь, отличающая типы друг от друга — это их размер. Например, номер недели в году не может быть больше 52, а вот количество секунд, истекших с начала года, как правило, представляет собой астрономическую величину. Следовательно, для эффективного хранения и обработки столь разных значений в памяти компьютера можно выделять разные по размеру участки. Это приводит нас к пониманию того, что за обобщающим термином "число" могут, на самом деле, скрываться разные типы.

MQL5 позволяет использовать несколько числовых типов, различающихся не только размером ячеек памяти, выделяемых под них, но и дополнительными признаками. В частности, некоторые числа могут принимать отрицательные значения (например, плавающая прибыль в пунктах), а другие нет (номер счета). Кроме того, некоторые величины не способны иметь дробную часть и

потому их экономичнее представлять более строгим типом "целых чисел", в противоположность типам произвольных "чисел с дробной частью". Например, баланс счета или цена торгового инструмента, в общем случае, имеют значения с дробной частью. А количество ордеров в истории или опять же номер счета — это всегда целые числа.

MQL5 поддерживает набор универсальных типов, аналогичным тем, что присутствуют в подавляющем большинстве языков программирования. В набор входят типы целых чисел (разного размера), два типа — вещественных чисел (с дробной частью) разной точности, строки и одиночные символы, а также логический тип, состоящий всего из двух возможных значений: "истина" (*true*) и "ложь" (*false*). Кроме того, MQL5 предоставляет свои собственные, специфические типы, работающие со временем и цветом.

Для полноты картины заметим, что MQL5 позволяет расширять набор типов, объявляя в коде прикладные типы: структуры, классы и другие сущности, характерные для ООП, но мы рассмотрим их позднее.

Поскольку размер ячейки, где хранится значение, является важным атрибутом типа, затронем принцип организации памяти.

Минимальной единицей измерения памяти в компьютерах считается байт. Иными словами, байт — это минимальный размер ячейки, которую программа способна выделить под отдельное значение. Внутри байт состоит из 8 более мелких "частиц" — битов, каждый из которых может быть в двух состояниях: включенном (1) или выключенном (0). Все современные компьютеры используют на нижнем уровне такие биты, потому что подобное двоичное (бинарное) представление информации удобно для воплощения в "железе" (в оперативной памяти, в процессорах, во время передачи по сетевым кабелям или WiFi).

Обработка значений разных типов обеспечивается за счет разной интерпретации состояний битов в ячейках памяти. Об этом заботится компилятор. Программист обычно не опускается до уровня битов, хотя в языке имеются средства для этого (см. Побитовые операции).

Для описания типов в MQL5 применяются специальные зарезервированные слова. С некоторыми из них (*void*, *int*, *string*) мы уже познакомились в первой части. Полный список типов приведен ниже, с кратким описанием и размером в байтах.

Их можно условно разделить по назначению на числовые и символьные (помечены в соответствующих колонках), а также остальные, специализированные — строчный, логический (булевый), дата/время и цвет. Особняком стоит тип void, обозначающий отсутствие какого-либо значения. Помимо скалярных типов в MQL5 встроено несколько объектных типов для работы с комплексными числами, матрицами и векторами — complex, vector и matrix. Эти типы используются при решении множества задач из линейной алгебры, математического моделирования, в машинном обучении и т.д. Мы изучим их отдельно в 4-ой Части книги.

Тип	Размер (байты)	Число	Символ	Примечание
char	1	+	+	Однобайтовый символ или целое число со знаком
uchar	1	+	+	Однобайтовый символ или целое число без знака

Тип	Размер (байты)	Число	Символ	Примечание
short	2	+	+	Двухбайтовый символ или целое число со знаком
ushort	2	+	+	Двухбайтовый символ или целое число без знака
int	4	+		Целое число со знаком
uint	4	+		Целое число без знака
long	8	+		Целое число со знаком
ulong	8	+		Целое число без знака
float	4	+		Вещественное число со знаком
double	8	+		Вещественное число со знаком
enum	4	(int)		Перечисление
datetime	8	(ulong)		Дата и время
color	4	(uint)		Цвет
bool	1	(uchar)		Логический
string	10+ переменный			Строка
void	0			Пустота
complex	16	+		Структура с двумя полями типа double
vector	длина вектора x размер типа	+		Одномерный массив вещественного или комплексного типа
matrix	строки x столбцы x размер типа	+		Двумерный массив вещественного или комплексного типа

В зависимости от размера числового типа, в нем могут храниться величины в разных диапазонах. Помимо этого диапазон существенно отличается для целых и вещественных чисел одного и того же размера, потому что для них используется разное внутреннее представление значений. Все эти нюансы будут раскрыты в разделах о конкретных типах.

Программист волен выбирать числовой тип, исходя из предполагаемых значений, соображений эффективности и экономичности. В частности, меньший размер типа дает возможность уместить

больше таких значений в памяти, а целочисленные значения обрабатываются быстрее вещественных.

Обратите внимание, что числовые и символьные типы частично пересекаются. Происходит это потому, что символ хранится в памяти как целое число — код в соответствующей таблице символов: ANSI для однобайтовых символов или Unicode для двухбайтовых символов. ANSI — это стандарт, который назван по имени института (American National Standards Institute), а Unicode, как нетрудно догадаться, означает Universal Code (Character Set). Символы Unicode применяются в MQL5 для составления строк (тип *string*). Однобайтовые символы требуются обычно при интеграции программ с внешними источниками данных, например, из Интернета.

Как мы упомянули выше, числовые типы, можно разделить на целочисленные и вещественные. Рассмотрим их более подробно.

2.2.1 Целые числа

Целочисленные типы предназначены для хранения чисел без дробной части. Их следует выбирать, когда прикладной смысл значения исключает дроби. Например, количество баров на графике или количество открытых позиций всегда равно целому числу.

MQL5 позволяет выбрать целые типы с размером от 1 до 4 байт с помощью ключевых слов *char*, *short*, *int*, *long* соответственно. Все они являются знаковыми, то есть могут содержать как положительные, так и отрицательные значения. При необходимости целые типы, при тех же размерах, могут объявляться беззнаковыми (их названия начинаются с буквы 'u', unsigned): *uchar*, *ushort*, *uint*, *ulong*.

В зависимости от размера и "знаковости" типа, диапазоны возможных значений приведены в следующей таблице.

Тип	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

Запоминать вышеприведенные пограничные значения для каждого целочисленного типа нет необходимости. В MQL5 существует много предопределенных именованных констант, которые можно использовать в коде вместо "магических" чисел, включая и минимальные/максимальные целые. Данная технология рассматривается в разделе, посвященном препроцессору. А здесь мы лишь перечислим соответствующие именованные константы: CHAR_MIN, CHAR_MAX,

UCHAR_MAX, SHORT_MIN, SHORT_MAX,USHORT_MAX, INT_MIN, INT_MAX, UINT_MAX, LONG_MIN, LONG_MAX, ULONG_MAX.

Поясним, каким образом получаются эти значения. Для этого потребуется вернуться к битам и байтам.

Количество возможных сочетаний различных состояний 8 битов (включенных и выключенных) внутри одного байта равно 256. Это дает диапазон значений от 0 до 255, которые может хранить байт. Но их интерпретация зависит от типа, под который выделен данный байт. Разную интерпретацию обеспечивает компилятор, согласно инструкциям программиста.

Младший бит в байте (самый правый) обозначает число 1, второй бит — 2, третий — 4, и так далее вплоть до старшего бита, который соответствует 128. Не трудно заметить, что эти числа равны двойке, возведенной в степень, равную номеру бита (нумерация идет с 0). Это следствие использования двоичной системы.

Биты	**старшие**				**младшие**			
Номер	7	6	5	4	3	2	1	0
Значение	128	64	32	16	8	4	2	1

Когда все биты взведены, это дает сумму всех степеней двойки, то есть 255 — максимальное значение для байта. Если все биты сброшены, получаем ноль. Когда включен младший бит, число нечетное.

При кодировании знаковых чисел старший бит используется для пометки отрицательных значений. Поэтому для однобайтового целого в положительном диапазоне максимальным значением становится 127. Для отрицательного диапазона остается 128 возможных сочетаний, т.е. минимальное значение равно -128. Когда все биты в байте взведены, это интерпретируется как -1. Если у такого числа сбросить младший бит, получим -2, и так далее. Когда взведен только старший бит (знак), а все остальные сброшены — получаем -128.

Такое, на первый взгляд, нелогичное кодирование называется "дополнительным". Оно позволяет унифицировать вычисления знаковых и беззнаковых чисел на аппаратном уровне. Кроме того, оно позволяет не терять одно значение, что произошло бы, если бы положительная и отрицательная области кодировались одинаково: тогда у нас оказалось бы два значения для нуля — положительный 0 и отрицательный 0, что кроме прочего вносило бы двусмысленность.

Числа с бОльшим количеством байтов (2, 4, 8) имеют аналогичную сквозную нумерацию битов и прогрессию соответствующих им значений. Во всех случаях признаком отрицательности числа является взведенный старший бит старшего байта.

Итак, мы можем использовать байт для хранения целого числа без знака (*uchar*, сокращение от unsigned character) в диапазоне 0-255. А можем записать в байт целое число со знаком (для чего опишем его тип как *char*), и в этом случае компилятор поделит доступное количество сочетаний 256 поровну между положительными и отрицательными значениями, отобразив его на область от -128 до 127 (256-е значение — это ноль). Как нетрудно догадаться, значения от 0 до 127 будут кодироваться на уровне битов одинаково для знакового и беззнакового байтов. Однако бОльшие абсолютные значения, начиная со 128, "превратятся" в отрицательные (по схеме, описанной во врезке выше). Данное "превращение" происходит только в момент чтения или выполнения операций над хранимым значением, при одинаковом внутреннем представлении данных (состоянии битов).

Мы изучим этот вопрос подробнее в разделе, посвященном приведению типов.

По аналогии с однобайтовыми целыми числами, легко вычислить, что количество сочетаний битов для 2 байт составляет 65536. Отсюда формируются диапазоны для знакового и беззнакового двухбайтового целого (*short* и *ushort*). Остальные типы позволяют хранить еще большие значения за счет увеличения размера в байтах.

Обратите внимание, что использование беззнакового типа при том же размере позволяет увеличить в два раза максимальное положительное значение. Это бывает необходимо для хранения потенциально очень больших величин, для которых исключено появление отрицательных значений. Например, номер приказа в MetaTrader 5 является значением типа *ulong*.

Примеры описания целочисленных переменных мы уже встречали в Части 1. Там, в частности, был определен входной параметр *GreetingHour* типа *uint*:

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

За исключением дополнительного ключевого слова input, которое делает переменную видимой в списке параметров MQL-программы, остальные компоненты — тип, имя, и опциональная инициализация после знака '=' — свойственны всем переменным.

Про синтаксис описания переменных будет подробно рассказано в разделе Переменные. Пока же обратим внимание на способ записи константных значений целого типа. При описании переменной они могут быть указаны в качестве значения по умолчанию (в примере выше это 0). Кроме того, константы могут использоваться в выражениях, например, при вычислении по формуле.

Напомним, что константы любого типа, вставленные в исходный код, называются литералами (дословно, "буквальными"). Их название происходит от того факта, что они внедряются в программу "как есть" и используются непосредственно по месту описания. Литералы, в отличие от многих других элементов языка, в частности, переменных, не имеют имен и на них нельзя сослаться из других мест программы.

Для отрицательных чисел указание знака минус '-' перед числом обязательно, но для положительных чисел знак плюс '+' можно опускать, то есть форма +100 и просто 100 — идентичны.

Стоит отметить, что числовые значения обычно записываются в исходном коде в привычной нам десятичной системе счисления. Однако MQL5 допускает использование и другой — шестнадцатеричной. Она удобна при обработке информации на уровне битов (см. Побитовые операции).

Если в константах десятичной системы допустимы цифры от 0 до 9 во всех разрядах, то для шестнадцатеричных — кроме цифр дополнительно используются латинские символы от *A* до *F* или от *a* до *f* (то есть регистр не имеет значения). "Шестнадцатеричная цифра" *A* соответствует числу 10 десятичной системы, *B* — 11, *C* — 12 и так далее вплоть до *F*, равного 15.

Отличительной чертой шестнадцатеричной константы является то, что она начинается с префикса 0x или 0X, за которым следуют значащие разряды числа. Например, число 1 в шестнадцатеричной системе записывается 0x1, а 16 — 0x10 (требуется дополнительный старший разряд, потому что 16 больше 15, то есть 0xF). Десятичное 255 превращается в 0xFF.

Приведем еще несколько примеров, иллюстрирующих различные ситуации с использованием целочисленных типов при описании переменных (прилагаются в скрипте *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
}
```

Переменная *x* правильно инициализируется допустимым отрицательным значением -10.

Переменная *y* является беззнаковой, и потому попытка записать в неё отрицательное значение приводит к интересному эффекту. Число -1 имеет представление в битах, которое программа интерпретирует в соответствии с беззнаковым типом *uint* и потому получается число 4294967295 (оно, кстати, равно UINT_MAX).

Переменной *z* присваивается вещественное число 1.23 (они будут рассмотрены в следующем разделе), и компилятор выдает предупреждение об отбрасывании дробной части. В результате в переменную попадает целое значение 1.

Переменная *h* успешно инициализируется константой в шестнадцатеричной форме (0x1000 = 4096).

Большое значение 10000000000 записывается в переменные *p* и *w*, первая из которых длинного целого типа (*long*) и обрабатывается успешно, в вторая — обычного (*int*), и потому вызывает предупреждение компилятора. Поскольку константа превышает максимальное значение для *int*, компилятор отсекает лишние старшие разряды (биты) и фактически в *w* попадает величина 1410065408.

Такое поведение является одним из возможных негативных проявлений преобразований типов, которые могут подразумеваться программистом, а могут — и нет. В последнем случае это чревато потенциальной ошибкой. Разумеется, в данном конкретном примере некорректные значения были выбраны намеренно для демонстрации предупреждений. В реальной программе не всегда столь очевидно, какое значение программа попытается сохранить в целочисленную переменную, и потому предупреждения компилятора стоит изучить очень внимательно, и постараться от них избавиться, поменяв тип или явно указав требуемое приведение типов. Об этом будет рассказано в разделе о приведении типов.

Для целых типов определены арифметические, побитовые и прочие виды операций (см. главу Выражения).

2.2.2 Вещественные числа

Числа с дробной частью, или иначе — вещественные, используются нами в повседневной жизни не менее часто, чем целые. Само название "вещественные" говорит о том, что с помощью таких чисел можно выразить нечто осязаемое из реального мира — вес, длину, температуру тела — то, что измеряется не ровным количеством единиц измерения, а "с хвостиком".

В трейдинге мы также часто оперируем вещественными числами. Например, в них выражаются цены инструментов или объемы в торговых приказах (обычно, допускающие дробные части целого лота).

В MQL5 предусмотрено 2 вещественных типа: *float* — нормальной точности и *double* — удвоенной точности.

В исходном коде константные значения типов *float* и *double* обычно записываются как целая и дробная части (каждая из них — это последовательность цифр), разделенные символом '.', например, 1.23, -789.01. Целая или дробная часть может отсутствовать (но не обе вместе), только точка является обязательной. Например, .123 означает 0.123, а 123. — 123.0. Просто 123 создаст константу целого типа.

Однако существует и другая форма записи вещественных констант — показательная. В ней после целой и дробной частей идет символ 'E' или 'e' (регистр не важен) и целое число, представляющее собой степень, в которую возводится 10 для получения дополнительного множителя. Например, следующие варианты представляют одно и то же число 0.57 в показательной форме:

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

При записи вещественных констант они по умолчанию определяются как тип *double* (занимают 8 байт). Для того чтобы задать тип *float*, справа к константе следует добавить суффикс 'F' (или 'f').

Типы *float* и *double* отличаются размером, диапазоном значений и точностью представления чисел. Все это указано в таблице ниже.

Тип	Размер (байты)	Минимальное	Максимальное	Точность (разряды)
float	4	$\pm 1.18 * 10^{-38}$	$\pm 3.4 * 10^{38}$	6-9, обычно 7
double	8	$\pm 2.23 * 10^{-308}$	$\pm 1.80 * 10^{308}$	15-18, обычно 16

Диапазон значений для них указан в абсолютных величинах: минимум и максимум определяют размах допустимых значений как в положительной, так и в отрицательной области. Также как и для целочисленных типов, для этих предельных величин имеются встроенные именованные константы: FLT_MIN, FLT_MAX, DBL_MIN, DBL_MAX.

Обратите внимание, что вещественные числа — всегда со знаком, то есть для них нет беззнаковых аналогов.

Под точностью понимается количество значащих цифр (десятичных разрядов), которые способно без искажений сохранить вещественное число соответствующего типа.

Действительно, числа вещественных типов не столь точны как целых типов. Это плата за их универсальность и гораздо больший диапазон возможных значений. Например, если 4-байтовое беззнаковое целое (*uint*) имеет максимальное значение 4294967295, то есть примерно 4 миллиона или $4.29*10^9$, то 4-байтовое вещественное (*float*) — $3.4 * 10^{38}$, что на 29 порядков больше. Для 8-байтовых типов разница еще заметнее: *ulong* способен вместить

18446744073709551615 ($18.44*10^{18}$ или ~18 квинтиллионов), а *double* — $1.80 * 10^{308}$, то есть на 289 порядков больше. Подробнее про точность рассказано во врезке.

Мантисса и показатель степени

Внутреннее представление вещественных чисел в памяти (в выделенных под них байтах) довольно хитроумное. Старший бит используется как маркер отрицательного знака (мы видели это и у целых типов). Все остальные биты поделены на две группы. Одна, более крупная, содержит мантиссу числа — значащие разряды (имеются в виду двоичные разряды, то есть биты). Вторая, более мелкая, хранит показатель степени, в которую нужно возвести 10, чтобы после домножения на мантиссу получить хранимое число. В частности, для типа *float* размер мантиссы — 24 бита (FLT_MANT_DIG), а для *double* — 53 (DBL_MANT_DIG). В переводе на привычные нам десятичные разряды (цифры) получим ту самую точность, что была указана в таблице выше: для *float* минимальное количество значащих цифр — 6 (FLT_DIG), а для *double* — 15 (DBL_DIG). Но в зависимости от конкретного числа, в нем могут быть "удачные" сочетания битов, соответствующие большему количеству десятичных цифр. Размеры показателей — 8 и 11 бит для *float* и *double* соответственно.

За счет степенного показателя вещественные числа получают гораздо больший диапазон значений. Вместе с тем, при увеличении показателя "удельный вес" младшего разряда мантиссы также увеличивается, что означает, что два соседних вещественных числа, которые возможно представить в памяти компьютера, существенно отличаются. Например, для числа 1.0 "удельный вес" младшего бита равен 1.192092896e–07 (FLT_EPSILON) в случае *float*, и 2.2204460492503131e-016 (DBL_EPSILON) в случае *double*. Иными словами, число 1.0 неотличимо от любого числа в его окрестности меньше 1.192092896e–07. Это кажется не очень важным или страшным, но для более крупных чисел данная область неопределенности также становится больше. Если сохранить во *float* число около 1 миллиарда ($1*10^9$), последние 2 цифры перестанут надежно сохраняться и восстанавливаться из памяти (см. пример кода ниже). Но, в принципе, проблема не в абсолютной величине числа, а в максимальном количестве цифр в нем, которые нужно воспроизвести без потерь. С тем же "успехом" мы можем попытаться уместить во *float* число вида 1234.56789 (которое по структуре очень похоже на цену финансового инструмента), и два его последних разряда "будут гулять" из-за недостатка точности во внутреннем представлении.

Для *double* похожая ситуация начнет проявляться для гораздо больших чисел (или для гораздо большего количества значащих цифр), но она все равно возможна и часто случается на практике. Это необходимо учитывать при работе с очень большими или очень маленькими вещественными числами и писать программы с дополнительными проверками на возможную потерю точности. В частности, сравнивать вещественное число на ноль нужно особым образом. Мы коснемся этого в разделе про операции сравнения.

Внимательному читателю можно показаться, что размеры мантиссы и показателя указаны выше неверно. Поясним на примере типа *float*. Он хранится в ячейке памяти размером 4 байта, то есть занимает 32 бита. При этом размер мантиссы (24) и показателя (8) в сумме уже дают 32. А где же тогда бит со знаком? Дело в том, компьютерщики договорились хранить мантиссу в нормализованном виде. Понять, что это такое будет проще, если для начала рассмотреть показательную форму записи обычного десятичного числа. Скажем, число 123.0 можно представить как 1.23E2, 12.3E1, 0.123E3. Нормализованной формой считается запись, когда перед точкой стоит только одна значащая цифра, то есть не ноль. Для данного числа это — 1.23E2. Значащими цифрами в десятичной системе по определению считаются цифры от 1 до 9. Теперь мы плавно переходим в двоичную систему счисления. В ней значащая

цифра только одна — 1. Получается, что нормализованная форма в двоичном представлении всегда имеет первую цифру 1 и её можно опустить (не тратить на неё память). Таким образом, удается сберечь один бит в мантиссе. Фактически она содержит 23 бита (еще одна старшая единица неявно присутствует всегда и добавляется автоматически при реконструкции числа и его извлечении из памяти). Уменьшение мантиссы на 1 бит освобождает место под бит со знаком.

В подавляющем большинстве случаев, когда нужно использовать вещественный тип, выбор происходит в пользу *double*, как более точного. Тип *float* применяется только в целях экономии памяти, например, при работе с очень большими массивами данных.

Несколько примеров использования констант вещественных типов приведено в скрипте *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 = 0.123456789012345678901234567890; // 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
}
```

Переменные *a0*, *a1*, *a2*, *a3* содержат одинаковые числа (123.0), записанные разными способами.

В константе для переменной *b* опущен незначащий ноль перед точкой. Кроме того, здесь демонстрируется запись отрицательного числа с помощью знака минус '-'.

В переменную *q* сделана попытка сохранить максимальное целое число. В этом месте компилятор выдает предупреждение, потому что *double* не способен точно представить LONG_MAX: вместо 9223372036854775807 получится 9223372036854776000. Это наглядно показывает, что хоть диапазоны значений *double* намного превосходят диапазоны целых чисел, это достигается за счет потери младших разрядов.

Для сравнения, максимальное целое, которое *double* способен хранить без искажений, приведено в качестве значения переменной *d*. После него в последовательности целых чисел начнут встречаться пропуски, если мы используем для них *double*.

Переменная *z* напоминает еще раз об ограничении на максимальное количество значащих цифр (16) — более длинная константа будет усечена.

Переменные *y1* и *y2*, в которые записывается одно и то же число, но в разных форматах (*double* и *float*), позволяют увидеть потерю точности из-за перехода к *float*.

Переменные *m* и *n* будут по факту равны, потому что 999999975.0 сохраняется во внутреннем представлении приблизительно и превращается в 1000000000.0.

Числовые типы часто используются для расчетов по формулам, для них определен широкий набор операций (см. главу Выражения).

Вычисления иногда могут приводить к некорректным результатам, то есть их невозможно представить числом. Например, корень из отрицательного числа или логарифм от нуля не определены. Для таких случаев вещественные типы умеют хранить специальную величину NaN (Not A Number). На самом деле таких величин — несколько разных видов, позволяющих, например, отличить плюс бесконечность от минус бесконечности. MQL5 предоставляет специальную функцию *MathIsValidNumber*, которая проверяет, является значение *double* числом или одним из NaN.

2.2.3 Символьные типы

Символьные типы предназначены для хранения отдельных символов (букв), из которых формируются строки (см. раздел Строки). MQL5 имеет 4 символьных типа: два размером 1 байт (*char*, *uchar*) и два размером 2 байта (*short*, *ushort*). Типы с приставкой 'u' являются беззнаковыми.

Символьные типы являются по сути целочисленными, так как хранят целочисленный код символа из соответствующей таблицы символов: в случае *char* это таблица ASCII-символов (коды от 0 до 127), в случае *uchar* — расширенная ASCII (коды от 0 до 255), а в случае *short/ushort* — таблица Unicode (до 65535 символов в беззнаковом варианте). Если вам интересно, ASCII расшифровывается как American Standard Code for Information Interchange.

Для строк MQL5 используются 2-байтовые символы *ushort*. 1-байтовые *uchar* обычно применяются для интеграции с внешними программами при передаче массивов произвольных данных, которые упаковываются и распаковываются в другие типы согласно прикладным протоколам (например, для подключения к криптовалютной бирже).

Константы символов записываются как буквы в обрамлении одинарных кавычек. Но также допустимо использовать рассмотренную выше (см. раздел Целые числа) целочисленную нотацию. При этом целое число должно быть в диапазоне значений для 1- или 2-байтового формата.

Дополнительно существует возможность использовать нотацию управляющих последовательностей. В ней в качестве первого символа используется обратная косая черта ('\'), после которой идет один из предопределенных управляющих символов и/или числовой код. MQL5 поддерживает следующие управляющие последовательности:

- \n — новая строка
- \r — возврат каретки
- \t — табуляция
- \\ — обратная наклонная черта
- \" — двойная кавычка
- \' — одинарная кавычка
- \X или \x — префикс для последующего указания числового кода в шестнадцатеричном формате
- \0 — префикс для последующего указания числового кода в восьмеричном формате

Основные способы использования констант символьных типов приведены в скрипте *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
}
```

Переменные *a1* и *a2* получают значение символа 'a' (английская буква), двумя разными способами.

В переменную *b* производится попытка записать символ '£', но его код 163 находится вне диапазона *char* (127) и потому он "преобразуется" в -93 со знаком (компилятор выдает предупреждение). Идущие следом переменные типов *uchar* (c) и *short* (d) нормально воспринимают данный код.

Остальные переменные инициализированы с помощью управляющих последовательностей.

Символы можно обрабатывать теми же операциями, что и целые числа (см. главу Выражения).

2.2.4 Строковый тип

Строковый тип предназначен для хранения текстовой информации и обозначается ключевым словом *string*. Строка представляет собой последовательность символов типа *ushort* и поддерживает весь спектр Unicode (включая множество национальных алфавитов). В частности, названия финансовых инструментов и комментарии в торговых приказах — это строки.

В силу специфики строки, её размер является переменной величиной, равной удвоенной длине текста (количеству символов, умноженному на "ширину" одного символа — 2 байта) плюс еще один символ. Этот дополнительный символ отводится под так называемый "терминальный ноль" (символ с кодом 0), который обозначает конец строки. Кроме того, MQL5 использует некоторое место для хранения служебной информации — ссылки на то место в памяти, где начинается строка.

> В MQL5 нельзя получить адрес строки или любой другой переменной, в отличие от C++. Прямой доступ к памяти в MQL5 запрещен.

Строковый литерал записывается в исходном коде как последовательность символов, заключенная в двойные кавычки. Например, "EURUSD" или "$". Следует различать строки из одного символа (как "$") и те же самые одиночные символы ('$') — это разные типы данных.

Пустая строка выглядит так — "". С учетом неявного завершающего нуля на неё тратится 2 байта (не считая служебной информации).

Если необходимо внутри строки использовать символ двойной кавычки, он предваряется символом обратной косой черты, превращаясь в управляющую последовательность — "Нажмите кнопку \"OK\"".

Примеры инициализации строк приведены в скрипте *MQL5/Scripts/MQL5Book/p2/TypeString.mq5*.

```
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";
}
```

В переменную *h* помещается строка "Hello".

В переменную *b* записывается текст, содержащий двойные кавычки.

Переменная *z* инициализируется пустой строкой. Это, в принципе, эквивалентно описанию *z* без инициализации, но здесь существуют нюансы. Чуть позже в разделе про Инициализацию переменных мы узнаем, что строки без инициализации получают специальное значение NULL (в отличие от "", для которой, как было сказано выше, выделяется память под "терминальный ноль"). Такое различие сказывается при выполнении операции сравнения строк и некоторых других. Мы затронем все такие особенности по мере продвижения по материалу.

Переменная *t* получит текст, который при выводе в журнал с помощью функции *Print* или отображении другими способами, будет разделен на 2 строки.

Строка "123", записанная в переменную *n*, не является числом, хоть и выглядит таковым. В MQL5 имеется несколько функций для преобразования текста в числа и обратно (см. раздел Преобразование данных). Кроме того, существует отдельный набор функций для работы со строками.

Длинные литералы могут быть записаны для удобства на нескольких строчках, как для переменной *m*. Общее правило такое: все литералы вплоть до точки с запятой, которая обозначает конец описания переменной, склеиваются компилятором. Самое главное при таком форматировании — не забыть добавить разделительный пробел внутри каждого фрагмента строки, если он нужен (в частности, для разделения слов сообщения, как в приведенном примере).

Для строк определена операция суммирования или конкатенации, обозначаемая символом '+', — мы поговорим об этом в главе про выражения (см. Арифметические операции).

Символы строки можно прочитать по отдельности, обращаясь к ним как к элементам массива (см. Использование массивов): если *s* — строка, то *s[i]* — код i-го символа в ней, типа *ushort*.

2.2.5 Логический тип

Логический тип предназначен для хранения признаков, имеющих только 2 возможных состояния: "включен"/"выключен". Их интерфейсным аналогом являются опции в диалогах настройки

многих программ (MetaTrader 5, в том числе): каждый флаг может быть либо включен, либо выключен. Проверка состояния таких признаков позволяет ветвить логику выполнения программ, отсюда и название типа.

Логический тип определен в MQL5 под ключевым словом *bool* и занимает в памяти 1 байт. Для типа зарезервированы две константы: *true* ("истина") и *false* ("ложь"). Кроме того, допустимы ситуации (и этим часто пользуются программисты), когда *bool* является результатом вычислений с целыми и вещественными числами, при этом значение 0 интерпретируется как *false*, а любые другие — как *true*.

Обратная интерпретация значения типа *bool* как числа при вычислениях также поддерживается: *true* считается равным 1, а *false* — 0.

Примеры переменных логического типа есть в файле *MQL5/Scripts/MQL5Book/p2/TypeBool.mq5*.

```
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
}
```

Для логического типа предусмотрен набор специальных логических операций (см. Логические операции и Операции сравнения).

2.2.6 Дата и время

MQL5 предоставляет специальный тип для хранения времени *datetime*. Как следует из названия, значения *datetime* включают и дату, и время, но при необходимости могут содержать только дату или только время внутри дня.

Значения этого типа могут использоваться в программах для отслеживания событий, таких как расписание торговых сессий, выход новостей, таймауты для временного отключения торговли советника после неудачных сделок.

Размер *datetime* в памяти — 8 байт. Внутреннее представление данных полностью идентично типу *ulong*, поскольку внутри хранится количество секунд, прошедшее с 1 января 1970 года. Максимальная поддерживаемая дата — 31 декабря 3000 года.

Константы *datetime* записываются в виде литеральной строки, заключенной в одинарные кавычки, перед которой ставится символ 'D'. Внутри строки выделено 6 полей с числами для всех компонент даты и времени в следующих форматах:

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

Здесь YYYY — год, MM — месяц, DD — день, HH — часы, mm — минуты, ss — секунды. Либо дату, либо время можно опустить. Также можно не указывать секунды или минуты с секундами.

Для максимально разрешенного значения даты в MQL5 заведена особая константа DATETIME_MAX, равная целочисленному значению 0x793406fff, что и соответствует D"3000.12.31 23:59:59".

Примеры записи значений типа *datetime* представлены в файле *MQL5/Scripts/MQL5Book/p2/TypeDateTime.mq5*.

```
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
}
```

Первые четыре переменных вызывают предупреждение компилятора о некорректной дате. В случае *blank* литерал полностью пустой. В переменной *noday* пропущен день. В обоих случаях компилятор подставляет в константу дату компиляции. Переменные *feb30* и *mon22* содержат некорректные номера дня и месяца. Компилятор их автоматически исправляет, перенося переполнение в старшее поле (30 февраля превращается во 2-е марта, а 22-й месяц — в 10-й месяц следующего года). Однако от предупреждений, как всегда, рекомендуется избавляться.

Переменная *dt0* демонстрирует инициализацию значения *datetime* целым числом.

Тип *datetime* поддерживает набор операций, свойственных целым числам (см. главу Выражения). Это, например, позволяет прибавлять к времени заданное количество секунд (получая некий момент в будущем) или вычислять разницу между датами.

2.2.7 Цвет

MQL5 имеет специальный тип для работы с цветом. Это дает возможность раскрашивать графические объекты.

Для обозначения типа используется ключевое слово *color*. Под значение типа *color* выделяется 4 байта памяти. Внутреннее представление — это целое беззнаковое число, содержащее цвет в формате RGB (от Red, Green, Blue), то есть с раздельными уровнями интенсивности для красного, зеленого и синего цветов. Смешение этих трех компонент позволяет получить любой видимый оттенок. Зеленый с красным дадут желтый, красный с синим — фиолетовый, и так далее.

Под каждую компоненту выделен 1 байт, то есть она может принимать значения от 0 до 255. Например, три нуля во всех компонентах дают черный цвет, а три максимальных величины 255 смешиваются в белый.

Если представить *color* как *uint* в шестнадцатеричной системе, то распределение цветов выглядит следующим образом: 0x00BBGGRR, где RR, GG, BB — однобайтовые беззнаковые целые числа.

Для удобства пользователя MQL5 поддерживает особый вид литералов для записи цветовых констант. Литерал представляет собой тройку чисел, разделенных запятыми и заключенную в одинарные кавычки. Перед литералом ставится символ 'C'. Например, C'0,128,255' означает цвет с 0 для красной составляющей, 128 — для зеленой и 255 — для синей. Можно использовать и шестнадцатеричную запись чисел: C'0x00,0x80,0xFF'.

Помимо этого в MQL5 в виде служебных слов встроен длинный перечень предопределенных цветовых оттенков — все они начинаются с префикса *clr*. Например, clrMagenta, clrLightCyan, clrYellow. В нем, разумеется, есть и основные: clrRed, clrGreen, clrBlue. Полный перечень можно найти в Справке MetaEditor.

Вот несколько примеров задания цветов (доступны также в файле *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 Перечисления

Перечисления — это группа встроенных в MQL5 типов, каждый из которых содержит набор именованных констант для описания родственных прикладных понятий или свойств. Данные константы называются также элементами перечисления.

Например, перечисление ENUM_DAY_OF_WEEK содержит константы для всех дней недели:

Идентификатор	Описание	Значение
SUNDAY	Воскресенье	0
MONDAY	Понедельник	1
TUESDAY	Вторник	2
WEDNESDAY	Среда	3
THURSDAY	Четверг	4
FRIDAY	Пятница	5
SATURDAY	Суббота	6

А перечисление ENUM_ORDER_TYPE описывает все поддерживаемые типы ордеров в MetaTrader 5:

Идентификатор	Описание	Значение
ORDER_TYPE_BUY	Рыночный ордер на покупку	0
ORDER_TYPE_SELL	Рыночный ордер на продажу	1
ORDER_TYPE_BUY_LIMIT	Отложенный ордер Buy Limit	2
ORDER_TYPE_SELL_LIMIT	Отложенный ордер Sell Limit	3
ORDER_TYPE_BUY_STOP	Отложенный ордер Buy Stop	4
ORDER_TYPE_SELL_STOP	Отложенный ордер Sell Stop	5
ORDER_TYPE_BUY_STOP_LIMIT	По достижении цены ордера выставляется отложенный ордер Buy Limit по цене StopLimit	6
ORDER_TYPE_SELL_STOP_LIMIT	По достижении цены ордера выставляется отложенный ордер Sell Limit по цене StopLimit	7
ORDER_TYPE_CLOSE_BY	Ордер на закрытие позиции встречной позицией	8

Всего имеется несколько десятков разных перечислений. Их названия начинаются с префикса "ENUM_". Мы будем изучать их по мере продвижения по соответствующим предметным областям.

Каждое перечисление является самостоятельным типом, но их внутреннее представление одинаковое — четырехбайтовое целое (*int*). Каждая из констант перечисления кодируется тем или иным числом, но программисту в большинстве случаев не нужно запоминать эти числа, потому что смысл использования перечислений как раз в том, чтобы заменить внутреннее представление на понятные идентификаторы.

Компилятор следит за тем, чтобы значением перечисления всегда была одна из предопределенных констант. В противном случае возникнет предупреждение или ошибка компиляции (в зависимости от контекста, см. пример).

Вот как выглядит "внутри" перечисление ENUM_DAY_OF_WEEK (скрипт *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
}
```

Все константы дней недели кодируются числами от 0 до 6, причем отсчет начинается с воскресенья. В принципе, константы не обязательно должны иметь последовательные номера или начинаться с 0. Существуют перечисления, где это не так.

Обратите внимание, что одни и те же константы могут означать в разных типах перечислений разные вещи. Например, для ордеров ORDER_TYPE_BUY и ORDER_TYPE_SELL в перечислении ENUM_ORDER_TYPE используются те же значения (0 и 1), что и для дней недели SUNDAY и MONDAY в перечислении ENUM_DAY_OF_WEEK.

При копировании значения из простой целочисленной переменной *i* в переменную перечисления *x* компилятор выдает предупреждение, так как на стадии выполнения программы в переменной *i* может оказаться величина, отличная от разрешенных констант.

В переменную *y* мы записываем число 1, которое соответствует MONDAY, и компилятор считает это корректной операцией.

Попытка записать константу одного перечисления в переменную другого перечисления (как MONDAY для переменной *type* в вышеприведенном примере) может вызвать предупреждение о неявном преобразовании одного типа в другой. Происходит это, если записываемая константа имеет такое же значение, что и один из элементов целевого перечисления. Иными словами в каждом из двух перечислений есть свой собственный элемент с соответствующим значением. Тогда компилятор делает неявное преобразование автоматически за программиста, но с помощью предупреждения просит обратить внимание и проверить, так ли все задумывалось: то, что MONDAY будет заменен на ORDER_TYPE_SELL действительно выглядит странно, но здесь мы сделали это намеренно для демонстрации.

В том случае, если копируемый элемент не совпадает по значению ни с одним элементом другого перечисления, генерируется ошибка компиляции, так как неявная конвертация невозможна (например, при записи ORDER_TYPE_CLOSE_BY в переменную *day*).

Закомментированная строка с переменной *z* также вызывает ошибку компиляции, поскольку значение 10 не принадлежит ENUM_DAY_OF_WEEK. Если программист уверен, что в каком-то экзотическом случае все же требуется записать в переменную типа перечисления произвольное значение, он может воспользоваться явным приведением типов.

О явных и неявных приведениях типов рассказывается в разделе Приведение типов.

MQL5 позволяет программисту объявлять собственные прикладные перечисления с помощью ключевого слова *enum* — данная возможность описана в следующем разделе Пользовательские перечисления (*enum*).

2.2.9 Пользовательские перечисления

Пользовательские перечисления конструктивно основаны на типе *int* и принцип их использования полностью совпадает с тем, что было рассказано в предыдущем разделе о встроенных перечислениях. Поэтому мы описываем пользовательские перечисления здесь, хотя они и не являются, строго говоря, встроенными.

Для того чтобы описать собственное перечисление в коде MQL5 используется ключевое слово *enum*. Простейшая форма описания имеет следующий вид:

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

Такое определение регистрирует в программе тип перечисления с именем *name* и указанными в фигурных скобках, через запятую, элементами (их количество ограничено лишь максимальным значением *int*, что для практических задач можно расценивать как отсутствие ограничений). Идентификаторы *element1*, *element2*, *element3* можно далее использовать в программе в пределах того контекста, где произведено определение: глобально (т.е. снаружи всех функций) или внутри какой-либо функции (см. раздел Контекст, область видимости и время жизни переменных).

Обратите внимание на точку с запятой после закрывающей скобки. Она нужна, потому что описание перечисления — это отдельная инструкция, а после любой инструкции языка MQL5 необходимо ставить точку с запятой.

Идентификаторы получают по умолчанию константные значения, начиная с 0, каждый следующий — на 1 больше предыдущего. При необходимости программист может задать для каждого элемента конкретное значение, после знака '=' справа от идентификатора. Например, вышеприведенная запись эквивалентна следующей:

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

В качестве значений допускается указывать только константы или выражения, которые компилятор способен вычислить на стадии компиляции (подробнее об этом — в примере ниже).

Если значения заданы не для всех элементов, то пропущенные значения автоматически вычисляются на основе ближайших известных (предыдущих) путем увеличения на 1. Например,

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

Здесь первые два элемента получают значения 1 и 2 (вычислено), а начиная с третьего — 10 (указано явно), 11 и 12 (два последних вычислены от 10).

В скрипте *TypeUserEnum.mq5* приведено несколько примеров описания пользовательских перечислений.

```
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;
}
```

Перечисление INTERNAL демонстрирует возможность описать его внутри функции и тем самым ограничить область видимости/доступности этого типа, что полезно для исключения конфликтов имен.

Перечисление RISK показывает, что можно назначать элементам отрицательные значения. Закомментированный элемент OFF не может быть описан из-за попытки его инициализации неконстантным выражением: в данном случае там указана переменная *zero*, значение которой компилятор не способен вычислить.

В перечислении INCOME элемент ENORMOUS успешно инициализируется значением из элемента MILLION другого перечисления, определенного выше. Перечисления создаются в момент компиляции и потому доступны в выражениях инициализации.

Перечисление с MILLION не имеет имени — такие перечисления называются анонимными. Их основное применение — декларация констант. Однако чаще для констант все же используются именованные перечисления, так как они позволяют группировать элементы по смыслу.

Поскольку в примере определено 2 перечисления с элементами, имеющими одинаковые имена, указание идентификатора LOW при объявлении переменной *x* приводит к ошибке компиляции "неоднозначный доступ" ("ambiguous access") — не понятно, элемент какого перечисления имеется в виду. Следует учесть, что идентификаторы могут иметь (и имеют в данном случае) разные значения.

Для разрешения этой проблемы существует специальный оператор контекста — два двоеточия "::". С помощью него формируется полный идентификатор элемента языка, в нашем случае элемента перечисления: сперва указывается имя перечисления, далее оператор "::" и затем идентификатор элемента. Например, RISK::LOW и INCOME::LOW. Позднее мы познакомимся со всеми операторами в соответствующем разделе.

2.2.10 Тип void

Тип *void* — особенный тип. Он обозначает пустоту (отсутствие типа) и не занимает в памяти место. Он используется только для описания функций, которые не возвращают значений или не имеют параметров. Мы познакомились с примером подобной функции — *OnStart* — в скрипте *HelloChart*, в первой части. Более подробно об этом будет рассказано в разделе Функции.

Тип *void* невозможно использовать для описания переменных, однако он является базовым типом при описании указателей на произвольные объекты классов. Данная возможность описывается в Части 3, посвященной объектно-ориентированному программированию.

2.3 Переменные

В этой главе мы изучим базовые принципы работы с переменными в MQL5, ограничившись для начала встроенными типами данных. В частности, рассмотрим объявление и определение переменных, особенности инициализации в зависимости от контекста, время жизни и основные модификаторы, меняющие свойства переменных. Позднее, опираясь на эти знания, мы расширим возможности переменных, благодаря новым пользовательским типам (объединениям, пользовательским перечислениям, алиасам), классам, указателям и ссылкам.

Переменные в MQL5 предоставляют механизм для хранения данных различных типов, играя важную роль в организации логики программ и работы с рыночной информацией. Раздел включает в себя следующие подразделы:

Объявление и определение переменных:

- Объявление переменных — это шаг создания их в программе. В этом разделе мы рассматриваем способы объявления и определения переменных, включая указание их типов.

Контекст, область видимости и время жизни переменных:

- Переменные могут существовать в различных контекстах и областях видимости, что влияет на их доступность и время жизни. Этот подраздел углубляется в эти аспекты, помогая понять, как переменные взаимодействуют с кодом.

Инициализация:

- Инициализация переменных — это присвоение им начальных значений. Мы изучаем методы инициализации, что помогает избежать неопределенного поведения программы.

Статические переменные:

- Статические переменные сохраняют свое значение между вызовами функций. В этом разделе рассказывается о том, как использовать статические переменные для хранения информации между различными исполнениями кода.

Переменные-константы:

- Переменные-константы представляют собой значения, которые не изменяются в ходе выполнения программы. Этот раздел подробно рассматривает их использование и особенности.

Входные переменные:

- Входные переменные применяются в торговых роботах для настройки параметров стратегии. Мы изучаем, как использовать их для создания гибких и настраиваемых торговых систем.

Внешние переменные:

- Внешние переменные предоставляют пользователю возможность взаимодействия с программой, изменяя их значения без необходимости внесения изменений в код. Этот раздел раскрывает принципы работы внешних переменных.

2.3.1 Объявление и определение переменных

Переменная — это именованная ячейка памяти для хранения данных конкретного типа. Для того чтобы программа могла работать с переменной, программист должен объявить и/или определить её в исходном коде. Термины "объявление" и "определение" означают, в общем случае, разные вещи в отношении элементов программ, но для переменных они почти всегда совпадают. Эти тонкости будут освещены, когда мы познакомимся с функциями, классами, и особыми — внешними переменными. Здесь же будем использовать оба термина взаимозаменяемо, наравне с обобщающим — "описанием".

В первом приближении можно сказать, что объявление содержит описание элемента программы со всеми его атрибутами, необходимыми для использования в программе. Определение же содержит конкретную реализацию этого элемента, отвечающую объявлению.

Объявления позволяют компилятору связать друг с другом все элементы программы. А на основании определений компилятор генерирует исполняемый код.

В случае переменных их объявление практически всегда выступает и в качестве определения, поскольку обеспечивает выделение памяти и интерпретацию содержимого в соответствии с типом (а это и есть реализация переменной). Исключение составляют лишь объявления переменных со словом extern (об этом подробнее в разделе Внешние переменные).

Только после описания переменной можно с помощью специальных инструкций записывать в неё значения, считывать их, ссылаться по имени на переменную для передачи её из одной части программы в другую.

В простейшем случае инструкция, описывающая переменную, выглядит следующим образом:

```
type name;
```

Здесь имя *name* должно отвечать требованиям составления идентификаторов. В качестве типа *type* может быть указан любой из встроенных типов, которые мы рассмотрели в предыдущем

разделе, или другие пользовательские типы — как их создавать, мы изучим чуть позднее. Например, целочисленная переменная *i* объявляется так:

```
int i;
```

При необходимости можно описать сразу несколько однотипных переменных. В этом случае их имена указываются в инструкции через запятую.

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

Важным фактором является место в программе, где расположена инструкция с описанием переменной. Это влияет на время жизни переменной и её доступность из разных частей программы.

2.3.2 Контекст, область видимости и время жизни переменных

Язык MQL5 относится к числу языков программирования, которые используют фигурные скобки для группирования инструкций в блоки кода.

Напомним, что программа состоит из блоков с инструкциями, и один блок должен существовать обязательно. В примерах скриптов из первой Части мы видели функцию *OnStart*. Тело данной функции (текст внутри фигурных скобок после имени функции) как раз представляло собой такой обязательный блок кода.

Внутри каждого блока формируется локальный контекст — область, которая ограничивает видимость и время жизни переменных, описанных внутри неё. До сих пор мы встречали только примеры, где фигурные скобки определяли тело функций. Однако они также могут использоваться для формирования составных операторов, в синтаксисе описания классов и пространств имен. Все эти способы тоже определяют области видимости и будут рассмотрены в соответствующих разделах. А мы на данном этапе ограничимся одним видом локальных блоков — внутри функций.

Кроме локальных областей в любой программе существует также и один глобальный контекст — область с определениями переменных, функций и других сущностей, которые сделаны вне других блоков.

Если взять простой скрипт, в котором Мастер MQL создал единственную пустую функцию *OnStart*, то в нем будет только 2 области: глобальная и локальная (внутри тела функции *OnStart*, хоть она и пуста). Это иллюстрирует следующий скрипт с помощью комментариев.

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

Обратите внимание, что глобальная область простирается везде кроме функции *OnStart* (и до неё, и после). Вообще говоря, она включает в себя всё за пределами любых функций (если бы их было много), но в данном скрипте кроме *OnStart* ничего нет.

Мы можем описать в начале файла переменные (например, *i, j, k*), и они станут глобальными.

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

Глобальные переменные создаются сразу после запуска MQL-программы в терминале и существуют на протяжении всего периода, пока программа выполняется.

Программист может записывать и считывать содержимое глобальных переменных из любого места программы.

В принципе рекомендуется описывать глобальные переменные именно в начале, однако это не обязательно. Если перенести декларацию ниже всей функции *OnStart*, ничего принципиально не изменится. Просто другим программистам будет сложно сразу понять суть кода с переменными, до определений которых еще надо добраться.

Интересно отметить, что сама функция *OnStart* тоже объявлена в глобальном контексте. Если добавить еще одну функцию, то она тоже будет объявлена в глобальном контексте. Вспомним, как мы создали функцию *Greeting* в первой части и вызывали её из функции *OnStart*. Это следствие того, что имя функции и способ обращения к ней (как её выполнять) известны во всем исходном коде. Некоторые нюансы к этому правилу добавляют пространства имен, но их мы изучим позднее.

Локальная область внутри каждой функции принадлежит только ей: внутри *OnStart* — одна локальная область, внутри *Greeting* — другая, её собственная, отличная и от локальной области *OnStart*, и от глобальной.

Переменные, описанные в теле функции, называются локальными. Они создаются согласно их описаниям в тот момент, когда в процессе выполнения программы происходит вызов соответствующей функции. Локальные переменные можно использовать только внутри содержащего их блока. Снаружи они не видны и не доступны. При выходе из функции локальные переменные уничтожаются.

Пример описания локальных переменных *x, y, z* внутри функции *OnStart*:

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

Следует отметить, что пары фигурных скобок могут использоваться не только в определении функции и других инструкций, но и сами по себе — для формирования внутреннего блока кода. Уровень вложенности блоков не ограничен.

Обычно вложенные блоки добавляют, чтобы до минимума ограничить видимость переменных, которые применяются в логически обособленном, небольшом участке кода (если он не оформлен по тем или иным причинам функцией). Это позволяет снизить вероятность ошибочной

модификации переменной там, где это не предусматривалось, или нежелательных побочных эффектов из-за попытки приспособить одну и ту же переменную под разные нужды (это плохая практика).

Ниже приведен пример функции, где уровень вложенности блоков равен 2 (если считать блок с телом функции первым уровнем), и таких блоков создается 2 — они будут выполнены последовательно.

```
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
}
```

Внутри обоих блоков описана переменная *p*, которая в них используется для разных целей. Фактически, это две разных переменных, хотя они и имеют одно и то же имя, видимое внутри каждого блока.

Если бы переменная была вынесена в начальный список локальных переменных функции, то она могла содержать некое остаточное значение после выхода из первого блока и тем самым нарушить работу второго блока. Более того, программист мог бы случайно задействовать *p* еще для чего-то в самом начале функции, и тогда побочные эффекты могли бы произойти и в первом блоке.

Вне любого из двух вложенных блоков переменная *p* неизвестна и потому попытка обратиться к ней из общего блока функции приводит к ошибке компиляции ("неизвестный идентификатор" — "undeclared identifier").

Отметим также, что переменная может быть описана не в самом начале блока, а в середине или даже ближе к концу. Тогда она определена не во всем блоке, а только ниже своего определения. Следовательно, при обращении к переменной выше её описания, возникнет такая же ошибка.

Таким образом, область видимости переменной может отличаться от контекста (блока целиком).

Оба варианта проблемы проиллюстрированы в примере: попробуйте включить любую из строк с инструкциями $p = x$ и $y = p$ и откомпилировать исходный код.

Память под все локальные переменные функции выделяется, как только управление передается внутрь функции. Но на этом их создание не заканчивается. Далее осуществляется их

инициализация (установка начальных значений), которая может определяться явным образом программистом или неявно по умолчаниям компилятора. При этом существенную роль играет контекст, в котором описаны переменные.

2.3.3 Инициализация

При описании переменных существует возможность задать начальное значение — оно указывается после имени переменной и символа '=', и должно соответствовать типу переменной или приводиться к нему (о приведении типов — в соответствующем разделе).

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

Здесь *i* и *k* проинициализированы явно, а *j* — нет.

В качестве начального значения можно указать не только константу (литерал соответствующего типа), но и выражение — своего рода формулу для расчета. Про выражения мы подробно поговорим отдельно. Пока лишь приведем простой пример:

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

Здесь переменная *j* получает такое же значение, что и переменная *i*, а переменная *k* — сумму *i* и *j*. Строго говоря, во всех трех случаях мы здесь видим выражения, но константа (3) — это особый вырожденный вариант выражения. Во втором случае выражением является единственное имя переменной, т.е. результатом выражения будет значение этой переменной без преобразований. В третьем случае в выражении происходит обращение к двум переменным *i* и *j*, и над их значениями выполняется операция сложения, после чего результат попадает в переменную *k*.

Поскольку инструкция с описанием нескольких переменных обрабатывается слева направо, компилятор уже знает имена предыдущих переменных при разборе очередного описания.

Обычно в программе много инструкций с описаниями переменных. Они считываются компилятором естественным образом сверху вниз, и в более поздних инициализациях могут использоваться имена из более ранних описаний. Вот те же переменные, описанные двумя раздельными инструкциями.

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

Переменные без явной инициализации тоже получают некие начальные значения, но они зависят от места описания переменной — её контекста.

Локальные переменные при отсутствии инициализации получают в момент создания случайные значения: компилятор просто выделяет под них память согласно размеру типа, а что будет находиться по конкретному адресу — неизвестно (разные участки памяти компьютера часто повторно выделяются для использования в разных программах, после того как стали не нужны другим программам, выполнявшимся ранее).

Обычно предполагается, что в локальные переменные без инициализации будет произведена запись рабочих значений когда-то позднее в коде алгоритма, например, с помощью операции присваивания, о которой будет рассказано далее. Она похожа по синтаксису на инициализацию, так как тоже использует знак равно '=' для переноса значения из той "конструкции", что стоит справа (это может быть константа, переменная, выражение, вызов функции), в переменную, которая упомянута слева. Слева от '=' может находиться только переменная.

Программист должен следить за тем, чтобы чтение из неинициализированной переменной происходило только после присвоения ей осмысленного значения. Компилятор выдает предупреждение, если это не так ("возможное использование неинициализированной переменной" — "possible use of uninitialized variable").

С глобальными переменными все иначе.

Примером глобальной переменной является входной параметр *GreetingHour* скрипта *GoodTime2* из первой Части. То, что переменная была описана с ключевым словом *input*, не влияет на её прочие свойства как переменной. Мы могли бы исключить её инициализацию и описать следующим образом:

```
input uint GreetingHour;
```

Это ничего не поменяло бы в программе, потому что глобальные переменные неявно инициализируются компилятором с помощью нуля в отсутствии явной инициализации (а у нас ранее была явная инициализация тоже нулем).

Вне зависимости от типа переменной неявная инициализация всегда производится значением, эквивалентным нулю. Например, переменной типа *bool* будет по умолчанию установлено значение *false*, а переменной типа *datetime* — D'1970.01.01 00:00:00'. Для строк существует специальное значение NULL. Если хотите, это "еще более пустая строка", чем пустые кавычки "", потому что под них всё же выделяется память, куда в самое начало помещается единственный терминальный нулевой символ.

Кроме локальных и глобальных существует еще один вид переменных — статические. Компилятор также неявно инициализирует их нулем, если программист не прописал явно начальное значение. О них мы расскажем в следующем разделе.

Создадим новый скрипт *VariableScopes.mq5* с примерами описания локальных и глобальных переменных (*MQL5/Scripts/MQL5Book/VariableScopes.mq5*).

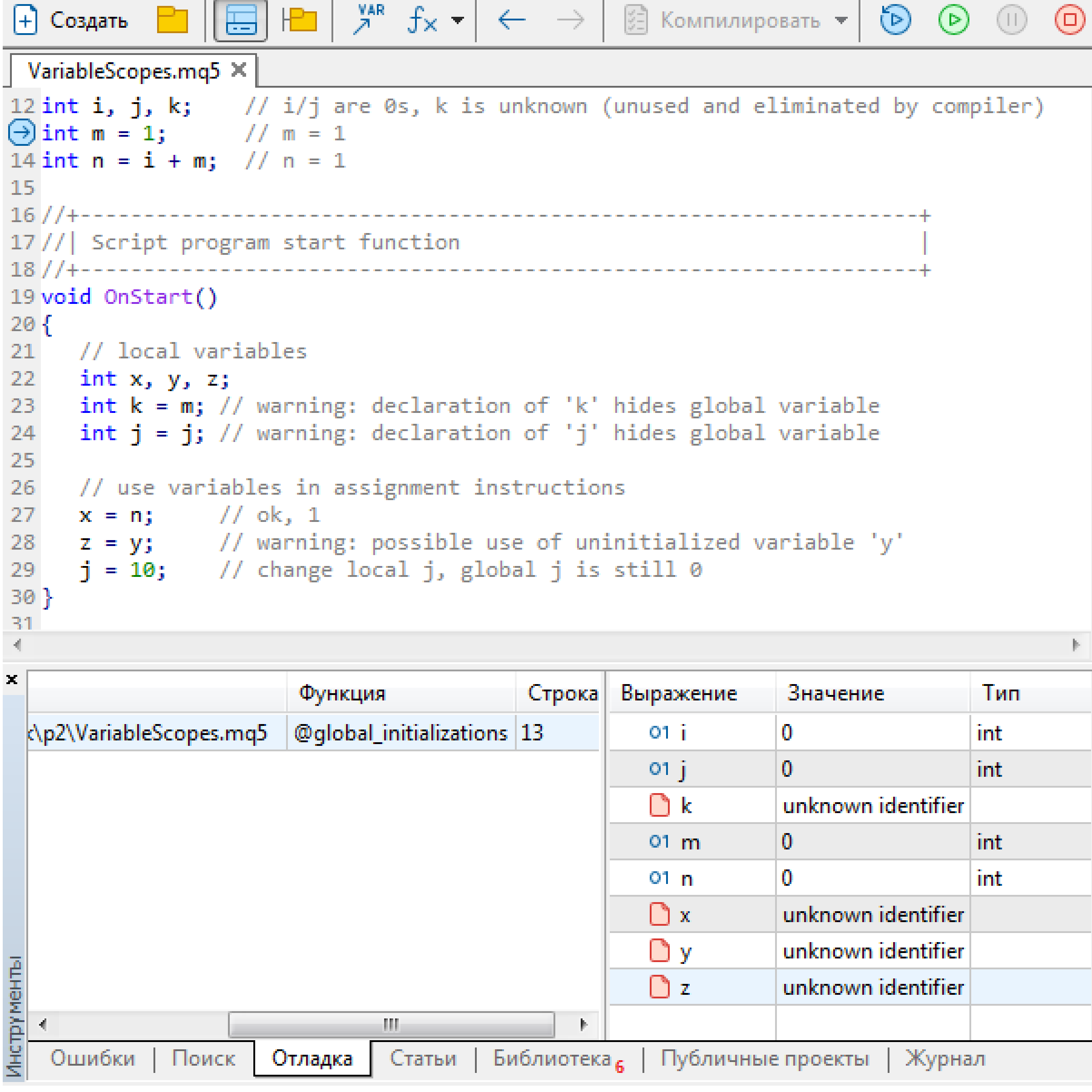
```
// global variables
int i, j, k;    // all are 0s
int m = 1;      // m = 1                (place breakpoint on this line)
int n = i + m;  // n = 1
void OnStart()
{
  // local variables
  int x, y, z;
  int k = m; // warning: declaration of 'k' hides global variable
  int j = j; // warning: declaration of 'j' hides global variable
  // use variables in assignment instructions
  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
```

Напомним, что в момент запуска MQL-программы терминал сначала инициализирует все глобальные переменные и затем вызывает функцию, являющуюся точкой входа для соответствующего типа программ, в данном случае, *OnStart* для скриптов.

Здесь глобальными являются переменные *i*, *j*, *k*, *m*, *n*, т.к. они описаны вне функции (в данном случае у нас только одна функция — обязательная для скриптов функция *OnStart*). *i*, *j*, *k* неявно получают значение 0. *m* и *n* содержат 1.

Вы можете запустить скрипт в режиме отладки в пошаговом режиме и убедиться, что значения переменных меняются именно так. Для этого нужно предварительно установить точку остановки на строку с инициализацией одной из глобальных переменных, например *m*. Поставьте текстовый курсор на эту строку и выполните команду *Отладка -> Переключить точку останова* (F9), и строка будет подсвечена в левом поле синим значком, сигнализирующим, что здесь выполнение программы остановится, если она будет работать под отладчиком.

Далее следует, собственно, запустить программу под отладчиком, для чего выполните команду *Отладка -> Начать на реальных данных* (F5). В терминале в этот момент открывается новый график, на котором начинает выполняться данный скрипт (в правом верхнем углу — надпись "VariableScopes (Отладка)"), но тут же приостанавливается, и мы попадаем обратно в MetaEditor. В нем мы должны увидеть примерно следующую картину.



Пошаговая отладка и просмотр переменных в MetaEditor

Строка с точкой остановки теперь помечена значком со стрелочкой — это текущая инструкция, которую программа готовится выполнить, но еще не выполнила. Внизу слева показан текущий стек программы, состоящий пока только из одной записи: @global_initializations. Внизу справа можно ввести выражения для наблюдения за их значениями в реальном времени. Нас интересуют значения переменных, так что введем последовательно *i, j, k, m, n, x, y, z* (каждую в отдельной строке).

В дальнейшем вы увидите, что MetaEditor автоматически добавляет для просмотра переменные из текущего контекста (например, все локальные переменные и входные параметры функции, когда выполняются инструкции внутри функции). Но сейчас мы добавим *x*, *y*, *z* вручную и заранее, просто чтобы показать, что они не определены вне функции.

Обратите внимание, что для локальных переменных вместо значения написано "Неизвестный идентификатор" ("Unknown identifier"), потому что блок функции *OnStart*, где они находятся, еще не существует. Глобальные переменные *i* и *j* будут иметь сначала нулевые значения. Глобальная переменная *k* нигде не используется и потому исключена компилятором.

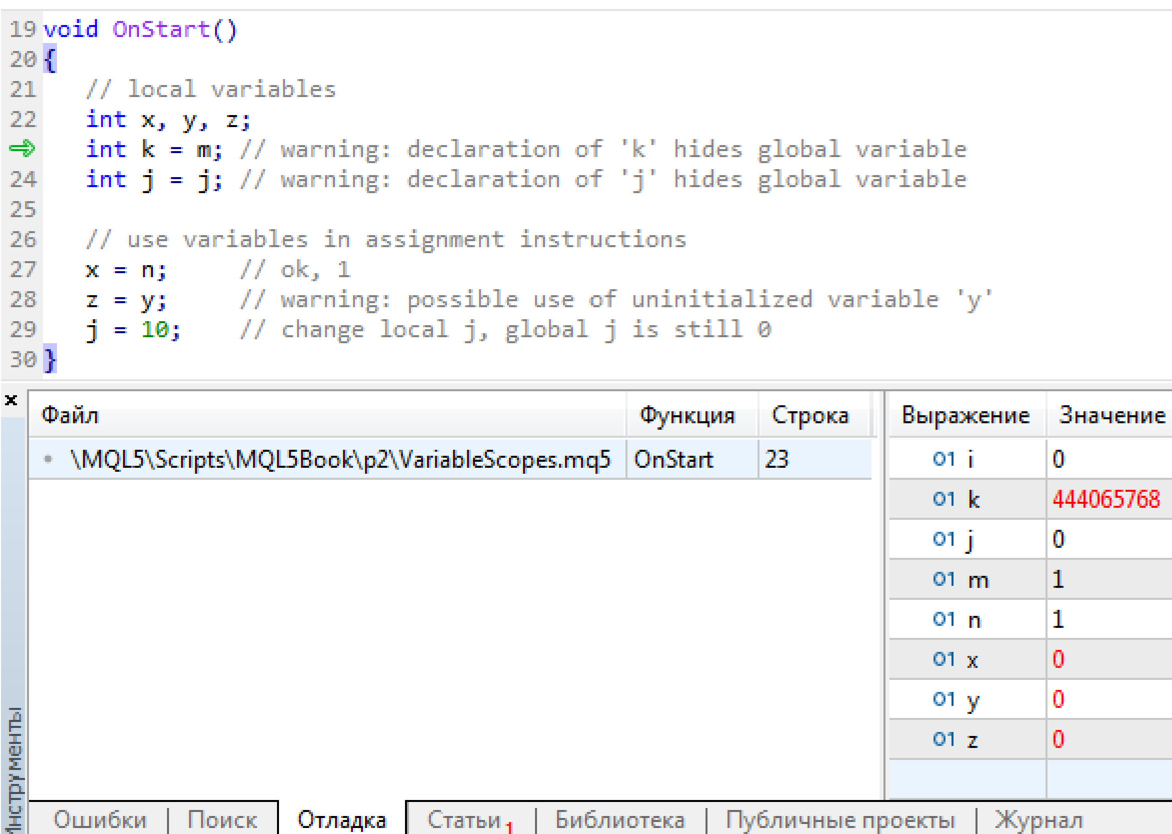
Если выполнить один шаг исполнения программы (выполнить инструкцию на текущей строке кода) с помощью команды *Шаг с заходом* (F11) или *Шаг с обходом* (F10), мы увидим как переменная *m* получит значение 1. Еще один шаг продолжит инициализацию уже для переменной *n*, и она тоже станет равной 1.

На этом описания глобальных переменных заканчиваются, а как мы знаем, после завершения инициализации глобальных переменных терминал вызывает функцию *OnStart*. В данном случае, чтобы зайти в функцию *OnStart* в пошаговом режиме нажмите еще раз F11 (или можно поставить другую точку останова в начале функции *OnStart*).

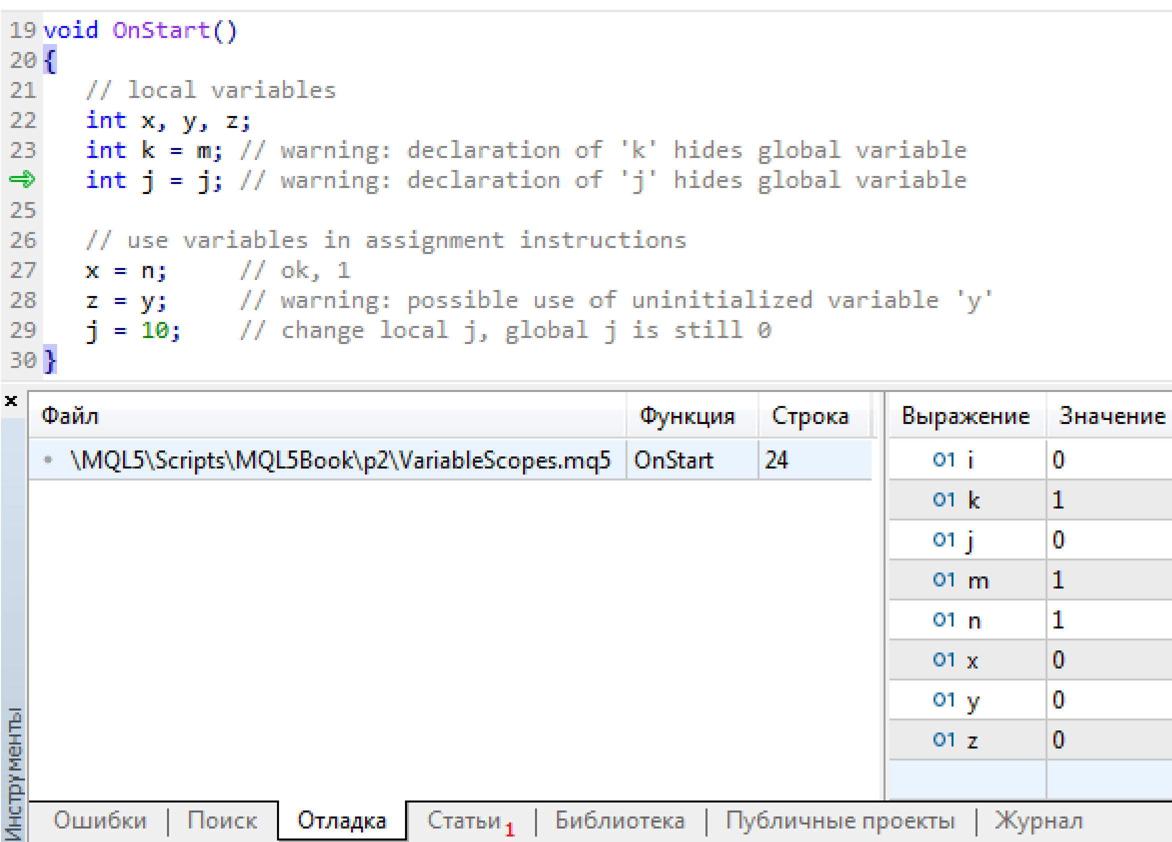
Инициализация локальных переменных происходит в тот момент, когда выполнение инструкций программы приходит в блок кода, где они определены. Поэтому переменные *x, y, z* создаются только после захода в функцию *OnStart*.

Когда отладчик окажется внутри функции *OnStart*, Вы сможете увидеть, если повезет, что в *x*, *y* и *z* действительно находятся изначально случайные значения. "Везение" здесь заключается в том, что эти случайные значения могут вполне оказаться нулевыми, и тогда их нельзя будет отличить от неявной инициализации нулем, которую компилятор делает для глобальных переменных. Если повторить запуск скрипта несколько раз, "мусор" в локальных переменных, скорее всего, будет разным и более наглядным. Они не инициализированы явно и потому их содержимое может оказаться любым.

На последовательности изображений ниже можно наблюдать эволюцию переменных с помощью пошагового режима отладчика. Текущая строка, ожидающая своего выполнения (но еще не выполненная), помечена зелёной стрелочкой на полях с нумерацией.



Пошаговая отладка и просмотр переменных в MetaEditor (строка 23)



Пошаговая отладка и просмотр переменных в MetaEditor (строка 24)

В коде далее демонстрируется, как эти переменные могли бы использоваться простейшим образом в операторах присваивания. Значение глобальной переменной *n* копируется в локальную *x* без проблем, потому что *n* была проиницилизирована. Однако в строке, где содержимое переменной *y* копируется в переменную *z*, возникает предупреждение компилятора, так как *y* является локальной, и в неё к этому моменту еще ничего не было записано (явной инициализации нет, как и прочих операторов, способных установить её значение).

Внутри функции допускается описывать переменные с теми же именами, которые уже использовались для глобальных переменных. Аналогичная ситуация может возникать во вложенных локальных блоках, если во внутреннем блоке создается переменная с именем, существующем во внешнем блоке. Однако такая практика не рекомендуется, поскольку может приводить к логическим ошибкам. Компилятор выдает в таких случаях предупреждения ("определение скрывает глобальную/локальную переменную" — "declaration hides global/local variable").

В результате подобного переопределения локальная переменная (как переменная *k* в примере выше) перекрывает внутри функции одноименную глобальную. Это две разных переменных, несмотря на одно и то же имя. Локальная *k* известна внутри *OnStart*, а глобальная *k* — везде кроме *OnStart*. Иными словами, любые операции с переменной *k* внутри блока будут влиять только на локальную переменную. Поэтому после выхода из функции *OnStart* (если бы это не была единственная и главная функция скрипта), мы обнаружили бы, что глобальная переменная *k* по-прежнему равна нулю.

Локальная переменная *j* не только перекрывает глобальную переменную *j*, но и инициализируется значением последней. В строке с описанием *j* внутри *OnStart* локальная версия *j* еще находится в процессе создания, когда начальное значение для неё считывается из глобальной версии *j*. После успешного определения локальной *j*, это имя перекрывает глобальную версию, и последующие изменения переменной *j* относятся именно к локальной версии.

В конце исходного кода закомментирована попытка объявить еще одну глобальную переменную — *bad*, в инициализации которой происходит обращение к значению переменной *x*. Эта строка вызывает ошибку компилятора, так как переменная *x* неизвестна вне функции *OnStart*, где она определена.

2.3.4 Статические переменные

Иногда бывает необходимо описать переменную внутри функции, но обеспечить её существование на протяжении всего времени выполнения программы. Например, мы хотим подсчитывать количество раз, сколько данная функция вызывалась.

Такая переменная не может быть локальной, потому что тогда она потеряет свою "долговременную память", поскольку будет создаваться каждый раз при вызове функции и удаляться при выходе из неё. Чисто технически её можно было бы описать глобально, но если переменная используется только в этой функции, такой подход неверен с точки зрения дизайна программ.

Во-первых, глобальную переменную можно по недосмотру изменить из любого места программы.

Во-вторых, представьте себе, в какой "зоопарк" переменных превратилась бы глобальная область программы, если по каждому поводу объявлять глобальную переменную. Вместо этого рекомендуется объявлять переменные в наименьшем по размеру блоке (если их несколько вложенных), в котором они используются.

Следовательно, счетчик выполнений функции следует описывать внутри самой функции. Здесь и приходит на помощь новый атрибут переменных — статичность.

Специальное ключевое слово (модификатор) *static*, поставленное перед типом переменной в её объявлении, позволяет увеличить её время жизни до всего периода выполнения программы, то есть делает её похожей на глобальные. Статическая переменная, как правило, определяется только локально, в одной из функций. Поэтому её видимость ограничена соответствующим блоком кода, как у обычной локальной переменной.

Статические переменные могут быть описаны и на глобальном уровне, но ничем не отличаются от обычных глобальных (по крайней мере, на момент написания книги). Это разнится с их поведением в C++: там их видимость ограничивается тем файлом, в котором они описаны. В MQL5 программа собирается на основе одного главный mq5-файла и, возможно, нескольких подключаемых (см. директиву *#include*), поэтому и статические, и обычные глобальные переменные доступны из всех исходных файлов программы.

Локальная статическая переменная создается единожды — в тот момент, когда программа первый раз входит в функцию, где эта переменная описана. А удалится такая переменная только при выгрузке программы. Если какая-либо функция ни разу не вызывалась, то локальные статические переменные, описанные в ней (если такие есть), так и не будут созданы.

В качестве примера модифицируем функцию *Greeting* из первой части книги таким образом, чтобы она выдавала разные приветствия при каждом обращении. Назовем новый скрипт *GoodTimes.mq5*.

Уберем входной параметр скрипта *GreetingHour* и параметр самой функции *Greeting*. Внутри функции *Greeting* опишем новую статическую переменную *counter* целого типа, с начальным значением 0. Напомним, что это именно инициализация, и она выполнится лишь однажды, потому что переменная статическая.

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

Поскольку мы теперь знаем модификатор *static*, имеет смысл применить его и для массива *messages*. Дело в том, что ранее он был объявлен локальным, и при многократном вызове функции *Greeting* создавался бы каждый раз заново (и удалялся бы при выходе). Это не эффективно.

Напомним, что массив — это именованный набор нескольких однотипных переменных, доступных по индексу (который указывается в квадратных скобках после имени). Многое из того, что было сказано о переменных, напрямую относится и к массивам. Дополнительные нюансы работы с массивами будут освещены в разделе Массивы.

Но вернемся к нашей текущей задаче. Выбор варианта из массива на основе значения переменной *counter* производится в инструкции *return* и выглядит пока довольно кабалистически:

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

Операция деления по модулю с помощью символа '%' уже освещалась вскользь в первой части. С помощью неё мы здесь гарантируем, что индекс элемента не сможет превысить размер массива: каким бы ни был counter, деление его по модулю на 3 даст либо 0, либо 1, либо 2.

Что же касается конструкции *counter++*, она означает увеличение значения переменной на 1 (единичный инкремент).

Важно отметить, что в указанной нотации инкремент будет происходить уже после вычисления всего выражения, в данном случае — после деления *counter % 3*. Это означает, что счет пойдет, начиная с нуля (начального значения). Существует возможность делать инкремент до вычисления выражения, написав *++counter % 3*. Тогда счет начинался бы с 1. Мы изучим операции такого типа в разделе Инкремент и декремент.

Вызовем функцию *Greeting* из *OnStart* 3 раза подряд.

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

В результате мы увидим в журнале ожидаемые три строки со всеми приветствиями, по очереди.

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

Если продолжать вызовы функции, счетчик будет увеличиваться, а сообщения пойдут по кругу.

Попытка обратиться к переменной *counter* в конце *OnStart* (закомментирована) не даст коду скомпилироваться, поскольку статическая переменная хоть и продолжает существовать, доступна только внутри функции *Greeting*.

Обратите внимание, что фигурные скобки используются не только для формирования блоков кода, но и для инициализации массивов. Следует различать эти варианты их применения. О массивах будет подробно рассказано в соответствующем разделе. Но это далеко не все варианты применения фигурных скобок: с помощью них мы позднее научимся определять пользовательские типы, структуры и классы. Внутри структур и классов также можно определить статические переменные.

2.3.5 Переменные-константы

Каким бы оксюмороном это ни звучало, большинство языков программирования поддерживает концепцию переменных-констант. В MQL5 они описываются путем добавления модификатора *const*. Он ставится в описании переменной, перед её типом, и означает, что значение переменной нельзя никаким образом изменить после её инициализации начальным значением. Всё своё время существования переменная будет иметь одно и то же значение — константу.

Компилятор просто не даст присвоить константе какое-либо значение: в соответствующей строке появится ошибка "константа не может модифицироваться" ("constant cannot be modified").

Смысл модификатора *const* в том, чтобы явно указать намерение программиста не менять соответствующую переменную, если в ней хранится некая общеизвестная постоянная величина, такая как коэффициент евро для расчета индекса доллара, количество недель в году и т.д. Рекомендуется всегда применять модификатор const, если переменную не предполагается менять. Это помогает избежать в дальнейшем потенциальных ошибок, если сам программист или кто-то из его коллег случайно попытается записать в константу что-то другое.

Например, мы можем добавить модификатор *const* для массива *messages* в функции *Greeting*. Это не кажется очевидно полезным для столь малой программы. Но поскольку программы имеют тенденцию разрастаться, любая строка рано или поздно может оказаться в гораздо более сложном программном окружении (добавленные инструкции, режимы работы и т.д.). Поэтому имеет смысл заранее подстраховаться, тем более что это так просто.

```
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];
}
```

В закомментированной строке тестируется запись строки "Good night" в первый элемент массива (напомним, нумерация идет с 0). В данном случае смысл такого действия — исключительно саботажный: убедиться, что компилятор не даст это сделать.

Легко заметить, что модификаторы *static* и *const* можно комбинировать. Порядок их записи не важен.

Кстати говоря, в MQL5 переменные становятся константами не только при помощи модификатора *const*, но и при объявлении их входными переменными программы.

2.3.6 Входные переменные

Все программы на MQL5 имеют возможность запросить у пользователя параметры при своем запуске. Исключение составляют лишь библиотеки, которые выполняются не самостоятельно, а в составе какой-либо другой программы (подробнее про Библиотеки — в посвященном им разделе).

Входные параметры MQL-программы — это глобальные переменные, описанные в коде со специальным модификатором *input* или *sinput*. Они становятся доступными в диалоге свойств программы, для ввода значений пользователем. Мы встречали описание входной переменной *GreetingHour* в скриптах первой части.

Особенностью входной переменной является то, что её значение не может быть изменено в коде программы, то есть она ведет себя аналогично константам.

Входные переменные могут быть только простых встроенных типов или перечислениями. При использовании перечисления ввод осуществляется через выпадающий список, а во всех

остальных случаях через поле ввода. Недопустимо описывать как *input*: массивы, структуры или объединения, а также классы.

Разработчик может задать для входного параметра имя, отличное от идентификатора переменной, которое будет использоваться для показа пользователю в диалоге свойств программы. Подробное описание следует добавить в виде однострочного комментария после определения входного параметра.

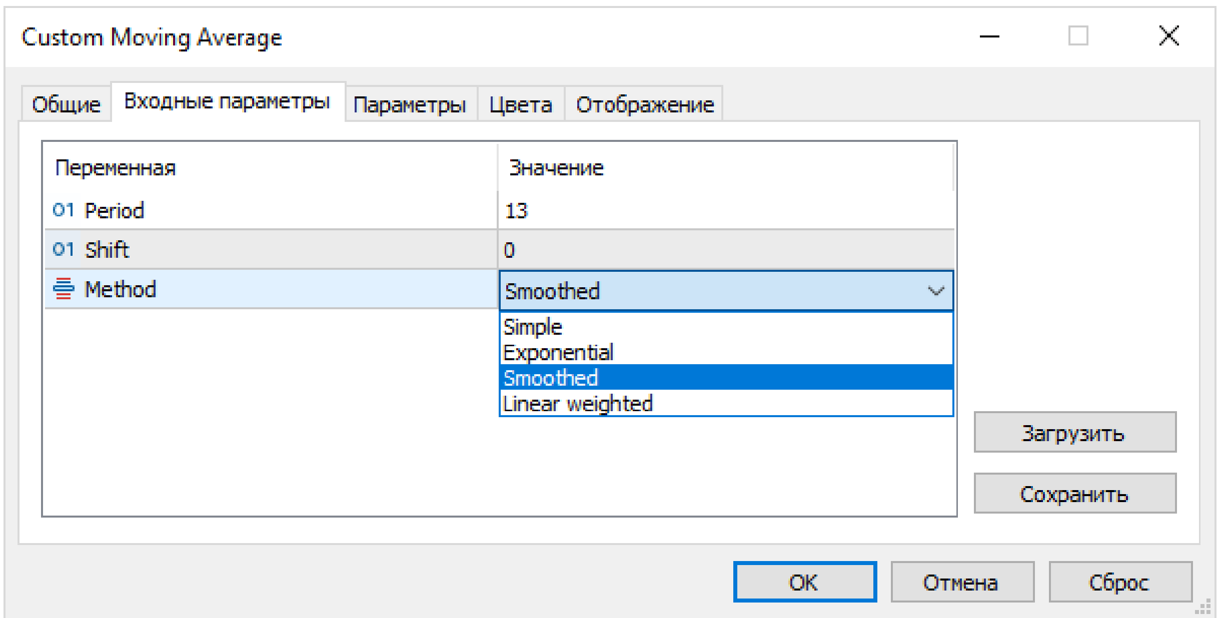
```
input int HourStart = 0; // Начало торговли (час, включая):
input int HourStop = 0;  // Конец торговли (час, исключая):
```

Это позволяет сделать интерфейс более понятным, подробным и не подверженным синтаксическим ограничениям, которые MQL5 накладывает на идентификаторы. Кроме того, названия (как и комментарии) могут быть на национальном языке.

Например, вместе с MetaTrader 5 поставляется исходный код индикатора *MQL5/Indicators/Examples/Custom Moving Average.mq5* со входными переменными:

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

Такое описание генерирует приведенный ниже диалог свойств.



Пример диалога свойств MQL-программы

Максимальная длина текстового представления входной переменной как пары "идентификатор=значение", включая символ "=", не должна превышать 255 символов (это ограничение накладывается внутренними протоколами обмена данными терминала и агентов тестирования). Данный лимит особенно важен для строковых переменных, поскольку значения остальных типов в него укладываются. Как мы знаем, длина идентификатора ограничена 63 символами, и потому на значение строковой входной переменной остается от 191 до 253 символов, в зависимости от длины идентификатора. Весь текст, превышающий объединенный порог 255 символов, может быть обрезан при передаче в тестер. Если в MQL-программу требуется ввести строку большего размера, используйте несколько полей ввода (с

продолжением) или предоставьте пользователю возможность указать имя файла, из которого следует прочитать текст.

Для удобства работы с MQL-программами входные параметры можно объединять в именованные блоки с помощью ключевого слова *group* (точка с запятой в конце строки группы не обязательна).

```
input group "название_группы"
input тип идентификатор = значение;
...
```

Все переменные с модификатором *input*, идущие следом за описанием группы (вплоть до описания другой группы или конца файла), визуально отображаются вложенным списком под заголовком группы в диалоге свойства MQL-программы. Более того, в тестере стратегий (применимом к индикаторам и экспертам) группы параметров можно раскрывать и сворачивать щелчком мыши.

Ключевое слово *sinput* является сокращением от "*static input*" — обе формы эквивалентны.

Переменные, описанные с модификаторами *sinput* или *static input* не могут участвовать в оптимизации. Их имеет смысл применять только в экспертах — это единственный тип MQL-программ, которые поддерживают оптимизацию. Подробнее об этом — в разделе о Тестировании и оптимизации экспертов.

2.3.7 Внешние переменные

> Материал данного раздела является одновременно сложным и необязательным. Он требует знания концепций, некоторые из которых основываются на аналогии с языком C++, а некоторые изучаются позднее в этой книге. В то же время, эффект от описываемой языковой конструкции может быть достигнут другим способом, а её гибкость — потенциальный источник ошибок.

MQL5 позволяет описывать переменные внешними. Это делается с помощью ключевого слова *extern* и допускается только в глобальном контексте.

Для внешней переменной синтаксис в целом повторяет обычное описание, с добавлением extern, но инициализация при этом запрещена:

```
extern type identifier;
```

Описание переменной как внешней означает, что её определение отложено и должно встретиться далее в исходных кодах, как правило в другом файле (о подключении файлов с помощью директивы *#include* будет рассказано в главе про препроцессор). Несколько разных исходных файлов могут иметь описание одной и той же внешней переменной, то есть с одинаковым типом и идентификатором. Все такие описания ссылаются на одну и ту же переменную.

Предполагается, что в одном из файлов данная переменная будет полностью определена. Если переменная нигде не определена без ключевого слова *extern*, выдается ошибка компиляции "unresolved extern variable" (аналогично ошибке линковщика в C++ в подобных случаях).

Описание внешней переменной позволяет полноценно её использовать в исходном коде обособленного файла. Иными словами, оно обеспечивает компиляцию отдельно взятого модуля, несмотря на то, что переменная в этом модуле не создается.

Применение *extern* в MQL5 не является столь же насущным, как в C++, и может быть, в большинстве случаев, заменено на включение заголовочного файла с общими определениями тех переменных, которые предполагалось объявить *extern*. Эти определения достаточно выполнить обычным способом. Компилятор гарантирует, что каждый подключенный файл добавляется в исходный код лишь однажды. А учитывая, что в MQL5 программа всегда состоит из одного компилируемого модуля mq5, здесь не существует проблемы C++ с потенциальной ошибкой множественного определения одной и той же переменной за счет подключения заголовка в разные модули.

Даже если в директиве *#include* подключен некий дополнительный mq5-файл (а не *mqh*), он не конкурирует на равных с главным модулем, для которого запущена компиляция, а рассматривается как один из заголовков.

В отличие от C++, MQL5 не позволяет указывать для внешней переменной начальное значение (в C++ инициализация приводит к тому, что слово *extern* игнорируется). Попытка указать начальное значение сгенерирует при компиляции ошибку "extern variable initialization is not allowed".

В целом, описание переменной внешней можно рассматривать как своего рода "мягкое" описание: оно обеспечивает появление переменной и исключает ошибку переопределения, которая возникла бы, если описать переменную в нескольких файлах без модификатора *extern*.

Однако это можно быть и источником ошибок. Если в разных заголовочных файлах по совпадению описаны одинаковые переменные для разных целей, то отсутствие *extern* позволяет выявить коллизию, а при наличии *extern* переменные сольются в одну, и логика работы программы, скорее всего, будет нарушена.

Внешними могут быть описаны не только переменные, но и функции (они будут рассмотрены далее). Для функций описание с атрибутом внешней является рудиментом (т.е. оно компилируется, но не привносит изменений). Следующие два объявления функции эквивалентны:

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

В этом смысле наличие или отсутствие *extern* можно использовать лишь для того, чтобы стилистически различать упреждающее описание функции из текущего модуля (*extern* нет) или из внешнего (*extern* есть).

Вы можете использовать *extern* как в компилируемом mq5-модуле, так и в подключаемых заголовочных файлах.

Рассмотрим несколько вариантов применения *extern*: они разнесены по разным файлам — главный скрипт *ExternMain.mq5* и 3 подключаемых файла: *ExternHeader1.mqh*, *ExternHeader2.mqh*, *ExternCommon.mqh*.

В главном файле подключены только *ExternHeader1.mqh* и *ExternHeader2.mqh*, а файл *ExternCommon.mqh* нам потребуется чуть позже.

```
// исходный код из mqh-файлов будет неявным образом подставлен
// в главный mq5-файл вместо этих директив
#include "ExternHeader1.mqh"
#include "ExternHeader2.mqh"
```

В заголовочных файлах определены две условно-полезные функции: в первом — функция *inc* для инкремента переменной *x*, во втором — функция *dec* для декремента переменной *x*. Как раз переменная *x* описана в обоих файлах как внешняя:

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

За счет этого описания каждый из mqh-файлов нормально компилируется. И когда они вместе включены в mq5-файл, вся программа целиком также компилируется.

Если бы переменная была определена в каждом файле без слова *extern*, возникла бы ошибка повторного определения при компиляции программы как целого. Если бы мы перенесли определение *x* из заголовочных файлов в главный модуль, перестали бы компилироваться отдельные заголовочные файлы (вероятно, для кого-то это не проблема, но в проектах побольше некоторые разработчики любят проверять компилируемость сиюминутных правок без компиляции проекта целиком).

В главном скрипте определяем переменную (в данном случае, с начальным значением 2, но можно и без него, тогда по умолчанию будет 0) и производим вызов условно-полезных функций и печать значения *x*.

```
int x = 2;

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

В файле *ExternHeader1.mqh* есть определение переменной *short z* (без *extern*). В главном скрипте закомментировано похожее определение. Если эту строку сделать активной, мы получим упомянутую раньше ошибку о повторном определении ("variable already defined"). Это сделано для наглядной демонстрации потенциальной проблемы.

Также в *ExternHeader1.mqh* описана *extern long y*. При этом в файле *ExternHeader2.mqh* одноименная внешняя переменная имеет другой тип: *extern short y*. Если бы не тот факт, что

последнее описание предусмотрительно "убрано" в комментарий, здесь возникла бы ошибка о несовместимости типов ("variable 'y' already defined with different type"). Резюме: типы либо должны совпадать, либо переменные не должны быть внешними. Если оба варианта не подходят, значит — опечатка в имени одной из переменных.

Дополнительно следует обратить внимание на то, что переменная *y* не инициализируется явно. Однако главный скрипт успешно к ней обращается и выводит в журнал 0:

```
long y;

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

Наконец, в скрипте предусмотрена возможность попробовать альтернативу внешним переменным-двойникам на примере уже известной переменной *x*. Вместо описания *extern int x* каждый из файлов *ExternHeader1.mqh* и *ExternHeader2.mqh* способен подключить другой общий заголовок *ExternCommon.mqh*, в котором находится определение *int x* (без *extern*). Оно становится единственным определением *x* в проекте.

Этот альтернативный режим сборки программы включается при активации макроса USE_INCLUDE_WORKAROUND: он присутствует в комментарии в начале скрипта:

```
#define USE_INCLUDE_WORKAROUND // эта строка была в комментарии
#include "ExternHeader1.mqh"
#include "ExternHeader2.mqh"
```

В такой конфигурации компилируемость отдельных включаемых файлов сохранится, как и всего проекта в целом. В реальном проекте, при использовании данного способа, общий mqh-файл был бы подключен в *ExternHeader1.mqh* и *ExternHeader2.mqh* безусловно (без условий на USE_INCLUDE_WORKAROUND). В данном примере переключение между двумя ветками инструкций на основе USE_INCLUDE_WORKAROUND нужно только для демонстрации обоих режимов. Например, *ExternHeader2.mqh* в упрощенной версии должен выглядеть так:

```
// ExternHeader2.mqh
#include "ExternCommon.mqh" // int x; теперь здесь

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

В журнале MetaEditor можно убедиться, что файл *ExternCommon.mqh* подгружается единожды, несмотря на то, что на него есть ссылка и в *ExternHeader1.mqh*, и в *ExternHeader2.mqh*.

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

Если переменная *x* "прописалась" в *ExternCommon.mqh*, мы уже не имеем права переопределять её (без *extern*) в главном или любом другом модуле, т.к. будет ошибка компиляции, но мы можем просто присвоить ей нужное значение в начале алгоритма.

2.4 Массивы

Массив — это средство для группового хранения и обработки данных произвольных типов. Они поддерживаются практически в любом языке программирования. В MQL5 они особенно важны, потому что представляют собой удобный способ организации серийных данных, актуальных для задач трейдинга. Котировки, показания индикаторов, торговая история счета с приказами и сделками, новостные события — все это примеры серийных данных, то есть последовательностей значений меняющихся во времени.

Массив можно назвать переменной-контейнером: она способна содержать предопределенное количество однотипных значений, которые идентифицируются не только по имени, но и по индексу (порядковому номеру).

В данном разделе мы изучим общий синтаксис описания массивов и обращения к ним на примере встроенных типов данных. В следующих частях книги, по мере получения знаний о расширении системы типов за счет объектно-ориентированной технологии, мы используем массивы в связке с ними для получения новых возможностей.

2.4.1 Характеристики массивов

Прежде чем приступать к изложению синтаксических особенностей объявления и приемов работы с массивами в MQL5, затронем базовые концептуальные вопросы построения массивов.

Главной характеристикой массива является количество измерений. В одномерном массиве элементы размещены один за другим, как шеренга солдат, и для обращения к ним достаточно одного номера (индекса). Побаровые цены открытия какого-либо финансового инструмента на заданную глубину истории можно сохранить в таком массиве.

В двумерном массиве элементы расходятся в двух логически перпендикулярных направлениях, образуя своего рода квадрат (или прямоугольник, в общем случае), причем для доступа к каждому элементу нужно два индекса — по одному в каждом измерении. Такой массив можно было бы использовать для хранения четверок цен (Open, High, Low, Close) для каждого бара истории. По первому измерению отсчитывались бы номера баров, а по второму — число от 0 до 3 (включительно), обозначающее один из типов цен.

Трехмерный массив — это эквивалент куба (или более строго с позиций геометрии — прямоугольного параллелепипеда) с тремя осями. Продолжая пример с массивом побаровых цен, мы могли бы добавить в него третье измерение, отвечающее за перебор финансовых инструментов из Обзора рынка.

По каждому из измерений массив имеет некоторую протяженность (размер), задающую диапазон возможных индексов. Если бы, предположим, история была загружена для 1000 баров для 10

инструментов, мы получили бы массив с размером 1000 элементов в первом измерении, 4 элемента во втором (OHLC) и 10 — в третьем.

Произведение размеров по всем измерениям дает общее количество элементов массива, в данном случае — 40000. В MQL5 оно не может быть больше 2147483647 (максимум для int).

Для массива с 4-мя измерениями уже трудно представить пространственную фигуру, потому что мы привыкли жить в трехмерном мире. Однако MQL5 допускает создание массивов с количеством измерений вплоть до четырех.

Следует отметить, что всегда можно использовать одномерный массив вместо многомерного массива с произвольным количеством измерений (в том числе, больше 4) — это всего лишь вопрос организации пересчета нескольких индексов в один сквозной. Например, если двумерный массив имеет 10 колонок (измерение 1, ось X) и 5 строк (измерение 2, ось Y), его можно преобразовать в одномерный массив с таким же количеством элементов, то есть 50. При этом индекс элемента получается по формуле:

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

Здесь N — количество элементов по первому измерению, в данном случае 10, это размер каждой строки; Y — номер строки (0..4); X — номер колонки (0..9) в строке.

Размеры по всем измерениям — еще одна характеристика, отличающая массив от переменной. Таким образом, количество измерений и размер по каждому из них должны некоторым образом указываться при описании, в дополнении к названию массива и типу данных (см. следующий раздел).

Следует различать размер переменной (элемента массива) в байтах и размер массива как количество элементов в нем. Полный размер массива, который он занимает в памяти, должен теоретически быть равен произведению размера одного элемента (зависит от типа данных) на количество элементов. Однако на практике эта формула работает не всегда. В частности, поскольку строки могут иметь разную длину, оценить объем памяти, занятый строковым массивом, затруднительно.

По способу выделения памяти массивы делятся на динамические и фиксированного размера.

Фиксированный массив описывается в коде с точными размерами по всем измерениям. Изменить его размер потом нельзя. Однако на практике часто возникают задачи, когда количество данных для обработки заранее не известно, и потому желательно изменять размер массива в процессе работы алгоритма. Для этих целей и существуют динамические массивы. Как мы увидим далее, они описываются без указания размера по первому измерению и могут затем "растягиваться" или "сжиматься" с помощью специальных функций MQL5 API.

В документации MQL5 используется неоднозначная терминология, называющая массивы фиксированного размера статическими. Данное понятие также используется для модификатора static, который может быть применен к массиву. Если такой массив объявлен динамическим, тогда он одновременно нестатический с точки зрения выделения памяти и статический с точки зрения модификатора static. Чтобы исключить неоднозначность, в данной книге под статичностью будем понимать исключительно атрибут декларации.

Помимо динамических и фиксированных массивов в MQL5 существуют массивы особого рода для хранения котировок и буферов технических индикаторов. Такие массивы называются массивами с временными рядами (таймсериями), поскольку их индексы соответствуют отсчетам во времени. По сути эти массивы являются одномерными динамическими, но в отличие от прочих

динамических массивов, память под них выделяется самим терминалом. Мы изучим их в разделах про таймсерии и индикаторы.

2.4.2 Описание массивов

Описание массивов наследует некоторые черты описания переменных. Прежде всего следует отметить, что массивы могут быть глобальными и локальными в зависимости от места их объявления. Также по аналогии с переменными, при описании массива могут быть использованы модификаторы *const* и *static*. Для одномерного фиксированного массива синтаксис объявления выглядит следующим образом:

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

Здесь, *type* и *static1D* обозначают название типа элементов и идентификатор массива соответственно, а *size* в квадратных скобках — это целочисленная константа, задающая размер.

Для многомерных массивов необходимо указывать несколько размеров, по количеству измерений:

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

Динамические массивы описываются аналогично, за исключением того, что в первых квадратных скобках делается пропуск (перед использованием такого массива нужно выделить под него требуемый объем памяти с помощью функции *ArrayResize*, см. раздел про динамические массивы).

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

Для фиксированных массивов разрешается делать инициализацию: начальные значения для элементов указываются после знака равно, списком через запятую, причем весь список заключается в фигурные скобки. Например:

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

Здесь целочисленный массив размером 3 получает значения 10, 20, 30.

При наличии списка инициализации необязательно указывать размер массива в квадратных скобках (для первого измерения) — компилятор автоматически определит размер по длине списка. Например:

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

Начальными значениями могут быть не только константы, но и константные выражения — формулы, которые компилятор способен рассчитать в момент компиляции. Например, следующий массив заполняется количеством секунд в минуте, часе, сутках и неделе (представление в виде формул более наглядное, чем 86400 или 604800):

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

Обычно такие значения оформляются как макрос препроцессора в начале кода, и потом везде, где нужно по тексту, вместо формул вставляется имя макроса. Эта возможность описана в разделе Препроцессор.

Количество инициализирующих элементов не должно превышать размер массива. Иначе компилятор выдаст ошибку "слишком много инициализаторов" ("too many initializers"). Если количество значений меньше размера массива, оставшиеся элементы инициализируются нулем. Поэтому существует сокращенная нотация для инициализации всего массива нулями:

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

Или просто пустые скобки:

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

Она работает вне зависимости от количества измерений.

Для инициализации многомерных массивов списки значений должны быть вложенными. Например:

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

Здесь массив имеет размер 3 по первому измерению, поэтому внутри внешних фигурных скобок две запятых выделяют 3 элемента. Однако поскольку массив двумерный, каждый из его элементов — это в свою очередь массив, причем размер каждого из них равен 2. Поэтому каждый элемент представляет собой список в фигурных скобках, в каждом списке по 2 значения.

Если, предположим, нам требуется транспонированный массив (первый размер равен 2, второй — 3), то его инициализация изменится:

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

При необходимости можно пропустить одно или более значений в списке инициализации, обозначив их место запятыми. Все пропущенные элементы будут также проинициализированы нулем.

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

Здесь первые два элемента будут равны 0.

Синтаксисом языка разрешено ставить запятую и после последнего элемента:

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

Это упрощает добавление новых элементов, особенно при многострочной записи. В частности, если забыть вставить запятую перед новым добавленным элементом в строковом массиве, окажется, что старая и новая строки склеятся в одном элементе (с прежним индексом), а нового элемента не появится. Кроме того, какие-то массивы могут генерироваться автоматически (другой программой или макросами), поэтому унифицированный вид всех элементов получается естественным.

"Куча" и "стек"

В случае массивов, которые могут потенциально иметь большие размеры, важно отметить разницу между глобальным и локальным размещением в памяти.

Под глобальные переменные и массивы память распределяется в так называемой куче — свободной памяти, доступной программе. Эта память практически ничем не ограничена кроме физических характеристик компьютера и операционной системы. Название "куча" объясняется тем, что участки памяти разных размеров постоянно то выделяются, то освобождаются программой, в результате чего свободные области оказываются произвольным образом разбросаны в общей массе.

Локальные переменные и массивы размещаются в стеке — это ограниченный участок памяти, заранее выделяемый программе специально для локальных элементов. Название стек происходит от того, что в процессе выполнения алгоритма часто происходят вложенные вызовы функций, накапливающие свои внутренние данные по принципу "нагромождения друг на друга": например, *OnStart* вызывается терминалом, из *OnStart* вызывается некая функция из вашего прикладного кода, из неё, в свою очередь, вызывается другая ваша функция и так далее. При этом в момент входа в каждую из функций на стеке создаются её локальные переменные, и они продолжают там оставаться, когда происходит вызов вложенной функции. Та тоже создает локальные переменные, которые попадают на стек как бы сверху предыдущих. В результате стек обычно содержит несколько слоев локальных данных из всех функций, которые активировались на пути к текущей строчке кода. Только когда функция, находящаяся на вершине стека, завершается, её локальные данные оттуда удаляются. В общем, стек — это хранилище, работающее по схеме FILO/LIFO (First In Last Out, Last In First Out).

Поскольку размер стека ограничен, в нем рекомендуется создавать только локальные переменные. Массивы же могут быть достаточно большими, чтобы быстро исчерпать весь стек. При этом выполнение программы завершается с ошибкой. Поэтому массивы следует описывать на глобальном уровне, статическими (*static*) или выделять под них память динамически (это тоже делается из кучи).

2.4.3 Использование массивов

Запись и чтение значений в элементы массива производится с помощью похожего синтаксиса с указанием требуемых индексов в квадратных скобках. Чтобы положить значение в элемент, используем операцию присваивания '='. Например, для замены значения 0-го элемента одномерного массива:

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

Индексация ведется с 0. Индекс последнего элемента равен количеству элементов минус 1. Разумеется, в качестве индекса можно использовать не только константу, а любое выражение, приводимое к целому типу (подробнее о выражениях в следующей главе), например, целочисленную переменную, вызов функции или элемент другого массива с целыми (косвенная адресация).

```
int index;
// ...
// index = ... // как-либо присваиваем index
// ...
array1D[index] = 11;
```

Для многомерных массивов требуется указать индексы для всех измерений.

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

Допустимые целочисленные типы для индексов исключают *long* и *ulong*. При попытке использовать в качестве индекса значение "длинного целого", оно будет неявно конвертироваться в *int*, в связи с чем компилятор выдаст предупреждение "возможна потеря данных из-за конвертации" ("possible loss of data due to type conversion").

Доступ к элементам массива на чтение организуется по такому же принципу. Например, вот как можно вывести в журнал элемент массива:

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

В скрипте *GoodTimes* мы уже видели описание локального статического массива *messages* со строками приветствий (внутри функции *Greeting*) и использование его элементов в операторе *return*.

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

При выполнении *return* производится чтение элемента с индексом, определяемым выражением: *counter++ % 3*. Деление по модулю 3 (обозначаемое символом '%'), обеспечивает, что увеличиваемый каждый раз на 1 счетчик *counter* будет приведен в диапазон корректных значений индексов: 0, 1 или 2. Если бы деления по модулю не было, индекс запрашиваемого элемента превысил бы размер массива, начиная с 4-го вызова данной функции. В таких случаях происходит ошибка времени выполнения программы ("выход за пределы массива" — "array out of range"), и она выгружается с графика.

MQL5 API включает универсальные функции для многих операций с массивами: выделение памяти (под динамические массивы), заполнение, копирование, сортировка, поиск в массивах — они рассматриваются в разделе Работа с массивами. Но одну из них мы представим сейчас: *ArrayPrint* позволяет в удобном виде (с учетом измерений) вывести элементы массива в журнал.

Скрипт *Arrays.mq5* демонстрирует некоторые примеры описания массивов и результаты выводятся в журнал. Манипуляции с элементами массивов мы разберем позднее, после знакомства с циклами и выражениями.

```
void OnStart()
{
  char array[100];      // без инициализации
  int array2D[3][2] =
  {
    {1, 2},             // форматирование для наглядности
    {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}; // ошибка: too many initializers
  ArrayPrint(array);    // вывод произвольных "мусорных" значений
  ArrayPrint(array2D);  // показываем 2D-массив в журнале
  ArrayPrint(array2Dt); // "транспонированный" вид тех же данных 2D
  ArrayPrint(prices);   // узнаем значения элементов ценового перечисления
}
```

Один из вариантов записей в журнале представлен ниже.

```
[ 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][,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
```

Массив *array* не имеет инициализации и потому выделяемая под него память может содержать случайные значения. При каждом запуске скрипта значения будут меняться. Рекомендуется всегда инициализировать локальные массивы, на всякий случай.

Массивы *array2D* и *array2Dt* выводятся в журнал в наглядном виде, как матрицы. Это никак не связано с тем, что мы отформатировали таким же образом списки инициализации в исходном коде.

Массив *prices* имеет тип встроенного перечисления ENUM_APPLIED_PRICE. В принципе, массивы могут быть любых типов, включая структуры, указатели на функции и другие вещи, которые нам еще предстоит изучить. Поскольку перечисления строятся на основе типа *int*, значения

отображены цифрами, а не названиями элементов (для получения имени конкретного элемента перечисления есть функция EnumToString, но её режим не поддерживается в *ArrayPrint*).

В строке с описанием массива *d* — ошибка: количество начальных значений превышает размер массива.

2.5 Выражения

Выражение — основополагающий элемент любого языка программирования. Какой-бы прикладной смысл ни был заложен в алгоритме, он в конечном счете сводится к обработке данных, то есть к вычислениям. Выражение описывает вычисление некоего результата из одной или более заданных величин. Величины называются операндами, а суть производимых над ними действий обозначают операции или операторы.

В качестве операторов, которые позволяют манипулировать операндами, в выражениях используются отдельные символы или их последовательности, например, '+' для сложения или '*' для умножения. Все они образуют несколько групп, в частности, арифметические, побитовые, сравнения, логические и некоторые специализированные.

Мы уже пользовались выражениями в предыдущих разделах книги, например, для инициализации переменных. В простейшем случае, выражение — это константа (литерал), которая и является единственным операндом, и результат вычисления равен значению операнда. Однако операндами также могут быть переменные, элементы массивов, результаты вызова функции (для чего функция вызывается прямо из выражения), вложенные выражения и прочие сущности.

Все операторы подставляют (возвращают) результат своей работы в родительское выражение, непосредственно в то место, где находились операнды, что позволяет комбинировать их, составляя довольно сложные иерархические конструкции. Например, в следующем выражении результат произведения переменных *b* и *c* складывается со значением переменной *a*, после чего полученная величина сохраняется в переменной *v*:

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

В этом разделе мы рассмотрим общие принципы построения и вычисления выражений, а также стандартный набор операторов, поддерживаемый в MQL5 для встроенных типов. Позднее, в части, посвященной ООП, мы узнаем, как можно перегрузить (переопределить) операторы для пользовательских типов — структур и классов, что позволит использовать в выражениях объекты и выполнять над ними нестандартные действия.

В русскоязычной литературе одинаково часто встречаются два устоявшихся варианта перевода оригинального термина "operator": операция и оператор. При этом инструкции ("statements"), которые мы изучим в одноименной главе, также часто называют по-русски операторами. Это может создавать проблемы, потому что операторы-операции, составляющие выражения, "живут" внутри операторов-инструкций, организующих выполнение разных строк программы. Кроме того, слова "операция" и "оператор" настолько похожи, что их легко перепутать. Поэтому далее в книге постараемся придерживаться следующих правил.

Мы будем использовать термин "оператор" в качестве синонима "операции" там, где не упоминаются "инструкции". Также мы будем использовать "оператор" как общеупотребительный аналог "инструкции" в отсутствии неоднозначности с "операциями".

2.5.1 Базовые понятия

Прежде чем приступать к описанию конкретных групп операторов, необходимо ввести несколько базовых понятий, которые присущи всем операторам и влияют на их применимость и поведение в тех или иных случаях.

Прежде всего, по количеству требуемых операндов операторы делятся на унарные и бинарные. Как можно понять по этим названиям, унарные обрабатывают один операнд, а бинарные — два. В случае бинарных, оператор всегда располагается между операндами. Среди унарных встречаются операторы, которые должны стоять перед операндом, и такие, которые стоят после. Например, оператор унарного минуса ('-') позволяет поменять знак величины на противоположный:

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

При этом существует также и бинарный оператор вычитания, использующий тот же символ '-'.

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

Выбор компилятором правильного оператора (действия) в конкретном случае определяется контекстом его применения в выражении.

Каждому оператору присвоен приоритет. Он определяет порядок, в котором операторы рассчитаются в сложных выражениях, где имеется более одного оператора. Операторы с более высоким приоритетом рассчитываются первыми, а с более низким — последними. Например, в выражении *1 + 2 * 3* есть две операции (сложение и умножение) и три операнда. Поскольку приоритет умножения выше приоритета сложения, сначала рассчитается произведение *2 * 3*, а затем — его сложение с единицей.

Позднее мы приведем полную таблицу операций с приоритетами.

Дополнительно каждый оператор характеризуется ассоциативностью. Она бывает левая и правая, и определяет, в каком порядке выполняются последовательно идущие операторы с равным приоритетом. Например, выражение *10 - 7 - 1* можно, чисто теоретически, рассчитать двумя способами:

- сначала вычесть 7 из 10, и потом из получившейся 3-ки вычесть 1, что даст результат 2;
- сначала вычесть 1 из 7, что даст 6, и потом вычесть 6 из 10, с результатом 4.

В первом случае вычисления проводились слева направо, что соответствует левой ассоциативности, и поскольку операция вычитания действительно левоассоциативна, то первый ответ — правильный.

Второй порядок вычислений соответствует правой ассоциативности, и не будет применен.

Рассмотрим еще один пример, в котором одновременно задействованы приоритет и ассоциативность: *11 + 5 * 4 / 2 + 3*. Оба типа операций — суммирование и умножение — выполняются слева направо. Если бы не разный приоритет, мы получили бы 35, хотя правильный ответ 24. А смена ассоциативности на правую дала бы 14.

Для явного переопределения приоритетов в выражениях можно применять круглые скобки, например: *(11 + 5) * 4 / (2 + 3)*. То, что стоит в скобках, вычисляется раньше, и промежуточный результат подставляется в выражение для участия в остальных операциях. Группы в скобках могут быть вложенными. Подробнее об этом в разделе Группировка с помощью круглых скобок.

В качестве примера оператора с правой ассоциативности можно привести унарный оператор логического отрицания '!'. Суть его работы: делать из *true* — *false*, и наоборот. Как и для других унарных операторов, ассоциативность в этом случае означает, с какой стороны от оператора должен стоять операнд. Символ '!' ставится перед операндом, то есть операнд находится справа.

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

В данном случае логическое отрицание делается дважды: первый раз по отношению к переменной *x* (правый '!'), а второй раз — по отношению к результату предыдущего отрицания (левый '!'). Такое двойное отрицание позволяет преобразовать любое ненулевое значение в 1, благодаря конвертации в *bool* и обратно.

В итоговой таблице операций будет также указана и ассоциативность.

Наконец, последним, но не менее важным нюансом обработки выражений является порядок вычисления операндов. Его следует отличать от приоритета, который относится к операции, а не операндам. Порядок вычисления операндов бинарных операций явно не определен, что дает простор компилятору для оптимизации и повышения эффективности кода. Компилятор гарантирует только то, что операнды будут вычислены перед выполнением операции.

Существует ограниченный набор операций, для которых порядок вычисления операндов определен, в частности, для логических И ('&&') и ИЛИ ('||') он — "слева направо", причем правая часть может быть опущена, если ничего не решает из-за значения левой части. А вот для тернарного условного оператора '?:' порядок и вовсе более "хитрый", потому что после вычисления первого условия, в зависимости от его истинности, будет вычислена либо одна, либо другая ветка. Подробнее об этом — в последующих разделах.

Порядок вычисления операндов наглядно иллюстрирует ситуация, когда в выражении имеется несколько вызовов функций. Например, пусть в выражении используется 4 функции:

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

Правила приоритетов и ассоциативности будут применены только к промежуточным результатам вызовов этих функций, а сами вызовы могут быть сгенерированы компилятором в любом порядке, который он "сочтет нужным", исходя из особенностей исходного кода и настроек компилятора. Например, функции *b* и *c*, задействованные в произведении, могут вызываться в последовательности [*b(), c()*] или наоборот [*c(), b()*]. Если функции в процессе своего выполнения могут влиять на одни и те же данные, их состояние после вычисления выражения будет неоднозначным.

Аналогичную проблему можно наблюдать при работе с массивами и операторами инкремента (см. Инкремент и декремент).

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

В зависимости от того, левый или правый операнд разности будет рассчитан первым, мы можем получить -1 (*a[1] - a[2]*) или +1 (*a[2] - a[1]*). Поскольку компилятор MQL5 постоянно совершенствуется, нет гарантии, что текущий результат (-1) сохранится в будущем.

Чтобы избежать потенциальных проблем, рекомендуется не использовать повторно операнд, если тот уже модифицировался в том же выражении.

Во всех выражениях, как правило, встречаются операнды разных типов. Это приводит к необходимости приводить их к некоему общему типу, прежде чем выполнять над ними действия. В отсутствии явного приведения типов MQL5 производит неявную конвертацию там, где это необходимо. Причем для разных сочетаний типов правила конвертации отличаются. Явное и неявное приведение типов будет рассмотрено в отдельном разделе.

2.5.2 Операция присваивания

Результат вычисления выражений обычно должен где-то сохраняться. Для этой цели в языке предназначен оператор присваивания, обозначаемый знаком равенства '='. Слева от него ставится имя переменной или элемента массива, куда следует сохранить результат, а справа — само выражение, фактически, формула для расчета.

Мы уже использовали этот оператор для инициализации переменных, которая выполняется лишь однажды, в момент их создания. Однако присваивание позволяет изменять значения переменных в ходе алгоритма произвольное количество раз. Например:

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

Переменные *x* и *y* были проинициализированы значениями 1 и 2, после чего с помощью вспомогательной третьей переменной *z* и трех присвоений был произведен обмен значениями *x* и *y*.

Оператор присваивания, как и все операторы, возвращает в выражение результат своей работы. Это дает возможность записывать присваивания в цепочку.

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

Здесь единица будет присвоена сперва переменной *z*, затем переменной *y* и, наконец, переменной *x*. Очевидно, что данный оператор является правоассоциативным, потому что присваиваемое значение "перетекает" в выражении справа налево.

Мы можем использовать присваивание как часть выражения, но поскольку приоритет его ниже всех других операторов (исключая только оператор "запятая", см. Приоритеты операций), необходимо заключать его в круглые скобки (подробности см. в разделе Группировка с помощью круглых скобок). Этот аспект делает возможными ситуации, когда опечатки в выражениях (скажем, одинарный символ '=' вместо '==') приводят к выполнению инструкций не так, как было задумано. Пример такого поведения приводится в разделе про инструкцию *if*.

Оператор присваивания накладывает определенные ограничения на то, что может стоять слева от знака '=', а что — справа. В программировании эти сущности для облегчения запоминания так и называются: LValue и RValue (от Left и Right, левое и правое).

LValue и RValue

LValue — представляет собой сущность, для которой выделена память, и как следствие, в неё можно записать значение. Известными нам примерами LValue являются переменная и элемент массива. После изучения ООП мы узнаем еще одного представителя этой категории: объект, в котором может быть перегружен оператор присваивания. Обязательным признаком

LValue является наличие идентификатора.

Следует учитывать, что переменные и массивы могут быть описаны с ключевым словом const, и тогда они не могут выступать в качестве LValue, потому что модификация констант запрещена.

RValue — временная величина, используемая в выражении, такая как литерал или значение, возвращенное вследствие вызова функции или расчета фрагмента выражения.

Категория LValue носит расширительный характер: принадлежность к ней разрешает ставить соответствующий объект слева от знака '=', но не запрещает использовать его наравне с RValue справа от '='.

Категория RValue, напротив, имеет ограничительный характер: любая величина RValue может стоять только справа от '='.

Когда некий LValue-элемент используется справа от знака '=', его идентификатор обозначает фактически его текущее содержимое, помещенное в формулу выражения.

Когда же LValue-элемент используется слева от знака '=', его идентификатор указывает на адрес памяти (ячейку), куда следует записать новое значение — результат вычисления выражения.

Различные операторы имеют различные ограничения относительно того, могут ли они применяться к операндам-LValue или операндам-RValue. Например, операторы инкремента '++' и декремента '--' (см. Инкремент и декремент) допустимо использовать только с LValue.

Вот несколько примеров того, что можно и что нельзя делать с оператором присваивания (скрипт *ExprAssign.mq5*):

```
// описание переменных
const double cx = 123.0;
int x, y, a[5] = {1};
string s;
// присваивание
a[2] = 21;          // ok
x = a[0] + a[1] + a[2]; // ok
s = Symbol();       // ok
cx = 0;             // нельзя изменить const переменную
                    // error: 'cx' - constant cannot be modified
5 = y;              // 5 — это число (литерал)
                    // error: '5' - l-value required
x + y = 3;          // слева RValue (результат вычисления выражения)
                    // error: l-value required
Symbol() = "GBPUSD"; // слева RValue с результатом вызова функции
                     // error: l-value required
```

Компилятор выдает ошибку при нарушении правил использования оператора.

2.5.3 Арифметические операции

К арифметическим операциям относятся 5 бинарных: сложение, вычитание, умножение, деление и деление по модулю, а также 2 унарных: плюс и минус. В таблице приведены символы, использующиеся для каждой из этих операций.

В колонке с примерами *e1*, *e2* — это произвольные подвыражения. Ассоциативность помечена буквами 'L' (слева направо) и 'R' (справа налево). Номер в первой колонке можно расценивать, как очередность выполнения операции.

П	Символы	Описание	Пример	А
2	+	Унарный плюс	+e1	R
2	-	Унарный минус	-e1	R
3	*	Умножение	e1 * e2	L
3	/	Деление	e1 / e2	L
3	%	Деление по модулю	e1 % e2	L
4	+	Сложение	e1 + e2	L
4	-	Вычитание	e1 - e2	L

Порядок в таблице соответствует понижению приоритетов: унарные плюс и минус вычисляются раньше, чем умножение и деление, а те, в свою очередь — раньше, чем сложение и вычитание.

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

Унарный плюс фактически не имеет эффекта при вычислениях, но может использоваться для большей наглядности выражения. Унарный минус меняет знак своего операнда на противоположный.

Арифметические операции применяются к числовым типам или приводимым к ним. Результат вычисления является RValue. В процессе вычислений часто производится расширение разрядности целочисленных операндов до наиболее "большого" из использованных целых или *int* (если все целые типы были меньшего размера), а также приведение к общему типу. Подробнее об этом — в разделе Приведение типов.

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

В данном примере переменная *b1* "расширяется" до типа *int* со значением 1. Обращение знака дает -1, что при обратном приведении типа к *bool* дает *true* (потому что -1 не ноль). Применение логического типа в арифметических вычислениях не приветствуется.

Деление целых чисел дает целое число, то есть дробная часть (если она возникает) отбрасывается. Проверить это можно с помощью скрипта *ExprArithmetic.mq5*.

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

Несмотря на то, что переменная *c* описана как *double*, в выражении для её инициализации стоят целые числа и потому деление выполняется целочисленное. Для того чтобы выполнялось деление с дробной частью, необходимо, чтобы хотя бы один операнд был вещественного типа (второй будет неявно преобразован также к нему).

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

Оператор '%' вычисляет остаток от целочисленного деления (применим только к двум операндам целого типа).

```
int x = 11 % 5;    // ok: x = 1
int y = 11 % 5.0; // нельзя использовать вещественное число
                   // error: '%' - illegal operation use
```

Для случаев, когда операнды имеют разные знаки, операторы '*' и '/' дают отрицательное число. Для оператора '%' действуют следующие правила:

- когда делитель оператора '%' отрицателен, знак "уходит";
- когда делимое оператора '%' отрицательно, результат получается отрицательным;

Это легко проверить, пользуясь альтернативным расчетом деления по модулю: *m % n = m - m / n * n*. Здесь следует помнить, что деление *m / n* для целых чисел будет с округлением, поэтому *m / n * n* в общем случае не равно *m*.

В разделе Характеристики массивов мы поднимали вопрос о том, что многомерный массив может быть представлен одномерным массивом за счет пересчета индексов элементов. Там приводилась формула для получения сквозного индекса в одномерном массиве по координатам (номеру столбца X и строки Y при длине строки N) двумерного массива.

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

Операция '%' позволяет нам более удобно произвести обратный расчет, т.е. узнать X и Y по сквозному индексу:

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

Если в процессе вычисления выражения на каком-то этапе получился непредставимый результат NaN (Not A Number, например, бесконечность, корень из отрицательного числа и т.д.), все последующие операции с ним также дают NaN. Отличить его от нормального числа можно с помощью функции MathIsValidNumber (см. Математические функции).

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

Тут вычитание уже производится из NaN (полученного при делении) и дает в результате тоже NaN.

Операция сложения определена для строк и выполняет конкатенацию, то есть их объединение.

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

Прочие операции для строк запрещены.

2.5.4 Инкремент и декремент

Операторы инкремента и декремента позволяют в упрощенном виде записать увеличение или уменьшение операнда на 1. Они чаще всего встречаются внутри циклов для модификации индексов при доступе к массивам и прочим объектам, поддерживающим перечисление.

Инкремент обозначается двумя подряд символами плюса '++'. Декремент обозначается двумя подряд минусами '--'.

Существует два вида таких операторов: префиксные и постфиксные.

Префиксные, как следует из названия, записываются перед операндом (++*x*, --*x*). Они изменяют значение операнда, и это новое значение участвует в дальнейшем расчете выражения.

Постфиксные записываются после операнда (*x*++, *x*--). Они подставляют в выражение копию текущего значения операнда, после чего производят изменение его величины (новое значение не попадает в выражение). Простые примеры приведены в скрипте *ExprIncDec.mq5*.

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

Постфиксная форма бывает полезна для более компактной записи выражений, сочетающих обращение к предыдущему значению операнда и его побочную модификацию (альтернативная запись того же потребовала бы двух отдельных инструкций). Во всех остальных случаях рекомендуется использовать префиксную форму (она не создает временную копию "старого" значения).

В следующем примере у элементов массива последовательно меняется знак, пока не встретится нулевой элемент. Продвижение по индексам массива обеспечивает постфиксный инкремент *k*++ внутри цикла *while*. За счет постфикса, выражение *a[k++] = -a[k]* сначала обновляет k-й элемент, а затем увеличивает *k* на 1. Затем результат присваивания проверяется на неравенство нулю (!= 0, см. следующий раздел).

```
int k = 0;
int a[] = {1, 2, 3, 0, 5};
while((a[k++] = -a[k]) != 0){}
// a[] = {-1, -2, -3, 0, 5};
```

В следующей таблице представлены операторы инкремента и декремента в порядке приоритета:

П	Символы	Описание	Пример	А
1	++	Постфикс инкремент	e1++	L
1	--	Постфикс декремент	e1--	L
2	++	Префикс инкремент	++e1	R
2	--	Префикс декремент	--e1	R

Все операции инкремента и декремента имеют более высокий приоритет, чем арифметические операции. Префиксы менее приоритетны, чем постфиксы. В следующем примере "старое" значение *x* суммируется со значением *y*, после чего *x* инкрементируется. Если бы приоритет

префикса был выше, выполнился бы инкремент *y*, после чего новое значение 6 просуммировалось бы с *x*, и мы получили бы *z = 6, x = 0* (прежний).

```
int x = 0, y = 5;
int z = x+++y; // "x++ + y" : z = 5, x = 1
```

2.5.5 Операции сравнения

Как и следует из названия, данные операции предназначены для сравнения двух операндов и возврата логического признака — *true* или *false* — в зависимости от выполнения условия в сравнении.

В таблице приведены все операции сравнения и их свойства: используемые символы, приоритеты, примеры и ассоциативность.

П	Символы	Описание	Пример	А
6	<	Меньше	e1 < e2	L
6	>	Больше	e1 > e2	L
6	<=	Меньше или равно	e1 <= e2	L
6	>=	Больше или равно	e1 >= e2	L
7	==	Равно	e1 == e2	L
7	!=	Не равно	e1 != e2	L

Принцип работы каждой операции — сравнение двух операндов с помощью критерия из колонки с описанием. Например, запись "*x < y*" означает проверку на то, что "*x* меньше *y*". Соответственно, результат сравнения будет равен *true*, если *x* действительно меньше *y*, и *false* — во всех остальных случаях.

Сравнения работают для операндов любых типов (при различии типов делается их приведение).

С учётом левой ассоциативности и возврата результата типа *bool*, построение цепочки сравнений работает не столь очевидным образом. Например, гипотетическое выражение для проверки того, лежит ли величина *y* между значениями *x* и *z*, могло бы, казалось бы, выглядеть так:

```
int x = 10, y = 5, z = 2;
bool range = x < y < z;   // true (!)
```

Однако такое выражение обрабатывается иначе. Даже компилятор выделяет его предупреждением "небезопасное применение 'bool'" ("unsafe use of type 'bool' in operation").

В силу левой ассоциативности сначала проверяется левое условие *x < y*, и его результат в виде временного значения типа *bool* подставляется в выражение, которое становится таким: *b < z*. Далее значение *z* уже сравнивается с *true* или *false* во временной переменной *b*. Для того чтобы проверить, лежит ли *y* в диапазоне между *x* и *z*, следует использовать две операции сравнения, объединенных логической операцией И (она будет рассмотрена в следующем разделе).

```
int x = 10, y = 5, z = 2;
bool range = x < y && y < z;    // false
```

При использовании проверок на равенство и неравенство следует учитывать особенности типов операндов. В частности, числа с плавающей запятой часто содержат "приблизительные" значения после вычислений (мы рассматривали точность представления *double* и *float* в разделе Вещественные числа). Например, сумма 0.6 и 0.3 не равна строго 0.9:

```
double p = 0.3, q = 0.6;
bool eq = p + q == 0.9;         // false
double diff = p + q - 0.9;      // -0.000000000000000111
```

Разница составляет $1*10^{-16}$, но её достаточно, чтобы операция сравнения дала false.

Поэтому проверки вещественных чисел на равенство и неравенство следует проводить с помощью операций больше/меньше для их разницы и допустимого отклонения, которое подбирается вручную, исходя из особенностей расчетов, или берется универсальное. Напомним, что для *double* и *float* определены встроенные константы точности DBL_EPSILON и FLT_EPSILON, валидные для значения 1.0. Их следует масштабировать для сравнения других величин. В скрипте *ExprRelational.mq5* представлена одна из возможных реализаций функции *isEqual* для сравнения вещественных чисел, которая учитывает этот аспект.

```
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;
}
```

Здесь используются функции получения абсолютного значения без знака (*MathAbs*) и максимального значения из двух (*MathMax*) — они будут описаны в 4-й части в разделе Математические функции. Абсолютная разница между параметрами функции *isEqual* сравнивается с откалиброванным допуском в переменной *eps* с помощью операции '<'.

Эту функцию все равно нельзя использовать для сравнения с абсолютным нулем. Для этих целей можно воспользоваться таким подходом (его, вероятно, потребуется адаптировать под конкретные нужды):

```
bool isZero(const double x)
{
   return MathAbs(x) < DBL_EPSILON;
}
```

Сравнение строк выполняется лексикографически, то есть побуквенно. Код каждого символа сравнивается с кодом символа в той же позиции во второй строке. Сравнение производится до тех пор, пока не обнаружится различие в кодах или одна из строк не закончится. Соотношение строк будет равно соотношению первых отличных символов или более длинная строка будет считаться больше короткой. Напомним, что заглавные и строчные буквы имеют разные коды, и как ни странно "большие" буквы имеют меньшие коды, чем "маленькие".

Пустая строка "" (на самом деле хранит один терминальный символ 0) не равна специальному значению NULL, обозначающему отсутствие строки.

```
bool cmp1 = "abcdef" > "abs";     // false, [2]: 's' > 'c'
bool cmp2 = "abcdef" > "abc";     // true,  по длине
bool cmp3 = "ABCdef" > "abcdef";  // false, по регистру
bool cmp4 = "" == NULL;           // false
```

Кроме того, для сравнения строк MQL5 предоставляет несколько функций, которые будут описаны в разделе Работа со строками.

При проверке равенства и неравенства не рекомендуется использовать константы *bool*: *true* и *false*. Дело в том, что в выражениях вида *v == true* или *v == false* операнд *v* может интуитивно интерпретироваться как логический тип, будучи на самом деле числом. Как известно, для чисел нулевое значение расценивается как *false*, а все остальные — как *true* (это зачастую хочется использовать, как признак наличия какого-то результата или отсутствия). Однако в данном случае приведение типов к общему происходит в обратную сторону: *true* или *false* "расширяются" до числового типа *v* и становятся фактически равны 1 и 0 соответственно. Такое сравнение будет иметь результат, отличный от предполагаемого (например, ложным получится сравнение *100 == true*).

2.5.6 Логические операции

Логические операции производят вычисления над операндами логического типа и возвращают результат того же типа.

П	Символы	Описание	Пример	А
2	!	Логическое НЕ	!e1	R
11	&&	Логическое И	e1 && e2	L
12	||	Логическое ИЛИ	e1 || e2	L

Логическое НЕ обращает *true* в *false*, а *false* — в *true*.

Логическое И равно *true*, если оба операнда равны *true*.

Логическое ИЛИ равно *true*, если хотя бы один операнд равен *true*.

Операторы И и ИЛИ всегда вычисляют операнды слева направо и по возможности применяют сокращение вычислений. Если левый операнд равен *false*, то оператор И пропускает второй операнд, потому что он ни на что не влияет — результат уже точно равен *false*. Если левый операнд равен *true*, то оператор ИЛИ пропускает второй операнд по аналогичной причине, так как результат в любом случае будет равен *true*.

Это часто используется в программах для предотвращения ошибок во втором операнде (и последующих операндах). Например, мы можем обезопасить себя от ошибки обращения к несуществующему элементу массива:

```
index < ArraySize(array) && array[index] != 0
```

Здесь используется встроенная функция *ArraySize*, возвращающая длину массива. Только в том случае, если *index* меньше длины, происходит чтение элемента с этим индексом и его сравнение с нулем.

Обратная по смыслу проверка с помощью '||' также применяется, например:

```
ArraySize(array) == 0 || array[0] == 0
```

Условие сразу выполнится, если массив пуст. И только если есть элементы, продолжится дополнительная проверка на содержимое.

Когда выражение состоит из множества операндов, объединенных логическим ИЛИ, то по первому *true* (если такое найдется) будет сразу получен общий результат *true*. Когда же операнды объединены логическим И, то по первому *false* будет сразу получен общий результат *false*.

Разумеется, можно комбинировать разные операции в одном выражении, учитывая при этом их разный приоритет: отрицание выполняется прежде всего, затем — условия по И, и в конце — условия по ИЛИ. Если требуется другая последовательность, необходимо её явно указывать с помощью скобок.

Например, следующее выражение без скобок — A && B || C && D — фактически эквивалентно: (A && B) || (C && D). Чтобы логическое ИЛИ выполнялось первым, нужно заключить его в скобки: A && (B || C) && D. Более подробно об использовании скобок см. раздел Группировка с помощью круглых скобок.

В скрипте *ExprLogical.mq5* приведены простые примеры для проверки логических операций на практике.

```
int x = 3, y = 4, z = 5;
bool expr1 = x == y && z > 0;  // false, x != y, z не важно
bool expr2 = x != y && z > 0;  // true,  оба условия соблюдены
bool expr3 = x == y || z > 0;  // true,  достаточно того, что z > 0
bool expr4 = !x;               // false, x должен быть 0, чтобы получить true
bool expr5 = x > 0 && y > 0 && z > 0; // true, все 3 выполнены
bool expr6 = x < 0 || y > 0 && z > 0; // true, y и z достаточно
bool expr7 = x < 0 || y < 0 || z > 0; // true, z достаточно
```

В строке с вычислением *expr6* компилятор выдает предупреждение "проверьте приоритеты, возможна ошибка; используйте скобки для уточнения" ("check operator precedence for possible error; use parentheses to clarify precedence").

Не следует путать логические операции '&&' и '||' с побитовыми операциями '&' и '|' (рассматриваются в следующем разделе).

2.5.7 Побитовые операции

Иногда требуется обрабатывать числа на уровне битов. Для этих целей существует группа побитовых операций, которые применимы только к целочисленным типам.

В таблице, в порядке убывания приоритетов, приведены все символы и описания побитовых операторов с указанием ассоциативности.

П	Символы	Описание	Пример	А
2	~	Побитовое отрицание (инверсия)	~e1	R
5	<<	Сдвиг влево	e1 << e2	L
5	>>	Сдвиг вправо	e1 >> e2	L
8	&	Побитовое И	e1 & e2	L
9	^	Побитовое исключающее ИЛИ	e1 ^ e2	L
10	|	Побитовое ИЛИ	e1 | e2	L

Из всей группы только операция побитового отрицания '~' является унарной, все остальные — бинарные.

Во всех случаях, если операнд по размеру меньше *int/uint*, то он предварительно расширяется до *int/uint*-а, путем добавления 0-битов в старшие разряды. В зависимости от знаковости типа операнда старший бит может влиять на знак.

Понять представление чисел на битовом уровне может помочь стандартное приложение Windows Калькулятор. Если в меню Вид выбрать режим работы Программист, в программе появляются группы переключателей для выбора отображения числа в шестнадцатеричной (Hex), десятичной (Dec), восьмеричной (Oct) и бинарной (Bin) форме. Последняя как раз показывает биты. Кроме того, можно выбрать размер числа: 1, 2, 4 или 8 байт. Кнопки позволяют выполнять все рассматриваемые операции: Not ('~'), And ('&'), Or ('|'), Xor ('^'), Lsh ('<<'), Rsh ('>>').

Поскольку Калькулятор использует знаковые числа, при переключении в десятичный режим возможно появление отрицательных значений (напомним, старший бит интерпретируется как знак). Для удобства анализа имеет смысл исключить появление минуса, для чего необходимо выбирать размер в байтах на одну градацию больше. Например, для проверки значений в диапазоне до 255 (uchar, беззнаковое однобайтовое целое), следует выбрать 2 байта (иначе положительными будут только десятичные значения до 127 включительно, а остальные отобразятся в отрицательную область).

Побитовое отрицание создает значение, в котором на месте всех 1-битов стоит 0-бит, а на месте 0-битов — 1-бит. Например, отрицание байта со всеми нулевыми битами, дает байт со всеми единичными битами. Число 50 в побитовом формате выглядит как '00110010' (байт). Его инверсия дает '11001101'.

Единица в шестнадцатеричном представлении — это 0x0001 (для *short*). Инверсия этих битов дает 0xFFFE (см. скрипт *ExprBitwise.mq5*).

```
short v = ~1;  // 0xfffe = -2
ushort w = ~1; // 0xfffe = 65534
```

Побитовое И проверяет каждый бит в обоих операндах, и в тех позициях, где обнаруживаются два взведенных бита (1), сохраняет в результат 1-бит. Во всех остальных случаях (когда взведенный бит есть только в одном операнде или они сброшены и там, и там) в результат записывается 0-бит.

Побитовое ИЛИ записывает в результат 1-биты на тех позициях, на которых взведенный бит есть хотя бы в одном из двух операндов.

Побитовое исключающее ИЛИ записывает в результат 1-биты на тех позициях, на которых взведенный бит есть либо в первом, либо во втором операнде, но не одновременно. Ниже показано бинарное представление двух чисел X и Y и результаты побитовых операций с ними.

```
X        10011010   154
Y        00110111    55

X & Y    00010010    18
X | Y    10111111   191
X ^ Y    10101101   173
```

При написании сложных выражений из нескольких разных операторов используйте группировку с помощью круглых скобок, чтобы не путаться с приоритетами.

Операции сдвига перемещают биты влево ('<<') или вправо ('>>') на количество битов, заданное во втором операнде, который должен быть неотрицательным целым числом. В результате левые биты (для '<<') или правые биты (для '>>') отбрасываются, так как выходят за границы ячейки памяти. При сдвиге влево, справа добавляется соответствующее количество 0-битов. При сдвиге вправо, слева добавляются либо 0-биты (если операнд беззнаковый), либо размножается бит знака (если операнд знаковый). Во втором случае для положительных чисел слева добавляются 0-биты, а для отрицательных — 1-биты, то есть, знак сохраняется.

```
short q = v << 5;  // 0xffc0 = -64
ushort p = w << 5; // 0xffc0 = 65472
short r = q >> 5;  // 0xfffe = -2
ushort s = p >> 5; // 0x07fe = 2046
```

На примере выше первоначальный сдвиг влево "уничтожил" старшие биты переменной *p*, а последующий сдвиг вправо на то же количество битов заполнило их нулями, в результате чего значение уменьшилось с 0xffc0 до 0x07fe.

Размер сдвига (количество битов) должен быть меньше размера типа операнда (с учетом возможного расширения). В противном случае все исходные биты будут потеряны.

Сдвиг на 0 битов оставляет число неизменным.

Не следует путать побитовые операции '&' и '|' с логическими операциями '&&' и '||' (рассмотрены в предыдущем разделе).

2.5.8 Операции модификации

Модификация, называемая также составным присваиванием, позволяет объединить в одном операторе арифметические или побитовые операции с обычным присваиванием.

П	Символы	Описание	Пример	А
14	+=	Сложение с присваиванием	e1 += e2	R
14	-=	Вычитание с присваиванием	e1 -= e2	R
14	*=	Умножение с присваиванием	e1 *= e2	R
14	/=	Деление с присваиванием	e1 /= e2	R
14	%=	Деление по модулю с присваиванием	e1 %= e2	R
14	<<=	Сдвиг влево с присваиванием	e1 <<= e2	R
14	>>=	Сдвиг вправо с присваиванием	e1 >>= e2	R
14	&=	Побитовое И с присваиванием	e1 &= e2	R
14	|=	Побитовое ИЛИ с присваиванием	e1 |= e2	R
14	^=	Побитовое ИИЛИ с присваиванием	e1 ^= e2	R

Данные операторы выполняют соответствующее действие для операндов *e1* и *e2*, после чего результат сохраняется в *e1*.

Выражение вида *e1 @= e2*, где @ — любой оператор из таблицы, примерно эквивалентно *e1 = e1 @ e2*. Слово "примерно" подчеркивает наличие нюансов.

Во-первых, если на месте *e2* стоит выражение с оператором, у которого более низкий приоритет, чем у @, *e2* все равно вычисляется раньше. То есть, если обозначить приоритет скобками, получим *e1 = e1 @ (e2)*.

Во-вторых, если в выражении *e1* присутствуют побочные модификации переменных, они производятся лишь единожды. Это демонстрирует следующий пример.

```
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
                           // не эквивалентно!
b[++j] = b[++j] * (j + 1); // b = {1, 2, 4, 4, 5}, j = 2
```

Здесь массивы *a* и *b* содержат одинаковые элементы и обрабатываются с помощью индексных переменных *i* и *j*. Причем выражение для массива *a* использует операцию '*=', а выражение для массива *b* — "эквивалент". Результаты не равны: отличаются как индексные переменные, так и массивы.

В задачах с манипуляциями на уровне битов пригодятся другие операторы. Так для установки конкретного бита в 1 можно использовать следующее выражение:

```
ushort x = 0;
x |= 1 << 10;
```

Здесь производится сдвиг 1 ('0000 0000 0000 0001') на 10 битов влево, в результате чего получается число с одним взведенным 10-м битом ('0000 0100 0000 0000'). Операция побитового ИЛИ копирует этот бит в переменную *x*.

Для сброса того же бита напишем:

```
x &= ~(1 << 10);
```

Здесь к 1, сдвинутой на 10 битов влево (которую мы видели в предыдущем выражении), применяется операция инверсии, в результате чего все биты меняют свое значение: '1111 1011 1111 1111'. Операция побитового И сбрасывает обнуленные биты (в данном случае один) в переменной *x*, все остальные биты в *x* остаются без изменений.

2.5.9 Условный тернарный оператор

Условный тернарный оператор позволяет описать в едином выражении два варианта вычислений в зависимости от некоторого условия. Синтаксис оператора такой:

```
condition ? expression_true : expression_false
```

Логические условие должно быть указано в первом операнде condition. Это может быть произвольное сочетание операций сравнения и логических операций. Обе ветви обязательно должны присутствовать.

Если условие истинно, будет вычисляться выражение *expression_true*, а если ложно — *expression_false*.

Данный оператор гарантирует, что только одно из выражений *expression_true* и *expression_false* выполнится.

Типы двух выражений должны быть одинаковыми, в противном случае будет предпринята попытка их неявного приведения к общему типу.

Обратите внимание, что результат обработки выражений в MQL5 всегда представляет собой RValue (в C++, если в выражениях стоят только LValue, то и результатом оператора также будет LValue). Так, следующий код компилируется в C++, но выдает ошибку в MQL5:

```
int x1, y1; ++(x1 > y1 ? x1 : y1); // '++' - l-value required
```

Условные операторы могут быть вложенными, то есть в качестве условия или любой из веток (*expression_true* и *expression_false*) разрешено использовать другой условный оператор. При этом визуально может быть не всегда понятно, к чему относятся условия (если не используются круглые скобки для явного обозначения группировки). Рассмотрим примеры из *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)
```

В данном случае первое логическое условие представляет собой сравнение *x > y*. При его истинности выполняется ветвь с переменной *z*. При его ложности проверяется дополнительное логическое условие *p != 0 && q != 0*, также с двумя вариантами выражений.

Ниже приведено еще несколько операторов, в которых заглавными буквами обозначены логические условия, а строчными — варианты расчета. Здесь для простоты сделано, что все они — переменные (из примера выше). В реальности каждая из трех компонент бывает более витиеватым выражением.

Вы можете проследить для каждой строки, как получается результат, показанный в комментарии.

```
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
```

Поскольку оператор имеет свойство правой ассоциативности, составное выражение разбирается справа налево, то есть самая правая конструкция с тремя операндами, объединенными символами '?' и ':', становится операндом внешнего условия, которое записано левее. Далее с учетом этой "подмены", выражение снова разбирается справа налево и так далее, пока не получится последняя полная конструкция '?:' верхнего уровня.

Поэтому вышеприведенные выражения разбиваются на группы следующим образом (скобками обозначена неявная интерпретация компилятора, но такие скобки могли бы быть добавлены в выражения для наглядности исходного кода, и такой подход рекомендуется).

```
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);
```

Для переменной *r5* первое условие *A ? f : h* вычисляет логическое условие для последующего выражения, поэтому преобразуется в *bool*. Поскольку A равно *false*, значение берется из переменной *h*. Она не равна 0, и потому первое условие считается истинным. В результате срабатывает ветка *(B ? x : y)*, из которой возвращается значение переменной *y*, так как B равно *false*.

В операторе обязательно должны присутствовать все 3 компонента (условие и 2 варианта), в противном случае компилятор сгенерирует ошибку "неожиданный токен" ("unexpected token"):

```
// ';' - unexpected token
// ';' - ':' colon sign expected
int r6 = A ? B ? x : y; // не хватает альтернативы
```

"Токен" на языке компилятора — это неделимый фрагмент исходного кода с самостоятельным смыслом или назначением (название типа, идентификатор, символ пунктуации и т.д.). Весь исходный код разбивается компилятором на последовательность токенов. Знаки рассматриваемых операторов тоже токены. В вышеприведенном коде имеется два символа '?' и должно быть два парных им символа ':', однако он только один. Поэтому компилятор "говорит", что символ окончания инструкции ';' преждевременный, и "уточняет", чего именно не хватает: "ожидалось двоеточие" ("colon sign expected").

Из-за того, что условный оператор имеет очень низкий приоритет (13 в полной таблице, см. Приоритеты операций), рекомендуется заключать его в скобки. Так проще избежать ситуаций,

когда операнды условного оператора могли бы быть "захвачены" соседними операциями с более высоким приоритетом. Например, если требуется рассчитать значение некоторой переменной *w* через сумму двух тернарных операторов, прямолинейный подход мог бы выглядеть так:

```
int w = A ? f : h + B ? x : y;                          // 1
```

Это будет работать иначе, чем задумывалось. Сумма *h + B* из-за более высокого приоритета рассматривается как единое выражение. Учитывая разбор справа налево, эта сумма оказывается в роли условия и приводится к типу *bool*, о чем компилятор даже выдает предупреждение "выражение не является логическим" ("expression not boolean"). Трактовку компилятора можно для наглядности выделить скобками:

```
int w = A ? f : ((h + B) ? x : y);                      // 1
```

Для решения проблемы необходимо расставить скобки по-своему.

```
int v = (A ? f : h) + (B ? x : y);                      // 9
```

Глубокая вложенность условных операторов негативно сказывается на понятности кода. Следует избегать уровней вложенности больше двух-трех.

2.5.10 Запятая

Оператор запятая, очевидно обозначаемый ',', ставится между двумя выражениями, которые вычисляются независимо друг от друга слева направо. Иными словами, данный оператор не производит никаких действий сам по себе, а лишь позволяет указать последовательность двух и более выражений в одной инструкции.

Выражения, расположенные правее в последовательности, могут пользоваться результатами расчетов выражений слева, так как они уже обработаны.

Результатом оператора является результат самого правого выражения. Оператор имеет низший приоритет.

В данный момент применение оператора в MQL5 ограничено заголовком цикла for.

Пример:

```
for(i=0,j=99; i<100; i++,j--)
   Print(array[i][j]);
```

Повторим еще раз ключевые аспекты оператора запятая в MQL5:

Порядок вычислений:

- Выражения обрабатываются в порядке слева направо. Таким образом, выражения справа могут использовать результаты выражений слева, поскольку они уже обработаны.

Результат и Приоритет:

- Результатом оператора является значение самого правого выражения. Важно учитывать, что оператор запятая имеет низший приоритет, что означает, что другие операторы в выражении могут иметь более высокий приоритет.

2.5.11 Специальные операторы sizeof и typename

sizeof

Оператор *sizeof* возвращает размер своего операнда в байтах. Синтаксис оператора: *sizeof(x)*, где *x* может быть типом или выражением. Выражение при этом не вычисляется, так как оператор *sizeof* выполняется на стадии компиляции, и вместо него в выражение фактически подставляется константа.

Для массивов с фиксированным размером оператор возвращает общий объем выделенной памяти, то есть произведение количества элементов по всем измерениям на размер типа в байтах. Для динамических массивов возвращается размер внутренней структуры, хранящей свойства массива.

Приведем несколько примеров с пояснениями (*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 (количество элементов)
Print(sizeof(dynamic1));                          // 52
Print(sizeof(dynamic2));                          // 52
```

В комментариях помечен результат, выводимый в журнал.

Тип *double* занимает 8 байт. Размер типа *string* — 12. В этих 12 байтах хранится служебная информация, о которой мы говорили в разделе про тип *string*. Такая память резервируется под любую строку (даже неинициализированную). Обратите внимание, что строка с текстом длиной 29 букв, тоже имеет размер 12. Так происходит, потому что внутренняя структура, предназначенная для хранения ссылки на память, имеется как у пустой строки, так и у строки с содержимым. Для получения длины текста следует использовать функцию *StringLen*.

Размер массива фиксированного размера действительно рассчитан как произведение количества элементов (2*2=4) на размер типа *double* (8) — итого 32. И как следствие, выражение вида *sizeof(array) / sizeof(double)* позволяет узнать количество элементов в нем.

Для динамических массивов размер внутренней структуры равен 52 байтам. Различия в описаниях массивов *dynamic1* и *dynamic2* никак не сказываются на этой величине.

Оператор *sizeof* особенно полезен для получения размера классов и структур.

typename

Оператор *typename* возвращает строку с названием типа переданного ей параметра, который может быть типом или выражением. Для массивов, помимо ключевого слова типа данных, выводится признак в виде пары скобок (или нескольких, в зависимости от размерности массива).

```
Print(typename(double));                      // double
Print(typename(array));                       // double [2][2]
Print(typename(dynamic1));                    // double [][1]
Print(typename(1 + 2));                       // int
```

Для пользовательских типов, таких как классы, структуры и другие (которые мы изучим в 3-ей Части), имя типа предваряется категорией сущности: например, "class MyCustomType". Кроме того, для констант в строковое описание будет добавлен модификатор "const".

Поэтому, чтобы узнать краткое имя типа, состоящее из одного слова, воспользуйтесь макросом TYPENAME из прилагаемого файла *TypeName.mqh*.

Узнать имя типа бывает необходимо в так называемых шаблонах, которые способны генерировать из исходного кода похожие реализации для разных типов, задаваемых в параметрах шаблонов.

2.5.12 Группировка с помощью круглых скобок

В предыдущих разделах мы уже не раз видели, что некоторые выражения могут вызывать неожиданные результаты из-за приоритетов операций. Для явного изменения порядка расчета необходимо пользоваться круглыми скобками. Заключенная в них часть выражения получает повышенный приоритет по сравнению с окружением, невзирая на приоритеты по умолчанию. Пары скобок могут быть вложенными, однако делать уровень вложенности больше 3-4 не рекомендуется. Слишком сложные выражения лучше разделить на несколько отдельных.

В скрипте *ExprParentheses.mq5* показана эволюция расстановки скобок в одном выражении. Исходный замысел для него в том, чтобы взвести бит в переменной *flags* с помощью операции сдвига влево '<<'. Номер бита берется из переменной *offset*, если она не нулевая, или равным 1 в противном случае (напомним, нумерация идет с нуля). Затем полученное значение умножается на *coefficient*. Прикладного смысла в этом примере искать не стоит, но на практике встречаются и более хитрые конструкции.

```
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
```

Первый вариант без скобок кажется подозрительным даже компилятору. Тот выдает уже знакомое нам предупреждение "выражение не является логическим" ("expression not boolean"). Дело в том, что минимальным приоритетом из всех операторов здесь обладает тернарный условный. Из-за этого вся левая часть до знака '?' считается его условием. Внутри условия вычисления идут в следующем порядке: умножение, побитовый сдвиг, сравнение "больше", побитовое ИЛИ, в результате чего получается целое число. Оно, конечно, может использоваться как *true* или *false*, но такие намерения желательно всегда "сообщать" компилятору с помощью явного приведения типов. В его отсутствии компилятор считает выражение подозрительным, и не зря. Результат первого расчета равен 8. Он неверный.

Добавим скобки вокруг тернарного оператора. Предупреждение компилятора пропадет. Однако выражение по-прежнему считается неправильно. Поскольку приоритет умножения выше побитового ИЛИ, переменные *coefficient* и *flags* перемножаются до того, как применяется битовая маска, получаемая сдвигом влево. Результат равен 256.

Наконец, добавив еще одни скобки, мы получим правильный результат 2560.

2.5.13 Приоритеты операций

Мы приводим полную таблицу всех операций в порядке приоритетов.

П	Символы	Описание	Пример	А
0	::	Разрешение контекста	n1 :: n2	L
1	()	Группировка	(e1)	L
1	[]	Индекс	[e1]	L
1	.	Разыменование	n1.n2	L
1	++	Постфикс инкремент	e1++	L
1	--	Постфикс декремент	e1--	L
2	!	Логическое НЕ	!e1	R
2	~	Побитовое отрицание (инверсия)	~e1	R
2	+	Унарный плюс	+e1	R
2	-	Унарный минус	-e1	R
2	++	Префикс инкремент	++e1	R
2	--	Префикс декремент	--e1	R
2	(type)	Приведение типа	(n1)	R
2	&	Взятие адреса	&n1	R
3	*	Умножение	e1 * e2	L
3	/	Деление	e1 / e2	L
3	%	Деление по модулю	e1 % e2	L
4	+	Сложение	e1 + e2	L
4	-	Вычитание	e1 - e2	L
5	<<	Сдвиг влево	e1 << e2	L
5	>>	Сдвиг вправо	e1 >> e2	L
6	<	Меньше	e1 < e2	L
6	>	Больше	e1 > e2	L
6	<=	Меньше или равно	e1 <= e2	L
6	>=	Больше или равно	e1 >= e2	L
7	==	Равно	e1 == e2	L

П	Символы	Описание	Пример	А
7	!=	Не равно	e1 != e2	L
8	&	Побитовое И	e1 & e2	L
9	^	Побитовое исключающее ИЛИ	e1 ^ e2	L
10	|	Побитовое ИЛИ	e1 | e2	L
11	&&	Логическое И	e1 && e2	L
12	||	Логическое ИЛИ	e1 || e2	L
13	?:	Условный тернарный	c1 ? e1 : e2	R
14	=	Присваивание	e1 = e2	R
14	+=	Сложение с присваиванием	e1 += e2	R
14	-=	Вычитание с присваиванием	e1 -= e2	R
14	*=	Умножение с присваиванием	e1 *= e2	R
14	/=	Деление с присваиванием	e1 /= e2	R
14	%=	Деление по модулю с присваиванием	e1 %= e2	R
14	<<=	Сдвиг влево с присваиванием	e1 <<= e2	R
14	>>=	Сдвиг вправо с присваиванием	e1 >>= e2	R
14	&=	Побитовое И с присваиванием	e1 &= e2	R
14	|=	Побитовое ИЛИ с присваиванием	e1 |= e2	R
14	^=	Побитовое И/ИЛИ с присваиванием	e1 ^= e2	R
15	,	Запятая	e1 , e2	L

Квадратные скобки, как мы видели, используются для указания индексов элементов массивов, и потому имеют один из высших приоритетов.

Помимо операторов, которые были рассмотрены ранее, здесь также имеется несколько пока неизвестных.

Оператор разрешения контекста '::' мы изучим в рамках объектно-ориентированного программирования (ООП). Тогда же нам станет необходим и оператор разыменования '.'. Их операндами выступают не выражения, а идентификаторы типов (классов) и их свойств.

Оператор взятия адреса '&' предназначен для передачи параметров функций по ссылке и получения адреса объекта в ООП. В обоих случаях оператор применяется к переменной (LValue).

Об операции явного приведения типов будет рассказано в следующей главе.

2.6 Приведение типов

В данном разделе мы рассмотрим концепцию приведения типов, ограничившись пока встроенными типами данных. Позднее, после изучения ООП, мы дополним её нюансами, свойственными объектным типам.

Приведение типов в MQL5 — это процесс изменения типа данных переменной или выражения. В языке MQL5 поддерживаются три основных вида приведения типов: неявное, арифметическое и явное.

Неявное приведение типов:

- Происходит автоматически в случаях, когда переменная одного типа используется в контексте, ожидающем другой тип. Например, целочисленные значения могут неявно приводиться к вещественным.

Арифметическое приведение типов:

- Возникает при выполнении арифметических операций с операндами разных типов. Компилятор пытается сохранить максимальную точность, но предупреждает о потенциальных потерях данных. Например, при делении целых чисел результат приводится к вещественному типу.

Явное приведение типов:

- Предоставляет программисту контроль над приведением типов. Используется двумя формами записи: C-стиль ((target)) и "функциональный" стиль (target()). Применяется, когда требуется явно указать компилятору на конвертацию между типами, например, при округлении вещественных чисел или при необходимости последовательных приведений типов.

Приведение типов важно для обеспечения корректного выполнения операций и избегания потерь данных. Понимание различий между неявным, арифметическим и явным приведением типов помогает программистам эффективно использовать этот механизм в разработке на MQL5.

2.6.1. Неявное приведение типов

Приведение типов происходит автоматически, если в некотором месте исходного кода используется один тип, но ожидается другой, и существуют правила конвертации между ними. Такое приведение называется неявным и может не всегда отвечать замыслу программиста. Кроме того, некоторые операции конверсии имеют побочные эффекты, и компилятор, не зная, является ли их применение намеренным, выделяет соответствующие строки кода предупреждениями. Для решения этих проблем существует синтаксис явного приведения типов (см. далее Явное приведение типов).

Некоторые из правил неявного приведения типов мы уже встречали при изучении типов и переменных.

В частности, если значение типа, отличного от логического, присваивается переменной *bool*, то значение 0 расценивается как *false*, а всё остальное — как *true*. В более общем случае, все выражения, которые предполагают наличие логических условий, приводятся к типу *bool*. Например, первый операнд тернарного условного оператора всегда конвертируется в *bool*.

Если же значение типа *bool* присваивается числовому типу, то *true* становится равно 1, а *false* — 0.

Когда вещественное число присваивается переменной целого типа, дробная часть отбрасывается (компилятор выдает предупреждение). Когда целое число, наоборот, присваивается переменной вещественного типа, возможна потеря точности (компилятор также выдает предупреждение). Об этом мы уже говорили в разделах про Целые числа и Вещественные числа.

При наличии целых чисел и чисел с плавающей запятой, всё приводится к числам с плавающей запятой максимального использованного размера (обычно *double*, если нет явного указания *float* или в числовом литерале не стоит суффикс 'f', например — 1234.56789f).

С целыми разных размеров также существуют правила конвертации: они при необходимости расширяются, то есть увеличиваются до размера максимально большого целого типа, использованного в выражении (см. Арифметические преобразования типов).

Помимо выражений потребность в неявном приведении типов часто возникает при инициализации и присваивании, когда типы справа и слева от знака '=' не совпадают. Такие же правила конвертации применяются и при передаче значений через параметры функций и возврате результатов из функций (подробнее об этом в разделе про функции).

Учитывая вышеизложенное, в одной строке кода может выполняться большое количество преобразований. Если при этом возникают предупреждения компилятора, желательно убедиться, что конвертация сделана намеренно, и ликвидировать предупреждения, вставив явное приведение типов.

```
short s = 10;
long n = 10;
int p = s * n + 1.0;
```

В этом примере при выполнении умножения тип переменной *s* расширяется до типа второго операнда *long* и получается промежуточный результат типа *long*. Поскольку константа 1.0 имеет тип *double*, результат произведения конвертируется перед сложением в *double*. Общий результат также имеет тип *double*, однако переменная *p* имеет тип *int*, и потому выполняется неявное приведение типа из *double* в *int*.

Специальные типы *datetime* и *color* обрабатываются по правилам целых длиной 8 и 4 байта соответственно. Но для даты и времени существует более строгое ограничение на максимальную величину — 32535244799, что соответствует D'3000.12.31 23:59:59'.

Большинство типов может неявно конвертироваться в строки и обратно, но результаты не всегда адекватны, поэтому компилятор выдает предупреждения "неявное преобразование числа в строку" и "неявное преобразование строки в число" ("implicit conversion from 'number' to 'string'", "implicit conversion from 'string' to 'number'"), чтобы программист их проверил. Например, преобразование строки в целое число допускает наличие в строке только цифр и знаков '+'/'-' в начале. Конвертация из строки в вещественное — разрешает помимо цифр наличие точки '.' и формы записи с "экспонентой" ('e' или 'E', например, +1.2345e-1). Если в строке встретится посторонний символ (например, буква), весь остаток строки отбрасывается.

Например, дату и время в виде строки ("2021.12.12 00:00") не удастся без потерь присвоить переменной типа *datetime*, поскольку *datetime* — это целое число (количество секунд). И чтение числа из строки завершится по достижении первой точки, то есть число получит значение 2021. Это количество секунд соответствует 34-й минуте 1970-го года.

Для подобных преобразований существуют специальные функции (см. раздел Преобразование данных).

Единственное направление неявного и явного приведения типов, которое запрещено, это из *string* в *bool*. Компилятор в таких случаях выдает ошибку "нельзя неявно преобразовать 'string' в 'bool'" ("cannot implicitly convert type 'string' to 'bool'").

Примеры из данной главы можно найти в скрипте *TypeConversion.mq5*.

2.6.2. Арифметические преобразования типов

В выражениях арифметических вычислений и сравнений часто случается ситуация, когда в качестве операндов задействованы величины разных типов. Для их правильной обработки необходимо привести типы в некоему "общему знаменателю". Компилятор пытается сделать это без участия программиста, если тот не указал явных правил преобразования (см. Явное приведение типов). При этом компилятор, по возможности, пытается сохранить максимальную точность, когда речь идет о числах. В частности, он производит увеличение разрядности целых чисел и переход от целых чисел к вещественным (если они задействованы).

Целочисленное расширение подразумевает преобразование *bool*, *char*, *unsigned char*, *short*, *unsigned short* к *int* (или *unsigned int*, если *int* недостаточно для хранения конкретных чисел). Большие значения могут преобразовываться к *long* и *unsigned long*.

Если тип переменной не способен сохранить результат того типа, который получился при вычислении выражения, компилятор выдаст предупреждение:

```
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
```

Выражение для инициализации переменных *x* и *y* содержит вещественное число 1.0, поэтому другие операнды (в данном случае, константы 10) преобразуются в *double*, и результат деления также будет типа *double*. Однако тип переменных — *int*, и потому происходит неявное преобразование к нему.

Вычисление 1.0 / 10 компилятор выполняет во время компиляции и потому получает константу типа *double* (0.1). Конечно, на практике маловероятен такой код, чтобы инициализирующая константа превышала размер приемной переменной. Поэтому предупреждение компилятора "усечение константы" ("truncation of constant value") можно считать экзотическим. Просто оно демонстрирует проблему в наиболее упрощенном виде.

Однако в результате вычислений на основе переменных аналогичная потеря данных также может произойти. Второе предупреждение компилятора, которое мы здесь видим ("возможна потеря точности из-за конвертации типов" — "possible loss of data due to type conversion"), возникает гораздо чаще. Причем потеря возможна не только при приведении из вещественного типа в целый, но и обратно.

```
double f = LONG_MAX; // truncation of constant value
long m1 = 1000000000;
f = m1 * m1;         // possible loss of data due to type conversion
```

Как мы знаем, тип *double* не может точно представить большие целые числа (несмотря на то, что диапазон его допустимых значений гораздо больше *long*).

Еще одно предупреждение, с которым мы можем столкнуться из-за несоответствия типов: "переполнение целой константы" ("integral constant overflow").

```
long m1 = 1000000000;
long m2 = m1 * m1;               // ok: m2 = 1000000000000000000
long m3 = 1000000000 * 1000000000; // integral constant overflow
                                 // m3 = -1486618624
```

Целочисленные константы имеют в MQL5 тип *int*, поэтому умножение миллиона на миллион выполняется с учетом диапазона этого типа, а он равен INT_MAX (2147483647). Значение 1000000000000000000 вызывает переполнение, и в *m3* попадает остаток от деления этого значения на диапазон (подробнее об этом — во врезке ниже).

То, что приемная переменная *m3* имеет тип *long* — не означает, что значения в выражении должны заранее приводиться к нему. Это происходит только в момент присваивания. Для того чтобы умножение выполнялось по правилам long, требуется каким-либо образом указать тип *long* непосредственно в самом выражении. Это можно сделать с помощью явного приведения или путем использования переменных. В частности, получение того же произведения с помощью переменной *m1* типа *long* (как *m1 * m1*) приводит к правильному результату в *m2*.

Знаковые и беззнаковые целые

Программы не всегда пишутся идеально, с защитой от всех возможных сбоев. Поэтому иногда бывает, что получившееся в ходе вычислений целое число не умещается в переменную выбранного целого типа. Тогда в неё попадает остаток от деления этого значения на максимальную величину (M), которую можно записать в соответствующее количество байтов (размер типа), плюс 1. Так для целых типов размером от 1 до 4 байтов M+1 составляет, соответственно, 256, 65536, 4294967296 и 18446744073709551616.

Но для знаковых типов есть нюанс. Как мы знаем, у знаковых чисел общий диапазон значений поделен примерно поровну между положительной и отрицательной областями. Поэтому новое "остаточное" значение может в 50% случаев превысить положительный или отрицательный лимит. В таком случае число превращается в "противоположное": у него меняется знак и оно оказывается на удалении M от исходного.

Важно понимать, что это превращение происходит только из-за разной интерпретации состояния битов во внутреннем представлении, а само состояние одно и то же для знаковых и беззнаковых чисел.

Поясним на примере для самых маленьких целых типов: *char* и *uchar*.

Так как *unsigned char* способен хранить значения от 0 до 255, то 256 отображается в 0, -1 — в 255, 300 — в 44, и так далее. При попытке записать 300 в обычный знаковый *char* мы также получим 44, потому что 44 лежит в диапазоне от 0 до 127 (положительная область значений *char*). Однако если присвоить переменным *char* и *uchar* значение 3000, картина получится разная. Остаток от деления 3000 на 256 равен 184. Он попадает в *uchar* без изменений. Однако для *char* точно такое же сочетание битов дает -72. Нетрудно проверить, что 184 и -72 отличаются на 256.

В следующем примере легко выявить проблему благодаря предупреждению компилятора.

```
char c = 3000;       // truncation of constant value
Print(c);            // -72
uchar uc = 3000;     // truncation of constant value
Print(uc);           // 184
```

Однако если получить сверхбольшое число в ходе вычислений, предупреждения уже не будет.

```
char c55 = 55;
char sm = c55 * c55;  // ok!
Print(sm);            // 3025 -> -47
uchar um = c55 * c55; // ok!
Print(um);            // 3025 -> 209
```

Похожий эффект может происходить, когда знаковое и беззнаковое целое одинакового размера используются в одном выражении, поскольку знаковый операнд преобразуется к беззнаковому. Например:

```
uint u = 11;
int i = -49;
Print(i + i); // -98
Print(u + i); // 4294967258 = 4294967296 - 38
```

Когда суммируются два отрицательных целых, мы получаем ожидаемый результат. Во втором выражении сумма -38 отображается в "противоположное" беззнаковое число 4294967258.

> Смешивать знаковые и беззнаковые типы в одном выражении не рекомендуется из-за подобных потенциальных проблем.

Кроме того, если мы вычитаем что-то из беззнакового целого, мы должны быть уверены, что результат не получится отрицательным. В противном случае он будет преобразован в положительное число и способен исказить идею алгоритма, в частности, цикла *while* с проверкой переменной на условие "больше или равно нулю": так как беззнаковые числа всегда неотрицательные, легко получить бесконечный цикл, то есть зависание программы.

2.6.3. Явное приведение типов

Для явного приведения типов в MQL5 поддерживается две формы записи: в стиле языка C и "функциональный". C-стиль имеет следующий синтаксис:

```
target t = (target)s;
```

Здесь *target* — это имя целевого типа. В качестве источника данных *s* может выступать любое выражение. Если в нем выполняются какие-то операции, необходимо заключить выражение в скобки, чтобы приведение типа применялось ко всему выражению.

Альтернативный "функциональный" синтаксис выглядит так:

```
target t = target(s);
```

Рассмотрим пару примеров.

```
double w = 100.0, v = 7.0;
int p = (int)(w / v);      // 14
```

Здесь результат деления двух вещественных чисел явно приводится к типу *int*. Тем самым программист подтверждает свое намерение отбросить дробную часть, и компилятор не будет

выдавать предупреждений. Стоит отметить, что в MQL5 имеется группа функций для округления вещественных чисел различными способами (см. Математические функции).

Если наоборот требуется выполнить операцию над целыми числами с получением вещественного результата, нужно применить приведение типа к операндам (в самом выражении):

```
int x = 100, y = 7;
double d = (double)x / y;  // 14.28571428571429
```

Приведения одного из операндов достаточно, чтобы остальные автоматически конвертировались в тот же тип.

При необходимости можно выполнять несколько приведений типа последовательно. Поскольку операция приведения типа обладает правой ассоциативностью, целевые типы будут применяться в порядке справа налево. В следующем примере мы преобразуем частное к типу *float* (это преобразование позволяет более компактно, с меньшим количеством знаков, отобразить значение), а затем к *string*. Без явного приведения к *string*, мы получили бы предупреждение компилятора "неявное преобразование числа в строку".

```
Print("Result:" + (string)(float)(w / v)); // Result:14.28571
```

Не используйте явное приведение типов только для того, чтобы подавить предупреждение компилятора. Если это не имеет под собой прикладного основания, вы тем самым маскируете потенциальную ошибку в программе.

2.7 Инструкции

К данному моменту мы изучили типы данных, описание переменных и их использование в выражениях для вычислений. Однако это лишь маленькие кирпичики в том здании, с которым можно сравнить программу. Даже самая простая программа состоит из более крупных блоков, позволяющих группировать связанные друг с другом операции по обработке данных и управлять последовательностью их выполнения. Такие блоки называются инструкциями, и мы, на самом деле, уже использовали некоторые из них.

В частности, объявление переменной (или нескольких переменных) — это инструкция. Присваивание переменной результата вычисления некоторого выражения — тоже инструкция. Строго говоря, в состав выражения входит и сама операция присваивания, поэтому такую инструкцию корректнее называть инструкцией-выражением. Между прочим, выражение может не иметь в своем составе операции присваивания (например, если в нем просто вызывается некая функция, не возвращающая значение, вроде *Print("Hello");*).

Выполнение программы представляет собой поступательное выполнение инструкций: сверху вниз и слева направо (если в одной строке несколько инструкций). В простейшем случае их последовательность выполняется линейно, одна за другой. Для большинства программ этого недостаточно, поэтому существуют различные управляющие инструкции. Они позволяют организовать в программах циклы (повторяющиеся расчеты) и выбор вариантов работы алгоритма в зависимости от условий.

Инструкции представляют собой особые синтаксические построения, то есть исходный текст, написанный по правилам. Для инструкций конкретного вида — свои правила, но есть и кое-что общее. Инструкции всех видов оканчиваются символом ';', за исключением одной — составной инструкции. Она "обходится" без точки с запятой, потому что её начало и конец задает пара

фигурных скобок. Важно отметить, что благодаря составной инструкции мы можем включать наборы инструкций внутрь других инструкций, выстраивая произвольные иерархические структуры алгоритмов.

В данной главе мы познакомимся со всеми видами управляющих инструкций MQL5, а также закрепим особенности инструкций объявления и выражений.

В русскоязычной компьютерной литературе существует распространенное альтернативное название для инструкций — операторы. Мы будем использовать его как синоним, но лишь в тех случаях, когда это не вызывает ассоциаций с операциями.

2.7.1 Составные инструкции (блоки кода)

Составная инструкция представляет собой универсальный контейнер для других инструкций, заключенных в фигурные скобки '{' и '}'. Такой блок кода может использоваться для определения тела функции, после заголовка других инструкций управления, если в них требуется указать больше одной контролируемой инструкции, а также просто как самостоятельный вложенный блок внутри тела функции или другой инструкции. Это позволяет создать локальную, ограниченную область для переменных. Мы уже рассказывали об этом в разделе Контекст, область видимости и время жизни переменных.

В обобщенном виде составную инструкцию можно описать так:

```
{
   [инструкции]
}
```

В таком схематическом описании любой фрагмент, заключенный в полукруглые скобки и с верхним индексом ^{opt}, обозначает, что он необязателен. В данном случае, внутри блока может не быть ни одной вложенной инструкции.

В следующих разделах мы увидим, как составные инструкции применяются в комбинации с другими видами инструкций и что могут содержать.

Есть одна тонкость, которую стоит подчеркнуть: после описания составной инструкции точка с запятой ';' не требуется. Это отличает её от всех остальных инструкций.

2.7.2 Инструкции объявления/определения

Объявление переменной, массива, функции или любого другого именованного элемента программы (включая структуры и классы, которые будут рассмотрены в 3-ей части) является инструкцией.

Объявление обязательно содержит тип и идентификатор элемента (см. Объявление и определение переменных), а также опциональное начальное значение для инициализации. Также при объявлении могут указываться дополнительные модификаторы, меняющие те или иные характеристики элемента. В частности, мы уже знаем модификаторы *static* и *const*, но скоро их число пополнится. Для массивов мы должны дополнительно указать размерность и количество элементов (см. Описание массивов), для функций — список параметров (подробнее об этом — в разделе про функции).

Инструкцию объявления переменной можно в обобщенном виде представить следующим образом:

```
[модификаторы] тип идентификатор
    [= выражение_инициализации];
```

Для массива она выглядит так:

```
[модификаторы] тип идентификатор [ [размер_1]] [ [размер_N] ](3)
    [ = { список_инициализации } ] ;
```

Основное отличие — обязательное наличие хотя бы одной пары квадратных скобок (внутри них может быть указан размер или нет, в зависимости от чего мы получим фиксированный или динамически распределяемый массив). Всего разрешено использовать до 4-х пар квадратных скобок (4 — максимальное поддерживаемое количество измерений).

Во многих случаях объявление одновременно может выступать в качестве определения, то есть оно резервирует под элемент память, определяет его поведение и делает возможным использование в программе. Как раз объявление переменной или массива является также и определением. С этой точки зрения инструкцию объявления можно в равной степени называть инструкцией определения, однако это не стало устоявшейся практикой.

Наших базовых познаний в функциях достаточно, чтобы достоверно предположить, как должно выглядеть и их определение:

```
тип идентификатор ( [список_аргументов] )
{
    [инструкции]
}
```

Тип, идентификатор и список аргументов составляют заголовок функции.

Обратите внимание, что это именно определение, поскольку данное описание содержит как внешние атрибуты функции (интерфейс), так и инструкции, задающие её внутреннюю суть (реализацию). Последнее делается с помощью блока кода, сформированного парой фигурных скобок и идущего сразу следом за заголовком функции. Как не трудно догадаться, это и есть один из примеров составной инструкции, которую мы упоминали в предыдущем разделе. В данном случае не обойтись без терминологической тавтологии, поскольку она совершенно обоснована: составная инструкция входит в состав инструкции определения функции.

Чуть позже мы узнаем, для чего и каким образом можно отделить описание интерфейса от реализации и тем самым добиться объявления функции без её определения. Также мы продемонстрируем разницу между объявлением и определением на примере классов.

Инструкция объявления делает доступным новый элемент по его имени в контексте того блока кода (см. Контекст, область видимости и время жизни переменных), внутри которого эта инструкция находится. Напомним, что блоки формируют локальную область видимости объектов (переменных, массивов). В первой части книги мы сталкивались с этим при описании функции приветствия.

Кроме локальных областей всегда существует глобальная область видимости, в которой тоже можно использовать инструкции объявления для создания элементов, доступных из любого места программы.

Если в инструкции объявления отсутствует модификатор *static* и она находится в каком-то локальном блоке, то соответствующий элемент создается и инициализируется в момент выполнения инструкции (строго говоря, память под все локальные переменные внутри функции

выделяется, в угоду эффективности, сразу при входе в функцию, но они в этот момент еще не сформированы).

Например, следующее описание переменной *i* в начале функции *OnStart* гарантирует, что такая переменная будет создана с указанным начальным значением (0), как только функция получит управление (т.е. терминал вызовет её, потому что это главная функция скрипта).

```
void OnStart()
{
   int i = 0;
   Print(i);
   
   // error: 'j' - undeclared identifier
   // Print(j);
   int j = 1;
}
```

Благодаря декларации в первой инструкции, переменная *i* известна и доступна в последующих строках функции, в частности, во второй строке с вызовом функции *Print*, которая выводит содержимое переменной в журнал.

Переменная *j*, описанная в последней строке функции, будет создана непосредственно перед завершением функции (это, конечно, бессмысленно, но наглядно). Поэтому эта переменная не известна во всех более ранних строках этой функции. Попытка вывести *j* в журнал с помощью закомментированного вызова *Print* закончится ошибкой компиляции "неизвестный идентификатор" ("undeclared identifier").

Элементы, объявленные подобным образом (внутри блоков кода и без модификатора *static*) называются автоматическими, потому что программа сама выделяет под них память при заходе в блок и уничтожает их при выходе из блока (в нашем случае, после выхода из функции). Поэтому участок памяти, в котором это происходит, называется стеком ("положил сверху — взял сверху").

Создание автоматических элементов происходит в порядке исполнения инструкций объявления (сначала *i*, потом *j*). Уничтожение выполняется в обратном порядке (сначала *j*, потом *i*).

Если переменная описана без инициализации и начинает использоваться в последующих инструкция (например, справа от знака '=') без предварительной записи в неё осмысленного значения, компилятор выдает предупреждение: "вероятно используется неинициализированная переменная" ("possible use of uninitialized variable").

```
void OnStart()
{
   int i, p;
   i = p; // warning: possible use of uninitialized variable 'p'
}
```

Если в инструкции объявления имеется модификатор *static*, соответствующий элемент создается единожды при первом выполнении инструкции и остается в памяти, невзирая на выход и возможные последующие входы и выходы в тот же блок кода. Все такие статические элементы удаляются только при выгрузке программы.

Несмотря на увеличенный срок жизни, область видимости таких переменных по-прежнему ограничена локальным контекстом, в котором они определены, и обращаться к ним можно только из более поздних инструкций (расположенных ниже по коду).

В отличие от этого инструкции объявления в глобальном контексте создают свои элементы в порядке упоминания в исходном коде, сразу после загрузки программы (до вызова какой-либо стандартной функции запуска, такой как *OnStart* для скриптов). Удаляются глобальные объекты в обратном порядке при выгрузке программы.

Для демонстрации изложенного создадим более "хитрый" пример (*StmtDeclaration.mq5*). Вспомнив навыки, полученные в первой части, напишем в дополнение к *OnStart* простую функцию *Init*, которая будет использована в выражениях инициализации переменных и выведет в журнал последовательность вызовов.

```
int Init(const int v)
{
   Print("Init: ", v);
   return v;
}
```

Функция *Init* принимает единственный параметр *v* целого типа *int*, значение которого возвращает в вызывающий код (инструкция *return*).

Это позволяет использовать её как обёртку для установки начального значения переменной, например, для двух глобальных переменных:

```
int k = Init(-1);
int m = Init(-2);
```

Значение переданного аргумента попадает в переменные *k* и *m* посредством вызова функции и возврата из неё. Однако внутри *Init* мы дополнительно выводим значение с помощью *Print*, и таким образом сможем отследить, как создаются переменные.

Обратите внимание, что мы не можем использовать функцию *Init* в инициализации глобальных переменных выше её определения. Если попробовать перенести декларацию переменной *k* над описание *Init*, получим ошибку "'Init' - неизвестный идентификатор". Это ограничение срабатывает только для инициализации глобальных переменных, потому что функции тоже определяются глобально, а компилятор строит список таких идентификаторов за один проход. Во всех остальных случаях порядок определения функций в коде не важен, потому что компилятор сначала их все регистрирует во внутреннем списке, а потом взаимно увязывает их вызовы из блоков. В частности, вы можете перенести всю функцию *Init* и декларацию глобальных переменных *k* и *m* ниже функции *OnStart* — это ничего не сломает.

Внутри функции *OnStart* опишем с использованием *Init* еще несколько переменных: локальные *i* и *j*, а также статическую *n*. Для простоты все переменные получают уникальные значения, чтобы их можно было отличить.

```
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);
}
```

Здесь также закомментированы ошибочные попытки обратиться к соответствующим переменным до их определения.

Запустим скрипт и получим следующий журнал:

```
Init: -1
Init: -2
-1
Init: 1
1
Init: 0
Init: 2
2
0
```

Как мы видим, инициализация глобальных переменных произошла до вызова функции *OnStart*, и именно в том порядке, в котором они встретились в коде. Внутренние переменные создавались в той же последовательности, в какой записаны инструкции их объявления.

Если переменная определена, но нигде не используется, компилятор выдаст предупреждение "переменная 'имя' не используется" ("variable 'name' not used"). Это признак потенциальной ошибки программиста.

Забегая вперед, скажем, что с помощью инструкций объявления/определения в программу могут вводиться не только элементы данных (переменные, массивы) или функции, но и новые пользовательские типы (структуры, классы, шаблоны, пространства имен), которые нам еще не известны. Такие инструкции можно делать только на глобальном уровне, то есть вне всех функций.

Определить функцию внутри функции также нельзя. Следующий код не скомпилируется:

```
void OnStart()
{
   int Init(const int v)
   {
      Print("Init: ", v);
      return v;
   }
   int i = 0;
}
```

Компилятор сгенерирует ошибку: "определение функций разрешено только на глобальном уровне, в классах или пространствах имен" ("function declarations are allowed on global, namespace or class scope only").

2.7.3 Простые инструкции (выражения)

Простые инструкции содержат выражения, в частности, присваивание переменным новых значений или результатов вычислений, а также вызовы функций.

Формально синтаксис выглядит так:

```
выражение ;
```

Здесь важно наличие точки с запятой в конце. Поскольку исходные коды MQL5 поддерживают свободное форматирование, символ ';' является единственным разделителем, который сообщает компилятору, где закончилась предыдущая инструкция и началась следующая. Как правило, инструкции пишутся на отдельных строках, например, так:

```
int i = 0, j = 1, k;    // инструкция объявления
++i;                    // простая инструкция
j += i;                 // простая инструкция
k = (i + 1) * (j + 1);  // простая инструкция
Print(i, " ", j);       // простая инструкция
```

Однако правила не запрещают написать код компактно:

```
int i=0,j=1;++i;j+=i;k=(i+1)*(j+1);Print(i," ",j);
```

Если бы не ';', соседние выражения могли бы втихую "склеиваться" и приводить к непреднамеренным результатам. Например, выражение *x = y - 10 * z* вполне могло бы быть двумя: *x = y;* и *-10 * z;* (-10 с унарным минусом). Как такое возможно?

Дело в том, что синтаксически допустимо написать инструкцию, которая фактически работает впустую, то есть не сохраняет результат. Вот еще один пример:

```
i + j; // warning: expression has no effect
```

Компилятор при этом выдает предупреждение "выражение бесполезно" ("expression has no effect"). Возможность конструировать подобные выражения нужна потому, что объектные типы, которые мы освоим в Части 3, позволяют осуществлять перегрузку операций, то есть подменять привычный смысл символов-операторов некими специфическими действиями. Тогда, если тип *i* и *j* будет не *int*, а каким-то классом с переопределенной операцией сложения, такая запись будет иметь эффект, и компилятор не станет выдавать предупреждение.

Простые инструкции можно писать только внутри составных инструкций. Например, вызвать функцию *Print* за пределами какой-либо функции не получится:

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

Мы получим целый град ошибок:

```
'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
```

Самая подходящая, в данном случае, — последняя: "выражения недопустимы в глобальном контексте".

2.7.4 Обзор управляющих инструкций

Управляющие инструкции предназначены для организации нелинейного выполнения других инструкций, включая объявления, выражения и вложенные управляющие инструкции. Их можно условно разделить на 3 вида:

- инструкции повторения, или циклы;
- условные инструкции для выбора одной из нескольких веток альтернативных действий;
- инструкции перехода, меняющие при необходимости стандартное поведение инструкций первых двух видов.

Инструкции повторения и выбора состоят из заголовка (у каждого — свой синтаксис) и последующей управляемой инструкции. Если в управляемой части необходимо указать несколько инструкций, в ней используется составная инструкция. Эта особенность недоступна для инструкций перехода. Согласно своему названию, они лишь перемещают внутренний указатель, по которому программа определяет, какую именно инструкцию в данный момент надо выполнить, по особым правилам, которые мы рассмотрим в следующих разделах.

В простейшем случае, без управляющих инструкций, операторы выполняются последовательно, один за другим, как они записаны в блоке кода (в частности, в теле главной функции *OnStart* для скриптов). Если в блоке кода встречается выражение с вызовом другой функции, программа по такому же линейному принципу начинает выполнять инструкции внутри вызванной функции, а когда они все будут выполнены, произойдет возврат в вызывающий блок кода, и выполнение продолжится на следующем операторе после вызова функции. Управляющие инструкции могут существенно поменять эту логику работы.

Вы можете использовать выбор внутри циклов или наоборот, причем уровень вложенности не ограничен. Вместе с тем, слишком большой уровень вложенности делает программу трудно понятной для программиста. Поэтому рекомендуется выделять (переносить) блоки кода в функции (в одну или несколько): внутри каждой функции имеет смысл поддерживать уровень вложенности не более 2-3.

В MQL5 поддерживаются следующие инструкции повторения:

- цикл *for*
- цикл *while*
- цикл *do*

Все циклы позволяют выполнять одну или несколько инструкций заданное количество раз или пока выполняется некое логическое условие. Однократное выполнение содержимого цикла называется итерацией. Как правило, в циклах обрабатываются массивы или производятся периодические повторяющиеся действия (как правило, в скриптах или сервисах).

К условным инструкциям относятся:

- выбор с помощью *if*
- выбор с помощью *switch*

Первая из них позволяет указать одно или несколько условий, в зависимости от истинности или ложности которых будут выполняться приписанные к ним варианты действий (одна или несколько инструкций). Вторая вычисляет выражение целого типа, и по его значению выбирает одну из нескольких альтернатив.

Наконец, переходы осуществляются операторами:

- *break*
- *continue*
- *return*

Далее мы опишем подробно каждый из них.

В отличие от C++, MQL5 не имеет инструкции *go to*.

2.7.5 Цикл for

Данный цикл реализуется инструкцией с ключевым словом *for* — отсюда и название. В обобщенном виде её можно описать так:

```
for ( [инициализация]; [условие]; [выражение] )
   тело цикла
```

В заголовке после слова 'for' в круглых скобках указываются:

- инициализация: инструкция для однократной инициализации перед началом цикла;
- условие: логическое условие, которое проверяется в начале каждой итерации, и цикл работает, пока оно истинно;
- выражение: формула вычислений, производимых в конце каждой итерации, когда пройдены все инструкции в теле цикла.

Тело цикла — это простая или составная инструкция.

Все три компоненты заголовка являются опциональными и могут быть опущены в произвольном сочетании, включая и полное их отсутствие.

Инициализация может заключаться в объявлении переменных (вместе с установкой начальных значений) или операции присваивания значений уже существующим переменным. Такие

переменные называются переменными цикла. Если они объявлены в заголовке, то их область видимости и время жизни ограничены циклом.

Цикл начинает выполняться, если после инициализации условие равно *true*, и продолжает выполняться, пока оно верно в начале каждой последующей итерации. Если же во время очередной проверки условие нарушается, происходит выход из цикла, то есть управление передается инструкции, которая написана после цикла и его тела. Если условие равно *false* перед началом цикла (после инициализации), он не выполнится ни разу.

Условие и выражение обычно включают переменные цикла.

Выполнение цикла означает выполнение его тела.

Наиболее распространенная форма цикла *for* подразумевает единственную переменную цикла, которая управляет количеством итераций. В следующем примере мы рассчитываем для чисел в массиве *a* их квадраты.

```
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
```

Выполнение данного цикла производится по следующим шагам:

1. Создается переменная *i* с начальным значением 0.
2. Проверяется условие, что переменная *i* меньше размера цикла *n*. Пока оно верно, цикл продолжается. Если оно нарушено, произойдет переход к инструкции с вызовом функции *ArrayPrint*.
3. При истинности условия выполняются инструкции тела цикла, в данном случае, в *i*-й элемент массива записывается произведение исходного значения этого элемента на самого себя, т.е. значение каждого элемента заменяется на его квадрат.
4. Происходит увеличение переменной *i* на 1.

Далее всё повторяется, начиная с шага 2. После выхода из цикла его переменная *i* уничтожается, и попытка обратиться к ней вызовет ошибку.

Выражение для шага 4 может быть произвольной сложности, а не только инкрементом переменной цикла. Например, для прохода только по четным или нечетным элементам можно было бы написать *i += 2*.

Вне зависимости от того, какое количество инструкций составляет тело цикла, рекомендуется писать его на отдельной строке (строках) от заголовка. Это облегчает пошаговую отладку.

Инициализация может включать объявление нескольких переменных, но они должны иметь один и тот же тип, потому что это одна инструкция. Например, для перестановки элементов в обратном порядке можно написать такой цикл (это всего лишь демонстрация цикла, для обращения порядка в массиве существует встроенная функция *ArrayReverse*, см. раздел Копирование и редактирование массивов):

```
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
```

Вспомогательная переменная *temp* создается и удаляется при каждом проходе цикла, однако компилятор распределяет под неё память только один раз, как и для всех локальных переменных, при входе в функцию. Эта оптимизация хорошо работает для встроенных типов. Однако если в цикле будет описан объект пользовательского класса, то его конструктор и деструктор будут вызываться на каждой итерации.

Изменять переменную цикла в теле цикла допустимо, но этот прием используется только в очень экзотических случаях. Делать это не рекомендуется, так как это чревато ошибками (в частности, пропуском обрабатываемых элементов или попаданием в бесконечный цикл).

Чтобы продемонстрировать возможность опускать компоненты заголовка, представим следующая задачу: необходимо найти количество элементов того же массива, сумма которых меньше 100. Для этого нам потребуется переменная-счетчик *k*, определенная до цикла, потому что она должна продолжить существование после его завершения. Также заведем переменную *sum* для подсчета суммы нарастающим итогом.

```
int k = 0, sum = 0;
for( ; sum < 100; )
{
  sum += a[k++];
}

Print(k, " ", sum - a[k]); // 3 94
```

Таким образом, в заголовке не требуется делать инициализацию. Кроме того, счетчик *k* увеличивается с помощью постфиксного инкремента непосредственно в выражении, подсчитывающем сумму (при обращении к элементу массива). Поэтому нам не нужно и выражение в заголовке.

По завершении цикла мы выводим *k* и сумму за вычетом последнего прибавленного элемента, поскольку именно он переполнил наш лимит 100.

Обратите внимание, что мы используем составной блок, несмотря на то, что в теле цикла всего одна инструкция. Это полезно делать потому, что при разрастании программы всё уже готово для добавления дополнительных инструкций внутрь скобок. Кроме того, такой подход гарантирует единый стиль для всех циклов. Но выбор, в любом случае, за программистом.

В предельном, максимально сокращенном варианте заголовок цикла может выглядеть так:

```
for( ; ; )
{
   // ...        // периодические действия
   Sleep(1000); // приостановить работу программы на 1 секунду
}
```

Если в теле такого цикла нет инструкций, которые производили бы прерывание цикла по каким-то условиям, он будет выполняться постоянно. Способы прерывания и проверки условий мы изучим соответственно в разделах Переход *break* и Выбор *if*.

Подобные зацикленные алгоритмы встречаются, как правило, в сервисах (они предназначены для постоянной фоновой работы) и производят мониторинг состояния терминала или внешних сетевых ресурсов. В них обычно вставляют инструкции, приостанавливающие работу программы с заданной периодичностью, например, с помощью встроенной функции *Sleep*. Без этой предосторожности бесконечный цикл загрузит на 100% одно ядро процессора.

Скрипт *StmtLoopsFor.mq5* содержит в конце бесконечный цикл, но только в целях демонстрации.

```
for( ; ; )
{
   Comment(GetTickCount());
   Sleep(1000); // 1000 мс

   // выйти из цикла можно только удалив скрипт по команде пользователя
   // после 3 секунд ожидания получим сообщение 'Abnormal termination'
}
Comment("");  // эта строка никогда не выполнится
```

В цикле раз в секунду выводится внутренний таймер компьютера (*GetTickCount*) с помощью функции *Comment*: значение высвечивается в левом верхнем углу графика. Прервать цикл может только пользователь, удалив весь скрипт с графика (кнопка "Удалить" в диалоге экспертов). Данный код не проверяет внутри цикла такие запросы пользователя на остановку, хотя для этих целей имеется встроенная функция *IsStopped*. Она возвращает *true*, если пользователь дал команду остановиться. В программе, особенно при наличии циклов и долгосрочных вычислений, желательно предусмотреть проверку значения этой функции и добровольно завершить цикл и всю программу по получении *true*. В противном случае терминал принудительно завершит скрипт после 3 секунд ожидания (с выводом в журнал "Abnormal termination"), что и случится в данном примере.

Более правильный вариант этого цикла должен быть таким:

```
for( ; !IsStopped(); ) // продолжаем, пока пользователь не прервал
{
   Comment(GetTickCount());
   Sleep(1000); // 1000 мс
}
Comment("");    // очистит комментарий
```

Однако данный цикл лучше было бы реализовать с помощью другой инструкции повторения *while*. По негласному правилу, цикл *for* следует использовать только при наличии очевидной переменной цикла и/или определенном заранее количестве повторов. В данном случае эти условия не выполняются.

Обычно переменные цикла являются целыми, хотя допускаются и другие типы, например, *double*. Это связано с тем, что сама логика работы цикла подразумевает нумерацию итераций. Кроме того, из целого индекса всегда можно рассчитать необходимые вещественные числа, причем с большей точностью. Например, следующий цикл производит перебор значений от 0.0 до 1.0 с шагом 0.01:

```
for(double x = 0.0; x < 1.0; x += 0.01) { ... }
```

Его можно заменить аналогичным циклом с целочисленной переменной:

```
for(int i = 0; i < 100; ++i) { double x = i * 0.01; ... }
```

В первом случае, при сложениях *x += 0.01* постепенно накапливается погрешность вычислений с плавающей запятой. Во втором случае каждое значение *x* получается за одну операцию *i * 0.01*, с максимально доступной точностью.

Переменным цикла принято давать следующие имена из одной буквы, например: *i, j, k, m, p, q*. Несколько имен требуется, когда циклы являются вложенными или внутри одного цикла высчитываются как прямой (увеличивающийся), так и обратный (уменьшающийся) индексы.

Кстати, вот и пример вложенных циклов. Следующий фрагмент подсчитывает и сохраняет таблицу умножения в двумерном массиве.

```
int table[10][10] = {0};
for(int i = 1; i <= 10; ++i)
{
   for(int j = 1; j <= 10; ++j)
   {
      table[i][j] = i * j;
   }
}
ArrayPrint(table);
```

2.7.6 Цикл while

Данный цикл описывается с помощью ключевого слова *while*. Он повторяет выполнение управляемых инструкций до тех пор, пока логическое выражение в его заголовке истинно.

```
while ( условие )
   тело цикла
```

Условие — произвольное выражение логического типа. Наличие условия обязательно. Если перед началом цикла условие равно *false*, цикл не выполнится ни разу.

В отличие от C++, MQL5 не поддерживает определение переменных в заголовке цикла while.

Переменные, включенные в условие, должны быть определены до цикла.

Тело цикла — это простая или составная инструкция.

Цикл *while* обычно применяется, когда количество повторов не определено. Так, пример с ежесекундным выводом счетчика компьютерного таймера может быть записан с помощью цикла *while* и проверкой флага остановки (вызовом функции *IsStopped*) следующим образом (*StmtLoopsWhile.mq5*):

```
while(!IsStopped())
{
   Comment(GetTickCount());
   Sleep(1000);
}
Comment("");
```

Также цикл *while* удобен, когда условие окончания цикла можно совместить с модификацией переменных в одном выражении. Следующий цикл выполняется, пока переменная *i* не достигнет нуля (0 трактуется как *false*).

```
int i = 5;
while(--i) // warning: expression not boolean
{
   Print(i);
}
```

Однако в данном случае выражение в заголовке не является логическим (и неявным образом преобразуется в *false* или *true*), о чем компилятор выдает предупреждение. Желательно всегда составлять выражения с соблюдением ожидаемых (согласно правилам) характеристик. Здесь цикл правильнее записать так:

```
int i = 5;
while(--i > 0)
{
   Print(i);
}
```

Цикл можно использовать и с простой инструкцией (без блока):

```
while(i < 10)
   Print(++i);
```

Заметьте, что простая инструкция завершается точкой с запятой. Здесь также демонстрируется, что изменение переменной, проверяемой в заголовке, производится внутри цикла.

При работе с циклами нужно с осторожностью использовать беззнаковые целые числа. Например, следующий цикл никогда не закончится, потому что его условие всегда истинно (в принципе, компилятор мог бы выдать в таких местах предупреждения, но не выдает). После нуля счетчик "превратится" в большое положительное число (UINT_MAX) и цикл продолжится.

```
uint i = 5;
while(--i >= 0)
{
   Print(i);
}
```

С точки зрения пользователя MQL-программа зависнет (перестанет реагировать на команды), хотя будет по-прежнему потреблять ресурсы (процессор и память).

Циклы *while* допускают вложенность, как и остальные виды инструкций повторения.

2.7.7 Цикл do

Данный цикл похож на цикл *while*, но его условие проверяется после тела цикла. За счет этого управляемые инструкции выполняться обязательно хотя бы один раз.

Для описания цикла используется два ключевых слова *do* и *while*:

```
do
   тело цикла
while ( условие ) ;
```

Таким образом, заголовок цикла разделен, а после логического условия в скобках должна стоять точка с запятой. Условие не может быть опущено. Когда оно становится ложным, происходит выход из цикла.

Переменные, включенные в условие, должны быть определены до цикла.

Тело цикла — это простая или составная инструкция.

Следующий пример рассчитывает последовательность чисел, начиная с 1, в которой каждое следующее число получается путем умножения предыдущего на квадратный корень из двух, предопределенная константа M_SQRT2 (*StmtLoopsDo.mq5*).

```
double d = 1.0;
do
{
   Print(d);
   d *= M_SQRT2;
}
while(d < 100.0);
```

Процесс завершается, когда число превышает 100.

2.7.8 Выбор if

Оператор *if* имеет несколько форм. В простейшем случае он выполняет зависимую инструкцию, если истинно заданное условие:

```
if ( условие )
   инструкция
```

Если условие ложно, инструкция пропускается, и сразу же происходит переход к оставшейся части алгоритма (последующим инструкциям, если они есть).

Инструкцией может быть простая или составная инструкция. Условие представляет собой выражение логического или приводимого к нему типа.

Вторая форма позволяет указать две ветки действий: не только для истинного условия (инструкция_А), но и для ложного (инструкция_Б):

```
if ( условие )
   инструкция_А
else
   инструкция_Б
```

Какая бы из управляемых инструкций ни выполнилась, алгоритм затем продолжится, следуя инструкциям ниже оператора *if/else*.

Например, скрипт может придерживаться разной стратегии в зависимости от таймфрейма графика, на котором он размещен. Для этой цели достаточно анализировать значение, возвращаемое встроенной функцией *Period*. Значение имеет тип перечисления ENUM_TIMEFRAMES. Если оно меньше PERIOD_D1, подразумеваем краткосрочную торговлю, а в противном случае — долгосрочную (*StmtSelectionIf.mq5*).

```
if(Period() < PERIOD_D1)
{
   Print("Intraday");
}
else
{
   Print("Interday");
}
```

В качестве инструкции в ветке *else* допускается указывать следующий оператор *if*, и таким образом нанизывать их в цепочку последовательных проверок. Например, следующий фрагмент подсчитывает в строке количество заглавных букв и символов пунктуации (точнее, не являющихся буквами латинского алфавита).

```
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);
```

Цикл организован по всем символам строки (нумерация начинается с 0), функция *StringLen* возвращает длину строки. Первый *if* проверяет каждый символ на принадлежность диапазону от 'A' до 'Z' и в случае успеха увеличивает на 1 счетчик заглавных букв capital. Если символ не попадает в этот диапазон, запускается второй *if*, в котором условие на принадлежность диапазону строчных букв (*s[i] >= 'a' && s[i] <= 'z'*) подвергается инверсии с помощью '!'. Иными словами, условие означает, что символ не лежит в заданном диапазоне. С учетом двух последовательных проверок, если символ не заглавная буква (*else*) и не строчная буква (второй *if*), можно заключить, что символ не является буквой латинского алфавита — в этом случае увеличиваем счетчик *punctuation*.

Те же проверки можно было бы записать в более развернутом виде и использовать блоки '{...}' для наглядности.

```
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;
      }
   }
}
```

Применение фигурных скобок помогает избежать логических ошибок, связанных с тем, что программисты в их отсутствии ориентируются на отступы в коде. В частности, наиболее распространена проблема под названием "висящий" *else*.

Когда операторы *if* вложены друг в друга, иногда получается, что ветвей *else* меньше, чем *if*. Вот один пример:

```
factor = 0.0;
if(mode > 10)
   if(mode > 20)
      factor = +1.0;
else
   factor = -1.0;
```

Отступы указывают, какую логику подразумевал программист: *factor* должен стать равным +1 при *mode* больше 20, остаться равным 0 при *mode* от 10 до 20, и измениться на -1 в остальных случаях (*mode <= 10*). Но сработает ли код именно так?

В MQL5 каждый *else* считается относящимся к ближайшему предыдущему *if* (у которого нет *else*). В результате компилятор воспримет инструкции следующим образом:

```
factor = 0.0;
if(mode > 10)
   if(mode > 20)
      factor = +1.0;
   else
      factor = -1.0;
```

Значит, *factor* будет равен -1 в диапазоне *mode* от 10 до 20, и 0 при *mode <= 10*. Самое интересное, что программа не выдает никаких формальных ошибок, ни при компиляции, ни при исполнении. И все же она работает неверно.

Исключить такие трудноуловимые логические проблемы позволяет расстановка фигурных скобок.

```
if(mode > 10)
{
   if(mode > 20)
      factor = +1.0;
}
else
   factor = -1.0;
```

Для единообразия оформления желательно использовать блоки во всех ветвях инструкции, если в ней уже потребовался хотя бы один блок.

При использовании в качестве условия проверки на равенство следует учесть возможность опечатки, когда вместо двух символов '==' написан один '='. При этом сравнение превращается в присваивание, и присвоенное значение анализируется в качестве логического условия. Например,

```
// должно было быть x == y + 1, что дало бы false и пропустило if
if(x = y + 1) // warning: expression not boolean
{
   // присвоило x = 5 и трактовало x как true, потому if выполняется
}
```

Компилятор в таких случаях выдает предупреждение "выражение не является логическим" ("expression not boolean").

2.7.9 Выбор switch

Оператор *switch* предоставляет возможность выбрать один из нескольких вариантов алгоритма. Как правило, количество вариантов гораздо больше двух, потому что в противном случае проще использовать оператор *if/else*. В принципе, цепочка операторов *if/else* позволяет во многих случаях (но не во всех) получить эквивалент *switch*. Важной особенностью *switch* является то, что все варианты выбираются (идентифицируются) на основе значения выражения целочисленного типа, как правило, переменной.

В обобщенном виде оператор *switch* выглядит так:

```
switch ( выражение )
{
   case константное выражение : инструкции [break; ]
   ...
   [ default : инструкции ]
}
```

Заголовок оператора начинается с ключевого слова *switch*. После него в круглых скобах обязательно задается выражение. Блок с фигурными скобками также обязательный.

Целочисленные значения, которые могут получаться при вычислении выражения, следует указывать в виде констант после ключевого слова *case*. Под константой понимается литерал любого из целых типов, например, *int* (10, 123), *ushort* (символы 'A', 's', '*' и т.д.) или элементы перечисления. Вещественные числа, переменные или выражения здесь запрещены.

Таких *case*-вариантов может быть много, но может и не быть вовсе — это обозначено полукруглыми скобками с индексом $^{opt(n)}$. Все варианты должны иметь уникальные константы (без повторений).

Для каждой альтернативы, задекларированной с помощью *case*, после двоеточия должна быть написана инструкция, которая и будет выполняться в случае, если значение выражения равно соответствующей константе. Как обычно, инструкция может быть простой инструкцией или составной. Кроме того, допустимо написать несколько простых инструкций, не заключая их в фигурные скобки — они при этом все равно выполнятся как группа (составная инструкция).

После этих одной или нескольких инструкций может быть написан оператор перехода *break*.

При наличии *break*, после выполнения предыдущей инструкции из ветки *case* происходит выход из оператора *switch*, то есть управление передаются инструкциям, идущим ниже *switch*.

При отсутствии *break* продолжают выполняться инструкции следующей ветки или нескольких веток *case*, то есть до первого встретившегося *break* или окончания блока *switch*. Это называется "проваливанием" (fall-through).

Таким образом, оператор *switch* не только позволяет разделить поток выполнения алгоритма на несколько альтернатив, но и скомбинировать их, что недоступно для оператора *if*. С другой стороны, в операторе *switch*, в отличие от *if*, нельзя выбрать диапазон значений в качестве условия активации альтернатив.

Ключевое слово *default* позволяет задать вариант алгоритма по умолчанию, то есть для любых других значений выражения, кроме констант из всех *case*-ов. Варианта *default* может не быть, или он должен быть только один.

Последовательность, в которой перечислены *case*-константы и *default*, может быть произвольной.

Даже если пока не предусмотрено алгоритма для ветки *default*, рекомендуется делать её в явном виде, пустой, т.е. содержащей *break*. Пустой *default* напомнит вам и другим программистам, что прочие варианты существуют, но считаются неважными, так как в противном случае ветка *default* должна была бы сигнализировать ошибку.

Несколько вариантов *case* с разными константами может быть указано один под другим (или слева направо) без инструкций, но последний из них должен иметь инструкцию. Такие совмещенные *case*-ы обозначены на схеме индексом (i).

Вот самый простой и бесполезный оператор *switch*:

```
switch(0)
{
}
```

Рассмотрим более сложный пример с различными режимами (*StmtSelectionSwitch.mq5*). В нем оператор *switch* помещен внутрь цикла, чтобы показать, как его работа зависит от значений управляющей переменной *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);
}
```

Вариант -1 не выполнится, потому что цикл меняет переменную *i* от 0 до 6 (включительно). Когда *i* равно 0, сработает ветка *default*. Она же перехватит управление при *i* равном 6. Все остальные возможные значения *i* распределяются по соответствующим директивам *case*. При этом после case 2 нет инструкции *break*, в связи с чем код для вариантов 3 и 4 выполнится в довесок к 2 (в таких случаях всегда рекомендуется оставлять комментарий, что это сделано намеренно).

Случаи 3 и 4 имеют общий блок инструкций. Но еще тут важно отметить, что если требуется объявить локальную переменную внутри одного из *case*-вариантов, нужно заключить инструкции во вложенный составной блок ('{...}'). Здесь таким образом определена переменная *local_var*.

Обратите внимание, в случае *default* отсутствует инструкция *break*. Она избыточна, потому что *default* в данном случае написан в последнюю очередь. Однако многие программисты советуют вставлять *break* в конце любого варианта, даже последнего, потому что он может перестать быть последним в процессе последующих модификаций кода, и тогда легко забыть добавить *break* — а это, вероятно, приведет к ошибке в логике программы.

Если в *switch* нет варианта *default*, и выражение в заголовке не совпадает ни с одной из *case*-констант, весь оператор *switch* пропускается.

В результате выполнения скрипта мы получим следующие сообщения в журнале:

```
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 Переход break

Оператор *break* предназначен для досрочного прерывания циклов *for*, *while*, *do*, а также выхода из инструкции выбора *switch*. Оператор может применяться только внутри указанных инструкций и влияет только на одну, непосредственно содержащую *break*, если их несколько вложенных. После обработки оператора *break* выполнение программы продолжается на инструкции, следующей за прерванным циклом или *switch*.

Синтаксис очень простой — ключевое слово *break* и точка с запятой:

```
break ;
```

При использовании внутри циклов *break* обычно помещается в одну из веток условного оператора *if/else*.

Рассмотрим скрипт, который выводит текущий счетчик системного времени раз в секунду, но не более 100 раз. В нем предусмотрена обработка прерывания процесса пользователем: для этого производится опрос функции *IsStopped* в условном операторе *if* и в его зависимой инструкции имеется *break* (*StmtJumpBreak.mq5*).

```
int count = 0;
while(++count < 100)
{
   Comment(GetTickCount());
   Sleep(1000);
   if(IsStopped())
   {
      Print("Terminated by user");
      break;
   }
}
```

В следующем примере заполняется диагональная матрица с таблицей умножения (верхний правый угол останется заполнен нулями).

```
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);
```

Когда переменная внутреннего цикла *j* больше переменной наружного цикла *i*, оператор *break* прерывает внутренний цикл. Разумеется, это не лучший способ заполнить диагонально матрицу: проще было бы запустить цикл по *j* от 0 до *i* без всякого *break*, но здесь это демонстрирует наличие эквивалентных конструкций с *break* и без *break*.

Хотя в рабочих проектах всё может быть не столь очевидно, рекомендуется по возможности избегать оператора *break* и заменять его на дополнительные переменные (например, логическую переменную с "говорящим" названием *needAbreak*), которые использовать в терминальных выражениях в заголовках циклов, чтобы прерывать их стандартным путем.

Представим, что два вложенных цикла используются для поиска повторяющихся символов в строке. Первый цикл последовательно делает каждый символ строки текущим, второй — пробегает по оставшимся (стоящим справа) символам.

```
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;
      }
   }
}
```

При совпадении символов на позициях *i* и *j*, запоминаем символ-дубликат и выходим из цикла по *break*.

Можно было бы предположить, что переменная *d* должна после выполнения этого фрагмента содержать букву 'l'. Однако если разместить скрипт на самом популярном инструменте "EURUSD", ответ будет 'U'. Дело в том, что *break* прерывает только внутренний цикл, и после нахождения первого дубликата ('ll' в слове "Hello"), цикл по *i* продолжается. Поэтому для выхода сразу из нескольких вложенных циклов необходимо принимать дополнительные меры.

Наиболее популярный способ — это включение в условие внешнего цикла (или всех внешних циклов) некой переменной, заполняемой во внутреннем цикле. В нашем случае такая переменная уже есть: *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;
      }
   }
}
```

Проверка *d* на 0 теперь остановит внешний цикл по нахождению первого дубликата. Но точно такая же проверка может быть добавлена и во внутренний цикл, что избавит нас от необходимости применять *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 Переход continue

Оператор *continue* прерывает выполнение текущей итерации самого внутреннего цикла, содержащего *continue*, и инициирует следующую итерацию. Оператор может использоваться только внутри циклов *for*, *while* и *do*. Выполнение *continue* внутри *for* приводит к очередному расчету выражения в заголовке цикла (инкремент/декремент переменной цикла), после чего проверяется условие на продолжение цикла. Выполнение *continue* внутри *while* или *do* сразу приводит к проверке условия в заголовке цикла.

Инструкция состоит из ключевого слова *continue* и точки с запятой:

```
continue ;
```

Она обычно помещается в одну из веток условного оператора *if/else* или *switch*.

Например, мы можем сгенерировать таблицу умножения с пропусками: когда произведение двух индексов нечетное, соответствующий элемент массива останется нулевым (*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);
```

А вот как можно подсчитать сумму положительных элементов массива.

```
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
```

Обратите внимание, что этот же цикл можно переписать без continue, но с большей вложенностью блоков кода:

```
for(int i = 0; i < 10; ++i)
{
   if(b[i] >= 0)
   {
      sum += b[i];
   }
}
```

Таким образом, оператор *continue* часто применяется для упрощения форматирования кода (особенно, если условий для "отсечки" несколько), но какой из двух подходов выбрать — вопрос личных предпочтений.

2.7.12 Переход return

Оператор *return* предназначен для возврата управления из функций. Учитывая, что все исполняемые инструкции находятся внутри той или иной функции, он может опосредованно использоваться для прерывания содержащих его циклов *for*, *while* и *do* любого уровня вложенности. При этом следует учитывать, что в отличие от *continue* и, тем более, *break* все инструкции, следующие за прерванными циклами внутри функции, также будут проигнорированы.

Синтаксис оператора *return*:

```
return ([выражение]);
```

Необходимость указывать выражение определяется сигнатурой функции (подробнее об этом будет рассказано в соответствующем разделе). Для общего понимания работы *return* в контексте управляющих инструкций ограничимся примером с главной функцией скриптов *OnStart*. Поскольку она имеет тип *void*, то есть ничего не возвращает, оператор для неё принимает форму:

```
return ;
```

В разделе, посвященном оператору *break*, мы реализовали алгоритм поиска дублирующихся символов в строке. Для прерывания двух вложенных циклов там пришлось использовать не только *break*, но и модифицировать условие внешнего цикла.

С помощью оператора *return* это можно сделать проще (*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");
}
```

При обнаружении равенства в операторе *if* — отображаем символ и завершаем работу функции. Если бы данный алгоритм находился в пользовательской функции, отличной от *OnStart*, мы могли бы определить для неё тип возвращаемого значения (например, *ushort* вместо *void*) и передавать найденный символ с помощью полной формы *return* в вызывающий код.

Поскольку в тестовой строке заведомо существует задвоенная буква 'l', инструкция после циклов (*Print*), не будет выполнена.

2.7.13 Пустая инструкция

Пустая инструкция — самая простая инструкция языка. Она состоит всего из одного символа — точка с запятой ';'.

Пустая инструкция используется в программе в тех местах, где синтаксис требует наличия инструкции, но логика алгоритма предписывает ничего не делать.

Например, следующий цикл *while* используется для поиска пробела в строке. Вся суть алгоритма выполняется непосредственно в заголовке цикла, так что его тело должно быть пустым. Мы могли бы написать пустой блок из фигурных скобок, но также здесь подойдёт и пустая инструкция (*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);
}
```

Обратите внимание, что если точку с запятой в конце заголовка *while* убрать (допустим, случайно), то инструкция *if* станет расцениваться как тело цикла. В результате не произойдет вывода в журнал функцией *Print*. Фактически, программа сработает некорректно, хотя и без заметных ошибок.

Обратная ситуация также возможна: лишняя точка с запятой после заголовка цикла (там, где она не должна была быть) "отцепит" тело цикла от заголовка, то есть в цикле будет выполняться только пустая инструкция.

В связи с этим следует проверять в коде необязательные точки с запятой, и везде, где они поставлены намеренно, оставлять комментарий с пояснениями.

Между прочим, с формальной точки зрения пустая инструкция используется и в операторе *for*, когда мы опускаем выражение инициализации. На самом деле инициализация есть всегда:

```
for ([инициализация] ; [условие окончания цикла] ; [выражение конца цикла])
   тело цикла
```

Первый символ ';' входит в состав инструкции инициализации, которая может быть выражением или пустой инструкцией: обе содержат знак ';' на конце, причем вторая не содержит ничего кроме ';'. Таким образом и достигается опциональность (пустота).

2.8 Функции

Функция — это именованный блок с инструкциями. Практически весь прикладной алгоритм программы содержится в функциях. Вне функций выполняются только вспомогательные операции, такие как создание и удаление глобальных переменных.

Выполнение инструкций внутри функции происходит, когда мы вызываем эту функцию. Некоторые функции — главные — терминал вызывает для нас автоматически при наступлении различных событий. Еще их называют точками входа в MQL-программу или обработчиками событий. В частности, мы уже знаем, что при загрузке скрипта на график терминал вызывает его главную функцию *OnStart*. В других типах программ существуют другие функции, вызываемые терминалом — о них мы подробно поговорим в пятой и шестой Частях, посвященных торговой архитектуре MQL5 API.

В этой главе мы узнаем как определить и объявить функцию, как описать и передать в неё параметры, а также возвращать из функции результат её работы.

Также мы поговорим о перегрузке функций, то есть возможности предоставить несколько функций с одним и тем же именем, и как это может быть полезно.

Наконец, мы познакомимся с новым типом — указателем на функцию.

2.8.1 Определение функции

Определение функции состоит из типа возвращаемого ею значения, идентификатора, списка параметров в круглых скобках и тела — блока кода с инструкциями. Параметры в списке перечисляются через запятую. Для каждого параметра указывается тип, имя и, опционально, значение по умолчанию.

```
тип_результата идентификатор_функции ( ⌈тип_параметра идентификатор_параметра
                                        = значение_по_умолчанию⌋opt ,... )
{
   ⌈инструкция⌋opt
   ...
}
```

Допускается создание функций без параметров: тогда список отсутствует, и после имени функции ставятся пустые скобки (их опускать нельзя). Желающие могут написать между скобок ключевое слово *void*, чтобы подчеркнуть, что параметров нет. Например, так:

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

Совокупность типа возвращаемого значения, количества и типов параметров в списке называется прототипом функции или сигнатурой. Разные функции могут иметь один и тот же прототип.

В предыдущих разделах мы уже встречали определения функций, например, *OnStart* и *Greeting*. Сейчас попробуем реализовать в виде тестовой функции вычисление чисел ряда Фибоначчи. Напомним, что его числа высчитываются по формуле:

```
f[0] = 1
f[1] = 1
f[i] = f[i - 1] + f[i - 2], i > 1
```

Первые два числа равны 1, а все последующие представляют собой сумму двух предыдущих. Приведем начало ряда: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...

Рассчитать число под заданным номером можно следующей функцией (*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;
}
```

Она принимает один параметр *n* типа *int* и возвращает результат типа *int*. Параметр *n* имеет модификатор *const*, потому что мы не собираемся менять *n* внутри функции (такая явная декларация ограничений в "правах" переменных приветствуется — она помогает избежать случайных ошибок).

В локальных переменных *prev* и *result* будут храниться текущие значения двух последних чисел ряда. В цикле по *i* мы вычисляем их сумму, получая очередное число ряда. Предварительно, старое значение *result* записывается в переменную *temp*, чтобы после суммирования, переложить его в *prev*.

После выполнения цикла заданное количество раз в переменной *result* оказывается искомое число. Его мы возвращаем из функции с помощью инструкции *result*.

Входной параметр функции — это тоже локальная переменная, которая будет инициализирована актуальным значением во время вызова функции. Это значение передается "снаружи" из инструкции с вызовом функции.

Имена параметров должны быть уникальными и не совпадать с именами локальных переменных.

Напомним, что тело функции представляет собой блок кода, который определяет видимость и время жизни локальных переменных. Принципы их определения и работы были рассмотрены в разделах Инструкции объявления/определения и Инициализация.

2.8.2 Вызов функции

Вызов функции происходит при упоминании её имени в выражении. После имени должна идти пара круглых скобок, в которых через запятую указываются аргументы, соответствующие параметрам функции (если в её определении есть список параметров).

Чуть позже мы рассмотрим тип "указатель на функцию", который позволяет создавать переменные, указывающие на функцию с конкретными характеристиками, и затем вызывать её не по имени, а посредством этой переменной.

Продолжая пример с функцией *Fibo*, осуществим её вызов из функции *OnStart*. Для этого заведем переменную *f* для хранения результирующего числа и в выражении её инициализации укажем имя функции *Fibo* и целое число (например, 10) в качестве аргумента, в круглых скобках.

```
void OnStart()
{
   int f = Fibo(10);
   Print(f); // 89
}
```

Напомним, что мы не обязаны заводить переменную для приема значения из функции. Вместо этого можно вызывать функцию непосредственно из выражения, например, "2*Fibo(10)" или "Print(Fibo(10))". Тогда её значение подставится в выражение на место вызова. Здесь вспомогательная переменная *f* заведена, чтобы выделить в отдельной инструкции вызов и возврат значения.

Процесс вызова включает следующие этапы:

- выполнение последовательности инструкций вызывающей функции (*OnStart*) приостанавливается;
- значение аргумента попадает во входной параметр *n* вызываемой функции (*Fibo*);
- начинают выполняться её инструкции;
- когда она полностью отработает, то передает результат обратно (вспоминаем инструкцию *return* внутри);
- результат попадет в переменную *f*;
- после этого продолжится выполнение функции *OnStart*, то есть вывод числа в журнал (*Print*).

Для каждого вызова функции компилятор генерирует вспомогательный двоичный код (программисту о нём заботиться не надо). Суть этого кода в том, что перед вызовом функции он

заносит в стек текущую позицию в программе, а после завершения вызова извлекает её и использует для возврата к инструкциям, следующим за вызовом функции. Когда одна функция вызывает другую, та вызывает еще одну, вторая вызывает третью и так далее, на стеке стопкой накапливаются обратные адреса переходов по всей иерархии вызванных функций (отсюда и название — стек). По мере отработки вложенных вызовов функций, стек будет в обратном порядке очищаться. Напомним, что на стеке также выделяется память для локальных переменных каждой функции.

2.8.3 Параметры и аргументы

Аргументы, передаваемые в функцию при вызове, являются начальными значениями соответствующих параметров функции. Количество, порядок и типы аргументов должны совпадать с прототипом функции. Вместе с тем, порядок вычисления аргументов не определен (см. раздел Базовые понятия). В зависимости от специфики исходного кода и соображений оптимизации, компилятор может выбрать удобный для него вариант. Например, при списке из двух параметров компилятор может вычислить сначала второй аргумент, а затем первый. Гарантированным является только то, что оба аргумента будут вычислены перед вызовом.

Каждый аргумент сопоставляется с соответствующим параметром по тем же правилам, по которым происходит инициализация переменных, причем при необходимости делается неявное приведение типов. Перед началом работы функции все её параметры гарантированно имеют заданные значения. Например, в зависимости от передаваемых аргументов, вызовы функции *Fibo* могут привести к следующим эффектам (описаны в комментариях):

```
// предупреждения
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'
// ошибки
Fibo();             // wrong parameters count
Fibo(0, 10);        // wrong parameters count
```

Все предупреждения касаются неявных преобразований, которые компилятор выполняет из-за того, что типы значений не соответствуют типам параметров. Их следует расценивать как потенциальные ошибки и устранять. Ошибка "неверное количество параметров" ("wrong parameters count") возникает, когда аргументов слишком мало или слишком много.

В принципе, параметр функции не обязан иметь имя, то есть для описания параметра достаточно только типа. Это звучит довольно странно, потому что к параметру без имени мы не сможем обращаться внутри функции. Однако при создании программ, основывающихся на неких стандартных интерфейсах, иногда приходится писать функции, которые должны отвечать заданным прототипам. При этом внутри функции некоторые параметры могут оказаться ненужными. Тогда для явного указания этого факта программист может опустить их имена. Например, MQL5 API предписывает реализовывать функцию-обработчик события *OnDeinit* со следующим прототипом:

```
void OnDeinit(const int reason);
```

Если в коде функции нам не требуется параметр *reason*, мы можем опустить его в описании:

```
void OnDeinit(const int);
```

Функцию обработки событий терминала обычно вызывает сам терминал, но если бы нам потребовалось вызвать аналогичную функцию (с анонимным параметром) из своего кода, то необходимо передать все аргументы, независимо от того, являются ли параметры именованными или нет.

2.8.4 Параметры-значения и параметры-ссылки

Передача аргументов в функцию может происходить двумя способами: по значению и по ссылке.

Все случаи, которые мы до сих пор рассматривали, относятся к передаче по значению. Этот вариант означает, что значение аргумента, подготовленное вызывающим фрагментом кода, копируется в новую переменную — соответствующую входную переменную функции. В остальном аргумент и входная переменная не связаны между собой. Все последующие манипуляции с переменной внутри функции никак не сказываются на аргументе.

Для описания параметра-ссылки необходимо добавить справа от типа знак амперсанда '&'. Многие программисты предпочитают присоединять амперсанд к имени параметра, тем самым подчеркивая, что именно параметр является ссылкой на заданный тип. Например, следующие записи эквивалентны:

```
void func(int &parameter);
void func(int & parameter);
void func(int& parameter);
```

При вызове функции для параметра-ссылки не создается соответствующая локальная переменная. Вместо этого аргумент, указанный для этого параметра, становится доступен внутри функции под именем (алиасом) входного параметра. Таким образом, значение не копируется, а используется по прежнему адресу в памяти. Поэтому модификации параметра внутри функции отражаются на состоянии связанного с ним аргумента. Из этого следует важная особенность.

В качестве аргумента для параметра-ссылки можно указывать только переменную (LValue, см. раздел Оператор присваивания). В противном случае мы получим ошибку "параметр передается по ссылке, ожидается переменная" ("parameter passed as reference, variable expected").

Передача по ссылке применяется в нескольких случаях:

- для повышения эффективности работы программы за счет исключения копирования значения;
- для передачи модифицированных данных из функции в вызывающий код, когда возврата отдельного значения с помощью *return* недостаточно.

Первый пункт особенно актуален для переменных с потенциально большим размером, таких как строки или массивы.

Чтобы разграничить первое и второе назначение параметра-ссылки, авторам функций рекомендуется добавлять модификатор const, когда изменение параметра внутри функции не предполагается. Это напомнит Вам и даст понять другим разработчикам, что передача переменной внутрь функции не приведет к побочным эффектам.

Неприменение модификатора *const* к параметрам-ссылкам, там где это возможно, способно тянуть за собой проблемы по всей иерархии вызовов функций. Дело в том, что вызовы подобных

функций будут требовать неконстантных аргументов. Иначе будет возникать ошибка "переменная-константа не может передаваться по ссылке" ("constant variable cannot be passed as reference"). В результате может постепенно оказаться, что все параметры во всех функциях следует лишить модификатора *const* в угоду компилируемости кода. На самом деле, это фактически расширяет простор для потенциальных ошибок с непреднамеренной порчей переменных. Исправлять ситуацию следует обратным образом: ставить *const* везде, где не требуется возврат и модификация значений.

Для сравнения способов передачи параметров в скрипте *FuncDeclaration.mq5* реализовано несколько функций: *FuncByValue* — передача по значению, *FuncByReference* — передача по ссылке, *FuncByConstReference* — передача по константной ссылке.

```
void FuncByValue(int v)
{
   ++v;
   // делаем что-то еще с v
}

void FuncByReference(int &v)
{
   ++v;
}

void FuncByConstReference(const int &v)
{
   // ошибка
   // ++v; // 'v' - constant cannot be modified
   Print(v);
}
```

В функции *OnStart* мы вызываем все эти функции и наблюдаем за их влиянием на переменную *i*, используемую как аргумент. Отметим, что передача параметра по ссылке не меняет синтаксис вызова функции.

```
void OnStart()
{
   int i = 0;
   FuncByValue(i);          // i не может измениться
   Print(i);                // 0
   FuncByReference(i);      // i меняется
   Print(i);                // 1
   FuncByConstReference(i); // i не может измениться, 1
   const int j = 1;
   // error
   // 'j' - constant variable cannot be passed as reference
   // FuncByReference(j);

   FuncByValue(10);         // ok
   // error: '10' - parameter passed as reference, variable expected
   // FuncByReference(10);
}
```

Литерал может быть передан только в функцию *FuncByValue*, так как остальные функции требуют ссылки, то есть переменной в качестве аргумента.

Функция *FuncByReference* не может быть вызвана с переменной *j*, поскольку последняя описана как константа, а данная функция декларирует возможность (или намерение) изменить свой параметр, так как он не снабжен модификатором *const*. Из-за этого генерируется ошибка "переменная-константа не может передаваться по ссылке" ("constant variable cannot be passed as reference").

Также в скрипте описана функция *Transpose*: она транспонирует матрицу 2x2, переданную в виде двумерного массива по ссылке.

```
void Transpose(double &m[][2])
{
   double temp = m[1][0];
   m[1][0] = m[0][1];
   m[0][1] = temp;
}
```

Её вызов из *OnStart* демонстрирует ожидаемое изменение содержимого локального массива **a**.

```
double a[2][2] = {{-1, 2}, {3, 0}};
Transpose(a);
ArrayPrint(a);
```

В MQL5 параметры-массивы всегда передаются в виде внутренней структуры динамического массива (см. раздел Характеристики массивов). Как следствие, описание такого параметра должно обязательно иметь открытый размер по первому измерению, то есть внутри первой пары квадратных скобок пусто.

Это не мешает передавать в функцию, при необходимости, фактический аргумент, являющийся массивом с фиксированным размером (как в нашем примере). Однако функции вроде *ArrayResize* не смогут изменить размер или как-то иначе реорганизовать такой замаскированный фиксированный массив.

Размеры массива по всем размерностям кроме первой должны совпадать у параметра и аргумента. В противном случае мы получим ошибку "конвертация параметра запрещена" ("parameter conversion not allowed"). В частности, в примере определена функция *TransposeVector*:

```
void TransposeVector(double &v[])
{
}
```

Попытка вызвать её для двумерного массива ***a*** закомментирована в *OnStart*, поскольку генерирует вышеуказанную ошибку: размерности массивов не совпадают.

> Помимо передачи параметров по значению или по ссылке существует еще один вариант — передача указателя. В отличие от C++, MQL5 поддерживает указатели только для объектных типов (классов). Мы рассмотрим эту особенность в третьей Части.

2.8.5 Необязательные параметры

MQL5 предоставляет возможность при описании функции указать для параметров значения по умолчанию. Для этого используется синтаксис инициализации, то есть литерал соответствующего типа справа от параметра, после знака '='. Например:

```
void function(int value = 0);
```

При вызове функции аргументы для таких параметров можно не указывать. Тогда их значения будут установлены равными величинам по умолчанию. Подобные параметры называются необязательными (опциональными).

Необязательные параметры должны располагаться в конце списка параметров. Иными словами, если *i*-й параметр описан с инициализацией, то все последующие также должны её иметь. В противном случае выдается ошибка компиляции "пропущено значение по умолчанию для параметра" ("missing default value for parameter"). Ниже показано описание функции с такой проблемой.

```
double Largest(const double v1, const double v2 = -DBL_MAX,
               const double v3);
```

Решений два: либо параметр *v3* также должен иметь значение по умолчанию, либо параметр *v2* должен стать обязательным.

Опускать необязательные аргументы при вызове функции можно только в направлении справа налево. То есть, если у функции два параметра и оба необязательные, то при вызове нельзя пропустить первый, но указать второй. Одиночное переданное значение будет сопоставлено с первым параметром, второй будет считаться опущенным. Если отсутствуют оба аргумента, пустые круглые скобки все равно нужны.

Рассмотрим функцию поиска максимального числа из трех. Первый параметр — обязательный, два последних — опциональные и равны по умолчанию минимально возможному числу типа *double*. Таким образом, каждый из них в отсутствии явно переданного значения будет заведомо меньше (или, в крайнем случае, равен) всех остальных параметров.

```
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);
}
```

Вот как её можно вызвать:

```
Print(Largest(1));       // ok: 1
Print(Largest(0, -2));   // ok: 0
Print(Largest(1, 2, 3)); // ok: 3
```

С помощью необязательных параметров в MQL5 реализуется концепция функций с переменным количеством параметров в пользовательских функциях.

MQL5 не поддерживает синтаксис определения функций с переменным числом параметров с помощью многоточия (ellipsis), как в C++. Вместе с тем, в MQL5 API существуют встроенные функции, которые описаны с использованием многоточия и принимают переменное количество произвольных параметров. В частности, к ним относится известная нам функция Print. Её прототип выглядит так: void Print(argument, ...). Поэтому мы можем вызвать её, указав через запятую вплоть до 64 аргументов (за исключением массивов) и она отобразит их в журнале.

2.8.6 Возврат значений

Функции способны возвращать значения встроенных типов, структуры с полями встроенных типов, а также указатели на функции и указатели на объекты классов. Название типа пишется в определении функции перед именем. Если функция ничего не возвращает, ей следует приписать тип *void*.

Для возврата из функции массивов необходимо использовать параметры, передаваемые по ссылке (см. Параметры-значения и параметры-ссылки).

Возврат значения производится с помощью инструкции *return*, в которой после ключевого слова *return* указывается выражение. Допустимо применять любую из двух форм:

```
return выражение ;
```

или:

```
return ( выражение ) ;
```

Если функция имеет тип *void*, то инструкция *return* упрощается:

```
return ;
```

Нельзя указывать в инструкции *return* какое-то выражение внутри *void*-функции: компилятор выдаст ошибку "'void' функция возвращает значение" ("'return' - 'void' function returns a value").

Для подобных функций, в принципе, не обязательно использовать *return* в конце блока с телом функции. Мы это видели на примере функции *OnStart*.

Если же функция имеет тип, отличный от *void*, то инструкция *return* должна быть обязательно. При её отсутствии возникнет ошибка компиляции "не все пути исполнения возвращают значение" ("not all control paths return a value").

```
int func(void)
{
   if(IsStopped()) return; // error: function must return a value
                           // error: not all control paths return a value
}
```

Важно отметить, что тело функции может иметь несколько операторов return. В частности, при досрочных выходах по условию. Любой оператор *return* прерывает выполнение функции в том месте, где он находится.

Если функция должна вернуть значение (т.к. она не типа *void*), а в операторе *return* оно не указано, компилятор выдаст ошибку "функция должна вернуть значение" ("function must return a value"). Корректная с точки зрения компилятора версия функции *func* приведена ниже (*FuncReturn.mq5*).

```
int func(void)
{
   if(IsStopped()) return 0;
   return 1;
}
```

Если возвращаемое значение отличается от указанного типа функции, компилятор сделает попытку неявного приведения типов. В случае, если типы требуют явного преобразования, будет сгенерирована ошибка.

Для возврата значения неявно создается временная переменная, которая становится доступной в вызывающем коде.

Когда мы изучим объектные типы (см. главу про Классы) и возможность возвращать указатели на объекты из функций, мы вернемся к нюансам их безопасной передачи. В отличие от C++, функции в MQL5 не способны возвращать ссылки. Попытка описать функцию с амперсандом в типе результата приводит к ошибке "ссылка не может использоваться" ("'&' - reference cannot used").

2.8.7 Объявление функции

Объявление функции описывает прототип без указания тела функции. Вместо блока с телом ставится точка с запятой.

Объявление необходимо для компилятора, чтобы можно было проверять в последующих фрагментах кода, насколько правильно производится вызов функцию по имени, передача в неё аргументов и получение результата.

Полное определение функции (вместе с телом) также является объявлением, поэтому объявлять функцию в дополнение к определению не требуется.

Например, объявление вышеприведенной функции *Fibo* могло бы выглядеть следующим образом.

```
int Fibo(const int n);
```

Раздельное объявление и определение функций используется при сборке программы из нескольких файлов с исходным текстом: тогда объявление делается в заголовочном файле с расширением *mqh* (см. раздел про директиву препроцессора *#include*), который включается в файлы, где функция используется, а определение функции реализуется лишь в одном из файлов.

Совпадение сигнатуры функции в объявлении и определении обеспечивает защиту от ошибок. Иными словами, единственное объявление гарантирует согласованность вносимых во весь исходный код правок.

Если мы объявим функцию и вызовем её где-то в коде, но не предоставим для неё полностью соответствующее определение, компилятор выдаст ошибку: "функция 'Имя' должна иметь тело" ("function 'Name' must have a body"). Это часто бывает при опечатках или неточностях либо в объявлении, либо в определении, а также в процессе изменения исходных кодов, когда часть исправлений уже внесена, а про другую часть, скорее всего, забыли.

Если функция объявлена и нигде не используется, компилятор не требует и её определения — такой элемент просто "вырезается" из бинарной программы.

В разделе Инструкции объявления/определения мы рассматривали пример функции *Init* (скрипт *StmtDeclaration.mq5*), которая использовалась для инициализации переменных. Там, в частности, была продемонстрирована проблема, связанная с тем, что глобальная переменная *k* не может быть определена до функции *Init*, поскольку начальное значение *k* получается в результате вызова *Init*. Компилятор выдавал ошибку "'Init' - неизвестный идентификатор".

Теперь мы знаем, что такую проблему можно решить с помощью объявления. В скрипте *FuncDeclaration.mq5* мы добавили такое предварительное объявление (forward declaration) функции *Init* перед переменной *k*, а определение *Init* оставили после *k*.

```
// предварительное объявление
int Init(const int v);
// до добавления предварительного объявления выше
// тут возникала ошибка: 'Init' - неизвестный идентификатор
int k = Init(-1);
int Init(const int v)
{
   Print("Init: ", v);
   return v;
}
```

Теперь скрипт компилируется нормально. В принципе, в данном случае можно было без предварительного объявления просто перенести функцию выше переменной. Мы сделали это для объяснения концепции. Однако бывают случаи взаимной зависимости элементов языка друг от друга (например, классов), когда без предварительного объявления в пределах одного файла не обойтись.

2.8.8 Рекурсия

Из инструкций внутри функции разрешено вызывать эту же функцию. Такие вызовы называются рекурсивными.

Вернемся к примеру с вычислением чисел Фибоначчи. Следуя формуле расчета каждого числа как суммы двух предыдущих (за исключением двух первых, которые равны 1), легко написать рекурсивную функцию расчета чисел Фибоначчи.

```
int Fibo(const int n)
{
   if(n <= 1) return 1;
   
   return Fibo(n - 1) + Fibo(n - 2);
}
```

В рекурсивной функции должен быть предусмотрен возврат управления без рекурсии, как в нашем случае внутри уловного оператора *if* для индексов 0 и 1. Иначе последовательность вызовов функции могла бы продолжаться бесконечно. На практике, поскольку незавершенные вызовы функции накапливаются в ограниченной области памяти под названием стек (см. раздел Инструкции объявления/определения, а также врезку "Куча"и "стек" в разделе Описание массивов), рано или поздно она закончится с ошибкой времени выполнения "Переполнения стека" ("Stack overflow"). Эта проблема продемонстрирована в функции *FiboEndless*.

```
int FiboEndless(const int n)
{
   return FiboEndless(n - 1) + FiboEndless(n - 2);
}
```

Обратите внимание, это не ошибка компиляции. Компилятор в данном случае не поможет даже предупреждением (хотя теоретически мог бы). Ошибка возникнет в процессе исполнения скрипта и будет выведена в журнал *Экспертов* в терминале.

Рекурсия может возникать не только при непосредственном вызове функции из неё самой. Например, если функция *F* вызывает функцию *G*, а та, в свою очередь, функцию *F*, то это тоже рекурсия, хоть и опосредованная. Таким образом, возможна рекурсия при цикличности вызовов любого порядка (глубины).

2.8.9 Перегрузка функций

MQL5 допускает определение в исходном коде функций с одним и тем же именем, но разным количеством или типом параметров. Такой подход называется перегрузкой функций. Он обычно применяется, когда одно и то же по смыслу действие может быть инициировано отличающимися входными данными. Благодаря различиям в сигнатурах компилятор способен автоматически определить, какую именно функцию следует вызвать, руководствуясь передаваемыми аргументами. Но существует несколько нюансов.

Функции не могут различаться только типом возвращаемого значения. Такая ситуация не запускает механизм перегрузки и будет генерировать ошибку "функция уже определена с другим типом" ("function already defined and has different type").

Кроме того, если одноименные функции отличаются количеством параметров, но "лишние" параметры объявлены необязательными, то компилятор не сможет "решить", какую из них вызвать. В результате мы получим ошибку "неоднозначный вызов перегруженной функции с одинаковыми параметрами" ("ambiguous call to overloaded function with the same parameters").

При вызове перегруженной функции компилятор производит сопоставление аргументов и параметров в имеющихся перегрузках. Если не удалось найти точного совпадения, компилятор "пробует" добавить/удалить модификатор *const*, выполнить расширение числового типа и арифметическое преобразование. В случае указателей на объекты используются правила наследования классов.

При различном количестве параметров или неродственных типах параметров в одной и той же позиции (например, число и строка), выбор обычно очевиден. Однако если типы параметров подлежат неявной конвертации из одного в другой, может появиться неоднозначность.

Например, представим две функции суммирования:

```
double sum(double v1, double v2)
{
   return v1 + v2;
}

int sum(int v1, int v2)
{
   return v1 + v2;
}
```

Тогда следующий вызов приведет к ошибке:

```
sum(1, 3.14); // ambiguous call to overloaded function
```

Здесь компилятору в равной степени неудобна каждая из перегрузок: для функции *double sum(double v1, double v2)* необходимо неявно преобразовать первый аргумент в *double*, а для функции *int sum(int v1, int v2)* — второй аргумент в *int*.

Термин перегрузка следует трактовать в том смысле, что многократно использованное имя "загружено" "обязанностями" в несколько раз сильнее, чем обычное имя, использованное только для одной функции.

Попробуем перегрузить функцию для транспонирования матрицы. У нас уже был пример для массива размером 2x2 (см. Параметры-значения и параметры-ссылки). Реализуем такую же операцию для массива 3x3. Размер многомерного параметра-массива в старших измерениях (ненулевом) изменяет тип, то есть double [][2] отличается от double [][3]. Таким образом, мы перегрузим старую версию функции:

```
void Transpose(double &m[][2]);
```

добавив новую (*FuncOverload.mq5*):

```
void Transpose(double &m[][3]);
```

В реализации новой версии нам удобно применить вспомогательную функцию *Swap* для обмена двух элементов матрицы по заданным индексам.

```
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;
}
```

Теперь мы сможем вызывать из *OnStart* обе функции, используя одну и ту же нотацию для массивов различных размеров. Компилятор сам сгенерирует вызов правильных версий.

```
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);
```

Важно отметить, что модификатор *const* у параметра хоть и меняет прототип функции, но не всегда является достаточным различием для перегрузки. Две одноименных функции, которые отличаются только присутствием и отсутствием *const* у какого-либо параметра, могут считаться одинаковыми. В результате возникнет ошибка "function already defined and has body". Данное поведение обусловлено тем, что для параметров-значений модификатор *const* отбрасывается при назначении аргумента (поскольку параметр-значение по определению не сможет изменить аргумент в вызывающем коде), и это не позволяет выбрать на его основе одну из нескольких перекрытых функций.

Чтобы продемонстрировать это, достаточно в скрипте добавить функцию:

```
void Swap(double &m[][3], int i, int j);
```

Она является неудачной перегрузкой для уже имеющейся:

```
void Swap(double &m[][3], const int i, const int j);
```

Различие двух функций заключается только в модификаторах *const* у параметров *i* и *j*. Поэтому они обе подходят для вызова с аргументами типа *int* и передачей по значению.

Когда параметры передаются по ссылке, перегрузка с различием только в *const*/не-*const* атрибутах совершается успешно, потому что для ссылок модификатор *const* важен (меняет тип и исключает возможность неявного преобразования). Это демонстрируется в скрипте парой функций:

```
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);
   }
}
```

Они оставлены в виде практически пустых заглушек, в которых печатается сигнатура каждой функции с помощью вызова *Print(__FUNCSIG__)*. Это дает возможность убедиться, что соответствующая версия функции вызывается в зависимости от атрибута *const* аргументов.

2.8.10 Указатели на функции (typedef)

MQL5 имеет ключевое слово *typedef*, которое позволяет описать специальный тип указателя на функцию.

> В отличие от C++, где *typedef* имеет гораздо более широкое применение, в MQL5 *typedef* используется только для указателей на функции.

Синтаксис описания нового типа таков:

```
typedef function_result_type ( *function_type )( [list_of_input_parameters] ) ;
```

Идентификатор *function_type* определяет имя типа, которое становится синонимом (алиасом) указателя на любую функцию, возвращающую значение заданного типа *function_result_type* и принимающую список входных параметров (*list_of_input_parameters*).

Например, у нас может быть 2 функции с одинаковыми прототипами (два входных параметра типа *double* и тип результата тоже *double*), выполняющие разные арифметические операции: сложение и вычитание (*FuncTypedef.mq5*).

```
double plus(double v1, double v2)
{
   return v1 + v2;
}

double minus(double v1, double v2)
{
   return v1 - v2;
}
```

Их общий прототип легко описать для использования в качестве указателя:

```
typedef double (*Calc)(double, double);
```

Данная запись вводит в программу тип *Calc*, с помощью которого можно определить переменную/параметр для хранения/передачи "ссылки" на любую функцию с таким прототипом, включая обе функции *plus* и *minus*. Указателем данный тип является потому, что в описании используется символ '*' (**Calc*). Об особенностях "звездочки" в применении к указателям мы более подробно узнаем при изучении ООП.

Подобный класс указателей удобно использовать для создания настраиваемых алгоритмов, способных "на лету", в зависимости от входных данных, вызывать разные функции, соответствующие алиасу.

В частности, у нас есть возможность ввести обобщенную функцию-вычислитель:

```
double calculator(Calc ptr, double v1, double v2)
{
   if(ptr == NULL) return 0;
   return ptr(v1, v2);
}
```

Её первый параметр описан с типом *Calc*. За счет этого мы можем передать в неё произвольную функцию с подходящим прототипом и в результате выполнить некую операцию, о сути которой сама функция *calculator* не знает. Она делает это путем делегирования вызова указателю: *ptr(v1, v2)*. Поскольку *ptr* это указатель на функцию, данный синтаксис не только напоминает вызов функции, но и реально вызывает функцию, которую указатель хранит.

Обратите внимание, что мы предварительно проверяем параметр *ptr* на специальное значение NULL (NULL — эквивалент нуля для указателей). Дело в том, что указатель может никуда не указывать, то есть быть непроинициализированным. Так, в скрипте у нас описана глобальная переменная:

```
Calc calc;
```

В ней как раз нет никакого указателя. Если бы не "защита" от NULL, вызов *calculator* с "пустым" указателем *calc* привел бы к ошибке времени исполнения "Вызов функции по некорректному указателю" ("Invalid function pointer call").

Вызовы функции *calculator* с разными указателями в первом параметре дадут следующие результаты (показаны в комментариях):

```
void OnStart()
{
   Print(calculator(plus, 1, 2));   //  3
   Print(calculator(minus, 1, 2));  // -1
   Print(calculator(calc, 1, 2));   //  0
}
```

Отметим, что все указатели на функции в отсутствие явной инициализации, заполняются нулевыми значениями. Это касается и глобальных, и локальных переменных данного типа.

Тип указателя, определенный с помощью *typedef*, можно возвращать из функций, например:

```
Calc generator(ushort type)
{
   switch(type)
   {
      case '+': return plus;
      case '-': return minus;
   }
   return NULL;
}
```

Кроме того, тип указателей на функции часто используется для функций обратного вызова (*callback*, см. *FuncCallback.mq5*). Представьте, что у нас есть функция *DoMath*, которая выполняет длительные вычисления (вполне вероятно, она реализована в отдельной библиотеке). С точки зрения удобства и дружественности пользовательского интерфейса было бы здорово показывать пользователю индикацию прогресса. Для этой цели можно определить специальный тип указателя на функцию для уведомлений о проценте выполненной работы (*ProgressCallback*), а в функцию *DoMath* добавить параметр этого типа. В коде *DoMath* следует периодически вызывать переданную функцию:

```
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);
      }

      // долгие вычисления
   }
}
```

Тогда вызывающий код может определить у себя требуемую функцию-callback, передать указатель на неё в *DoMath* и получать обновления по ходу вычислений.

```
void MyCallback(const float percent)
{
   Print(percent);
}

void OnStart()
{
   double data[] = {0};
   DoMath(data, MyCallback);
}
```

Указатели на функции работают только с пользовательскими функциями, определенными в MQL5. Они не могут указывать на встроенные функции MQL5 API.

2.8.11 Инлайнинг

В целях повышения эффективности кода современные компиляторы часто прибегают к следующему приему. При генерации исполняемого кода вызовы некоторых функций заменяются непосредственно телом функции (её инструкциями). Данная техника называется "встраиванием" ("инлайнингом", inline). Выигрыш в скорости достигается за счет того, что из кода исключаются накладные расходы, связанные с организацией вызова и возврата из функции. С точки зрения программиста инлайнинг ничего не меняет.

MQL5 поддерживает инлайнинг по умолчанию. При необходимости, его можно отключить, но только в режиме профилировки кода. Ключевое слово *inline* зарезервировано в языке MQL5 для совместимости с исходными кодами на C++. Его наличие или наоборот отсутствие перед определением функции не влияет на генерируемую программу.

2.9 Препроцессор

До сих пор мы изучали программирование на MQL5, подразумевая, что исходные коды обрабатываются компилятором, который преобразует их текстовое представление в двоичное (исполняемое терминалом). Однако первым инструментом, который читает и, при необходимости, преобразует исходные коды, является препроцессор. Эта встроенная в MetaEditor утилита управляется специальными директивами, вставленными непосредственно в исходный код, и позволяет решать целый ряд задач, с которыми сталкиваются программисты при подготовке исходных кодов.

По аналогии с препроцессором C++, MQL5 поддерживает определение макроподстановок (*#define*), условную компиляцию (*#ifdef*) и подключение других исходных файлов (*#include*). В данной главе мы рассмотрим эти возможности. В некоторых из них имеются ограничения по сравнению с C++.

Помимо стандартных директив препроцессор MQL5 имеет свои специфические, в частности, набор свойств MQL-программы (#property) и импорт функций из обособленных библиотек форматов EX5 и DLL (*#import*). К ним мы обратимся в пятой, шестой и седьмой Частях при изучении различных типов MQL-программ.

Все директивы препроцессора начинаются с символа решетки '#', за которым следует ключевое слово и дополнительные параметры, синтаксис которых зависит от типа директивы.

Рекомендуется начинать директиву препроцессора с самого начала строки или, по крайней мере, после отступа из пробельных символов (если директивы вложенные). Вставлять директиву внутрь инструкций исходного кода считается плохим стилем программирования (в отличие от MQL5 препроцессор C++ этого вовсе не позволяет).

Директивы препроцессора не являются инструкциями языка, их не следует завершать символом ';'. Директивы обычно продолжаются до конца текущей строки. В некоторых случаях их можно особым образом продлевать на следующие строки, о чем будет сказано отдельно.

Директивы выполняются последовательно, в порядке встречаемости в тексте и с учетом обработки предыдущих директив. Например, если к файлу подключен другой файл с помощью директивы *#include*, и в подключенном файле определено правило подстановки с помощью *#define*, то это правило начинает работать для всех последующих строк кода, в том числе, в подключенных позже заголовочных файлах.

Комментарии препроцессором не обрабатываются.

2.9.1 Включение исходных файлов (#include)

Директива *#include* предназначена для включения в исходный код содержимого другого, указанного в ней файла. Её действие эквивалентно тому, как если бы программист скопировал в буфер обмена текст из включаемого файла и вставил в то место текущего файла, где была директива.

Разделение исходных кодов на несколько файлов — обычная практика при написании сложных программ. Все они строятся по модульному принципу. В каждом модуле/файле содержится логически связанный код, решающий одну или несколько смежных задач.

Также включаемые файлы используются для распространения библиотек (наборов готовых алгоритмов). Одна и та же библиотека может быть включена в разные программы. При этом обновление библиотеки (её заголовочного файла) автоматически начнет работать во всех программах при их последующей компиляции.

Если основные файлы MQL-программ должны иметь расширение mq5, то для включаемых файлов существует общая практика давать им расширение *mqh* ('h' на конце от слова "header" — "заголовок"). Вместе с тем, допустимо использовать директиву *#include* и для других типов файлов с текстом, например, **.txt* (см. далее). В любом случае, после включения другого файла не должна нарушаться синтаксическая правильность итоговой программы, объединенной из основного mq5-файла и всех заголовков. В частности, включение файла с двоичной информацией (например, png-изображения) приведет к нарушению компиляции.

Существует два вида инструкции *#include*:

```
#include <имя_файла>
#include "имя_файла"
```

В первой из них имя файла обрамляется угловыми скобками. Такие файлы компилятор ищет в каталоге данных терминала в подпаке MQL5/Include/.

Для второй из них, с именем в кавычках, поиск осуществляется в том же каталоге, где расположен текущий файл, в котором написана инструкция #include.

В обоих случаях файл может располагаться не прямо в каталоге поиска, а где-то во вложенных папках — тогда в директиве нужно указать всю относительную иерархию папок перед именем

файла. Например, вместе с MetaTrader 5 поставляется множество общеупотребительных загрузочных файлов, среди которых есть *DateTime.mqh* с набором методов для работы с датой и временем (они оформлены в виде еще не изученной нами языковой конструкции — структуры, но здесь это и не важно: о структурах будет рассказано в Части 3, посвященной ООП). Файл *DateTime.mqh* расположен в папке *Tools*. Соответственно, чтобы включить его в свой исходный код, необходимо использовать следующую директиву:

```
#include <Tools/DateTime.mqh>
```

Для демонстрации включения заголовочного файла из той же папки, где лежит исходный файл с директивой, рассмотрим файл *Preprocessor.mq5*. В нем имеется директива:

```
#include "Preprocessor.mqh"
```

Она ссылается на файл *Preprocessor.mqh*, который действительно находится рядом с *Preprocessor.mq5*.

Включаемый файл может, в свою очередь, подключать другие файлы. В частности, внутри *Preprocessor.mqh* находится такой код:

```
double array[] =
{
   #include "Preprocessor.txt"
};
```

Он означает, что содержимое массива инициализируется из заданного текстового файла. Если заглянуть в *Preprocessor.txt*, мы увидим текст, отвечающий синтаксическим правилам инициализации массива:

```
1, 2, 3, 4, 5
```

Таким образом, существует возможность собирать исходный код из настраиваемых компонентов, в том числе, генерировать с помощью других программ.

Обратите внимание: если файл, указанный в директиве, не найден, компиляция закончится ошибкой.

Последовательность подключения нескольких файлов определяет порядок, в котором обрабатываются находящиеся в них директивы препроцессора.

2.9.2 Обзор директив макроподстановки

К директивам макроподстановки относятся две формы директивы #define:

- простая, как правило, для определения константы;
- определение макроса в виде псевдо-функции с параметрами.

Кроме того, существует директива *#undef* для отмены любого из предыдущих определений *#define*. Если *#undef* не используется, каждый определенный макрос действует до конца компиляции исходных кодов.

Макросы регистрируются и затем используются в коде по имени, составленном по правилам идентификаторов. По всеобщей договоренности имена макросов пишутся заглавными буквами. Имена макросов могут перекрывать имена переменных, функций и прочих элементов исходного кода. Целенаправленное использование этого факта позволяет гибко менять и генерировать

исходный код на лету. Однако непреднамеренное совпадение имени макроса с элементом программы приведет к ошибкам.

Принцип работы обеих форм макроподстановок общий. С помощью директивы *#define* вводится идентификатор, которому ставится в соответствие некий фрагмент текста — определение. Если препроцессор находит далее в исходном коде данный идентификатор, он заменяет его на связанный с ним текст. Подчеркнем, что имя макроса может быть использовано в компилируемом коде только после регистрации (это похоже на принципы объявления переменных, но только на стадии компиляции).

Замена имени макроса на его определение называется раскрытием или разверткой (от английского "expand"). Анализ исходного кода происходит поступательно и построчно в один проход, но развертка в каждой строке может производиться произвольное число раз, как в цикле, пока в получившемся результате вновь обнаруживаются имена макросов. Включать в определение макроса его же имя нельзя: при подстановке такой макрос приведет к ошибке "неизвестный идентификатор".

В третьей Части книги мы узнаем о шаблонах, которые также позволяют генерировать (фактически, реплицировать) исходный код, но по другим правилам. При наличии в исходном коде и директив макроподстановки, и шаблонов, первыми раскрываются макросы, а затем генерируется код по шаблонам.

Имена макросов подсвечиваются в редакторе MetaEditor красным цветом.

2.9.3 Простая форма #define

Простая форма директивы #define регистрирует идентификатор и последовательность символов, на которую этот идентификатор следует заменить везде в исходных кодах после директивы, вплоть до конца программы или до директивы #undef с тем же идентификатором.

Её синтаксис таков:

```
#define макро_идентификатор [текст]
```

Текст начинается после идентификатора и продолжается до конца текущей строки. Идентификатор и текст должно разделять произвольное количество пробелов или табуляций. Если требуемая последовательность символов слишком длинная, то для удобочитаемости можно переносить её на следующие строки, ставя в конце прерываемой строки символ обратной косой черты '\'.

```
#define макро_идентификатор текст_начало \
                            текст_продолжение \
                            текст_окончание
```

Текст может состоять из любых конструкций языка: констант, операторов, идентификаторов, знаков пунктуации. После подстановки в исходный код вместо найденных упоминаний *макро_идентификатора*, все они будут участвовать в компиляции.

Простая форма традиционно используется для:

- объявления флагов, которые затем используются для проверок условной компиляции;
- объявления именованных констант;
- сокращенной записи расхожих инструкций.

Первый пункт характерен тем, что после идентификатора можно ничего не указывать — наличия директивы с именем уже достаточно, чтобы соответствующий идентификатор был зарегистрирован и мог использоваться в условных директивах *#ifdef/ifndef*. Для них важно лишь, существует идентификатор или нет, то есть он работает в режиме флага: объявлен/не объявлен. Например, нижеприведенная директива определяет флаг DEMO:

```
#define DEMO
```

Его затем можно применять, предположим, для сборки демо-версии программы, из которой исключаются некоторые функции (см. пример в разделе условной компиляции).

Второй вариант использования простой директивы позволяет заменить в исходном коде "магические числа" понятными именами. "Магическими числами" называют вставленные в исходный текст константы, смысл которых не всегда ясен (потому что число — это просто число: его желательно хотя бы пояснить в комментарии). Кроме того, одно и то же значение может быть разбросано по разным частям кода, и если программист решит изменить его на другое, то ему придется делать это во всех местах (и надеяться, что он ничего не упустил).

При наличии макроса с именем эти две проблемы легко решаются. Например, скрипт может подготавливать массив с числами ряда Фибоначчи на некую максимальную глубину. Тогда имеет смысл определить макрос с предопределенным размером массива и использовать его в описании самого массива (*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;
   }
}
```

Если программист впоследствии решит, что размер массива нужно увеличить, ему достаточно сделать это в одном месте — в директиве *#define*. Таким образом, директива фактически определяет некий параметр алгоритма, который "зашит" в исходный код и недоступен для настройки пользователю. Потребность в этом возникает достаточно часто.

Может возникнуть вопрос, чем определение через *#define* отличается от переменной-константы в глобальном контексте. Действительно, мы могли бы описать переменную с таким же именем и назначением, и даже сохранить верхний регистр букв:

```
const int MAX_FIBO = 10;
```

Однако в этом случае MQL5 не позволит определить массив с указанным размером, так как в квадратных скобках допускаются только константы, т.е. литералы (а переменная-константа,

несмотря на свое похожее название, константой не является). Чтобы решить эту проблему, мы могли бы определить массив как динамический (без предварительного указания размера) и затем выделять для него память с помощью функции *ArrayResize* — здесь сложностей с передачей переменной в качестве размера не возникает.

Альтернативный способ определить именованную константу предоставляют перечисления, но он ограничен только целочисленными значениями. Например:

```
enum
{
   MAX_FIBO = 10
};
```

Макрос же способен содержать значение любого типа.

```
#define TIME_LIMIT     D'2023.01.01'
#define MIN_GRID_STEP  0.005
```

Поиск имен макросов в исходных текстах для замены производится с учетом синтаксиса языка, то есть неделимые элементы, такие как идентификаторы переменных или строковые литералы останутся без изменений, даже если в их состав входит подстрока, совпадающая с одним из макросов. Например, с учетом нижеприведенного макроса XYZ, переменная XYZAXES сохранится как есть, а имя XYZ (поскольку оно целиком совпадает с макросом) будет заменено на ABC.

```
#define XYZ ABC
int XYZAXES = 3; // int XYZAXES = 3
int XYZ = 0;     // int ABC = 0
```

С помощью макроподстановок можно встраивать свой код в исходный текст других программ. Обычно этот прием используется библиотеками, которые распространяются в виде заголовочных mqh-файлов и подключаются к программам с помощью директив *#include*.

В частности, для скриптов мы можем определить свою собственную, библиотечную реализацию функции *OnStart*, которая должна выполнить некие дополнительные действия, не затронув при этом исходный функционал программы.

```
void OnStart()
{
   Print("OnStart wrapper started");
   // ... дополнительные действия
   _OnStart();
   // ... дополнительные действия
   Print("OnStart wrapper stopped");
}

#define OnStart _OnStart
```

Допустим, эта часть находится в подключаемом заголовочном файле (*Preprocessor.mqh*).

Тогда оригинальная функция *OnStart* (в *Preprocessor.mq5*) будет переименована препроцессором в исходном коде на *_OnStart* (подразумевается, что там этот идентификатор не используется еще где-то по другому назначению). А новая версия *OnStart* из заголовка вызывает *_OnStart*, "обернув" её в дополнительные инструкции.

Третий распространенный способ использования простого *#define* — сокращение записи языковых конструкций. Например, заголовок бесконечного цикла можно обозначить одним словом LOOP:

```
#define LOOP for( ; !IsStopped() ; )
```

И потом применять в коде:

```
LOOP
{
   // ...
   Sleep(1000);
}
```

Данный способ также является основным приемом использования директивы *#define* с параметрами (см. далее).

2.9.4 Форма #define в виде псевдо-функции

Синтаксис параметрической формы #define напоминает функцию.

```
#define макро_идентификатор(параметр,...) текст_с_параметрами
```

Такой макрос имеет один или несколько параметров в круглых скобках. Параметры разделены запятыми. Каждый параметр — это простой идентификатор (часто однобуквенный). Причем у всех параметров одного макроса идентификаторы должны различаться.

Важно, чтобы между идентификатором и открывающей круглой скобкой не было пробела — иначе макрос будет воспринят как простая форма, в которой текст для замены начинается с открывающей круглой скобки.

После регистрации данной директивы препроцессор будет искать в исходных кодах строки вида:

```
макро_идентификатор(выражение,...)
```

В них вместо параметров могут быть указаны произвольные выражения. Количество аргументов должно совпадать с количеством параметров макроса. Все найденные вхождения будут заменены на *текст_с_параметрами*, в котором, в свою очередь, параметры будут заменены на переданные выражения. Каждый параметр может встретиться несколько раз, в произвольном порядке.

Например, следующий макрос позволяет найти максимальное из двух значений:

```
#define MAX(A,B) ((A) > (B) ? (A) : (B))
```

Если в коде будет инструкция:

```
int z = MAX(x, y);
```

она "раскроется" препроцессором в:

```
int z = ((x) > (y) ? (x) : (y));
```

Макроподстановка будет работать для любых типов данных (для которых допустимы примененные внутри макроса операции).

Вместе с тем, подстановка может иметь и побочные эффекты. Например, если в качестве фактического параметра указан вызов функции или инструкция с модификацией переменной

(скажем, *++x*), то соответствующее действие может быть выполнено несколько раз (вместо предполагаемого одного раза). В случае MAX это случится дважды: во время сравнения и при получении значений в одной из веток оператора '?:'. В связи с этим имеет смысл по возможности преобразовывать подобные макросы в функции (особенно учитывая, что в MQL5 для функций автоматически применяется инлайнинг).

Обратите внимание на круглые скобки вокруг параметров и вокруг всего определения макроса. Они нужны, чтобы гарантировать, что подстановка выражений в качестве параметров или самого макроса внутрь других выражений не исказит порядок вычислений в силу разных приоритетов. Допустим, макрос определяет умножение двух параметров (пока не заключенных в скобки):

```
#define MUL(A,B) A * B
```

Тогда использование макроса со следующими выражениями приведет к неожиданному результату:

```
int x = MUL(1 + 2, 3 + 4); // 1 + 2 * 3 + 4
```

Вместо умножения *(1 + 2) * (3 + 4)*, что дает 21, мы получили *1 + 2 * 3 + 4*, то есть 11. Свободное от этой проблемы определение макроса должно быть таким:

```
#define MUL(A,B) ((A) * (B))
```

В качестве параметра макроса можно указать другой макрос. Кроме того, в определении макроса также можно вставлять другие макросы. Все такие макросы будут подменяться последовательно. Например:

```
#define SQ3(X) (X * X * X)
#define ABS(X) MathAbs(SQ3(X))
#define INC(Y) (++(Y))
```

Тогда следующий код выведет 504 (*MathAbs* — встроенная функция возвращающая модуль числа, т.е. без знака):

```
int x = -10;
Print(ABS(INC(x)));
// -> ABS(++(Y))
// -> MathAbs(SQ3(++(Y)))
// -> MathAbs((++(Y))*(++(Y))*(++(Y)))
// -> MathAbs(-9*-8*-7)
// -> 504
```

В переменой *x* при этом останется значение -7 (за счет троекратного инкремента).

Определение макроса может содержать непарные скобки. Такой прием используется, как правило, в паре макросов, один из которых должен открывать некий фрагмент кода, а другой закрывать. При этом непарные в каждом из них скобки станут парными. В частности, в файлах стандартной библиотеки из поставки MetaTrader 5, в *Controls/Defines.mqh* определены макросы EVENT_MAP_BEGIN и EVENT_MAP_END. С помощью них формируется функция обработки событий в графических объектах.

Напомним, что препроцессор читает весь исходный текст программы строка за строкой, начиная с основного mq5-файла и вставляя по месту тексты из встретившихся заголовочных файлов. К моменту чтения любой строки кода формируется некий набор макросов, которые уже определены. При этом не важно, в какой последовательности макросы были определены: вполне допустимо, что один макрос ссылается в своем определении на другой, который был описан как

выше, так и ниже по тексту. Важно лишь, чтобы в той строке исходного кода, где используется имя макроса, были известны определения всех макросов, на которые есть ссылки.

Рассмотрим пример.

```
#define NEG(x) (-SQN(x))*TEN
#define SQN(x) ((x)*(x))
#define TEN 10
...
Print(NEG(2)); // -40
```

Здесь макрос NEG использует макросы SQN и TEN, которые описаны ниже него. И это не мешает успешно применять его в коде после всех трех *#define*-ов.

Однако если изменить относительное положение строк на следующее:

```
#define NEG(x) (-SQN(x))*TEN
#define SQN(x) ((x)*(x))
...
Print(NEG(2)); // error: 'TEN' - undeclared identifier
...
#define TEN 10
```

получим ошибку компиляции "неизвестный идентификатор" ("undeclared identifier").

2.9.5 Специальные операторы '#' и '##' внутри определений #define

Внутри определений макросов можно использовать два специальных оператора:

- Одиночный символ решетки '#', стоящий перед именем параметра макроса, превращает содержимое этого параметра в строку. Разрешен только в макросах-функциях.
- Сдвоенный символ решетки '##', стоящий между двумя словами (токенами) объединяет их. Причем если токен является параметром макроса, то подставляется его значение, а если токен — имя макроса, оно подставляется как есть, без раскрытия макроса. Если в результате "склейки" получается другое имя макроса, он раскрывается.

В примерах данной книги мы часто использовали следующий макрос:

```
#define PRT(A) Print(#A, "=", (A))
```

Он вызывает функцию *Print*, в которой переданное выражение отображается в виде строки благодаря #A, а после знака равно выводится актуальное значение A.

Для демонстрации '##' рассмотрим другой макрос:

```
#define COMBINE(A,B,X) A##B(X)
```

С помощью него мы можем фактически сгенерировать вызов определенного выше макроса SQN:

```
Print(COMBINE(SQ,N,2)); // 4
```

Литералы SQ и N объединяются, после чего макрос SQN раскрывается в ((2)*(2)) и дает результат 4.

Следующий макрос позволяет создать определение переменной в коде, сгенерировав её имя с учетом параметров макроса:

```
#define VAR(TYPE,N) TYPE var##N = N
```

Тогда строка кода:

```
VAR(int, 3);
```

эквивалентна записи:

```
int var3 = 3;
```

Сцепка токенов позволяет реализовать с помощью макроса краткую форму записи цикла по элементам массива.

```
#define for_each(I, A) for(int I = 0, max_##I = ArraySize(A); I < max_##I; ++I)

// описываем и как-то заполняем массив x
double x[];
// ...
// выполняем цикл по массиву
for_each(i, x)
{
   x[i] = i * i;
}
```

2.9.6 Отмена макроподстановки (#undef)

Зарегистрированные с помощью *#define* подстановки можно отменить, если они становятся не нужны после конкретного фрагмента кода. Для этих целей применяется директива *#undef*.

```
#undef макро_идентификатор
```

В частности, она полезна, если в разных частях кода требуется определить один и тот же макрос по-разному. Если идентификатор, указанный в *#define*, уже был зарегистрирован где-то в более ранних строках кода (другой директивой *#define*), то старое определение заменяется на новое, а препроцессор выдает предупреждение "переопределение макроса" ("macro redefinition"). Использование *#undef* позволяет избавиться от предупреждения, и явным образом документирует намерение программиста не использовать дальше по коду конкретный макрос.

С помощью *#undef* нельзя отменить встроенные макроопределения.

2.9.7 Предопределенные константы препроцессора

В MQL5 имеется несколько предопределенных констант, которые эквивалентны простым макросам, но определены они самим компилятором. В следующей таблице приведены некоторые их имена и значения.

Имя	Значение
__COUNTER__	Счетчик (каждое упоминание в тексте в процессе развертки макросов приводит к увеличению на 1)
__DATE__	Дата (день) компиляции

Имя	Значение
__DATETIME__	Дата и время компиляции
__FILE__	Название компилируемого файла
__FUNCSIG__	Сигнатура текущей функции
__FUNCTION__	Имя текущей функции
__LINE__	Номер строки в компилируемом файле
__MQLBUILD__, __MQL5BUILD__	Версия компилятора
__RANDOM__	Случайное число типа ulong
__PATH__	Путь к компилируемому файлу
_DEBUG	Определен при компиляции в отладочном режиме
_RELEASE	Определен при компиляции в штатном режиме

2.9.8 Условная компиляция (#ifdef/#ifndef/#else/#endif)

Директивы условной компиляции позволяют включать и исключать фрагменты кода из процесса компиляции. Директивы *#ifdef* и *#ifndef* обозначает начало управляемого ими фрагмента кода. Завершается фрагмент с помощью директивы *#endif*. В простейшем случае синтаксис *#ifdef* следующий:

```
#ifdef макро_идентификатор
   инструкции
#endif
```

Если макрос с указанным идентификатором определен выше по коду с помощью *#define*, то данный фрагмент кода будет участвовать в компиляции. В противном случае, он исключается. Помимо макросов, определенных в прикладном коде, среда предоставляет набор предопределенных констант, в частности флаги _RELEASE и _DEBUG (см. раздел Предопределенные константы): их имена также можно проверять в директивах условной компиляции.

Расширенная форма *#ifdef* позволяет указать два фрагмента кода: первый будет включен, если макро-идентификатор определен, а второй — в его отсутствии. Для этого между *#ifdef* и *#endif* вставляется разделитель фрагментов — *#else*.

```
#ifdef макро_идентификатор
   инструкции_истина
#else
   инструкции_ложь
#endif
```

Директива *#ifndef* работает аналогично, но фрагменты включаются и исключаются по обратной логике: если указанный в заголовке макрос не определен, компилируется первый фрагмент, а если определен — компилируется второй.

Например, в зависимости от наличия макроподстановки DEMO, мы можем вызывать или не вызывать функцию расчета чисел Фибоначчи.

```
#ifdef DEMO
   Print("Fibo is disabled in the demo");
#else
   FillFibo();
#endif
```

В данном случае при включенном режиме DEMO вместо вызова функции в журнал выводилось бы сообщение, но поскольку в скрипте *Preprocessor.mq5* и всех включенных файлах нет определения *#define* DEMO, компиляция идет по ветке *#else*, то есть в исполняемый ex5-файл попадает вызов функции *FillFibo*.

Директивы можно делать вложенными.

```
#ifdef _DEBUG
   Print("Debugging");
#else
   #ifdef _RELEASE
      Print("Normal run");
   #else
      Print("Undefined mode!");
   #endif
#endif
```

2.9.9 Общие свойства программ (#property)

С помощью директивы *#property* программист может установить некоторые свойства MQL-программы. Часть этих свойств носит общий характер, то есть применима к любой программе, и мы рассмотрим их здесь. Остальные свойства характерны для конкретных типов MQL-программ и будут рассмотрены в соответствующих разделах пятой Части при описании MQL5 API.

Директива *#property* имеет следующий формат:

```
#property ключ значение
```

В качестве ключа пишется одно из свойств, приведенных в следующей таблице, в первом столбце. Во втором столбце указано, как будет интерпретировано значение.

Свойство	Значение
copyright	Строка с информацией о правообладателе
link	Строка с ссылкой на сайт разработчика
version	Строка с номером версии программы (для Маркета MQL5 она должна быть в формате "X.Y", где X и Y — целые числа, соответствующие основному и дополнительному номеру сборки)
description	Строка с описанием программы (допускается несколько директив *#description*, их содержимое объединяется)
icon	Строка, путь к файлу с логотипом программы в формате ICO

Свойство	Значение
stacksize	Целое число, задающее размер стека в байтах (по умолчанию он равен от 4 до 16 Мбайт, в зависимости от типа программы и окружения, 1 МБайт = 1024*1024 байт); при необходимости размер увеличивается вплоть до 64 Мбайт (максимум)

Все вышеуказанные строковые свойства являются источником информации для диалога свойств программы, который открывается при её запуске. Однако для скриптов этот диалог по умолчанию не выводится. Чтобы изменить данное поведение, требуется дополнительно указать директиву *#property script_show_inputs*. Кроме того, информация о правах выводится во всплывающей подсказке при наведении курсора мыши на программу в *Навигаторе* MetaTrader 5.

Свойства *copyright*, *link* и *version* мы уже видели во всех предыдущих примерах данной книги.

Размер стека *stacksize* является рекомендательным: если компилятор обнаружит в исходном коде локальные переменные (как правило, массивы), превышающие заданное значение, стек будет автоматически увеличен в процессе компиляции, но не более чем до 64 МБайт. При превышении лимита программа не сможет даже стартовать: в журнале (вкладка *Журнал*, а не *Эксперты*) появится ошибка "Размер стека превысил 64Mb, уменьшите объем памяти локальных переменных" ("Stack size of 64MB exceeded. Reduce the memory occupied by local variables").

Обратите внимание, что такой анализ и предотвращение запуска учитывают только фиксированный "слепок" программы на момент старта. В случае рекурсивных вызовов функций расход памяти на стек может существенно возрасти и привести к ошибке переполнения стека, но уже на стадии выполнения программы. Подробнее о стеке см. врезку в разделе Описание массивов.

Директивы *#property* работают только в компилируемом mq5-файле, а во всех включенных с помощью *#include* — игнорируются.

Часть 3. Объектно-Ориентированное Программирование на MQL5

В процессе разработки программ рано или поздно возникает проблема с тем, что встроенных типов и набора функций становится недостаточно для эффективной реализации планов. Сложность управления множеством мелких сущностей, из которых состоит программа, нарастает как снежный ком, и требует применения каких либо технологий для повышения удобства, производительности и качества труда программиста.

Одна из таких технологий, реализуемая на уровне многих языков программирования, называется Объектно-Ориентированной, а программирование с её использованием — соответственно, Объектно-Ориентированным Программированием (ООП). Язык MQL5 также её поддерживает и потому относится к семейству объектно-ориентированных языков, как и C++.

Из самого названия технологии можно заключить, что она построена на объектах. По сути, объект — это переменная пользовательского типа, то есть типа, определенного программистом средствами MQL5. Возможность создавать типы, моделирующие предметную область, делает программы более понятными, упрощает их написание и сопровождение.

В MQL5 существует несколько способов определить новый тип, причем каждый способ характеризуется некоторыми особенностями, которые мы опишем в соответствующих разделах. В зависимости от способа описания, пользовательские типы делятся на классы, структуры и объединения. Все они могут сочетать в себе данные и алгоритмы, то есть описывают состояние и поведение прикладных объектов.

В первой Части книги мы приводили фразу одного из отцов программирования Никлауса Вирта о том, что программы — это симбиоз алгоритмов и структур данных. Получается, что и объекты представляют собой мини-программы — каждая ответственна за решение собственной, пусть и небольшой, но логически завершенной задачи. Компонуя объекты в единую систему, можно выстроить сервис или продукт произвольной сложности. Таким образом, в ООП мы находим новое прочтение принципа "разделяй и властвуй".

ООП следует рассматривать как более мощную и гибкую альтернативу процедурному стилю программирования, который мы изучили во второй Части. Вместе с тем, не следует противопоставлять оба подхода: при необходимости их можно комбинировать, а в наиболее простых задачах можно обойтись и без ООП.

Итак, в этой третьей Части книги мы изучим принципы ООП и возможности их практической реализации на MQL5. Кроме того, мы расскажем о шаблонах, интерфейсах и пространствах имен.

Программирование на MQL5 для трейдеров — исходные коды из книги: Часть 3

Примеры из книги также доступны в публичном проекте \MQL5\Shared Projects\MQL5Book

3.1 Структуры и объединения

Структура является наиболее простым для понимания объектным типом, поэтому мы начнем знакомство с ООП именно с него. У структур много общего с классами, которые являются основным строительным блоком в ООП, так что знание структур поможет в дальнейшем при переходе к классам. Вместе с тем, у структур есть некоторые отличия: часть из них можно

рассматривать как ограничения, а часть — как преимущества. В частности, структуры не могут иметь виртуальных функций, зато они применяются для интеграции со сторонними DLL-библиотеками.

Выбор между структурами и классами при реализации алгоритма традиционно базируется на требованиях к доступу к элементам объекта и наличию внутренней бизнес-логики. Если необходим простой контейнер со структурированными данными и их состояние не нужно проверять на корректность (в программировании это называется "инвариантом"), то подойдет структура. Если требуется ограничить доступ и поддерживать запись и чтение по некоторым правилам (которые формализуются в виде функций, приписанных к объекту, о чем мы далее поговорим), то лучше воспользоваться классами.

В MQL5 имеются встроенные типы структур, описывающие востребованные для трейдинга сущности, в частности, котировки (*MqlRates*), тики (*MqlTick*), дату и время (*MqlDateTime*), торговые запросы (*MqlTradeRequest*), результаты запросов (*MqlTradeResult*) и многие другие. О них мы поговорим в шестой Части книги.

3.1.1 Определение структур

Структура состоит из переменных, которые могут иметь встроенные или другие пользовательские типы. Назначение структуры — объединить логически связанные данные в едином контейнере. Представим, что у нас есть функция, выполняющая некий расчет и принимающая набор параметров: количество баров в истории котировок для анализа, дата начала анализа, тип цены, количество выделяемых сигналов (например, гармоник).

```
double calculate(datetime start, int barNumber,
                 ENUM_APPLIED_PRICE price, int components);
```

В реальности параметров может быть больше и передавать их в функцию списком будет неудобно. Кроме того, по результатам нескольких расчетов имеет смысл сохранить несколько лучших вариантов настроек в какой-то массив. Поэтому набор параметров удобно представить единым объектом.

Описание структуры с теми же переменными выглядит следующим образом:

```
struct Settings
{
   datetime start;
   int barNumber;
   ENUM_APPLIED_PRICE price;
   int components;
};
```

Описание начинается с ключевого слова *struct*, за которым идет выбранный нами идентификатор. Далее следует блок кода в фигурных скобках, а внутри него — описания переменных в составе структуры. Они еще называются полями или членами структуры. После фигурных скобок стоит точка с запятой, так как всё вместе это — инструкция определения нового типа, а после инструкций требуется ';'.

После определения типа мы можем его применять точно также как и встроенные типы. В частности, новый тип разрешает описать в программе переменную *Settings* привычным образом.

```
Settings s;
```

Важно отметить, что одиночное описание структуры позволяет создавать произвольное количество переменных-структур и даже массивов этого типа. В каждом экземпляре структуры будет выделен свой собственный набор элементов, и они будут содержать независимые значения.

Для обращения к членам структуры предусмотрен специальный оператор разыменования — символ точка '.'. Слева от него должна стоять переменная типа структуры, а справа — идентификатор одного из имеющихся в ней полей. Вот каким образом можно присвоить значение элементу структуры:

```
void OnStart()
{
   Settings s;
   s.start = D'2021.01.01';
   s.barNumber = 1000;
   s.price = PRICE_CLOSE;
   s.components = 8;
}
```

Существует более удобный способ заполнить структуру — агрегатная инициализация. В этом случае справа от переменной-структуры пишется знак '=' и далее в фигурных скобках список начальных значений всех полей через запятую.

```
   Settings s = {D'2021.01.01', 1000, PRICE_CLOSE, 8};
```

Типы значений должны совпадать с типами соответствующих элементов. Допускается указать меньшее количество значений, чем количество полей: тогда оставшиеся поля получат нулевые значения.

Обратите внимание, что этот способ работает только при инициализации переменной, во время её определения. Присвоить таким образом содержимое уже существующей структуре нельзя, мы получим ошибку компиляции.

```
   Settings s;
   // error: '{' - parameter conversion not allowed
   s = {D'2021.01.01', 1000, PRICE_CLOSE, 8};
```

С помощью оператора разыменования можно также и прочитать значение элемента структуры. Например, используем количество баров для расчета количества компонентов.

```
   s.components = (int)(MathSqrt(s.barNumber) + 1);
```

Здесь *MathSqrt* — это встроенная функция взятия квадратного корня.

Мы ввели новый тип *Settings* для упрощения передачи набора параметров в функцию. Теперь его можно использовать в качестве единственного параметра обновленной функции *calculate*:

```
double calculate(Settings &settings);
```

Обратите внимание на знак амперсанда '&' перед именем параметра, что означает передачу по ссылке. Структуры можно передавать в качестве параметров только по ссылке.

Структуры также бывают полезны, если из функции нужно вернуть не одно значение, а набор. Представим, что функция *calculate* должна вернуть не значение типа *double*, а несколько

коэффициентов и некие торговые рекомендации (направление сделок и вероятность успеха). Тогда мы можем определить тип структуры *Result* и использовать его в прототипе функции (*Structs.mq5*).

```
struct Result
{
   double probability;
   double coef[3];
   int direction;
   string status;
};

Result calculate(Settings &settings)
{
   if(settings.barNumber > 1000) // редактируем поля
   {
      settings.components = (int)(MathSqrt(settings.barNumber) + 1);
   }
   // ...
   // эмулируем получение результата
   Result r = {};
   r.direction = +1;
   for(int i = 0; i < 3; i++) r.coef[i] = i + 1;
   return r;
}
```

Пустые фигурные скобки в строке *Result r = {}* представляют собой минимальный агрегатный инициализатор: он заполняет все поля структуры нулями.

Определение и объявление типа структуры можно, при необходимости, сделать отдельно (как правило, объявление идет в заголовочном mqh-файле, а определение - в mq5-файле). Данный расширенный синтаксис будет рассмотрен в Главе про классы.

3.1.2 Функции (методы) в структурах

После получения результата из функции *calculate* было бы желательно его вывести в журнал, но функция *Print* не работает с пользовательскими типами: они должны сами предоставить способ отображения.

```
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
}
```

В комментариях представлены попытки вызвать функцию *Print* для структуры и что из этого получается. Первая ошибка вызвана тем, что экземпляры структур являются объектами, а объекты должны передаваться в функции по ссылке. Вместе с тем *Print* ожидает значение (одно или несколько). Использование амперсанда перед именем переменной во втором вызове *Print* обозначает в MQL5 получение указателя, а не ссылки, как можно было бы подумать. Указатели в

MQL5 поддерживаются только для объектов классов (но не структур), отсюда и вторая ошибка "ожидается класс" ("class type expected"). Мы изучим указатели в следующей главе (см. Классы и интерфейсы).

Мы могли бы указать в вызове *Print* все члены структуры по отдельности (с помощью разыменования), но это неудобно.

Для тех случаев, когда необходимо особым образом обработать содержимое структуры, предусмотрена возможность определить внутри структуры функции. Синтаксис их определения ничем не отличается от знакомых нам функций глобального контекста, но само определение располагается внутри блока структуры.

Подобные функции называются методами. Поскольку они расположены в контексте соответствующего блока, из них можно обращаться к полям структуры без оператора разыменования. Напишем для примера реализацию функции *print* в структуре *Result*.

```
struct Result
{
   ...
   void print()
   {
      Print(probability, " ", direction, " ", status);
      ArrayPrint(coef);
   }
};
```

Вызвать метод экземпляра структуры также просто как прочитать её поле: используется тот же оператор '.'.

```
void OnStart()
{
   Settings s = {D'2021.01.01', 1000, PRICE_CLOSE, 8};
   Result r = calculate(s);
   r.print();
}
```

Более подробно о методах будет рассказано в Главе про классы.

3.1.3 Копирование структур

Структуры одинаковых типов можно копировать друг в друга целиком с помощью оператора присваивания '='. Продемонстрируем данное правило на примере структуры *Result*. Первый экземпляр *r* мы получаем из функции *calculate*.

```
void OnStart()
{
   ...
   Result r = calculate(s);
   r.print();
   // выведет в журнал:
   // 0.5 1 ok
   // 1.00000 2.00000 3.00000
   ...
   Result r2;
   r2 = r;
   r2.print();
   // выведет в журнал те же значения:
   // 0.5 1 ok
   // 1.00000 2.00000 3.00000
}
```

Затем дополнительно была создана переменная *Result r2*, и в неё продублировано содержимое переменной *r*, всех полей одновременно. В правильности операции можно убедиться по выводу в журнал с помощью метода *print* (строки приведены в комментариях).

Следует отметить, что определение двух типов структур с одинаковым набором полей не делает их типы одинаковыми. Присвоить одну из них другой целиком нельзя, а только поэлементно.

Чуть позже мы поговорим о наследовании структур, которое предоставляет расширенные возможности для копирования. Дело в том, что копирование работает не только между структурами одного и того же типа, но и между родственными типами. Однако там есть важные нюансы: мы рассмотрим их в разделе Компоновка и наследование структур.

3.1.4 Конструкторы и деструкторы

В числе методов, которые можно определить для структуры, есть специальные: конструктор(ы) и декструктор.

Конструктор имеет имя, совпадающее с именем структуры, и не возвращает значение (тип *void*). Конструктор, если он определен, будет вызываться в момент инициализации для каждого нового экземпляра структуры. За счет этого в конструкторе можно особым образом рассчитать начальное состояние структуры.

Структура может иметь несколько конструкторов, отличающихся набором параметров, и компилятор выберет соответствующий, руководствуясь числом и типом аргументов при определении переменной.

Например, мы можем описать пару конструктор в структуре *Result*: один — без параметров, и второй с одним параметром типа строка, для задания статуса.

```
struct Result
{
   ...
   void Result()
   {
      status = "ok";
   }
   void Result(string s)
   {
      status = s;
   }
};
```

Кстати говоря, конструктор без параметров называется конструктором по умолчанию. Если явных конструкторов нет, компилятор неявным образом создает конструктор по умолчанию для любой структуры, в которой есть строки и динамические массивы, чтобы заполнить эти поля нулями.

Важно, что поля других типов (например, все числовые) не обнуляются, вне зависимости от того есть ли у структуры конструктор по умолчанию, а потому начальные значения элементов после выделения памяти будут случайными. Следует либо создать конструкторы, либо убедиться, что нужные значения присваиваются в коде сразу после создания объекта.

Наличие явных конструкторов делает невозможным использование синтаксиса агрегатной инициализации. В связи с этим строка *Result r = {};* в методе *calculate* перестанет компилироваться. Теперь мы имеем право использовать только один из конструкторов, которые сами предоставили. Например, следующие инструкции вызывают конструктор без параметров:

```
Result r1;
Result r2();
```

А создание структуры с заполненным статусом можно сделать так:

```
Result r3("success");
```

Конструктор по умолчанию (явный или неявный) также вызывается при создании массива структур. Например, следующая инструкция выделяет память под 10 структур с результатами и инициализирует их с помощью конструктора по умолчанию:

```
Result array[10];
```

Деструктор — это функция, которая будет вызвана при уничтожении объекта структуры. Деструктор имеет имя, совпадающее с именем структуры, но с префиксом в виде символа тильды '~'. Деструктор также как и конструктор не возвращает значения, но он и не принимает параметров.

Деструктор может быть только один.

Явно вызвать деструктор нельзя. Программа сама делает это при выходе из блока кода, где была определена локальная переменная-структура, или при освобождении массива структур.

Назначение деструктора — освободить какие-либо динамические ресурсы, если структура их распределяла в конструкторе. Например, структура может обладать свойством *персистентности*, то есть сохранять свое состояние в файл при выгрузке из памяти и восстанавливать его, когда программа снова её создаст. При этом во встроенных файловых функциях используется дескриптор, который необходимо открывать и закрывать.

Определим деструктор в структуре *Result* и попутно дополним конструкторы, чтобы во всех этих методах велся учет количества экземпляров объектов (по мере того, как они создаются и уничтожаются).

```
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);
   }
};
```

Три статических переменных с именем *count* существуют независимо друг от друга: каждая ведет подсчет в контексте своей функции.

В результате запуска скрипта мы получим следующий журнал:

```
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
```

Разберемся, что это значит.

Первый экземпляр структуры создается в функции *OnStart*, в строке с вызовом *calculate*. При входе в конструктор значение счетчика *count* однократно инициализируется нулем и далее инкрементируется при каждом исполнении конструктора, поэтому в первый раз выводится значение 1.

Внутри функции *calculate* определяется локальная переменная типа *Result*, она зарегистрирована под номером 2.

Третий экземпляр структуры не столь очевиден. Дело в том, что для передачи результата из функции компилятор создает в неявном виде временную переменную, куда копирует данные локальной переменной. Вероятно, это поведение изменится в дальнейшем, и тогда локальный экземпляр будет "перемещаться" из функции без дублирования.

Последний вызов конструктора происходит в методе со строковым параметром, поэтому в нем счетчик вызовов равен 1.

Важно, что общее количество вызовов обоих конструкторов совпадает с количеством вызовов деструктора: 4.

Более подробно про конструкторы и деструкторы мы поговорим в Главе про классы.

3.1.5 Упаковка структур в памяти и взаимодействие с DLL

Для хранения одного экземпляра структуры в памяти выделяется непрерывная область, достаточная, чтобы уместились все элементы.

В отличие от C++ элементы структуры идут в памяти один за другим и не выравниваются по границе 2, 4, 8 или 16 байт, в зависимости от размера самих элементов (алгоритмы выравнивания отличаются у разных компиляторов и режимов работы). Выравнивание элементов, размер которых меньше указанного блока, производится путем добавления фиктивных неиспользуемых переменных в состав структуры (программа не имеет к ним прямого доступа). Выравнивание используется для оптимизации скорости работы с памятью.

MQL5 позволяет изменять правила выравнивания при необходимости, в основном при интеграции MQL-программ со сторонними DLL-библиотеками, в которых описаны конкретные типы структур. Для них необходимо подготовить эквивалентное описание в MQL5 (см. раздел об импорте библиотек). Важно отметить, что структуры, предназначенные для интеграции, должны иметь в своем определении лишь поля ограниченного набора типов. Так, в них нельзя использовать строки, динамические массивы, а также объекты классов и указатели на объекты классов.

Управление выравниванием выполняется с помощью ключевого слова *pack*, добавляемого в заголовок структуры. Существует два варианта:

```
struct pack(размер) идентификатор
struct идентификатор pack(размер)
```

В обоих случаях размер — это целое число 1, 2, 4, 8, 16. Или в качестве размера можно использовать оператор *sizeof(встроенный_тип)*, например *sizeof(double)*.

Вариант *pack(1)*, то есть выравнивание по границе байта, идентичен поведению по умолчанию без модификатора *pack*.

Узнать смещение в байтах конкретного элемента структуры от её начала позволяет специальный оператор *offsetof()*. Он имеет 2 параметра: объект структуры и идентификатор элемента. Например,

```
Print(offsetof(Result, status)); // 36
```

Перед полем *status* в структуре *Result* находятся 4 величины типа *double* и одна — *int*: итого 36.

При проектировании собственных структур рекомендуется в начале располагать самые крупные элементы, и далее остальные — в порядке уменьшения их размера.

3.1.6 Компоновка и наследование структур

Структуры могут иметь в качестве своих полей другие структуры. Например, определим структуру *Inclosure* и используем этот тип для поля *data* в структуре *Main* (*StructsComposition.mq5*):

```
struct Inclosure
{
   double X, Y;
};

struct Main
{
   Inclosure data;
   int code;
};

void OnStart()
{
   Main m = {{0.1, 0.2}, -1}; // агрегатная инициализация
   m.data.X = 1.0;            // поэлементное присвоение
   m.data.Y = -1.0;
}
```

В списке инициализации поле *data* представлено дополнительным уровнем фигурных скобок со значениями полей *Inclosure*. Для обращения к полям такой структуры нужно использовать две операции разыменования.

Если вложенная структура более нигде не используется, её можно описать непосредственно внутри внешней.

```
struct Main2
{
   struct Inclosure2
   {
     double X, Y;
   }
   data;
   int code;
};
```

Другим способом компоновки структур является наследование. Этот механизм, как правило, применяется для построения иерархии классов (и будет подробно рассмотрен в соответствующем разделе), но также доступен и для структур.

При определении нового типа структуры программист может в её заголовке, после двоеточия, указать тип родительской структуры (он должен быть определен в исходном коде раньше). В результате этого все поля родительской структуры будут добавлены в структуру-наследник (в её начало), а собственные поля новой структуры расположатся в памяти за родительскими.

```
struct Main3 : Inclosure
{
   int code;
};
```

Родительская структура является здесь не вложенной, а неотъемлемой частью дочерней структуры. Поэтому заполнение полей не требует дополнительных фигурных скобок при инициализации или цепочки из нескольких операторов разыменования.

```
Main3 m3 = {0.1, 0.2, -1};
m3.X = 1.0;
m3.Y = -1.0;
```

Все три рассмотренные структуры *Main*, *Main2*, *Main3* имеют одинаковое представление в памяти и размер 20 байтов. Но это разные типы.

```
Print(sizeof(Main));   // 20
Print(sizeof(Main2));  // 20
Print(sizeof(Main3));  // 20
```

Как мы говорили ранее (см. Копирование структур), оператор присваивания '=' можно применять для копирования родственных типов структур, а точнее для тех, которые связаны цепочкой наследования. Иными словами, структуру родительского типа можно записать в структуру дочернего типа (при этом добавленные в производной структуре поля останутся нетронутыми), или наоборот, структуру дочернего типа — записать в структуру родительского типа (при этом "лишние" поля отсекаются).

Например:

```
Inclosure in = {10, 100};
m3 = in;
```

Здесь переменная *m3* имеет тип *Main3*, унаследованный от *Inclosure*. В результате операции присваивания *m3 = in*, поля *X* и *Y* (общая часть для обоих типов) скопируются из переменной *in* базового типа в поля *X* и *Y* в переменной *m3* производного типа. Поле *code* переменной *m3* останется без изменений.

Неважно является ли структура-наследник прямым потомком прародителя или отдаленным, т.е. цепочка наследований может быть длинной. Подобное копирование общих полей работает между "детьми", "внуками" и прочими вариантами сочетания типов из разных ответвлений "родословной".

Если родительская структура имеет только конструкторы с параметрами, при наследовании необходимо вызвать его из списка инициализации при конструкторе производной структуры. Например,

```
struct Base
{
   const int mode;
   string s;
   Base(const int m) : mode(m) { }
};

struct Derived : Base
{
   double data[10];
   // если удалить конструктор, получим ошибку:
   Derived() : Base(1) { } // 'Base' - wrong parameters count
};
```

В конструкторе *Base* мы заполняем поле *mode*. Поскольку оно имеет модификатор *const*, конструктор — единственный способ установить ему значение, причем делать это нужно в виде специального синтаксиса инициализации после двоеточия (присвоить константу в теле конструктора уже нельзя). Наличие явного конструктора приводит к тому, что компилятор не генерирует неявный конструктор (без параметров). Однако у нас в структуре *Base* нет явного конструктора без параметров, а в его отсутствии любой производный класс не обладает сведениями, как правильно вызвать конструктор *Base* с параметром. Поэтому в структуре *Derived* требуется явным образом производить инициализацию базового конструктора: это так же делается с помощью синтаксиса инициализации в заголовке конструктора, после ':' — в данном случае мы вызываем *Base(1)*.

Если убрать конструктор *Derived*, получим ошибку "неверное количество параметров" в базовом конструкторе, так как компилятор пытается вызвать для *Base* конструктор по умолчанию (а у него должно быть 0 параметров).

Более подробно мы рассмотрим синтаксис и механизм наследования в Главе про классы.

3.1.7 Права доступа

При необходимости в описании структуры можно использовать специальные ключевые слова — модификаторы доступа, которые ограничивают видимость полей извне структуры. Существует три модификатора: *public*, *protected* и *private*. По умолчанию все элементы структуры являются публичными, что эквивалентно следующей записи (на примере структуры *Result*):

```
struct Result
{
public:
   double probability;
   double coef[3];
   int direction;
   string status;
   ...
};
```

Все члены ниже модификатора, пока не встретится другой модификатор или не закончится блок структуры, получают соответствующие права доступа. Секций с разными правами может быть много, они могу чередоваться произвольным образом.

Защищенные с помощью *protected* члены станут доступны только из кода данной структуры и структур-наследников, то есть подразумевается, что в них должны быть публичные методы, иначе обратиться к таким полям никто не сможет.

Закрытые с помощью *private* члены доступны только из кода данной структуры. Например, если добавить *private* перед полем *status*, то, скорее всего, потребуется метод для чтения статуса внешним кодом (*getStatus*).

```
struct Result
{
public:
   double probability;
   double coef[3];
   int direction;

private:
   string status;

public:
   string getStatus()
   {
      return status;
   }
   ...
};
```

Установить статус можно будет только через параметр второго конструктора. Прямое обращение к полю приведет к ошибке "нет доступа к закрытому члену 'status' структуры 'Result'":

```
// error:
// cannot access to private member 'status' declared in struct 'Result'
r.status = "message";
```

В классах доступом по умолчанию является *private*. Это отвечает принципу инкапсуляции, который мы рассмотрим в Главе про классы.

3.1.8 Объединения

Объединение представляет собой пользовательский тип, составленный из полей, расположенных в одной и той же области памяти, за счет чего они накладываются друг на друга. Это дает возможность записать в объединение некое значение одного типа, а затем прочитать его внутреннее представление (на уровне битов) в интерпретации для другого типа. Таким образом можно обеспечить нестандартную конвертацию из одного типа в другой.

Поля объединения могут быть любых встроенных типов, за исключением строк, динамических массивов и указателей. Также в объединениях можно использовать структуры с такими же простыми типами полей и без конструкторов/деструкторов.

Компилятор выделяет под объединение ячейку памяти размером, равным максимальному размеру среди типов всех элементов. Так, для объединения с полями типа *long* (8 байтов) и *int* (4 байта) будет выделено 8 байтов.

Все поля объединения расположены по одному и тому же адресу памяти, то есть они выровнены по началу объединения (имеют смещение 0, что можно проверить с помощью offsetof, см. раздел Упаковка структур).

Синтаксис описания объединения аналогичен структуре, но использует ключевое слово *union*. За ним идет идентификатор и далее блок кода с перечнем полей.

Например, в алгоритме может использоваться массив типа *double* для хранения различных настроек, просто потому что тип *double* один из числа тех, что имеют максимальный размер в байтах: 8. Допустим, среди настроек есть числа типа *ulong*. Поскольку тип *double* не гарантирует точное воспроизведение больших значений *ulong*, требуется воспользоваться объединением для "упаковки" *ulong* в *double* и "распаковки" обратно.

```
#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
}
```

Размер структуры *ulong2double* равен 8, поскольку оба его поля имеют этот размер. Таким образом, поля U и D полностью перекрываются.

В области целых чисел значение 9007199254740992 является наибольшим, для которого гарантированно устойчивое хранение в *double*. В данном примере мы пытаемся сохранить в *double* на единицу большее число.

Стандартная конвертация из *ulong* в *double* приводит к потере точности: после записи 9007199254740993 в переменную *d* типа *double* мы читаем из неё уже "округленное" значение 9007199254740992 (о тонкостях хранения чисел в типе *double* см. раздел Вещественные числа).

При использовании конвертера число 9007199254740993 записывается в объединение "как есть", без конвертаций, поскольку мы присваиваем его полю U типа *ulong*. Его представление с точки зрения *double* доступно, опять-таки без конвертаций, из поля D. Мы можем его копировать в другие переменные и массивы типа *double* без опасений.

Хотя полученное значение *double* выглядит странно, оно в точности соответствует исходному целому, если его потребуется извлечь путем обратной конвертации: записываем в поле D типа *double*, потом читаем из поля U типа *ulong*.

Объединение может иметь конструкторы и деструкторы, а также методы. По умолчанию члены объединения имеют публичные права доступа, но это можно откорректировать с помощью модификаторов доступа, как в структуре.

3.2 Классы и интерфейсы

Классы являются основным строительным блоком при разработке программ по технологии ООП. В глобальном смысле термин класс обозначает совокупность чего-либо (вещей, людей, формул и т.д.), которые имеют некие общие характеристики. В контексте ООП эта логика сохраняется: один класс порождает объекты, обладающие одинаковыми наборами свойств и поведением.

В предыдущих частях книги мы познакомились со встроенными типами MQL5, такими как *double*, *int* или *string*. Компилятор знает, как хранить значения этих типов и какие операции над ними можно выполнять. Однако этими типами может быть не очень удобно пользоваться при описании какой-либо прикладной области. Например, трейдеру приходится оперировать такими сущностями как торговая стратегия, фильтр сигналов, корзина валют, портфель открытых позиций. Каждая из них состоит из целого набора связанных свойств, подчиняющихся конкретным правилам обработки и непротиворечивости.

Программа для автоматизации действий с этими объектами могла бы состоять только из встроенных типов и простых функций, но тогда пришлось бы придумывать хитрые способы хранения и взаимной увязки свойств. Именно здесь на помощь приходит технология ООП, которая предоставляет уже готовые, унифицированные и интуитивно понятные механизмы для этого.

ООП предлагает прописать все инструкции для хранения свойств, их корректного заполнения, и выполнения разрешенных операций над объектами конкретного пользовательского типа в едином контейнере с исходным кодом. В нем особым образом объединены переменные и функции. Контейнеры подразделяются на классы, структуры и объединения, если перечислять их в порядке убывания возможностей и востребованности.

Со структурами и объединениями мы уже познакомились в предыдущей главе. Эти знания пригодятся и для классов, однако классы предоставляют больше инструментов из арсенала ООП.

По аналогии со структурой, класс является описанием пользовательского типа с произвольным внутренним способом хранения и правилами работы с ним. На его основе программа может создавать экземпляры данного класса — объекты, которые следует рассматривать как составные переменные.

Все пользовательские типы в той или иной мере разделяют несколько базовых концепций, которые можно назвать теорией ООП, но для классов они особенно актуальны. К их числу относятся:

- абстракция

- инкапусляция
- наследование
- полиморфизм
- композиция (дизайн)

Несмотря на мудреные названия, они скрывают за собой довольно простые и привычные нормы реального мира, перенесенные в мир программирования. Мы начнем погружение в ООП с разбора этих концепций. А синтаксис описания классов и способы создания объектов рассмотрим позднее.

3.2.1 Основы ООП: абстракция

Мы часто используем в обычной жизни обобщающие понятия для передачи сути информации, не вдаваясь в детали. Например, на вопрос "как ты сюда добрался", человек может коротко ответить: "на машине". И для всех будет ясно, что речь о средстве передвижения на 4-х колесах, с двигателем и пассажирским кузовом. Какой конкретно марки была машина, какого цвета или года выпуска — нам не важно.

При работе с программой пользователю тоже, в принципе, все равно какого рода алгоритм применен внутри, лишь бы программа корректно выполняла свою задачу. Например, сортировку списка можно выполнить десятком разных способов.

Таким образом, абстракция заключается в предоставлении простого программного интерфейса, который оставляет скрытыми все сложности и частности реализации.

Под программным интерфейсом подразумевается набор функций, которые определены в контексте класса и выполняют набор действий согласно предназначению объектов. Помимо этих интерфейсных функций могут существовать и вспомогательные, более мелкие функции, но они доступны только внутри класса. По аналогии со структурами, для всех функций класса есть специальное название — методы.

Реализация, как правило, использует для хранения информации переменные или массивы, принадлежащие объекту (согласно описанию класса). Для них также имеется особое название — поля (этот термин происходит оттого, что свойства объектов часто связаны соотношением 1:1 с полями ввода в пользовательском интерфейсе или с полями в базах данных, где может сохраняться актуальное состояние объекта для возможности его восстановления при следующем запуске программы).

Поля и методы, хотя и описываются в классе, но относятся к конкретному объекту: под каждый экземпляр выделен собственный набор переменных, они имеют значения, независимые от состояния других объектов, и методы работают с полями своего экземпляра.

> Интерфейс и реализация должны быть независимы. При желании одну реализацию должно быть легко заменить на другую без всякого влияния на программный интерфейс. Также очень важно проектировать интерфейс исходя из требований конкретной задачи, а не подгонять под реализацию. Разработчик класса должен уметь рассматривать свое творение с двух точек зрения: 1) как автор внутренних алгоритмов и структур данных; 2) как потенциальный придирчивый заказчик, который использует класс как "черный ящик", и его панелью управления является интерфейс. Рекомендуется начинать разработку класса с продумывания программного интерфейса, до поиска и выбора методов реализации.

3.2.2 Основы ООП: инкапсуляция

Чтобы понять, что такое инкапсуляция, вернемся на минуту снова в реальность. Когда мы приобретаем какой-то бытовой прибор, то он обычно "опечатан" и находится на гарантии. Нам разрешено использовать его в штатных режимах, но производитель не приветствует, если мы вскроем корпус и начнем "копаться внутри", причем не обязательно в буквальном смысле: например, чтобы разогнать процессор компьютера можно использовать специальные утилиты, но это также лишает нас гарантии, потому что может привести оборудование к выходу из строя.

С разработкой классов — аналогичная ситуация. К внутренней реализации не следует никого допускать, чтобы не нарушить работу класса. Это и называется инкапсуляцией, то есть "заключением в капсуле" всего важного. В MQL5, как и в C++, существует 3 уровня прав доступа. По умолчанию, устройство класса является приватным (private), то есть скрыто от всех его пользователей. Доступ к содержимому имеет только исходный код самого класса.

Напомним, что пользователи класса — это тоже программисты. И даже если вы пишете класс для себя, имеет смысл воспользоваться максимальными ограничениями, чтобы не сломать класс случайно (в конце концов, людям свойственно ошибаться и забывать особенности собственных разработок спустя некоторое время, а у программ есть тенденция бесконечно разрастаться).

Второй уровень доступа позволяет заглядывать внутрь только "родственникам" (точнее, наследникам — о них через пару абзацев).

Наконец, третий уровень доступа, который можно выбрать — публичный. Он как раз предназначен для внешних программных интерфейсов, позволяющих из любой части программы применять объекты по их основному назначению.

Каждый метод или поле имеют один из трех уровней доступа: какой именно — определяет разработчик класса.

3.2.3 Основы ООП: наследование

Когда люди строят что-то большое и сложное, они ищут пути повышения эффективности процесса. Один из популярных путей заключается в использовании уже имеющихся наработок. Например, гораздо легче разработать проект дома не с нуля, а на основе предыдущей серии.

При создании программ повторное использование кодов, конечно, тоже в ходу. Мы уже знаем один такой прием — выделение фрагмента кода в функцию и последующий вызов ее из разных мест, где требуется соответствующий функционал. Но ООП предоставляет более мощный механизм: при разработке нового класса его можно наследовать от другого, вместе со всем внутренним устройством и внешним интерфейсом, и затем лишь слегка подредактировать. Таким образом, отталкиваясь от родительского класса, можно быстро "вырастить" класс-наследник с дополненными или уточненными способностями. Кроме того, любые последующие изменения родительского класса (например, усовершенствования или исправления ошибок) автоматически скажутся на всех дочерних классах.

Когда какой-либо класс выступает родительским для другого, его называют базовым. В свою очередь класс-наследник называют производным.

Разумеется, цепочку наследования (а точнее, генеалогическое дерево) можно продолжать: у каждого класса может быть несколько наследников, у тех в свою очередь — свои наследники, и так далее. Единственное, чего правила наследования не позволяют, так это циклы в родственных отношениях, например, внук не может быть родителем своего деда.

Отношение между каким-либо классом и его потомком любого поколения описывается словом "является" (английское "is a"), то есть потомок способен выступать в роли предка, но не наоборот. Это объясняется тем, что производный объект фактически содержит в себе модель данных предка и дополняет её новыми полями и поведением.

Наследуя множество классов друг от друга, мы получаем возможность унифицированно обрабатывать родственные объекты: ведь часть функций у них — общая.

Например, в гипотетической программе рисования может быть реализовано несколько типов фигур: круг, квадрат, треугольник и т.д. Каждый объект имеет координаты на экране (для простоты будем считать, что задается пара значений X и Y центра фигуры). Кроме того, для отрисовки каждой фигуры используется собственный цвет фона, цвет рамки и её толщина.

Значит, функции для установки координат и настройки стиля рисования достаточно один раз реализовать в родительском классе, описывающем абстрактную фигуру, и эти функции автоматически унаследуют все потомки.

Более того, в целях упрощения исходного кода желательно каким-то образом унифицировать не только настройки, но и рисование разных фигур. В этой фразе есть некоторое противоречие: поскольку фигуры разные, и каждая должна отображаться по-своему, то о какой унификации идет речь? — Об унификации программного интерфейса. Ведь согласно концепции абстракции следует отделять внешний интерфейс от внутренней реализации. А отображение конкретных фигур — это уже детали реализации.

Единый интерфейс и разные реализации для типов фигур плавно приводят нас к следующей концепции — полиморфизму.

3.2.4 Основы ООП: полиморфизм

Термин полиморфизм означает изменчивость, многоликость. Это обратная сторона абстракции, совмещенной с механизмом наследования. Когда мы имеем некий общий программный интерфейс, его могут реализовать разные классы, связанные отношениями наследования. Тогда вызов методов интерфейса приведет к выполнению задачи разными способами.

Например, представим себе семейство абстрактных средств передвижения, включающее пару конкретных: машину и вертолет. Команда переместиться из точки А в точку Б будет выполнена ими одинаково успешно, но машина проделает путь по земле, а вертолет — по воздуху.

Для большей наглядности продолжим пример с программой рисования. Можно сказать, что многоликость в неё заложена на уровне графических фигур. Пользователь волен нарисовать произвольное сочетание кругов, квадратов и треугольников. Каждый из этих объектов должен уметь отображать себя на экране по своим координатам и используя свой стиль, но самое главное — делать это в виде соответствующей формы.

В программе наверняка будет массив (или другой контейнер), хранящий все созданные пользователем фигуры, и вывод на экран рисунка целиком должен заключаться в последовательном рисовании каждой фигуры. Если мы сведем инструкции рисования для фигур в отдельный метод (пусть он так и называется *draw*), то для каждого класса получится своя реализация. Однако заголовок этих функций будет полностью идентичен, поскольку они выполняют одну и ту же задачу, и берут исходные данные из объектов.

Следовательно, мы получили возможность унифицировать исходный код, поскольку один и тот же вызов *draw* внутри цикла по фигурам проявляет полиморфизм: отображаемая форма будет зависеть от типа объекта.

3.2.5 Основы ООП: композиция (дизайн)

При проектировании программ с применением ООП встает задача нахождения оптимального (по некоторым заданным характеристикам) разбиения на классы и отношений между ними. В русском языке для обозначения этого лучше всего подошел бы термин композиция (как составление, соединение частей в единое целое). К сожалению, данный термин сильно перегружен в английской терминологии, и используется с разными значениями, включая и один из частных случаев "составления" классов. Это отступление необходимо, потому что при чтении другой компьютерной литературы Вы можете найти различные толкования термина "композиции": как в обобщенном, так и более узком смысле. Мы постараемся изложить данную концепцию, конкретизируя смысл терминов в каждом случае (когда подразумевается общий "дизайн/ проектирование" программного интерфейса, а когда — "композиционное агрегирование").

Итак, класс, как мы знаем, состоит из полей (свойств) и методов. Свойства, в свою очередь, могут быть описаны пользовательскими типами, то есть представлять собой объекты другого класса. При этом существует несколько способов логического соединения этих объектов:

- **Композиция** (полное **включение** или композиционное агрегирование) объектов-полей в объект-владелец. Связь таких объектов описывается отношением "целое-часть", причем часть не может существовать вне целого. Говорят, что объект-владелец "имеет" ("has a") объект-свойство, а объект-свойство "является частью" ("part of") объекта-владельца. Владелец создает и уничтожает свои части. Удаление владельца приводит к удалению всех его частей; владелец не может существовать без частей.
- **Агрегирование** объектов-полей объектом-владельцем представляет собой более "мягкое" включение. Хотя отношение также описывается как "целое-часть", владелец лишь содержит ссылки на части, которые могут назначаться, меняться и существовать в отрыве от целого. Более того, одна часть может использоваться в нескольких "владельцах".
- **Ассоциация**, то есть одно- или двухсторонняя связь независимых объектов, имеющая произвольный прикладной смысл. Говорят, что один объект "использует" другой.

Не следует забывать еще один тип отношений: "является" ("is a"), рассмотренный ранее в разделе про наследование.

В качестве примера полного включения можно привести машину и её двигатель. Здесь под машиной понимается полноценное средство передвижения. Без мотора это не так. И конкретный двигатель принадлежит в каждый момент времени только одной машине. Ситуации, когда двигателя в машине еще нет (на заводе) или уже нет (в автомастерской), эквивалентны тому, что мы сломали исходный код программы.

Примером агрегирования является состав групп студентов для занятий по определенным курсам: группа для каждого курса включает несколько студентов, причем любой из них может относиться и к другим группам (если слушает несколько предметов). Группа "имеет" слушателей. Выход студента из состава группы, не сказывается на учебном процессе группы (остальные продолжают учиться).

Наконец, для демонстрации идеи ассоциации рассмотрим компьютер и принтер. Можно сказать, что компьютер использует принтер для печати. Принтер может быть включен или выключен по

необходимости, причем один и тот же принтер можно использовать с разных компьютеров. Все компьютеры и принтеры существуют независимо друг от друга, но могут использоваться совместно.

Что касается характеристик, которыми принято руководствоваться при проектировании классов, то к наиболее известным относятся:

- DRY (Don't repeat yourself) — "не дублируй код": вместо этого выноси общие части в родительские (по возможности, абстрактные) классы;
- SRP (Single Responsibility Principle) — "разделяй сферы ответственности": один класс должен выполнять одну задачу, а если это не так — нужно его раздробить на более мелкие;
- OCP (Open-Closed Principle) — "пиши код открытым для расширения, но закрытым для модификации": если в классе X "зашито" несколько вариантов расчета и могут появиться новые, сделай базовый (абстрактный) класс для отдельного расчета и на его основе создавай специфические варианты ("расширение" функционала), подключаемые к классу X без его модификации.

Это лишь малая часть лучших практик проектирования классов. После освоения основ ООП, ограниченных рамками данной книги, может быть полезно познакомиться с другими специализированными источниками информации по данной теме, поскольку в них содержатся готовые решения для декомпозиции на объекты многих типичных ситуаций.

3.2.6 Определение класса

Инструкция для определения нового класса имеет много необязательных компонентов, которые влияют на его характеристики. В обобщенном виде её можно представить так:

```
class имя_класса [: модификатор_доступа имя_родительского_класса ...]
{
   [ модификатор_доступа:]
      [описание_члена ...]
   ...
};
```

Для простоты изложения мы начнем с минимального достаточного синтаксиса и будем дополнять его по мере продвижения по материалу.

В качестве полигона используем задачу с условной программой для рисования, поддерживающей несколько типов фигур.

Для определения нового класса используется ключевое слово class, после которого идет идентификатор класса и блок кода в фигурных скобках. Как и все инструкции, такое определение требуется завершить точкой с запятой.

Блок кода может быть пустым. Например, компилируемая заготовка класса *Shape* для программы рисования выглядит так:

```
class Shape
{
};
```

По предыдущей части книги мы знаем, что фигурные скобки обозначают контекст или область видимости переменных. Когда такие блоки встречаются в определении функции, они задают её

локальный контекст. Кроме него существует глобальный контекст, в котором определяются сами функции, а также глобальные переменные.

На этот раз скобки в определении класса задают новый вид контекста — контекст класса. Он является контейнером как для переменных, так и функций, описанных внутри класса.

Описание переменных для хранения свойств класса делается привычными нам инструкциями внутри блока (*Shapes1.mq5*).

```
class Shape
{
   int x, y;              // координаты центра
   color backgroundColor; // цвет заливки
};
```

Здесь мы задекларировали некоторые поля, о которых рассуждали в теоретических разделах: координаты центра фигуры и цвет заливки.

После такого описания пользовательский тип *Shape* становится доступным в программе наравне со встроенными типами. В частности, мы можем создать переменную этого типа, и она будет содержать внутри указанные поля. Однако сделать что-либо с ними и даже убедиться, что они там есть, мы пока не можем.

```
void OnStart()
{
   Shape s;
   // errors: cannot access private member declared in class 'Shape'
   Print(s.x, " ", s.y);
}
```

Члены класса по умолчанию являются закрытыми (приватными), и потому из других частей кода, внешних по отношению к классу, обращаться к ним нельзя. Это — принцип инкапсуляции в действии.

Если мы попытаемся вывести фигуру в журнал, результат нас разочарует по нескольким причинам.

Наиболее прямолинейный подход вызовет ошибку "объекты передаются только по ссылке" (мы это видели и со структурами):

```
Print(s); // 's' - objects are passed by reference only
```

Объекты могут состоять из множества полей и из-за большого размера их неэффективно передавать по значению. Поэтому компилятор требует передавать параметры объектных типов по ссылке, а *Print* принимает значения.

Из раздела про параметры функций (см. раздел Параметры-значения и параметры-ссылки) мы знаем, что для описания ссылок используется символ '&'. Логично было бы предположить, что для получения ссылки на переменную (в данном случае, объект *s* типа *Shape*) необходимо поставить тот же знак перед её именем.

```
Print(&s);
```

Эта инструкция благополучно компилируется и работает, но делает не совсем то, что ожидалось.

Программа в процессе выполнения выводит некое целое число, например 1 или 2097152 (оно будет, скорее всего, отличаться). Знак амперсанда перед именем переменной означает получение указателя на эту переменную, а не ссылки (в отличие от описания параметра функции).

Указатели будут подробно рассмотрены в отдельном разделе. Пока же отметим, что MQL5 не предоставляет прямого доступа к памяти, и указатель на объект является дескриптором, а по-простому — уникальным номером объекта (он назначается самим терминалом). Но даже если бы указатель вел на адрес в памяти (как это происходит в C++), это не обеспечило бы легального способа прочитать содержимое объекта.

Чтобы выводить содержимое объектов *Shape* в журнал или еще куда-либо, требуется функция-член класса. Назовем её *toString*: она должна вернуть строку с неким описанием объекта. Что в него выводить, мы можем решить позднее. Также зарезервируем метод для отрисовки фигуры — *draw*: пока он выступит декларацией будущего программного интерфейса объекта.

```
class Shape
{
   int x, y;              // координаты центра
   color backgroundColor; // цвет заливки

   string toString()
   {
      ...
   }

   void draw() { /* заглушка будущего интерфейса рисования */ }
};
```

Определение функций-методов делается привычным способом, с тем лишь отличием, что они находятся внутри блока кода, формирующего класс.

> В будущем мы узнаем, как можно разнести объявление функции внутри блока класса и её определение за пределами блока. Этот подход часто используется для вынесения объявлений в заголовочный файл, и "скрытия" определений в mq5-файле. Это делает код более понятным (за счет того, что программный интерфейс представлен отдельно, в компактном виде, без реализации). Кроме того, это позволяет, при необходимости, распространять библиотеки программ в виде ex5-файлов (без основных исходных кодов, но с предоставлением заголовочного файла, который достаточен для вызова методов внешнего интерфейса).

Поскольку метод *toString* является частью класса, он имеет доступ к переменным и может преобразовать их в строку. Например,

```
string toString()
{
  return (string)x + " " + (string)y;
}
```

Однако сейчас функции *toString* и *draw* такие же закрытые, как и остальные поля. Нам необходимо сделать их доступными извне класса.

3.2.7 Права доступа

Для изменения прав доступа к членам класса применяется особый синтаксис (мы с ним уже познакомились в главе про структуры). В любом месте блока, перед описанием членов класса можно вставить модификатор: одно из трех ключевых слов — *private*, *protected*, *public* — и двоеточие.

Все идущие за модификатором члены, пока не встретится другой модификатор, или вплоть до конца класса, получат соответствующее ограничение видимости.

Например, следующая запись идентична предыдущему описанию класса *Shape*, потому что для классов при отсутствии модификаторов подразумевается режим *private*:

```
class Shape
{
private:
   int x, y;               // координаты центра
   color backgroundColor; // цвет заливки
   ...
};
```

Если бы мы захотели открыть доступ ко всем полям, то поменяли бы модификатор на *public*:

```
class Shape
{
public:
   int x, y;               // координаты центра
   ...
};
```

Но это нарушило бы принцип инкапсуляции, и мы не станем так делать. Вместо этого вставим модификатор *protected*: он разрешает обращаться к членам из производных классов, но оставляет их скрытыми от внешнего мира. Мы планируем унаследовать от класса *Shape* несколько других классов фигур, и в них будет нужен доступ к переменным родителя.

```
class Shape
{
protected:
   int x, y;               // координаты центра
   color backgroundColor; // цвет заливки

public:
   string toString() const
   {
      return (string)x + " " + (string)y;
   }

   void draw() { /* заглушка будущего интерфейса рисования */ }
};
```

Попутно мы сделали обе функции публичными.

Модификаторы могут чередоваться в описании класса произвольным образом и повторяться много раз. Однако в целях улучшения читабельности кода рекомендуется делать по одной секции

открытых (*public*), защищенных (*protected*) и закрытых (*private*) членов, причем выдерживать один и тот же их порядок во всех классах проекта.

Обратите внимание, мы добавили ключевое слово *const* в конец заголовка функции *toString*. Оно означает, что функция не меняет состояние полей объекта. Хотя это и необязательно, но помогает предотвратить случайную порчу переменных, а также дает знать пользователям класса и компилятору, что вызов функции не приведет к каким-либо побочным эффектам.

В функции *toString*, как и в любом методе класса, поля доступны по их именам. Позднее мы познакомимся с возможностью описывать методы как статические: они относятся целиком к классу, а не к экземплярам-объектам, и потому в них нельзя обращаться к полям.

Теперь мы можем вызвать метод *toString* у переменной-объекта *s*:

```
void OnStart()
{
   Shape s;
   Print(s.toString());
}
```

Здесь мы видим использование символа точки '.' как специального оператора разыменования: он обеспечивает обращение к членам объекта — полям и методам. Слева от него должен стоять объект, а справа — идентификатор одного из доступных свойств.

Метод *toString* — публичный, и потому доступен из внешней по отношению к классу функции *OnStart*. Если бы мы попытались в *OnStart* через разыменование "достучаться" до полей *s.x* или *s.y*, то получили бы ошибку компиляции "невозможен доступ к защищенному члену класса 'Shape'" ("cannot access protected member declared in class 'Shape'").

Для знатоков C++ отметим, что MQL5 не поддерживает так называемых "друзей" (для остальных поясним, что в C++ можно, при необходимости, составлять своего рода "белый список" сторонних классов и методов, которые имеют расширенные права, хотя и не являются "родственниками").

Запустив программу, мы убедимся, что она выводит пару чисел. Однако значения координат будут случайными. Даже если вам повезет увидеть нули, это не гарантирует, что они появятся при следующем запуске скрипта. Как правило, если в терминале не меняется перечень выполняющихся MQL-программ, повторные запуски любого скрипта приводят к выделению ему одной и той же области памяти, из-за чего может сложиться обманчивое впечатление, что состояние объекта стабильное. На самом деле поля объекта, как и в случае локальных переменных, ничем не инициализируются по умолчанию (см. раздел Инициализация).

Чтобы их проинициализировать применяются специальные функции класса — конструкторы.

3.2.8 Конструкторы: по умолчанию, параметрический, копирования

Мы уже сталкивались с конструкторами в главе, посвященной структурам (см. раздел Конструкторы и деструкторы). Для классов они работают, во многом, аналогичным образом. Напомним основные моменты и дополним их.

Конструктор — это метод с именем, совпадающим с именем класса, и имеющий тип void, то есть он не возвращает значения. Обычно ключевое слово *void* перед именем конструктора опускают. В классе может быть несколько конструкторов: они должны различаться количеством или типом

параметров. В момент создания нового объекта, программа вызывает конструктор, чтобы в нем можно было установить начальные значения полям.

Один из способов создания объекта, который мы использовали — описание в коде переменной соответствующего класса. В этой строке и будет вызван конструктор. Это происходит автоматически.

В зависимости от наличия и типов параметров, конструкторы делятся на:

- конструктор по умолчанию: без параметров;
- конструктор копирования: с единственным параметром типа ссылки на объект того же класса;
- конструктор параметрический: с произвольным набором параметров, кроме одиночной ссылки для копирования из предыдущего пункта.

Конструктор по умолчанию

Самый простой конструктор — без параметров — называется конструктором по умолчанию. В отличие от C++, в MQL5 не считается конструктором по умолчанию такой конструктор, у которого есть параметры и все они имеют значения по умолчанию (то есть все параметры опциональные, см. раздел Необязательные параметры).

Определим конструктор по умолчанию для класса *Shape*.

```
class Shape
{
   ...
public:
   Shape()
   {
      ...
   }
};
```

Разумеется, его следует делать в публичной секции класса.

Иногда конструкторы намеренно делают защищенными или закрытыми, чтобы держать под контролем процесс создания объектов, например, через так называемые фабричные методы. Но в данном случае, мы рассматриваем стандартный вариант композиции класса.

Для установки начальных значений переменным объекта мы могли бы воспользоваться привычными инструкциями присвоения:

```
public:
   Shape()
   {
      x = 0;
      y = 0;
      ...
   }
```

Однако в синтаксисе конструкторов предусмотрен другой вариант. Он называется списком инициализации и пишется после заголовка функции, через двоеточие. Сам список представляет

собой последовательность имен полей, разделенных запятыми, причем справа от каждого имени в круглых скобках указывается нужное начальное значение.

Например, для конструктора *Shape* можно написать так:

```
public:
    Shape() :
        x(0), y(0),
        backgroundColor(clrNONE)
    {
    }
```

Данный синтаксис является более предпочтительным, чем присваивание переменных в теле конструктора по нескольким причинам.

Во-первых, присваивание в теле функции производится уже после того, как соответствующая переменная создана. В зависимости от типа переменной это может означать, что для неё был сначала вызван конструктор по умолчанию и потом перезаписано новое значение (а это — лишние накладные расходы). В случае списка инициализации переменная сразу создается с нужным значением. Вероятно, компилятор сможем оптимизировать присваивание и в отсутствии списка инициализации, но в общем случае это не гарантировано.

Во-вторых, некоторые поля класса могут быть объявлены с модификатором *const*. Тогда их можно установить только в списке инициализации.

В-третьих, переменные-поля пользовательских типов могут не иметь конструктора по умолчанию (то есть в их классе все доступные конструкторы имеют параметры). Это значит, что при создании переменной в неё нужно передать фактические параметры, и список инициализации позволяет это сделать: значения аргументов указываются внутри круглых скобок, как будто при явном вызове конструктора. Список инициализации можно использовать в определении конструкторов, но не других методов.

Параметрический конструктор

У параметрического конструктора, по определению, есть несколько параметров (один или больше).

Например, представим, что для координат *x* и *y* описана специальная структура с параметрическим конструктором:

```
struct Pair
{
    int x, y;
    Pair(int a, int b): x(a), y(b) { }
};
```

Тогда мы можем использовать поле *coordinates* нового типа *Pair* вместо двух целочисленных полей *x* и *y* в классе *Shape*. Такая конструкция объектов называется включением или композиционным агрегированием. Объект *Pair* является неотъемлемой частью объекта *Shape*. Пара координат автоматически создается и уничтожается вместе с объектом "хозяином".

Поскольку у *Pair* нет конструктора без параметров, поле *coordinates* должно быть указано в списке инициализации конструктора *Shape*, с двумя параметрами *(int, int)*:

```
class Shape
{
protected:
   // int x, y;
   Pair coordinates;  // координаты центра (включение объекта)
   ...
public:
   Shape() :
      // x(0), y(0),
      coordinates(0, 0), // инициализация объекта
      backgroundColor(clrNONE)
   {
   }
};
```

Без списка инициализации не удастся создать подобные автоматические объекты.

С учетом изменения способа хранения координат в объекте нам необходимо обновить метод *toString*:

```
   string toString() const
   {
      return (string)coordinates.x + " " + (string)coordinates.y;
   }
```

Но это не окончательная его версия: вскоре мы внесем еще некоторые правки.

> Напомним, что автоматические переменные описывались в разделе Инструкции объявления/определения. Они называются автоматическими, потому что компилятор создает их (выделяет память) автоматически, и также автоматически удаляет, когда выполнение программы покидает контекст (блок кода), в котором переменная была создана.
>
> В случае объектных переменных автоматическое создание означает не только выделение памяти, но и вызов конструктора. А автоматическое удаление объекта сопровождается вызовом его деструктора (см. далее раздел Деструкторы). Причем, если объект входит в состав другого объекта, то его время жизни совпадает с временем жизни своего "хозяина", как в случае поля *coordinates* — экземпляра *Pair* в составе объекта *Shape*.
>
> Статические (в том числе глобальные) объекты также управляются компилятором автоматически.
>
> Альтернативой автоматическому распределению является динамическое создание объектов и работа с ними через указатели.

В разделе про наследование мы узнаем, как можно унаследовать один класс от другого. В этом случае список инициализации — единственный способ вызвать параметрический конструктор базового класса (компилятор не способен автоматически сформировать вызов конструктора с параметрами, как он делает неявно для конструктора по умолчанию).

Добавим в класс *Shape* еще один конструктор, позволяющий задать конкретные значения переменным. Он как раз будет является параметрическим конструктором (их можно создать сколько угодно: для разных целей и с разным набором параметров).

```
    Shape(int px, int py, color back) :
       coordinates(px, py),
       backgroundColor(back)
    {
    }
```

Список инициализации гарантирует, что когда выполняется тело конструктора, все внутренние поля (включая вложенные объекты, если они есть) уже созданы и проинициализированы.

Порядок инициализации членов класса соответствует не списку инициализации, а последовательности их объявления в классе.

Если в классе описан конструктор с параметрами, и при этом требуется разрешить создание объектов без аргументов, программист должен реализовать конструктор по умолчанию явным образом.

В том случае, если в классе вообще нет конструкторов, компилятор неявным образом предоставляет конструктор по умолчанию в виде заглушки, которая ответственна за инициализацию полей следующих типов: строки, динамические массивы, автоматические объекты с конструктором по умолчанию. Если таких полей нет, неявный конструктор по умолчанию ничего не делает. Поля других типов неявный конструктор не "трогает", поэтому в них будет содержаться случайный "мусор". Чтобы этого избежать, программист должен описать конструктор в явном виде и установить начальные значения самостоятельно.

Конструктор копирования

Конструктор копирования позволяет создать объект на основе другого объекта, переданного по ссылке в качестве единственного параметра.

Например, для класса *Shape* конструктор копирования мог бы выглядеть так:

```
class Shape
{
   ...
   Shape(const Shape &source) :
      coordinates(source.coordinates.x, source.coordinates.y),
      backgroundColor(source.backgroundColor)
   {
   }
   ...
};
```

Обратите внимание, что защищенные и закрытые члены другого объекта доступны в текущем объекте, поскольку права доступа работают на уровне класса. Иными словами, два объекта одного класса могут обращаться к данным друг друга при наличии ссылки (или указателя).

При наличии такого конструктора можно создавать объекты с помощью одного из двух синтаксисов:

```
void OnStart()
{
   Shape s;
   ...
   Shape s2(s);   // ok: синтаксис 1 - копирование
   Shape s3 = s;  // ok: синтаксис 2 - копирование через инициализацию
                  //                    (если есть конструктор копирования)
                  //                  - или присваивание
                  //                    (если нет конструктора копирования,
                  //                     но есть конструктор по умолчанию)

   Shape s4;      // определение
   s4 = s;        // присваивание, а не конструктор копирования!
}
```

Следует различать инициализацию объекта при создании и присваивание.

Второй вариант (помечен комментарием "синтаксис 2") будет работать даже если конструктора копирования не существует, но есть конструктор по умолчанию. В этом случае компилятор сформирует менее эффективный код: сначала создаст с помощью конструктора по умолчанию пустой экземпляр приемной переменной (*s3*, в данном случае), а потом скопирует поля образца (*s*, в данном случае) поэлементно. Фактически, получится тот же случай, что с переменной *s4*, для которой определение и присваивание выполнены отдельными инструкциями.

Если конструктора копирования нет, то попытка использовать первый синтаксис приведет к ошибке "недопустимое преобразование параметра" ("parameter conversion not allowed"), так как компилятор попытается взять какой-либо другой конструктор из имеющихся, с отличным набором параметров.

Имейте в виду, что если в классе есть поля с модификатором *const*, присваивание таких объектов запрещено по понятной причине: константное поле нельзя менять, его можно только однократно установить при создании объекта. Поэтому конструктор копирования становится единственным способом продублировать объект.

В частности, в следующих разделах мы дополним наш пример *Shape1.mq5*, и в классе *Shape* появится такое поле (со строкой-описанием — *type*). Тогда оператор присваивания станет генерировать ошибки (в частности, для таких строк, как с переменной *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
```

Благодаря подробным формулировкам компилятора можно понять суть и причины происходящего: во-первых, упоминается оператор присваивания ('='), а не конструктор копирования; во-вторых, сообщается, что оператор присваивания был неявно удален из-за наличия модификатора *const*. Здесь нам встречаются пока неизвестные понятия, которые мы изучим позднее: перегрузка операторов в классах, приведение объектных типов и возможность помечать методы удаленными.

В разделе Наследование, после того как мы научимся описывать производные классы, потребуется сделать некоторые уточнения относительно конструкторов копирования в иерархиях классов.

3.2.9 Деструкторы

Мы познакомились с деструкторами в главе, посвященной структурам (см. раздел Конструкторы и деструкторы). Напомним вкратце, что деструктор — это метод, вызываемый в момент удаления объекта. Деструктор имеет имя, совпадающее с именем класса, но к нему спереди добавлен префикс в виде символа тильды '~'. Деструктор не возвращает значения и не имеет параметров. Деструкторов может быть не больше одного.

Даже если деструктора нет или он пустой, компилятор в любом случае неявным образом выполнит "уборку мусора" для полей следующих типов: строк, динамических массивов и автоматических объектов.

Обычно деструктор находится в публичной секции класса, однако в некоторых специфических случаях разработчик может перенести его в группу *private*- или *protected*-членов. Закрытый или защищенный деструктор не позволит описать в коде автоматическую переменную данного класса. Однако чуть позднее мы познакомимся с динамическим созданием объектов, и для них такой запрет может иметь смысл.

В частности, некоторые объекты могут быть реализованы так, что должны сами себя удалять, когда в них отпадает необходимость (принцип определения востребованности может быть разный). Иными словами, пока объекты используются какими-либо частями программы, они существуют, а как только задача выполнена, они самоудаляются (закрытый деструктор оставляет возможность удалить объект из методов класса).

Для знатоков C++ отметим, что в MQL5 деструкторы всегда виртуальные (мы обратимся к понятию виртуальных методов в разделе Виртуальные методы (virtual и override)). На синтаксис описания этот фактор не влияет.

В примере программы рисования, в принципе, не требуется деструктор для фигур, но мы его заведем, чтобы проследить последовательность вызовов конструкторов и деструкторов. Начнем с упрощенного наброска, в котором "печатается" полное название метода:

```
class Shape
{
   ...
   ~Shape()
   {
      Print(__FUNCSIG__);
   }
};
```

Скоро мы дополним этот и другие методы, чтобы можно было отличать один экземпляр объекта от другого.

Рассмотрим следующий пример. Пара объектов *Shape* описана в двух разных контекстах: глобальном (вне функций) и локальном (внутри *OnStart*). Конструктор глобального объекта будет вызван после загрузки скрипта и до вызова *OnStart*, а деструктор — перед выгрузкой скрипта. Конструктор локального объекта будет вызван в строке с определением переменной, а деструктор — при выходе из блока кода, содержащего это определение, в данном случае, из функции *OnStart*.

```
// конструктор и деструктор global связаны с загрузкой и выгрузкой скрипта
Shape global;

// ссылка на объект не создает копию и не влияет на время жизни
void ProcessShape(Shape &shape)
{
   // ...
}

void OnStart()
{
   // ...
   Shape local; // <- вызов конструктора local
   // ...
   ProcessShape(local);
   // ...
} // <- вызов деструктора local
```

Передача объекта по ссылке в другие функции не создает его копий, не вызывает конструктор и деструктор.

3.2.10 Ссылка на себя: this

В контексте каждого класса, в коде его методов, доступна специальная ссылка на текущий объект: *this*. По сути это неявно определенная переменная. К ней применимы все приемы работы с переменными-объектами. В частности, её можно разыменовать, чтобы обратиться к полю объекта или вызвать метод. Например, следующие операторы в каком-либо методе класса *Shape* идентичны (метода *draw* взят условно, только для демонстрации):

```
class Shape
{
   ...
   void draw()
   {
      backgroundColor = clrBlue;
      this.backgroundColor = clrBlue;
   }
};
```

Необходимость использовать длинную форму может возникнуть, если в том же контексте существуют другие переменные/параметры с тем же именем. Обычно такая практика не приветствуется, но если в ней возникла необходимость, то ключевое слово *this* позволяет обратиться к перекрытым членам объекта.

Компилятор выводит предупреждение, если имя какой-либо локальной переменной или параметра метода перекрывает имя переменной-члена класса.

В следующем гипотетическом примере мы реализовали метод *draw*, который принимает необязательный строковый параметр *backgroundColor* с названием цвета. Поскольку имя параметра совпадает с членом класса *Shape*, компилятор выдает первое предупреждение "определение 'backgroundColor' скрывает поле".

Следствием перекрытия является то, что последующее ошибочное присвоение значения *clrBlue* работает с параметром, а не членом класса, и поскольку типы значения и параметра не совпадают, выдается второе предупреждение "неявное преобразование числа в строку" (числом является константа *clrBlue*). Зато строка *this.backgroundColor = clrBlue* записывает значение в поле объекта.

```
void draw(string backgroundColor = NULL) // предупреждение 1:
              // declaration of 'backgroundColor' hides member
{
   ...
   backgroundColor = clrBlue; // предупреждение 2:
     // implicit conversion from 'number' to 'string'
   this.backgroundColor = clrBlue; // ok

   {
      bool backgroundColor = false; // предупреждение 3:
          // declaration of 'backgroundColor' hides local variable
      ...
      this.backgroundColor = clrRed; // ok
   }
   ...
}
```

Последующее определение локальной логической переменной *backgroundColor* (во вложенном блоке фигурных скобок) перекрывает прежние определения этого имени еще раз (из-за чего получаем третье предупреждение). Однако благодаря разыменованию *this* оператор *this.backgroundColor = clrRed* также обращается к полю объекта.

Без указания *this* компилятор всегда выбирает самое близкое (по контексту) определение имени.

Существует необходимость в *this* и иного рода: передать текущий объект как параметр в другую функцию. В частности, применяется подход, при котором объекты одного класса ответственны за создание/удаление объектов другого класса, причем подчиненный объект должен знать своего "начальника". Тогда зависимые объекты создаются в классе "начальника" с помощью конструктора, в который передается *this* объекта-"начальника". Эта техника, как правило, применяет динамическое распределение объектов и указатели, и потому соответствующий пример будет показан в разделе с описанием указателей.

Другой способ частого применения *this* заключается в возврате указателя на текущий объект из функции-члена. Это позволяет выстраивать вызовы функций-членов в цепочку. Пока мы не изучали указатели подробно, будет достаточно знать, что указатель на объект какого-либо класса описывается путем добавления к имени класса символа '*', а работать с объектом через указатель можно также, как и напрямую.

Например, мы можем предоставить пользователю несколько методов для установки свойств фигуры по отдельности: смена цвета, перемещение по горизонтали или вертикали. Каждый из них будет возвращать указатель на текущий объект.

```
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;
}
```

Тогда есть возможность удобно нанизывать вызовы этих методов в цепочку.

```
Shape s;
s.setColor(clrWhite).moveX(80).moveY(-50);
```

Когда свойств в классе много, данный подход позволяет компактно и выборочно настраивать объект.

В разделе Определение класса мы пробовали вывести в журнал переменную-объект, но обнаружили, что можем использовать её имя лишь с амперсандом (в вызове *Print*), чтобы получить указатель, а фактически уникальный номер (дескриптор). В контексте объекта тот же дескриптор доступен через *&this*.

Для целей отладки можно идентифицировать объекты по дескриптору. Мы собираемся изучить наследование классов, и когда их станет больше одного, идентификация окажется кстати. Поэтому во всех конструкторах и деструкторах добавим (и будем в будущем добавлять в производных классах) такой вызов *Print*:

```
~Shape()
{
   Print(__FUNCSIG__, " ", &this);
}
```

Теперь все этапы создания и удаления будут отмечаться в журнале именем класса и номером объекта.

Похожие конструкторы и деструкторы реализуем в структуре *Pair*, однако в структурах, к сожалению, не поддерживаются указатели, то есть запись *&this* невозможна. Поэтому их мы можем идентифицировать только по содержимому (в данном случае, по координатам):

```
struct Pair
{
   int x, y;
   Pair(int a, int b): x(a), y(b)
   {
      Print(__FUNCSIG__, " ", x, " ", y);
   }
   ...
};
```

3.2.11 Наследование

При определении класса разработчик может унаследовать его от другого класса, воплощая тем самым концепцию наследования. Для этого после имени класса ставится двоеточие, необязательный модификатор прав доступа (одно из ключевых слов *public*, *protected*, *private*), и имя родительского класса. Например, вот как мы можем описать класс *Rectangle*, производный от *Shape*:

```
class Rectangle : public Shape
{
};
```

Модификаторы доступа в заголовке класса управляют "видимостью" членов родительского класса, включенных в класс наследника:

- *public* — все унаследованные члены сохраняют свои права и ограничения;
- *protected* — меняет права унаследованных *public*-членов на *protected*;
- *private* — делает все унаследованные члены закрытыми (*private*).

Модификатор *public* используется в подавляющем большинстве определений. Два остальных варианта имеет смысл применять только в исключительных случаях, потому что они нарушают базовый принцип наследования: объекты производного класса должны обычно "являться" ("is a") полноценными представителями родительского семейства, а если мы "урезаем" их права, они утрачивают часть своих характеристик. Структуры также можно наследовать друг от друга по аналогичной схеме. Наследовать классы от структур или структуры от классов запрещено.

> В отличие от C++, MQL5 не поддерживает множественное наследование. Родителей у класса может быть не больше одного.

Объект производного класса имеет встроенный в себя объект базового класса. Учитывая, что базовый класс может в свою очередь быть унаследованным от какого-то другого класса-прародителя, создаваемый объект можно сравнить с матрешками, вложенными одна в другую.

В новом классе нам потребуется конструктор, который заполняет поля объекта аналогично тому, как это было сделано в базовом классе.

```
class Rectangle : public Shape
{
public:
   Rectangle(int px, int py, color back) :
      Shape(px, py, back)
   {
      Print(__FUNCSIG__, " ", &this);
   }
};
```

В данном случае список инициализации превратился в одиночный вызов конструктора *Shape*. В списке инициализации нельзя напрямую устанавливать переменные базового класса, потому что за их инициализацию отвечает базовый конструктор. Однако, при необходимости, мы могли бы изменить *protected*-поля базового класса из тела конструктора *Rectangle* (инструкции в теле функции выполняются уже после того, как отработал базовый конструктор в списке инициализации).

У прямоугольника есть два размера, поэтому добавим их в качестве защищенных полей *dx* и *dy*. Для установки их значений требуется дополнить список параметров конструктора.

```
class Rectangle : public Shape
{
protected:
   int dx, dy; // размеры (ширина, высота)

public:
   Rectangle(int px, int py, int sx, int sy, color back) :
      Shape(px, py, back), dx(sx), dy(sy)
   {
   }
};
```

Важно отметить, что в объектах *Rectangle* в неявном виде присутствует унаследованная от *Shape* функция *toString* (впрочем, как и *draw*, но та пока пуста). Поэтому корректен следующий код:

```
void OnStart()
{
   Rectangle r(100, 200, 50, 75, clrBlue);
   Print(r.toString());
};
```

Здесь продемонстрирован не только вызов *toString*, но и создание объекта-прямоугольника с помощью нашего нового конструктора.

Конструктор по умолчанию (без параметров) отсутствует в классе *Rectangle*. Это означает, что пользователь класса не может создавать объекты-прямоугольники простым способом, без аргументов:

```
   Rectangle r; // 'Rectangle' - wrong parameters count
```

Компилятор выдаст ошибку "Неверное количество аргументов".

Создадим еще один дочерний класс — *Ellipse*. Пока он ничем не будет отличаться от *Rectangle*, кроме имени. Позднее мы внесем в них различия.

```
class Ellipse : public Shape
{
protected:
   int dx, dy; // размеры (большой и малый радиусы)
public:
   Ellipse(int px, int py, int rx, int ry, color back) :
      Shape(px, py, back), dx(rx), dy(ry)
   {
      Print(__FUNCSIG__, " ", &this);
   }
};
```

Поскольку количество классов увеличивается, было бы здорово выводить имя класса в методе *toString*. В разделе Специальные операторы sizeof и typename мы описывали оператор *typename*. Попробуем его использовать.

Напомним, что *typename* ожидает один параметр, для которого и возвращается название типа. Например, если мы создадим пару объектов *s* и *r*, соответственно классов *Shape* и *Rectangle*, то можем узнать их тип следующим образом:

```
void OnStart()
{
   Shape s;
   Rectangle r(100, 200, 75, 50, clrRed);
   Print(typename(s), " ", typename(r));      // Shape Rectangle
}
```

Но нам нужно каким-то образом получить это имя внутри класса. Для этой цели добавим в параметрический конструктор *Shape* строковый параметр и будем сохранять его в новом строковом поле *type* (обратите внимание на секцию *protected* и модификатор *const*: это поле скрыто от внешнего мира и не может редактироваться после создания объекта):

```
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);
   }
   ...
};
```

В конструкторах производных классов станем заполнять этот параметр базового конструктора с помощью *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);
   }
};
```

Теперь мы можем усовершенствовать метод *toString* с использованием поля *type*.

```
class Shape
{
   ...
public:
   string toString() const
   {
      return type + " " + (string)coordinates.x + " " + (string)coordinates.y;
   }
};
```

Убедимся, что наша маленькая иерархия классов порождает объекты, как задумано, и выводит тестовые записи в журнал при вызове конструкторов и деструкторов.

```
void OnStart()
{
   Shape s;
   // настройка объекта цепочкой вызовов через '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());
}
```

В результате получим примерно следующие записи в журнале (пустые строки добавлены намеренно, чтобы разделить вывод от разных объектов):

```
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
```

По логу понятно, в какой последовательности вызываются конструкторы и деструкторы.

Для каждого объекта сначала создаются описанные в нем поля-объекты (если они есть), затем вызывается базовый конструктор и все конструкторы производных классов по цепочке наследования. Если в производном классе встречаются собственные (добавленные) поля неких объектных типов, конструкторы для них будут вызваны непосредственно перед конструктором этого производного класса. Когда объектных полей несколько, они создаются в той последовательности, в которой описаны в классе.

Деструкторы вызываются строго в обратном порядке.

В производных классах могут быть определены конструкторы копирования, которые мы изучили в разделе Конструкторы: по умолчанию, параметрический, копирования. Для конкретных типов фигур, например, прямоугольника, их синтаксис аналогичен:

```
class Rectangle : public Shape
{
   ...
   Rectangle(const Rectangle &other) :
      Shape(other), dx(other.dx), dy(other.dy)
   {
   }
   ...
};
```

А область применения слегка расширяется. Объект производного класса можно использовать для копирования в базовый класс (потому что производный содержит все данные для базового). Правда при этом, разумеется, игнорируются поля, добавленные в производном классе.

```
void OnStart()
{
   Rectangle r(100, 200, 75, 50, clrBlue);
   Shape s2(r);          // ok: копируем производный в базовый

   Shape s;
   Rectangle r4(s);      // error: no one of the overloads can be applied
                         // требуется явная перегрузка конструктора
}
```

Для копирования в обратном направлении нужно предоставить в базовом классе вариант конструктора со ссылкой на производный (что, в принципе, противоречит принципам ООП), иначе возникнет ошибка компиляции "нет подходящей перегрузки функции" ("no one of the overloads can be applied to the function call").

Сейчас мы можем описать в скрипте пару или большее количество переменных-фигур, чтобы затем "попросить" их нарисовать себя с помощью метода *draw*.

```
void OnStart()
{
   Rectangle r(100, 200, 50, 75, clrBlue);
   Ellispe e(100, 200, 50, 75, clrGreen);
   r.draw();
   e.draw();
};
```

Однако такая запись означает, что количество фигур, их типы и параметры жестко "зашиты" в программу, а в принципе пользователь должен выбирать, что и где рисовать. Отсюда следует необходимость создавать фигуры неким динамическим способом.

3.2.12 Динамическое создание объектов: new и delete

До сих пор мы пробовали создавать только автоматические объекты, то есть локальные переменные внутри *OnStart*. Объект, описанный в глобальном контексте (вне функции *OnStart* или какой-либо другой), также был бы автоматически создан (в момент загрузки скрипта) и удален (в момент выгрузки скрипта).

Кроме этих двух режимов мы затронули возможность описать поле объектного типа (в нашем примере это структура *Pair*, использованная для поля *coordinates* внутри объекта *Shape*). Все

такие объекты тоже являются автоматическими: они создаются для нас компилятором в конструкторе объекта-"хозяина" и удаляются в его деструкторе.

Однако в программах далеко не всегда удается обойтись только автоматическими объектами. В случае программы рисования нам потребуется создавать фигуры по запросу пользователя. Более того, фигуры нужно будет хранить в массиве, а для этого автоматические объекты должны были бы иметь конструктор по умолчанию (что в нашем случае не так).

Для подобных ситуаций MQL5 предоставляет возможность динамического создания и удаления объектов. Создание выполняется с помощью оператора *new*, а удаление — с помощью оператора *delete*.

Оператор *new*

После ключевого слова *new* следует имя требуемого класса и, в круглых скобках, перечень аргументов для вызова любого из существующих конструкторов. Выполнение оператора *new* приводит к созданию экземпляра класса.

Оператор *new* возвращает значение особого типа — указатель на объект. Для описания переменной такого типа следует после имени класса добавить символ звездочки '*'. Например:

```
Rectangle *pr = new Rectangle(100, 200, 50, 75, clrBlue);
```

Здесь переменная *pr* имеет тип указателя на объект класса *Rectangle*. Указатели будут более подробно рассмотрены в отдельном разделе.

Важно отметить, что описание самой переменной типа указатель на объект не выделяет память под объект и не вызывает его конструктор. Конечно, указатель занимает место — 8 байт, но по сути это беззнаковое целое *ulong*, которое система интерпретирует особым образом.

С указателем можно работать точно также, как с объектом, то есть вызывать через оператор разыменования доступные методы и обращаться к полям.

```
Print(pr.toString());
```

Переменная-указатель, которой еще не присвоен дескриптор динамического объекта (например, если вызов оператора *new* производится не в момент инициализации новой переменной, а перенесен на какие-то более поздние строки исходного кода), содержит специальный нулевой указатель: он обозначается как NULL (чтобы отличать его от чисел), но фактически равен 0.

Оператор *delete*

Полученные через *new* указатели следует освобождать по завершении алгоритма с помощью оператора *delete*. Например:

```
delete pr;
```

Если этого не сделать, экземпляр, выделенный оператором *new*, останется в памяти. Если таким образом будут создаваться все новые и новые объекты, и потом не удаляться, когда в них отпала необходимость, это приведет к лишнему расходу памяти. Оставшиеся неосвобожденные динамические объекты вызывают вывод предупреждений по завершении программы. Например, если не удалить указатель *pr*, получим в журнале после выгрузки скрипта примерно следующее:

```
1 undeleted objects left
1 object of type Rectangle left
168 bytes of leaked memory
```

Терминал сообщает, сколько объектов и какого класса было забыто программистом, а также какой объем памяти они занимали.

После того как для указателя вызван оператор *delete*, указатель становится недействительным, так как объект уже не существует. Последующая попытка обратиться к его свойствам вызывает ошибку времени исполнения "Обращение по неверному указателю":

```
Critical error while running script 'shapes (EURUSD,H1)'.
Invalid pointer access.
```

Работа MQL-программы при этом прерывается.

Это, однако, не значит, что ту же самую переменную-указатель больше нельзя использовать. Достаточно присвоить ей указатель на другой вновь созданный экземпляр объекта.

MQL5 имеет встроенную функцию, которая позволяет проверить правильность указателя в переменной — *CheckPointer*:

```
ENUM_POINTER_TYPE CheckPointer(object *pointer);
```

Она принимает один параметр типа указатель на класс и возвращает значение из перечисления ENUM_POINTER_TYPE:

- POINTER_INVALID — неверный указатель;
- POINTER_DYNAMIC — действующий указатель на динамический объект;
- POINTER_AUTOMATIC — действующий указатель на автоматический объект.

Выполнять оператор *delete* имеет смысл только для указателя, для которого функция вернула POINTER_DYNAMIC. Для автоматического объекта он не будет иметь эффекта (такие объекты удаляются автоматически при возврате управления из блока кода, в котором определена переменная).

Следующий макрос упрощает и обеспечивает корректность очистки указателя:

```
#define FREE(P) if(CheckPointer(P) == POINTER_DYNAMIC) delete (P)
```

Необходимость явным образом "подчищать за собой хвосты" — неизбежная плата за ту гибкость, которую предоставляют динамические объекты и указатели.

3.2.13 Указатели

Как мы уже говорили в разделе Определение класса, указатели в MQL5 — это некие дескрипторы (уникальные номера) объектов, а не адреса в памяти, как в C++. Для автоматического объекта мы получали указатель, поставив амперсанд перед его именем (в данном контексте, символ амперсанда является оператором "взятия адреса"). Так, в следующем примере переменная *p* указывает на автоматический объект *s*.

```
Shape s;         // автоматический объект
Shape *p = &s;   // указатель на тот же объект
s.draw();        // вызываем метод объекта
p.draw();        // делаем то же самое
```

В предыдущих разделах мы научились получать указатель на объект в результате динамического создания с помощью *new*. При этом для получения дескриптора амперсанд не нужен: значение указателя и есть дескриптор.

MQL5 API предоставляет функцию *GetPointer*, которая выполняет то же действие, что и оператор амперсанд '&', то есть возвращает указатель на объект:

```
void *GetPointer(Class object);
```

Какой именно из двух вариантов использовать — дело вкуса.

Обратиться к методам и свойствам объекта по указателю можно с помощью оператора разыменования ('.'). Однако для написания отказоустойчивой программы следует проверять работоспособность указателя перед его разыменованием. Это можно сделать как с помощью функции *CheckPointer*, рассмотренной в предыдущем разделе, так и в более краткой записи условных операторов сравнения с NULL — *if(pointer != NULL)* или просто *if(pointer)*. Важно, что сравнение с NULL является менее строгой проверкой, чем вызов *CheckPointer*, поскольку подразумевает, что любые ненулевые указатели валидны (а это может быть не так, если объект уже удален). Однако такой способ проверки работает быстрее, чем вызов *CheckPointer*. Кстати говоря, оператор вида *if(pointer)* также приводит к неявному вызову *CheckPointer* и возвращает результат её работы.

В некоторых случаях, например, при передаче в функцию параметра-ссылки на объект (именно ссылки, а не указателя!) или для правильного вызова перегруженного оператора может потребоваться преобразовать указатель в объект. Для этой цели применяют унарный оператор "звёздочка" перед указателем. Например:

```
void Function(const Object &object)
{
   ...
}

void OnStart()
{
   Object automatic;
   Object *dynamic = ...; // описание и инициализация
   ...
   Function(automatic);
   Function(*dynamic);    // из указателя в объект, хотя здесь '*' можно опустить
   ...
}
```

А вот пример перегрузки оператора '=' в связке с указателями, где наличие или отсутствие '*' меняет поведение:

```
class Object
{
public:
   void operator=(const Object *other)
   {
      Print("pointer: ", &this, " <- ", other);
   }
   void operator=(const Object &other)
   {
      Print("reference: ", &this, " <- ", &other);
   }
};

void OnStart()
{
   Object *dynamic1 = new Object();
   Object *dynamic2 = new Object();
   *dynamic1 = dynamic2;    // operator=(const Object *other)
   *dynamic1 = *dynamic2;   // operator=(const Object &other)
   dynamic1 = *dynamic2;    // operator=(const Object &other)
   dynamic1 = dynamic2;     // ни один из методов не вызывается, причем указатель в dy
                            // т.к. перезаписан указателем из dynamic2!
   ...
}
```

Скрипт выведет в журнал записи вида:

```
pointer: 2097152 <- 3145728
reference: 2097152 <- 3145728
reference: 2097152 <- 3145728
```

В русскоязычной литературе по программированию этот оператор также часто называется "разыменованием", но мы уже применили данный термин к оператору '.' для доступа к свойствам объекта. Поэтому назовем оператор '*' оператором "овеществления" или "объективации".

Указатели часто используются для взаимной увязки объектов. Проиллюстрируем идею создания подчиненных объектов, получающих указатель на *this* своего объекта-создателя (*ThisCallback.mq5*). Мы упоминали этот прием в разделе про ключевое слово *this*.

Попробуем с помощью него реализовать схему периодического уведомления "создателя" о проценте выполненных вычислений в подчиненном объекте: мы делали её аналог с помощью указателя на функцию. Вычислениями управляет класс *Manager*, а сами вычисления (вполне вероятно, по разным формулам) производятся в отдельных классах — в данном примере показан один класс *Element*.

```
class Manager; // предварительное объявление

class Element
{
   Manager *owner; // указатель

public:
   Element(Manager &t): owner(&t) { }

   void doMath()
   {
      const int N = 1000000;
      for(int i = 0; i < N; ++i)
      {
         if(i % (N / 20) == 0)
         {
            // передаем себя в метод управляющего класса
            owner.progressNotify(&this, i * 100.0f / N);
         }
         // ... массивные вычисления
      }
   }

   string getMyName() const
   {
      return typename(this);
   }
};

class Manager
{
   Element *elements[1]; // массив указателей (1 для демо)

public:
   Element *addElement()
   {
      // находим пустой слот в массиве
      // ...
      // передаем себя в конструктор подчиненного класса
      elements[0] = new Element(this); // динамическое создание объекта
      return elements[0];
   }

   void progressNotify(Element *e, const float percent)
   {
      // Manager выбирает способ уведомления пользователя:
      // вывод на экран, печать, отправка в Internet
      Print(e.getMyName(), "=", percent);
   }
};
```

Подчиненный объект может с помощью полученной ссылки уведомлять "начальника" о ходе работы. Достижение конца вычислений сигнализирует управляющему объекту, что можно удалить объект-вычислитель или дать поработать другому. Конечно, фиксированный массив из одного элемента в классе *Manager* выглядит не очень серьезно, но в качестве демонстрации он передает суть. Менеджер не только управляет раздачей вычислительных задач, но и предоставляет абстрактный уровень для уведомления пользователя: вместо вывода в журнал он может писать сообщения в отдельный файл, выводить на экран или отправлять в Интернет.

Кстати, обратите внимание на предварительное объявление класса *Manager* перед определением класса *Element*. Оно нужно, чтобы описать в классе *Element* указатель на класс *Manager*, который определен ниже по коду. Если предварительное объявление опустить, получим ошибку "'Manager' - неизвестный идентификатор, вероятно отсутствует тип?" ("'Manager' - unexpected token, probably type is missing?").

Необходимость предварительного объявления возникает, когда два класса ссылаются посредством своих членов друг на друга: в этом случае, в каком бы порядке мы ни располагали классы, невозможно полностью определить любой из них. Предварительное объявление позволяет зарезервировать имя типа без полного определения.

> Фундаментальным свойством указателей является то, что указатель на базовый класс может использоваться для указания на объект любого производного класса. Это одно из проявлений полиморфизма. Данное поведение возможно потому, что производные объекты содержат как матрешки встроенные "под-объекты" родительских классов.

В частности, для нашей задачи с фигурами легко описать динамический массив указателей *Shape* и добавлять в него объекты разных типов по запросу пользователя.

Количество классов расширим до пяти (*Shapes2.mq5*). Помимо *Rectangle* и *Ellipse* добавим *Triangle*, а также сделаем производный от *Rectangle* класс для квадрата (*Square*), и производный от *Ellipse* класс для круга (*Circle*). Очевидно, что квадрат представляет собой прямоугольник с равными сторонами, а круг — это эллипс с равными большим и малым радиусами.

Для передачи строкового имени класса по цепочке наследования добавим в *protected*-секциях классов *Rectangle* и *Ellipse* специальные конструкторы с дополнительным строковым параметром *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)
   {
   }
   ...
};
```

Тогда при создании квадрата установим не только равные размеры сторон, но и передадим *typename(this)* из класса *Square*:

```
class Square : public Rectangle
{
public:
   Square(int px, int py, int sx, color back) :
      Rectangle(px, py, sx, sx, back, typename(this))
   {
   }
};
```

Кроме этого перенесем конструкторы в классе *Shape* в *protected*-секцию: это запретит создание объекта *Shape* самого по себе — он может выступать только в качестве базового для своих классов-наследников.

Порождать фигуры поручим функции *addRandomShape*, которая возвращает указатель на вновь созданный объект. В демонстрационных целях в ней сейчас будет реализована случайная генерация фигур: их типов, позиций, размеров и цветов.

Типы поддерживаемых фигур сведены в перечисление SHAPES: они соответствуют пяти реализованным классам.

Случайные числа в заданном диапазоне возвращает функция *random* (в ней используется встроенная функция *rand*, которая при каждом вызове возвращает случайное целое число в диапазоне от 0 до 32767). Центры фигур генерируются в диапазоне от 0 до 500 пикселей, размеры фигур — в диапазоне до 200. Цвет формируется из трех RGB-составляющих (см. раздел Цвет), каждая в диапазоне от 0 до 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[];

   // имитируем создание произвольных фигур пользователем
   ArrayResize(shapes, 10);
   for(int i = 0; i < 10; ++i)
   {
      shapes[i] = addRandomShape();
   }

   // обрабатываем фигуры: пока просто вывод в лог
   for(int i = 0; i < 10; ++i)
   {
      Print(i, ": ", shapes[i].toString());
```

```
        delete shapes[i];
    }
}
```

Мы генерируем 10 фигур и выводим их в журнал (результат может отличаться из-за случайности выбора типов и свойств). Не забываем удалить объекты с помощью *delete*, поскольку они создавались динамически (здесь это делается в одном и том же цикле, потому что фигуры далее не используются; в реальной программе массив фигур будет, скорее всего, сохраняться каким-либо образом в файл для последующей загрузки и продолжения работы с изображением).

```
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
```

Фигуры успешно создаются и "рапортуют" о своих свойствах.

Теперь все готово для того, чтобы обратиться к прикладному интерфейсу наших классов, то есть методу *draw*.

3.2.14 Виртуальные методы (virtual и override)

Классы предназначены для описания внешних программных интерфейсов и предоставления их внутренней реализации. Поскольку функционал нашей тестовой программы заключается в рисовании различных фигур, мы описали в классе *Shape* и его наследниках несколько переменных для будущей реализации, а также зарезервировали метод *draw* для интерфейса.

В базовом классе *Shape* он ничего не должен и не может делать, потому что *Shape* — это не конкретная фигура: позднее мы преобразуем *Shape* в абстрактный класс (более подробно об абстрактных классах и интерфейсах мы поговорим позднее).

Перекроем метод *draw* в классах *Rectangle*, *Ellipse* и прочих наследниках (*Shapes3.mq5*), то есть фактически скопируем его и изменим содержимое. Многие называют такое перекрытие переопределением, однако мы будем разделять эти термины: переопределение оставим исключительно за виртуальными методами, о которых речь пойдет чуть позже.

Строго говоря, для перекрытия метода достаточно соответствия имени, но для унифицированного использования в коде нужно сохранить перечень параметров и возвращаемое значение.

```
class Rectangle : public Shape
{
   ...
   void draw()
   {
      Print("Drawing rectangle");
   }
};
```

Поскольку мы пока не умеем рисовать на экране, просто выведем сообщение в журнал.

Важно отметить, что предоставляя новую реализацию метода в классе наследнике, мы тем самым получаем 2 версии метода: одна относится к встроенному базовому объекту (внутренняя матрешка, *Shape*), и другая — к производному (внешняя матрешка, *Rectangle*).

Первая будет вызываться для переменной типа *Shape*, вторая — для переменной типа *Rectangle*.

В более длинной цепочке наследования метод может быть перекрыт и размножен еще большее количество раз.

У нового метода можно изменить тип доступа, например, сделать публичным, если он был защищенным, или наоборот. Но мы в данном случае оставили метод *draw* в публичной секции.

При необходимости программист может вызвать реализацию метода любого из классов-прародителей: для этого применяется специальный оператор разрешения контекста — два двоеточия '::'. В частности, мы могли бы вызвать из метода *draw* класса *Square* реализацию *draw* из класса *Rectangle*: для этого указываем имя нужного класса, '::' и имя метода, например, *Rectangle::draw()*. Вызов *draw* без уточнения контекста подразумевает метод текущего класса, и потому если делать это из самого метода *draw* — получится бесконечная рекурсия, а в конечном итоге — переполнение стека и падение программы.

```
class Square : public Rectangle
{
public:
   ...
   void draw()
   {
      Rectangle::draw();
      Print("Drawing square");
   }
};
```

Тогда вызов *draw* для объекта *Square* вывел бы в журнал две строки:

```
   Square s(100, 200, 50, clrGreen);
   s.draw(); // Drawing rectangle
             // Drawing square
```

Привязка метода к классу, в котором он описан, обеспечивает статическую диспетчеризацию (или статическое связывание): компилятор решает, какой метод вызвать, на стадии компиляции и "зашивает" найденное соответствие в двоичный код.

В процессе решения компилятор ищет вызываемый метод в объекте того класса, для которого выполняется разыменование ('.'). Если метод есть, он вызывается, а если нет, компилятор проверяет на наличие метода родительский класс, и так далее по всей цепочке наследования,

пока метод не будет обнаружен. Если метод не будет найден ни в одном из классов цепочки, случится ошибка компиляции "неизвестный идентификатор" ("undeclared identifier").

В частности, следующий код вызывает метод *setColor* для объекта *Rectangle*:

```
Rectangle r(100, 200, 75, 50, clrBlue);
r.setColor(clrWhite);
```

Однако данный метод определен только в базовом классе *Shape* и встроен в единственном числе во все классы наследники, и потому именно он здесь и будет выполнен.

Попробуем запустить в функции *OnStart* отрисовку произвольных фигур из массива (напомним, что мы продублировали и модифицировали метод *draw* во всех классах-наследниках).

```
for(int i = 0; i < 10; ++i)
{
   shapes[i].draw();
}
```

Как ни странно, в журнал ничего не выводится. Происходит это потому, что программа вызывает метод *draw* класса *Shape*.

Здесь проявляется один существенный недостаток статической диспетчеризации: когда мы используем указатель на базовый класс для хранения объекта производного класса, компилятор выбирает метод, руководствуясь типом указателя, а не объекта. Дело в том, что на стадии компиляции еще не известно, на объект какого класса он будет указывать во время выполнения программы.

Таким образом, возникает необходимость в более гибком подходе: динамической диспетчеризации (или связывании), которая отложила бы выбор метода (из числа всех перекрытых версий метода в цепочке наследников) до времени выполнения. Выбор должен осуществляться на основе анализа фактического класса объекта, находящегося по указателю. Именно динамическая диспетчеризация обеспечивает принцип полиморфизма.

Данный подход реализуется в MQL5 с помощью виртуальных методов. В описании такого метода необходимо в начале заголовка добавить ключевое слово *virtual*.

Объявим виртуальным метод *draw* в классе *Shape* (*Shapes4.mq5*). Это автоматически сделает виртуальными и все его версии в производных классах.

```
class Shape
{
   ...
   virtual void draw()
   {
   }
};
```

После виртуализации метода его модификации в производных классах называют переопределением, а не перекрытием. Переопределение требует соответствия имени, типов параметров и возвращаемого значения метода (с учетом наличия/отсутствия модификаторов *const*).

> Обратите внимание, что переопределение виртуальных функций отличается от перегрузки функций. Перегрузка использует одно и то же имя функции, но с разными параметрами (в частности, на примере структур мы видели возможность перегрузить конструктор, см. раздел

Конструкторы и деструкторы), а переопределение требует полного соответствия сигнатур функций.

Переопределяемые функции должны быть определены в разных классах, связанных наследственными отношениями. Перегружаемые функции должны быть в одном классе — иначе это будет не перегрузка, а, скорее всего, перекрытие (и работать будет по-другому, см. далее разбор примера *OverrideVsOverload.mq5*).

Если запустить новый скрипт, в журнале появятся ожидаемые строки, сигнализирующие вызовы специфических версий метода *draw* в каждом из классов.

```
Drawing square
Drawing circle
Drawing triangle
Drawing ellipse
Drawing triangle
Drawing rectangle
Drawing square
Drawing triangle
Drawing square
Drawing triangle
```

В производных классах, где производится переопределение виртуального метода, рекомендуется добавлять в его заголовок ключевое слово *override* (хотя это не обязательно).

```
class Rectangle : public Shape
{
   ...
   void draw() override
   {
      Print("Drawing rectangle");
   }
};
```

Это дает знать компилятору, что мы делаем переопределение метода намеренно. Если в будущем вдруг изменится программный интерфейс базового класса, и перекрытый метод перестанет быть виртуальным (или просто будет удален), компилятор выдаст сообщение об ошибке: "метод определен с модификатором 'override', но не переопределяет какой-либо метод в базовом классе" ("method is declared with 'override' specifier, but does not override any base class method"). Учтите, что даже добавление или удаление модификатора *const* у метода меняет его сигнатуру, и переопределение от этого может "сломаться".

Ключевое слово *virtual* перед переопределенным методом также допустимо, но не обязательно.

Для работы динамической диспетчеризации компилятор формирует по каждому классу таблицу виртуальных функций. В каждый объект добавляется неявное поле со ссылкой на данную таблицу своего класса. Таблица заполняется компилятором на основе информации о всех виртуальных методах и их переопределенных версиях по цепочке наследования конкретного класса.

Вызов виртуального метода кодируется в двоичном образе программы особым образом: сначала делается просмотр таблицы в поисках версии для класса конкретного объекта (расположенного по указателю), и далее — переход на подходящую функцию.

В связи с этим динамическая диспетчеризация выполняется медленнее, чем статическая.

В MQL5 классы всегда содержат таблицу виртуальных функций, вне зависимости от наличия виртуальных методов.

Если виртуальный метод возвращает указатель на класс, при его переопределении существует возможность поменять (сделать более конкретным, узкоспециализированным) объектный тип возвращаемого значения. Иными словами, тип указателя может быть не только тот же самый, что в изначальной декларации виртуального метода, но и любой его наследник. Такие типы называются "ковариантными" или взаимозаменяемыми.

Например, если бы мы сделали метод *setColor* виртуальным в классе *Shape*:

```
class Shape
{
   ...
   virtual Shape *setColor(const color c)
   {
      backgroundColor = c;
      return &this;
   }
   ...
};
```

то могли бы переопределить его в классе *Rectangle* следующим образом (только в качестве демонстрации технологии):

```
class Rectangle : public Shape
{
   ...
   virtual Rectangle *setColor(const color c) override
   {
      // вызываем оригинальный метод
      // (выполнив предварительно осветление цвета,
      // зачем бы это ни понадобилось)
      Rectangle::setColor(c | 0x808080);
      return &this;
   }
};
```

Обратите внимание, что возвращаемый тип — указатель на *Rectangle*, вместо *Shape*.

Подобный прием имеет смысл использовать, если переопределенная версия метода меняет что-то в той части объекта, который не принадлежит базовому классу, так что объект, по сути, уже не соответствует разрешенному состоянию (инварианту) базового класса.

Наш пример с рисованием фигур почти готов. Осталось наполнить виртуальные методы *draw* реальным содержимым. Мы сделаем это в главе Графические объекты (см. пример *ObjectShapesDraw.mq5*), а усовершенствуем после изучения графических ресурсов.

> С учетом наследования, процедура, по которой компилятор подбирает подходящий метод, выглядит слегка запутанной. Руководствуясь именем метода и конкретным перечнем аргументов (их типами) в инструкции вызова, составляется список всех имеющихся методов-кандидатов.

Для невиртуальных методов сначала анализируются только методы текущего класса. Если среди них нет подходящего, компилятор продолжит поиск в базовом классе (и затем в более отдаленных предках, пока не найдет соответствия). Если же среди методов текущего класса есть подходящий (даже с учетом, вероятно, необходимой неявной конвертации типов аргументов), будет выбран именно он. Если в базовом классе был метод с более подходящими типами аргументов (без конвертации или с меньшим количеством конвертаций), компилятор до него все равно не "доберется". Иными словами, невиртуальные методы анализируются начиная от класса текущего объекта в сторону предков до первого "рабочего" совпадения.

Для виртуальных методов компилятор сначала находит нужный метод по имени в классе указателя и далее выбирает в таблице виртуальных функций реализацию для наиболее конкретизированного класса (наиболее дальнего наследника), в котором этот метод переопределен в цепочке между типом указателя и типом объекта. При этом неявная конвертация аргументов также может быть задействована, если нет точного соответствия по типам аргментов.

Рассмотрим следующий пример (*OverrideVsOverload.mq5*). Имеется 4 класса, унаследованных по цепочке: *Base*, *Derived*, *Concrete* и *Special*. Все они содержат методы с аргументами типов *int* и *float*. В функции *OnStart* в качестве аргументов для вызовов всех методов используются целочисленная переменная *i* и вещественная *f*.

```
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
   // ошибка: '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);
   }
};
```

В начале мы создаем объект класса *Concrete* и указатель на него *Base *ptr*. Затем вызываем невиртуальные и виртуальные методы для них. Во второй части методы объекта *Special* вызываются через указатели классов *Base* и *Derived*.

```
void OnStart()
{
   float f = 2.0;
   int i = 1;

   Concrete c;
   Base *ptr = &c;

   // Тесты статического связывания

   ptr.nonvirtual(i); // Base::nonvirtual(float), конвертация int -> float
   c.nonvirtual(i);   // Derived::nonvirtual(int)

   // предупреждение: deprecated behavior, hidden method calling
   c.nonvirtual(f);   // Base::nonvirtual(float), т.к.
                      // подбор метода завершился в Base,
                      // Derived::nonvirtual(int) не подходит по f

   // Тесты динамического связывания

   // внимание: нет метода Base::process(int), и плюс к тому
   // нет переопределений process(float) в классах до Concrete (включительно)
   ptr.process(i);    // Base::process(float), конвертация int -> float
   c.process(i);      // Derived::process(int), т.к.
                      // в Concrete нет переопределения,
                      // а переопределение в Special не в счет

   Special s;
   ptr = &s;
   // внимание: нет метода Base::process(int) в ptr
   ptr.process(i);    // Special::process(float), конвертация int -> float
   ptr.process(f);    // Special::process(float)

   Derived *d = &s;
   d.process(i);      // Special::process(int)

   // предупреждение: deprecated behavior, hidden method calling
   d.process(f);      // Special::process(float)
}
```

Ниже показан вывод в журнал.

```
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
```

Вызов *ptr.nonvirtual(i)* происходит с помощью статического связывания, причем предварительно целое число *i* приводится к типу параметра — *float*.

Вызов *c.nonvirtual(i)* — также статический, и поскольку в классе *Concrete* метода *void nonvirtual(int)* нет, компилятор находит такой метод в родительском классе *Derived*.

Вызов на том же объекте одноименной функции со значением типа *float* приводит компилятор к методу *Base::nonvirtual(float)*, поскольку *Derived::nonvirtual(int)* не подходит (конвертация привела бы к потере точности). Попутно компилятор выводит предупреждение "устаревшее поведение, вызов перекрытого метода" ("deprecated behavior, hidden method calling").

Перекрытыми называются методы, которые выглядят как перегруженные (одноименные, но с разными параметрами), но не являются таковыми, потому что находятся в разных классах. Когда метод в производном классе перекрывает метод в родительском, это может вызывать неожиданные эффекты для программиста (он чаще всего ожидает от компилятора выбора другого подходящего метода), причем неважно виртуальные это методы или нет.

Для подавления предупреждения, если реализация родительского класса действительно нужна, её следует оформить в виде точно такой же функции в производном классе, и вызвать из неё базовую.

```
class Derived : public Base
{
public:
   ...
   // это переопределение подавит предупреждение
   // "deprecated behavior, hidden method calling"
   void nonvirtual(float v)
   {
      Base::nonvirtual(v);
      Print(__FUNCSIG__, " ", v);
   }
...
```

Вернемся к тестам в *OnStart*.

Вызов *ptr.process(i)* демонстрирует описанную выше проблему с путаницей переопределения и перекрытия. В классе *Base* есть виртуальный метод *process(float)*, а в классе *Derived* добавляется новый виртуальный метод *process(int)* — и это не переопределение, так как типы параметров различаются. Компилятор выбирает метод по имени в базовом классе и проверяет в таблице виртуальных функций наличие переопределений в цепочке наследования вплоть до класса *Concrete* (включительно, это класс объекта по указателю). Поскольку переопределений не найдено, компилятор взял *Base::process(float)* и применил преобразование типа аргумента к параметру (*int* во *float*).

Если бы мы придерживались правила всегда писать слово *override*, где подразумевается переопределение, и добавили его в *Derived*, то получили бы ошибку:

```
class Derived : public Base
{
   ...
   virtual void process(int v) override // ошибка!
   {
      Print(__FUNCSIG__, " ", v);
   }
};
```

Компилятор сообщил бы: "Метод 'Derived::process' определен с модификатором 'override', но не переопределяет ни один метод базового класса" ("'Derived::process' method is declared with 'override' specifier, but does not override any base class method"). Это послужило бы подсказкой к устранению проблемы.

Вызов *process(i)* для объекта *Concrete* выполняется с помощью *Derived::process(int)*. Хотя у нас имеется еще более "дальнее" переопределение в классе *Special*, оно не подходит, потому что сделано в цепочке наследования уже после класса *Concrete*.

Когда указатель *ptr* устанавливается на объект *Special*, вызовы *process(i)* и *process(f)* решаются компилятором как *Special::process(float)*. Выбор метода с параметром типа *float* происходит по той же причине, что описана выше, но здесь в ход вступает переопределение в классе *Special*.

Если же применить указатель *d* типа *Derived*, то мы, наконец-то, получим ожидаемый вызов *Special::process(int)* для строки *d.process(i)*. Дело в том, что *process(int)* определен в *Derived*, и попадает в область поиска компилятора.

Обратите внимание, что в классе *Special* производится и переопределение унаследованных виртуальных методов, и перегрузка двух методов (в самом классе).

> Не стоит вызывать виртуальную функцию из конструктора или деструктора! Хотя технически это возможно, виртуальное поведение в конструкторе и деструкторе полностью утрачивается, и вы можете получить неожиданные результаты. Избегать следует не только явные, но и опосредованные вызовы (например, когда из конструктора вызывается простой метод, а тот в свою очередь вызывает виртуальный).
>
> Разберем ситуацию более подробно на примере конструктора. Дело в том, что в момент работы конструктора объект "собран" еще не полностью по всей цепочке наследования, а только вплоть до текущего класса. Все производные части (внешние "матрешки") еще предстоит "досоздать" вокруг имеющегося ядра. Поэтому все более поздние переопределения виртуального метода (если они есть) еще недоступны в этот момент. В результате из конструктора будет вызываться текущая версия метода.

3.2.15 Статические члены

До сих пор мы рассматривали поля и методы класса, которые описывают состояние и поведение объектов данного класса. Однако в программах бывает необходимо хранить некоторые атрибуты или выполнять операции для класса целиком, а не для его объектов. Такие свойства класса называются статическими и описываются с помощью ключевого слова *static*, добавляемого перед типом. Они также поддерживаются в структурах и объединениях.

Например, мы можем подсчитывать количество фигур, созданных пользователем в программе рисования. Для этого в классе *Shape* опишем статическую переменную *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;
   }
};
```

Она определена в секции *private* и потому недоступна извне.

Для чтения текущего значения счетчика предусмотрен публичный статический метод *getCount()*. В принципе, поскольку статические члены определены в контексте класса, они получают ограничения видимости согласно модификатору секции, в которой расположены.

Будем увеличивать счетчик на 1 в параметрическом конструкторе *Shape*, а конструктор по умолчанию уберем. Таким образом, каждый экземпляр фигуры любого производного типа окажется учтенным.

Обратите внимание, что статическую переменную необходимо явным образом определить (и опционально проинициализировать) вне блока класса:

```
static int Shape::count = 0;
```

Статические переменные класса похожи на глобальные переменные и статические переменные внутри функций (см. раздел Статические переменные) в том плане, что создаются при запуске программы и удаляются перед её выгрузкой. Поэтому, в отличие от переменных объекта, они должны существовать в виде единственного экземпляра изначально.

В данном случае инициализацию нулем можно опустить, потому что, как мы знаем, глобальные и статические переменные по умолчанию получают нулевые значения. Статическими могут быть и массивы.

В определении статической переменной мы видим использование специального оператора выбора контекста '::'. С помощью него формируется полностью квалифицированное имя переменной. Слева от '::' стоит имя класса, к которому относится переменная, а справа — её идентификатор. Очевидно, что полное имя необходимо, потому что внутри разных классов могут

быть описаны статические переменные с одним и тем же идентификатором, и нужен способ однозначно обращаться к каждой из них.

Тот же оператор '::' используется для доступа не только к публичным статическим переменным класса, но и методам. В частности, для того чтобы вызвать метод *getCount* в функции *OnStart* используем следующий синтаксис — *Shape::getCount()*:

```
void OnStart()
{
   for(int i = 0; i < 10; ++i)
   {
      Shape *shape = addRandomShape();
      shape.draw();
      delete shape;
   }

   Print(Shape::getCount()); // 10
}
```

Поскольку сейчас генерируется заданное количество фигур (10), мы можем убедиться, что счетчик работает правильно.

При наличии объекта класса, можно обратиться к статическому методу или свойству через привычное разыменование (например, *shape.getCount()*), но такая запись может вводить в заблуждение (поскольку скрывает тот факт, что обращения к объекту на самом деле не происходит).

Отметим, что создание производных классов никак не влияет на статические переменные и методы: они всегда приписаны к тому классу, в котором были определены. Наш счетчик — единый для всех классов фигур, производных от *Shape*.

Внутри статических методов нельзя использовать *this*, поскольку они выполняются без привязки к конкретному объекту. Также из статического метода нельзя напрямую, без разыменования какой-либо переменной объектного типа, вызвать обычный метод класса или обращаться к его полю. Например, если вызвать *draw* из *getCount*, получим ошибку "доступ к нестатическому члену или функции":

```
   static int getCount()
   {
      draw(); // error: 'draw' - access to non-static member or function
      return count;
   }
```

По той же причине статические методы не могут быть виртуальными.

Можно ли, пользуясь статическими переменными, подсчитать не общее количество фигур, а их статистику в разбивке по типам? Да, можно. Эта задача оставлена для самостоятельной проработки. Желающие могут найти один из примеров реализации в скрипте *Shapes5stats.mq5*.

3.2.16 Вложенные типы, пространства имен и оператор контекста '::'

Классы, структуры и объединения могут быть описаны не только в глобальном контексте, но и внутри другого класса или структуры. И даже более того: определение можно сделать внутри

функции. Это позволяет описывать все сущности, необходимые для работы какого-либо класса или структуры, внутри соответствующего контекста и тем самым избежать потенциального конфликта имен.

В частности, в программе рисования структура для хранения координат *Pair* была до сих пор определена глобально. Когда программа разрастется, вполне возможна ситуация, что появится необходимость в другой сущности с именем *Pair* (особенно учитывая довольно общее название). Поэтому описание структуры желательно перенести внутрь класса *Shape* (*Shapes6.mq5*).

```
class Shape
{
public:
   struct Pair
   {
      int x, y;
      Pair(int a, int b): x(a), y(b) { }
   };
   ...
};
```

Вложенные описания получают права доступа согласно указанным модификаторам секций. В данном случае мы сделали имя *Pair* публично доступным. Внутри класса *Shape* обращением с типом структуры *Pair* никак не меняется из-за переноса. Однако во внешнем коде необходимо указывать полностью квалифицированное имя, включающее имя внешнего класса (контекста), оператор выбора контекста '::' и непосредственно идентификатор внутренней сущности. Например, чтобы описать переменную с парой координат потребуется написать:

```
Shape::Pair coordinates(0, 0);
```

Уровень вложенности при описании сущностей не ограничен, поэтому полностью квалифицированное имя может содержать идентификаторы множества уровней (контекстов), разделенные '::'. Например, мы могли бы все классы рисования заключить внутри внешнего класса *Drawing*, в секции *public*.

```
class Drawing
{
public:
   class Shape
   {
   public:
      struct Pair
      {
         ...
      };
   };
   class Rectangle : public Shape
   {
      ...
   };
   ...
};
```

Тогда полные имена типов (например, для использования в *OnStart* или других внешних функциях) удлинились бы:

```
Drawing::Shape::Rect coordinates(0, 0);
Drawing::Rectangle rect(200, 100, 70, 50, clrBlue);
```

С одной стороны это неудобно, но с другой является порой необходимостью в больших проектах с большим числом классов. В нашем маленьком проекте данный подход используется только для демонстрации технической возможности.

Для объединения логически связанных классов и структур в именованные группы MQL5 предоставляет более простой способ, чем включение их в "пустой" класс-обертку.

Пространство имен объявляется с помощью ключевого слова *namespace*, после которого идет имя и блок фигурных скобок, включающий все необходимые определения. Вот как выглядит та же программа рисования с использованием *namespace*:

```
namespace Drawing
{
   class Shape
   {
   public:
      struct Pair
      {
         ...
      };
   };
   class Rectangle : public Shape
   {
      ...
   };
   ...
}
```

Основных отличий два: внутреннее содержимое пространства всегда доступно публично (модификаторы доступа в нем не применимы) и после закрывающей фигурной скобки нет точки с запятой.

Добавим в класс *Shape* метод *move*, принимающий структуру *Pair* в качестве параметра:

```
class Shape
{
public:
   ...
   Shape *move(const Pair &pair)
   {
      coordinates.x += pair.x;
      coordinates.y += pair.y;
      return &this;
   }
};
```

Тогда в функции *OnStart* можно организовать сдвиг всех фигур на заданную величину с помощью вызова данной функции:

```
void OnStart()
{
   // рисуем случайный набор фигур
   for(int i = 0; i < 10; ++i)
   {
      Drawing::Shape *shape = addRandomShape();
      // сдвигаем все фигуры
      shape.move(Drawing::Shape::Pair(100, 100));
      shape.draw();
      delete shape;
   }
}
```

Обратите внимание, что типы *Shape* и *Pair* приходится описывать полными именами: *Drawing::Shape* и *Drawing::Shape::Pair* соответственно.

Блоков с одним и тем же названием пространства может встретиться несколько: всё их содержимое попадет в один логически единый контекст с указанным именем.

Идентификаторы, определенные в глобальном контексте, в частности все встроенные функции MQL5 API, также доступны через оператор выбора контекста, перед которым ничего не ставится. Например, вот как может выглядеть вызов функции *Print*:

```
::Print("Done!");
```

Когда вызов делается из какой-либо функции, определенной в глобальном контексте, необходимости в такой записи нет.

Необходимость может проявиться внутри какого-либо класса или структуры, если в них определен одноименный элемент (функция, переменная или константа). Например, добавим метод *Print* в класс *Shape*:

```
   static void Print(string x)
   {
      // пусто
      // (вероятно, собираемся потом выводить в отдельный лог-файл)
   }
```

Поскольку тестовые реализации метода *draw* в производных классах вызывают *Print*, то теперь они перенаправлены на данный метод *Print*: из нескольких одинаковых идентификаторов компилятор выбирает тот, что определен в более приближенном контексте. В данном случае, определение в базовом классе находится ближе к фигурам, чем глобальный контекст. В результате вывод в журнал из классов фигур будет подавлен.

При этом вызов *Print* из функции *OnStart* по-прежнему работает (потому что он вне контекста класса *Shape*).

```
void OnStart()
{
   ...
   Print("Done!");
}
```

Чтобы "починить" отладочную печать в классах, нужно все вызовы *Print* предварить оператором выбора глобального контекста:

```
class Rectangle : public Shape
{
   ...
   void draw() override
   {
      ::Print("Drawing rectangle"); // вновь печатаем через глобальный Print(...)
   }
};
```

3.2.17 Разнесение объявления и определения класса

В крупных программных проектах удобно разделять классы на краткое описание (объявление) и определение, включающее основные детали реализации. В некоторых случаях такое разделение становится необходимостью, если классы тем или иными образом ссылаются друг друга, то есть ни один не может быть полностью определен без предварительных объявлений.

Мы видели пример предварительного объявления в разделе Указатели (см. файл *ThisCallback.mq5*), где классы *Manager* и *Element* содержат взаимные указатели. Там было сделано предварительное объявление класса в краткой форме: в виде заголовка с ключевым словом *class* и именем:

```
class Manager;
```

Однако это — минимально возможное объявление. Оно регистрирует только имя и дает возможность отложить описание программного интерфейса до некоторых пор, но где-то позднее в коде это описание обязательно должно встретиться.

Более часто объявление включает в себя полное описание интерфейса: в нем указываются все переменные и заголовки методов класса, но без их тел (блоков кода).

Определения методов записываются отдельно: с заголовками, в которых используются полностью квалифицированные имена, включающие имя класса (или нескольких классов и пространств имен, если контекст метода имеет большой уровень вложенности). Имена всех классов и имя метода соединяются с помощью оператора выделения контекста '::'.

```
тип имя_класса [:: имя_вложенного_класса ...] :: имя_метода([параметры...])
{
}
```

В принципе, можно часть методов определить непосредственно в блоке описания класса (обычно так делают с малыми функциями), а часть — вынести отдельно (как правило, крупные функции). Но метод должен иметь только одно определение (то есть нельзя определить метод в блоке класса, и затем — еще раз отдельно) и одно объявление (определение в блоке класса является и объявлением).

Перечень параметров, возвращаемый тип, модификаторы *const* (если есть) должны полностью совпадать в объявлении и определении метода.

Посмотрим, как можно разнести описание и определение классов из скрипта *ThisCallback.mq5* (пример из раздела Указатели): создадим его аналог с именем *ThisCallback2.mq5*.

В начале по-прежнему будет идти предварительное объявление *Manager*. Далее оба класса *Element* и *Manager* объявлены без реализации: вместо блока кода с телом метода стоит точка с запятой.

```
class Manager; // предварительное объявление

class Element
{
   Manager *owner; // указатель
public:
   Element(Manager &t);
   void doMath();
   string getMyName() const;
};

class Manager
{
   Element *elements[1]; // массив указателей (заменить на динамический)
public:
   ~Manager();
   Element *addElement();
   void progressNotify(Element *e, const float percent);
};
```

Во второй части исходного кода представлены реализации всех методов (сами реализации — без изменений).

```
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)
{
   ...
}
```

Раздельное объявление и определение методов поддерживают также и структуры.

Обратите внимание, что список инициализации конструктора (после имени и ':') является частью определения и потому должен предшествовать телу функции (иными словами, список инициализации недопустим в объявлении конструктора, где присутствует только заголовок).

Раздельное написание объявления и определения позволяет разрабатывать библиотеки, исходный код которых должен быть закрытым. В этом случае объявления располагают в отдельном заголовочном файле с расширением *mqh*, а определения — в одноименном файле с расширением *mq5*. Программу компилируют и распространяют в виде ex5-файла, к которому прикладывается заголовочный файл с описанием внешнего интерфейса.

При этом может возникнуть вопрос, почему часть внутренней реализации, в частности организация данных (переменные), видна во внешнем интерфейсе. Строго говоря, это сигнализирует о недостаточном уровне абстракции в иерархии классов. Все классы, предоставляющие внешний интерфейс, не должны раскрывать никаких деталей реализации.

Иными словами, если бы мы ставили своей целью экспортировать вышеприведенные классы из некоей библиотеки, то нужно было бы выделить их методы в базовые классы, которые предоставили бы описание API (без полей с данными), а *Manager* и *Element* унаследовать от них. При этом в методах базовых классов мы не можем использовать никаких данных производных классов и, по большому счету, они вообще не могут иметь реализации. Как такое возможно?

Для этого существует технология абстрактных методов, абстрактных классов и интерфейсов.

3.2.18 Абстрактные классы и интерфейсы

Для изучения абстрактных классов и интерфейсов вернемся к нашему сквозному примеру программы рисования. Её программный интерфейс для простоты состоит из единственного виртуального метода *draw*. До сих пор он был пустым, но вместе с тем даже такая пустая реализация является конкретной реализацией. Однако объекты класса *Shape* не могут изображаться — их фигура не определена. Поэтому метод *draw* имеет смысл сделать абстрактным или, как иначе это называют, чисто виртуальным.

Для этого блок с пустой реализацией следует убрать, а в заголовке метода добавить "= 0":

```
class Shape
{
public:
   virtual void draw() = 0;
   ...
```

Класс, у которого есть хотя бы один абстрактный метод, также становится абстрактным, потому что создать его объект не получится: нет реализации. В частности, у нас конструктор *Shape* был доступен производным классам (благодаря модификатору *protected*), и их разработчики могли, гипотетически, создать объект *Shape*. Но так было раньше, а после объявления абстрактного метода мы такое поведение пресекли, как запрещенное нами — авторами интерфейса рисования. Компилятор будет выдавать ошибку:

```
'Shape' - cannot instantiate abstract class
      'void Shape::draw()' is abstract
```

Наиболее правильным подходом для описания интерфейса является создание под него абстрактного класса, содержащего только абстрактные методы. В нашем случае, метод *draw* следует вынести в новый класс *Drawable*, а класс *Shape* унаследовать от него (*Shapes.mq5*).

```
class Drawable
{
public:
   virtual void draw() = 0;
};

class Shape : public Drawable
{
public:
   ...
   // virtual void draw() = 0; // вынесен в базовый класс
   ...
};
```

Разумеется, интерфейсные методы должны находиться в секции *public*.

MQL5 предоставляет другой удобный способ описания интерфейсов — с помощью ключевого слова *interface*. Все методы в интерфейсе объявляются без реализации и считаются публичными и виртуальными. Описание интерфейса *Drawable*, эквивалентное вышеприведенному классу, выглядит так:

```
interface Drawable
{
   void draw();
};
```

В классах наследниках при этом ничего не приходится менять, если в абстрактном классе не было никаких полей (что было бы нарушением принципа абстракции).

Теперь пришло время расширить интерфейс и сделать тройку методов *setColor*, *moveX*, *moveY* также его частью.

```
interface Drawable
{
   void draw();
   Drawable *setColor(const color c);
   Drawable *moveX(const int x);
   Drawable *moveY(const int y);
};
```

Обратите внимание, что методы возвращают объект *Drawable*, потому что ничего не знаю о *Shape*. В классе *Shape* у нас уже имеются реализации, которые подходят для переопределения этих методов, поскольку *Shape* унаследован от *Drawable* (объекты *Shape* "являются своего рода" объектами *Drawable*).

Теперь сторонние разработчики могут добавить в программу рисования другие семейства классов *Drawable*, в частности, не только фигуры, но и текст, растровые картинки, а также, представьте себе, коллекции других *Drawable*, что позволяет вкладывать объекты друг в друга и составлять сложные композиции. Достаточно наследоваться от интерфейса и реализовать его методы.

```
class Text : public Drawable
{
public:
   Text(const string label)
   {
      ...
   }

   void draw()
   {
      ...
   }

   Text *setColor(const color c)
   {
      ...
      return &this;
   }
   ...
};
```

Если бы классы фигур распространялись в виде двоичной ex5-библиотеки (без исходных кодов), мы бы поставляли для неё заголовочный файл, содержащий только описание интерфейса, и никаких намеков о внутренних структурах данных.

Поскольку для виртуальных функций выполняется динамическое (позднее) связывание с объектом во время выполнения программы, есть вероятность получить критическую ошибку "вызов чистой виртуальной функции" ("Pure virtual function call"): программа при этом завершает работу. Это происходит, если программист по недосмотру "забыл" предоставить реализацию. Компилятор не всегда способен выявить такие упущения на стадии компиляции.

3.2.19 Перегрузка операторов

В главе Выражения мы изучили множество операций, определенных для встроенных типов. Например, для переменных типа *double* мы могли вычислить выражение:

```
double a = 2.0, b = 3.0, c = 5.0;
double d = a * b + c;
```

Было бы удобно использовать похожий синтаксис при работе с пользовательскими типами, например, матрицами:

```
Matrix a(3, 3), b(3, 3), c(3, 3); // создаем матрицы 3x3
// ... как-то заполняем a, b, c
Matrix d = a * b + c;
```

MQL5 предоставляет такую возможность за счет перегрузки операторов.

Данный технический прием организуется путем описания методов с именем, начинающимся с ключевого слова *operator*, и содержащего далее символ (или последовательность символов) одной из поддерживаемых операций. В обобщенном виде это можно представить так:

```
тип_результата operator@ ( [тип имя_параметра] );
```

Здесь @ - символ(ы) операций.

Полный перечень операций MQL5 был приведен в разделе Приоритеты операций, однако не все они разрешены для перегрузки.

Запрещены для перегрузки:

- двоеточия '::', разрешение контекста;
- круглые скобки '()', "вызов функции" или "группировка";
- точка '.', "разыменование";
- амперсанд '&', "взятие адреса", унарный оператор (однако доступен амперсанд как бинарный оператор "побитовое И");
- условный тернарный '?:';
- запятая ','.

Все остальные операторы доступны для перегрузки. Приоритеты операций при перегрузке нельзя изменить, они остаются равными стандартным приоритетам, поэтому следует, при необходимости, использовать группировку с помощью скобок.

Создать перегрузку для какого-то нового символа, не входящего в стандартный перечень, нельзя.

Все операторы перегружаются с учетом их унарности и бинарности, то есть количество требуемых операндов сохраняется. Как любой метод класса, перегрузка оператора может вернуть

значение некоторого типа. При этом сам тип следует выбирать исходя из планируемой логики использования результата функции в выражениях (см. далее).

Методы перегрузки операторов имеют следующий вид (вместо символа '@' подставляется символ(ы) требуемого оператора):

Название	Заголовок метода	Использование в выражении	Функциональный эквивалент
унарные префиксные	тип operator@()	@object	object.operator@()
унарные постфиксные	тип operator@(int)	object@	object.operator@(0)
бинарные	тип operator@(тип имя_параметра)	object@argument	object.operator@(argument)
индекс	тип operator[](тип имя_индекса)	object[argument]	object.operator[](argument)

Унарные операторы не принимают параметров. Из унарных только операторы инкремента '++' и декремента '--' поддерживают постфиксную форму в дополнение к префиксной, все остальные унарные операторы — только префиксные. Указание анонимного параметра типа *int* используется для обозначения постфиксной формы (чтобы отличить от префиксной), но сам параметр игнорируется.

Бинарные операторы должны принимать один параметр. Для одного и того же оператора возможно несколько перегруженных вариантов с параметром разного типа, включая и такой же тип, как класс текущего объекта. При этом объекты в качестве параметров могут передаваться только со ссылке или по указателю (последнее — только для объектов классов, но не структур).

Перегруженные операторы можно применять как с помощью синтаксиса операций в составе выражений (ради чего, собственно, и делается перегрузка), так и синтаксиса вызовов методов — оба варианта приведены в таблице выше. Функциональный эквивалент делает более очевидным, что с технической точки зрения оператор — не более чем вызов метода для объекта, причем для префиксных операторов объект стоит справа, а для всех остальных — слева от символа. Методу бинарного оператора будет передаваться в качестве аргумента то значение или выражение, что стоит справа от оператора (это может быть, в частности, другой объект или переменная встроенного типа).

Отсюда следует, что перегруженные операторы не обладают свойством коммутативности: *a@b* в общем случае не равно *b@a*, потому что для *a* оператор @ может быть перегружен, а для *b* — нет. Более того, если *b* является переменной или значением встроенного типа, то для него в принципе нельзя перегрузить стандартное поведение.

В качестве первого примера рассмотрим класс *Fibo* для генерации чисел из ряда Фибоначчи (мы уже делали одну реализацию данной задачи с помощью функций, см. Определение функции). В классе предусмотрим 2 поля для хранения текущего и предыдущего числа ряда: *current* и *previous* соответственно. Конструктор по умолчанию будет их инициализировать значениями 1 и 0. Также предусмотрим конструктор копирования (*FiboMonad.mq5*).

```
class Fibo
{
   int previous;
   int current;
public:
   Fibo() : current(1), previous(0) { }
   Fibo(const Fibo &other) : current(other.current), previous(other.previous) { }
   ...
};
```

Начальное состояние объекта: текущее число 1, предыдущее — 0. Для нахождения следующего числа в ряду перегрузим префиксные и постфиксные операции инкремента.

```
   Fibo *operator++() // prefix
   {
      int temp = current;
      current = current + previous;
      previous = temp;
      return &this;
   }

   Fibo operator++(int) // postfix
   {
      Fibo temp = this;
      ++this;
      return temp;
   }
```

Обратите внимание, что префиксный метод возвращает указатель не текущий объект *Fibo* после модификации числа, а постфиксный — на новый объект с сохраненным предыдущим счетчиком, что соответствует принципам работы постфиксного инкремента.

При необходимости, программист, конечно, может перегрузить любую операцию произвольным образом. Например, никто не запрещает в реализации инкремента вычислять произведение, выводить число в журнал или делать что-то еще. Однако рекомендуется придерживаться подхода, когда перегрузка операции выполняет интуитивно понятные действия.

Операции декремента реализуем по похожему принципу: они будут возвращать предыдущее число ряда.

```
Fibo *operator--() // prefix
{
   int diff = current - previous;
   current = previous;
   previous = diff;
   return &this;
}

Fibo operator--(int) // postfix
{
   Fibo temp = this;
   --this;
   return temp;
}
```

Для получения числа из ряда по заданному номеру перегрузим операцию доступа по индексу.

```
Fibo *operator[](int index)
{
   current = 1;
   previous = 0;
   for(int i = 0; i < index; ++i)
   {
      ++this;
   }
   return &this;
}
```

Для получения текущего числа, содержащегося в переменной current, перегрузим оператор '~' (как редко используемый).

```
int operator~() const
{
   return current;
}
```

Без этой перегрузки потребовалось бы все равно реализовать какой-либо публичный метод для чтения закрытого поля *current*. Мы воспользуемся этим оператором для вывода чисел с помощью *Print*.

Также для удобства следует перегрузить присваивание.

```
   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;
   }
```

Проверим, как это все работает.

```
void OnStart()
{
   Fibo f1, f2, f3, f4;
   for(int i = 0; i < 10; ++i, ++f1) // префиксный инкремент
   {
      f4 = f3++; // постфиксный инкремент и перегрузка присваивания
   }

   // сравниваем все значения, полученные инкрементами и по индексу [10]
   Print(~f1, " ", ~f2[10], " ", ~f3, " ", ~f4); // 89 89 89 55

   // отсчет в обратную сторону до 0
   Fibo f0;
   Fibo f = f0[10]; // конструктор копирования (из-за инициализации)
   for(int i = 0; i < 10; ++i)
   {
      // префиксный декремент
      Print(~--f); // 55, 34, 21, 13, 8, 5, 3, 2, 1, 1
   }
}
```

Результаты совпадают с ожидаемыми. Осталось рассмотреть один нюанс.

```
   Fibo f5;
   Fibo *pf5 = &f5;

   f5 = f4;   // вызов Fibo *operator=(const Fibo &other)
   f5 = &f4;  // вызов Fibo *operator=(const Fibo *other)
   pf5 = &f4; // ничего не вызывает, присваивает &f4 в pf5!
```

Перегрузка оператора присваивания для указателя работает только при обращении через объект. Если обращение идет через указатель, то происходит стандартное присваивание одного указателя другому.

В качестве возвращаемого типа перегруженного оператора можно использовать один из встроенных типов, объектный тип (класса или структуры) или указатель (только для объектов класса).

Для возврата объекта (экземпляра, а не ссылки) в классе должен быть реализован конструктор копирования. Такой способ вызовет дублирование экземпляра, что может сказаться на эффективности кода. По возможности следует возвращать указатель.

Вместе с тем, при возврате указателя нужно убедиться, что возвращается не локальный автоматической объект (который будет удален при выходе из функции, и указатель станет недействительным), а какой-либо уже существующий — как правило, возвращается *&this*.

Возврат объекта или указателя на объект позволяет "отправлять" результат работы одного перегруженного оператора в другой, и тем самым конструировать сложные выражения по аналогии с тем, как мы привыкли делать со встроенными типами. Возврат *void* приведет к невозможности использовать оператор в выражениях. Например, если оператор '=' будет определен с типом *void*, то перестанет работать множественное присваивание:

```
Type x, y, z = 1; // конструкторы и инициализация переменных некоего класса
x = y = z; // присваивания, ошибка компиляции
```

Цепочка присваиваний выполняется справа налево, а *y = z* вернет пустоту.

Если объекты содержат поля только встроенных типов (включая массивы), то оператор присваивания/копирования '=' из объектов того же класса переопределять не обязательно: MQL5 по умолчанию обеспечивает копирование всех полей "один в один". Не следует путать оператор присваивания/копирования с конструктором копирования и инициализацией.

Теперь обратимся ко второму примеру: работе с матрицами (*Matrix.mq5*).

> Отметим, между прочим, что недавно в MQL5 появились встроенные объектные типы матриц и векторов. Использовать ли встроенные типы или свои (а может быть, комбинировать их) — выбор каждого разработчика. Готовая и быстрая реализация многих востребованных методов во встроенных типах удобна и избавляет от рутинного кодирования. С другой стороны, собственные классы позволяют адаптировать алгоритмы под свои задачи. Здесь мы приводим класс *Matrix* в качестве учебного пособия.

В классе матрицы обеспечим хранение её элементов в одномерном динамическом массиве *m*. Под размеры выделим переменные *rows* и *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);
   }
```

Основной конструктор принимает два параметра (размеры матрицы) и выделяет память под массив. Также имеется конструктор копирования из другой матрицы *other*. Здесь и далее массово используются встроенные функции для работы с массивами (в частности, *ArrayCopy*, *ArrayResize*, *ArrayInitialize*) — они будут рассмотрены в отдельной главе.

Заполнение элементов организуем из внешнего массива, перегрузив оператор присваивания:

```
   Matrix *operator=(const double &a[])
   {
      if(ArraySize(a) == ArraySize(m))
      {
         ArrayCopy(m, a);
      }
      return &this;
   }
```

Для реализации сложения двух матриц перегрузим операции '+=' и '+':

```
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;
}
```

Обратите внимание, что оператор '+=' возвращает указатель на текущий объект после его модификации, а оператор '+' — новый экземпляр по значению (будет использован конструктор копирования), причем сам оператор имеет модификатор *const*, так как не меняет текущий объект.

Оператор '+' является по сути оберткой, которая делегирует всю работу оператору '+=', предварительно создав для его вызова временную копию текущей матрицы под именем *temp*. Таким образом, *temp* суммируется с *other* при помощи внутреннего вызова оператора '+=' (при этом *temp* модифицируется) и затем возвращается как результат оператора '+'.

Умножение матриц перегружается аналогично, с помощью двух операторов '*=' и '*'.

```
Matrix *operator*=(const Matrix &other)
{
   // условие перемножения: this.columns == other.rows
   // результатом будет матрица размером this.rows на 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;
         // суммируем попарные произведения i-х элементов
         // ряда 'r' текущей матрицы и столбца 'c' матрицы other
         for(int i = 0; i < columns; ++i)
         {
            t += m[r * columns + i] * other.m[i * other.columns + c];
         }
         temp.assign(r, c, t);
      }
   }
   // копируем результат в текущий объект матрицы this
   this = temp; // вызов перегруженного оператора присваивания
   return &this;
}

Matrix operator*(const Matrix &other) const
{
   Matrix temp(this);
   return temp *= other;
}
```

Наконец, умножение матрицы на число:

```
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;
}
```

Для сравнения двух матриц предоставим операторы '==' и '!=':

```
bool operator==(const Matrix &other) const
{
   return ArrayCompare(m, other.m) == 0;
}

bool operator!=(const Matrix &other) const
{
   return !(this == other);
}
```

Для отладочных целей реализуем вывод массива матрицы в журнал.

```
void print() const
{
   ArrayPrint(m);
}
```

Кроме описанных перегрузок в классе *Matrix* дополнительно имеется перегрузка оператора []: он возвращает объект вложенного класса *MatrixRow*, то есть строку с заданным номером.

```
MatrixRow operator[](int r)
{
   return MatrixRow(this, r);
}
```

Сам класс *MatrixRow* обеспечивает более "глубокий" доступ к элементам матрицы путем перегрузки того же оператора [] (то есть для матрицы можно будет естественным образом указать два индекса *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];
   }
};
```

Оператор [] для параметра типа *int* возвращает объект класса *MatrixElement*, через который можно записывать конкретный элемент в массиве. Для чтения элемента применяется оператор [] с параметром типа *uint*. Это похоже на трюк, но таково ограничение языка: перегрузки должны отличаться типом параметра. В качестве альтернативы для чтения элемента в классе *MatrixElement* предусмотрена перегрузка оператора '~'.

При работе с матрицами часто нужна единичная, поэтому создадим для неё производный класс:

```
class MatrixIdentity : public Matrix
{
public:
   MatrixIdentity(const int n) : Matrix(n, n)
   {
      for(int i = 0; i < n; ++i)
      {
         m[i * rows + i] = 1;
      }
   }
};
```

Теперь попробуем матричные выражения в действии.

```
void OnStart()
{
   Matrix m(2, 3), n(3, 2); // описание
   MatrixIdentity p(2);     // единичная
   
   double ma[] = {-1,  0, -3,
                   4, -5,  6};
   double na[] = {7,  8,
                  9,  1,
                  2,  3};
   m = ma; // заполнение данные
   n = na;
   
   // можем читать и записывать элементы отдельно
   m[0][0] = m[0][(uint)0] + 2; // вариант 1
   m[0][1] = ~m[0][1] + 2;      // вариант 2
   
   Matrix r = m * n + p;                    // выражение
   Matrix r2 = m.operator*(n).operator+(p); // эквивалент
   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
}
```

Здесь мы создали 2 матрицы размером 3 на 2 и 2 на 3 соответственно, затем заполнили их значениями из массивов и отредактировали выборочный элемент с помощью синтаксиса двух индексов [][]. Наконец вычислили выражение *m * n + p*, где все операнды — это матрицы. Строкой ниже показано то же самое выражение в форме вызовов методов. Результаты совпадают.

> В MQL5, в отличие от C++, не поддерживается перегрузка операторов на глобальном уровне. В MQL5 можно перегрузить оператор только в контексте класса или структуры, то есть с помощью их метода. Также MQL5 не поддерживает перегрузку приведения типов, операторов *new* и *delete*.

3.2.20 Приведение объектных типов: dynamic_cast и указатель void *

Объектные типы имеют собственные правила приведения, применяемые, когда типы переменных источника и приемника не совпадают. Правила для встроенных типов уже рассматривались в главе 2.6 Приведение типов. Особенности приведения типов структур при их копировании излагались в разделе Компоновка и наследование структур.

Как для структур, так и для классов главным условием допустимости приведения типов является то, чтобы они были связаны по цепочке наследования. Типы из разных веток иерархии или вообще не связанные родственными отношениями приводить друг к другу нельзя.

Правила приведения различаются для объектов (значений) и указателей.

Объекты

Объект одного типа A можно присвоить объекту другого типа B при наличии в последнем конструктора, принимающего параметр типа A (с вариациями по значению, ссылке или указателю, но как правило, вида *B(const A &a)*). Такой конструктор еще называется конструктором преобразования.

В отсутствие такого явного конструктора, компилятор попытается применить неявный оператор копирования, то есть вида *B::operator=(const B &b)*, при этом классы A и B должны быть в одной цепочке наследования, чтобы сработала неявная конвертация из A в B. Если A унаследован от B (в том числе, не напрямую, а опосредованно), то добавленные в A свойства пропадут при копировании в B. Если же B унаследован от A, то в него скопируется только та часть свойств, что имеется в A. В принципе, такие конвертации не приветствуются.

Кроме того, неявный оператор копирования не всегда может быть предоставлен компилятором. В частности, если в классе имеются поля с модификатором *const*, копирование считается запрещенным (см. далее).

В скрипте *ShapesCasting.mq5* мы используем иерархию классов фигур для демонстрации преобразований объектных типов. В классе *Shape* поле *type* намеренно сделано константным, поэтому попытка преобразовать (присвоить) объект *Square* в объект *Rectangle* заканчивается ошибкой компилятора с подробными пояснениями:

```
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
```

Согласно этому сообщению, метод копирования *Rectangle::operator=(const Rectangle&)* был неявным образом удален компилятором (который и предоставляет его реализацию по умолчанию), поскольку использует аналогичный метод базового класса *Shape::operator=(const Shape&)*, а тот, в свою очередь, был удален из-за наличия поля *type* с модификатором *const*. Такие поля можно установить только при создании объекта, и компилятор не знает, как копировать объект при таком ограничении.

Между прочим, эффект "удаления" методов доступен не только компилятору, но прикладному программисту: подробнее об этом будет рассказано в разделе Управление наследованием: final и delete.

Решить проблему можно было бы удалением модификатора *const* или предоставив собственную реализацию оператора присваивания (в ней *const*-поле не задействовано и сохранит содержимое с описанием типа: "Rectangle"):

```
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;
}
```

Обратите внимание, что это определение возвращает указатель на текущий объект, в то время как реализация, генерируемая компилятором по умолчанию, имела тип *void* (это видно из сообщения об ошибке). Это означает, что предоставляемые компилятором по умолчанию операторы присваивания нельзя использовать в цепочке $x = y = z$. Если вам требуется такая возможность, переопределите *operator*= явным образом и верните требуемый тип, отличный от *void*.

Указатели

Наиболее практичным является преобразование указателей на объекты разных типов.

В принципе, все варианты приведения указателей объектных типов можно свести к трем:

- от базового к производному, нисходящее приведение типов (downcast), потому что иерархию классов принято рисовать перевернутым деревом;
- от производного к базовому, восходящее приведение типов (upcast);
- между классами разных веток иерархии или вообще из разных семейств.

Последний вариант запрещен (получим ошибку компиляции). Первые два компилятор разрешает, но если "upcast" является естественным и безопасным, то "downcast" способен приводить к ошибкам на стадии выполнения программы.

```
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
   ...
};
```

Разумеется, когда используется указатель на объект базового класса, для него нельзя вызвать методы и свойства производного класса, даже если по указателю находится соответствующий объект. Получим ошибку компиляции "неизвестный идентификатор" ("undeclared identifier").

Однако для указателей поддерживается синтаксис явного приведения типов (см. C-стиль), который позволяет в выражениях "на лету" преобразовывать указатель в требуемый тип и разыменовывать его без создания промежуточной переменной.

```
Base *b;
Derived d;
b = &d;
((Derived *)b).derivedMethod();
```

Здесь мы создали объект производного класса (*Derived*) и указатель на него базового типа (*Base **). Для доступа к методу *derivedMethod* производного класса указатель временно преобразуется к типу *Derived*.

Тип указателя со звездочкой следует заключать в круглые скобки. Кроме того, само выражение с приведением, включая и имя переменной, также обрамляется еще одной парой круглых скобок.

Другую ошибку компиляции ("несоответствие типов" — "type mismatch") в нашем тесте генерирует строка, где мы пробуем привести указатель на *Rectangle* к указателю на *Circle*: они из разных веток наследования.

```
   c = r; // ошибка: type mismatch
```

Гораздо хуже обстоят дела, когда тип указателя, к которому выполняется приведение, не соответствует реальному объекту (хотя их типы совместимы, и потому программа компилируется нормально). Такая операция закончится ошибкой уже на стадии выполнения программы (то есть компилятор отловить её не может). Программа при этом выгружается.

Например, в скрипте *ShapesCasting.mq5* мы описали указатель на *Square* и присваиваем ему указатель на *Shape*, в котором находится объект *Rectangle*.

```
Square *s2;
// ОШИБКА ВРЕМЕНИ ВЫПОЛНЕНИЯ
s2 = p; // ошибка: Incorrect casting of pointers
```

Терминал выдает ошибку "Неправильное приведение типов указателей" ("Incorrect casting of pointers"). Указатель более конкретного типа *Square* не способен указывать на родительский объект *Rectangle*.

Чтобы избежать неприятностей во время выполнения и предотвратить "падение" программы, MQL5 предоставляет специальную языковую конструкцию *dynamic_cast*. С её помощью можно "осторожно" проверить, можно ли приводить указатель к требуемому типу. Если конвертация возможна, то она будет произведена. А если нет, мы получим нулевой указатель (NULL) и сможем его особым образом обработать (например, с помощью *if* как-то инициализировать или прервать выполнение функции, но не программы целиком).

Синтаксис *dynamic_cast* следующий:

```
dynamic_cast< Класс * >( указатель )
```

В нашем случае достаточно написать:

```
s2 = dynamic_cast<Square *>(p); // пытаемся привести тип, получим NULL при неудаче
Print(s2); // 0
```

Программа выполнится нормально.

В частности, можно еще раз попытаться привести прямоугольник к кругу и убедиться, что получим 0:

```
c = dynamic_cast<Circle *>(r); // пытаемся привести тип, получим NULL при неудаче
Print(c); // 0
```

В MQL5 существует специальный тип указателей, который способен хранить любой объект. Этот тип обозначается: *void **.

Продемонстрируем, как переменная *void ** работает с *dynamic_cast*.

```
void *v;
v = s;   // устанавливаем на экземпляр 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));
```

Первые три строки выведут в журнал значение указателя (дескриптор одного и того же объекта), а две последних — 0.

Теперь вернемся к примеру предварительного объявления в разделе Указатели (см. файл *ThisCallback.mq5*), где классы *Manager* и *Element* содержали взаимные указатели.

Тип указателя *void ** позволяет избавиться от предварительного объявления (*ThisCallbackVoid.mq5*). Закомментируем строку с ним, а тип поля *owner* с указателем на объект-менеджер поменяем на *void **. В конструкторе также поменяем тип параметра.

```
// class Manager;
class Element
{
   void *owner; // рассчитываем на совместимость с типом Manager *
public:
   Element(void *t = NULL): owner(t) { } // было Element(Manager &t)
   void doMath()
   {
      const int N = 1000000;

      // получаем нужный тип во время выполнения
      Manager *ptr = dynamic_cast<Manager *>(owner);
      // далее везде нужно проверять ptr на NULL перед использованием

      for(int i = 0; i < N; ++i)
      {
         if(i % (N / 20) == 0)
         {
            if(ptr != NULL) ptr.progressNotify(&this, i * 100.0f / N);
         }
         // ... lot of calculations
      }
      if(ptr != NULL) ptr.progressNotify(&this, 100.0f);
   }
   ...
};
```

Данный подход может предоставлять большую гибкость, но и требует большей осторожности, так как *dynamic_cast* способен вернуть NULL. Рекомендуется по возможности пользоваться стандартными средствами диспетчеризации (статической и динамической) с контролем типов, представляемых языком.

Указатели *void ** обычно становятся необходимы в исключительных случаях. И "лишняя" строка с предварительным описанием — не тот случай. Здесь оно было использовано только как наиболее простой пример универсальности указателя *void **.

3.2.21 Указатели, ссылки и const

После изучения встроенных и объектных типов, понятий ссылка и указатель, вероятно, имеет смысл сделать сравнение всех доступных модификаций типов.

Ссылки в MQL5 используются только при описании параметров функций и методов. Более того, параметры объектных типов обязательно должны передаваться по ссылке.

```
void function(ClassOrStruct &object) { }          // можно
void function(ClassOrStruct object) { }           // нельзя
void function(double &value) { }                  // можно
void function(double value) { }                   // можно
```

Здесь *ClassOrStruct* — имя класса или структуры.

В качестве аргумента для параметра типа ссылки допустимо передавать только переменные (LValue), но не константы или временные значения, полученные в результате вычисления выражения.

Создать переменную ссылочного типа или вернуть ссылку из функции нельзя.

```
ClassOrStruct &function(void) { return Class(); } // нельзя
ClassOrStruct &object;                            // нельзя
double &value;                                    // нельзя
```

Указатели в MQL5 доступны только для объектов классов. Указатели на переменные встроенных типов или структур не поддерживаются.

Вы можете объявить переменную или параметр функции типа указатель на объект, а также вернуть указатель на объект из функции.

```
ClassOrStruct *pointer;                                  // можно
void function(ClassOrStruct *object) { }                 // можно
ClassOrStruct *function() { return new ClassOrStruct(); } // можно
```

Однако нельзя возвращать указатель на локальный автоматический объект, так как последний будет освобожден при выходе из функции, и указатель станет недействительным.

Если функция вернула указатель на объект, распределенный динамически внутри функции с помощью new, то вызывающий код должен "не забыть" освободить указатель с помощью delete.

Указатель, в отличие от ссылки, может быть равен NULL. Параметры-указатели могут иметь значение по умолчанию, ссылки — нет (ошибка "reference cannot be initialized").

```
void function(ClassOrStruct *object = NULL) { }          // можно
void function(ClassOrStruct &object = NULL) { }          // нельзя
```

Ссылки и указатели могут быть совмещены в описании параметра. Так функция может принимать ссылку на указатель: тогда изменения указателя в функции станут доступны в вызывающем коде. В частности, подобным образом можно реализовать фабричную функцию, которая ответственна за создание объектов.

```
void createObject(ClassName *&ref)
{
   ref = new ClassName();
   // дальнейшая настройка ref
   ...
}
```

Правда, для возврата из функции одного указателя обычно принято использовать оператор return, так что данный пример — несколько искусственный. Однако в тех случаях, когда передать наружу необходимо массив указателей, ссылка на него в параметре становится предпочтительным вариантом. Например, в некоторых классах стандартной библиотеки для работы с классами-контейнерами типа карт с парами [ключ,значение] (*MQL5/Include/Generic/SortedMap.mqh*, *MQL5/Include/Generic/HashMap.mqh*) имеются методы *CopyTo* для получения массивов с элементами *CKeyValuePair*.

```
int CopyTo(CKeyValuePair<TKey,TValue> *&dst_array[], const int dst_start = 0);
```

Тип параметра *dst_array* может показаться незнакомым: это шаблон класса. Мы изучим шаблоны в следующей главе. Здесь для нас пока важно лишь то, что это ссылка на массив указателей.

Особое поведение для всех типов накладывает модификатор *const*. В отношении встроенных типов он был рассмотрен в разделе Переменные-константы. Для объектных типов есть свои особенности.

Если переменная или параметр функции описан как указатель или ссылка на объект (ссылка — только в случае параметра), то наличие у них модификатора *const* ограничивает набор методов и свойств, к которым можно обращаться, только теми, что также имеют модификатор *const*. Иными словами, через константные ссылки и указатели доступны только константные свойства.

При попытке вызвать неконстантный метод или изменить неконстантное поле компилятор выдаст ошибку: "вызов неконстантного метода для константного объекта" ("call non-const method for constant object") или "константу нельзя изменить" ("constant cannot be modified").

Неконстантный параметр-указатель может принять любой аргумент (как константу, так и неконстанту).

Следует иметь в виду, что в описании указателя можно ставить два модификатора *const*: один будет относиться к объекту, а второй — к указателю:

- *Class *pointer* — указатель на объект; объект и указатель работают без ограничений;
- *const Class *pointer* — указатель на константный объект; для объекта доступны только константные методы и чтение свойств, но указатель можно изменить (присвоить ему адрес другого объекта);
- *const Class * const pointer* — константный указатель на константный объект; для объекта доступны только константные методы и чтение свойств; указатель менять нельзя;
- *Class * const pointer* — константный указатель на объект; указатель менять нельзя, а свойства объекта — можно.

Рассмотрим в качестве примера следующий класс *Counter* (*CounterConstPtr.mq5*).

```
class Counter
{
public:
   int counter;

   Counter(const int n = 0) : counter(n) { }

   void increment()
   {
      ++counter;
   }

   Counter *clone() const
   {
      return new Counter(counter);
   }
};
```

В нем искусственно сделана публичной переменная *counter*. Также в классе имеется два метода, один из которых константный (*clone*), а второй — нет (*increment*). Напомним, что константный метод не имеет права менять поля объекта.

Следующая функция с параметром типа *Counter *ptr* может вызывать все методы класса и менять его поля.

```
void functionVolatile(Counter *ptr)
{
   // OK: все доступно
   ptr.increment();
   ptr.counter += 2;
   // удаляем клон сразу, чтобы не было утечки памяти
   // клон нужен только для демонстрации вызова константного метода
   delete ptr.clone();
   ptr = NULL;
}
```

Следующая функция с параметром *const Counter *ptr* вызовет пару ошибок.

```
void functionConst(const Counter *ptr)
{
   // ОШИБКИ:
   ptr.increment(); // call non-const method for constant object
   ptr.counter = 1; // constant cannot be modified

   // OK: доступны только const-методы, поля можно читать
   Print(ptr.counter); // чтение const-объекта
   Counter *clone = ptr.clone(); // вызов const-метода
   ptr = clone;     // изменяем non-const указатель ptr
   delete ptr;      // чистим память
}
```

Наконец, следующая функция с параметром *const Counter * const ptr* позволяет еще меньше.

```
void functionConstConst(const Counter * const ptr)
{
   // OK: доступны только const-методы, ptr нельзя менять
   Print(ptr.counter); // чтение const-объекта
   delete ptr.clone(); // вызов const-метода

   Counter local(0);
   // ОШИБКИ:
   ptr.increment(); // call non-const method for constant object
   ptr.counter = 1; // constant cannot be modified
   ptr = &local;    // constant cannot be modified
}
```

В функции *OnStart*, где мы описали два объекта *Counter* (один константный, а другой нет), можно вызывать эти функции с некоторыми исключениями:

```
void OnStart()
{
   Counter counter;
   const Counter constCounter;

   counter.increment();

   // ОШИБКА:
   // constCounter.increment(); // call non-const method for constant object
   Counter *ptr = (Counter *)&constCounter; // трюк: приведение типа без const
   ptr.increment();

   functionVolatile(&counter);

   // ОШИБКА: cannot convert from const pointer...
   // functionVolatile(&constCounter); // to nonconst pointer

   functionVolatile((Counter *)&constCounter); // приведение типа без const

   functionConst(&counter);
   functionConst(&constCounter);

   functionConstConst(&counter);
   functionConstConst(&constCounter);
}
```

Во-первых, обратим внимание, что для переменных точно также генерируется ошибка при попытке вызвать константный метод *increment* для неконстантного объекта.

Во-вторых, в функцию *functionVolatile* нельзя передать *constCounter* — получаем ошибку "нельзя конвертировать константный указатель в неконстантный" ("cannot convert from const pointer to nonconst pointer").

Однако обе ошибки можно обойти с помощью явного приведения типа без модификатора *const*. Хотя так делать не рекомендуется.

3.2.22 Управление наследованием: final и delete

MQL5 позволяет накладывать некоторые ограничения на наследование классов и структур.

Ключевое слово *final*

С помощью ключевого слова *final*, добавленного после имени класса, разработчик может запретить наследование от этого класса. Например (*FinalDelete.mq5*):

```
class Base
{
};

class Derived final : public Base
{
};

class Concrete : public Derived // ОШИБКА
{
};
```

Компилятор выдаст ошибку "нельзя наследоваться от 'Derived', потому что он определен как 'final'" ("cannot inherit from 'Derived' as it has been declared as 'final'").

Единого мнения по поводу пользы и сценариев использования такого ограничения, к сожалению, нет. Ключевое слово дает знать пользователям класса, что его автор по тем или иным соображениям не рекомендует брать его в качестве базового (например, его текущая реализация является черновой и будет сильно меняться, из-за чего потенциальные унаследованные проекты могут перестать компилироваться).

Некоторые пытаются таким образом стимулировать проектирование программ, в которых включение объектов (композиция) применяется вместо наследования. Чрезмерное увлечение наследованием действительно способно увеличить сцепление классов (т.е. взаимное влияние), поскольку все наследники тем или иным образом могут менять родительские данные или методы (в частности, переопределяя виртуальные функции). В результате возрастает сложность рабочей логики программы и вероятность непредвиденных побочных эффектов.

Дополнительным преимуществом от использования *final* может быть оптимизация кода компилятором: для указателей "финальных" типов он сможет заменить динамическую диспетчеризацию виртуальных функций на статическую.

Ключевое слово *delete*

Ключевое слово *delete* может быть указано в заголовке метода, чтобы сделать его недоступным в текущем классе и его наследниках. Виртуальные методы родительских классов удалять таким образом нельзя (это нарушило бы "контракт" класса, то есть наследники перестали бы "являться" ("is a") представителями того же рода).

```
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();

   // ОШИБКА:
   // attempting to reference deleted function 'void Derived::method()'
   //    function 'void Derived::method()' was explicitly deleted
   d.method();
}
```

Попытка его вызвать будет приводить к ошибке компиляции.

Мы видели похожую ошибку в разделе Приведение объектных типов, потому что компилятор обладает некоторым интеллектом, чтобы также "удалять" методы в определенных условиях.

Помечать удаленными рекомендуется следующие методы, для которых компилятор предоставляет неявные реализации:

- конструктор по умолчанию: *Class(void) = delete*;
- конструктор копирования: *Class(const Class &object) = delete*;
- оператор копирования/присваивания: *void operator=(const Class &object) = delete*.

Если вам требуется любой из них, следует его определить явным образом. В противном случае хорошим тоном считается отказаться от неявной реализации. Дело в том, что неявная реализация довольно прямолинейна и способна породить трудно локализуемые проблемы, в частности, при приведении объектных типов.

3.3 Шаблоны

В современных языках программирования существует множество встроенных средств, позволяющих избегать дублирования кода и, тем самым, минимизировать количество ошибок и повышать производительность программиста. В MQL5 к числу таких средств можно отнести уже известные нам функции, объектные типы с поддержкой наследования (классы и структуры), макроопределения препроцессора и возможность включения файлов. Но этот список был бы неполным без шаблонов.

Шаблон — это специальным образом составленное обобщенное определение функции или объектного типа, на базе которого компилятор способен автоматически генерировать рабочие экземпляры этой функций или объектного типа. Полученные экземпляры содержат один и тот же алгоритм, но оперируют переменными различных типов, соответствующих конкретным условиям использования шаблона в исходном коде.

Для знатоков C++ отметим, что шаблоны MQL5 не поддерживают многие возможности шаблонов C++, в частности:

- параметры, не являющиеся типами;
- параметры со значениями по умолчанию;
- переменное количество параметров;
- специализация классов, структур и объединений (полная и частичная);
- шаблоны шаблонов.

С одной стороны, это снижает потенциал шаблонов в MQL5, но, с другой стороны, упрощает освоение материала тем, кто незнаком с этими технологиями.

3.3.1 Заголовок шаблона

Шаблонными в MQL5 можно сделать функции, объектные типы (классы, структуры, объединения) или отдельные методы внутри них. В любом случае описание шаблона имеет заголовок:

```
template <typename T[, typename Ti, ... ]>
```

Заголовок начинается с ключевого слова *template*, за которым в угловых скобках, через запятую перечисляются формальные параметры: каждый параметр обозначается ключевым словом *typename* и идентификатором. Идентификаторы должны быть уникальными внутри конкретного определения.

Ключевое слово *typename* в заголовке шаблона сообщает компилятору, что идущий следом идентификатор следует рассматривать как заместитель типа. В будущем компилятор MQL5, вероятно, поддержит другие виды параметров, не являющихся типами, как это делает компилятор C++.

Не следует путать данное применение *typename* и встроенный оператор *typename*, возвращающий строку с названием типа переданного аргумента.

После заголовка шаблона следует привычное определение функции (метода) или класса (структуры, объединения), в котором формальные параметры шаблона (идентификаторы T, Ti) используются в инструкциях и выражениях в тех местах, где синтаксис требует наличия названия типа. Например, для шаблонных функций параметры шаблона описывают типы параметров функции или возвращаемого значения, а в шаблонном классе параметр шаблона может обозначать тип поля.

Шаблоном является всё определение целиком. Шаблон заканчивается вместе с определением сущности (функции, метода, класса, структуры, объединения), перед которой был заголовок *template*.

Для имен параметров шаблонов принято брать одно- или двухсимвольные идентификаторы в верхнем регистре.

Минимальное количество параметров — 1, максимальное — 64.

Основные варианты использования параметров (на примере параметра T) включают:

- тип при описании полей, локальных переменных в функциях/методах, их параметров и возвращаемых значений (*T имя_переменной*; *T функция(T имя_параметра)*);
- одна из составляющих полностью квалифицированного имени типа, в частности: *T::SubType*, *T.StaticMember*;
- конструирование новых типов с модификаторами: *const T*, указатель *T **, ссылка *T &*, массив *T[]*, *typedef* функции *T(*func)(T)*;
- конструирование новых шаблонных типов: *T<Type>*, *Type<T>*, в том числе и при наследовании от шаблонов (см. раздел Специализация шаблонов, которой нет);
- приведение типов *(T)* с возможностью добавления модификаторов и создание объектов через *new T()*;
- *sizeof(T)*, как примитивная замена параметрам-значениям, отсутствующим в MQL-шаблонах (на момент написания книги).

3.3.2 Общие принципы работы шаблонов

Вспомним перегрузку функций. Она заключается в определении нескольких версий функции с отличающимися параметрами, в том числе, если количество параметров одинаковое, но их типы — разные. Зачастую алгоритм таких функций — один и тот же для параметров разных типов. Например, в MQL5 имеется встроенная функция *MathMax*, которая возвращает наибольшее из двух переданных в неё значений:

```
double MathMax(double value1, double value2);
```

Несмотря на то, что прототип предоставлен только для типа *double*, функция на самом деле способна работать с парами аргументов других числовых типов, например, *int* или *datetime*. Иными словами, функция является перегруженной ядром для встроенных числовых типов. Если бы мы хотели достичь того же эффекта в своем исходном коде, нам пришлось бы перегрузить функцию за счет дублирования с разными параметрами, например, так:

```
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;
}
```

Все реализации (тела функций) одинаковы. Меняются только типы параметров.

В таких случаях и применяются шаблоны. С помощью них мы можем описать один образец алгоритма с требуемой реализацией, а компилятор сам сгенерирует по нему несколько экземпляров для конкретных типов, задействованных в программе. Генерация происходит на лету

в процессе компиляции и незаметна для программиста (если только в шаблоне не допущена ошибка). Полученный автоматическим способом исходный код не вставляется в текст программы, а напрямую преобразуется в двоичный код (ex5-файл).

В шаблоне в качестве одного или нескольких параметров вводятся формальные обозначения типов, для которых на стадии компиляции по особым правилам выведения типов будут подобраны реальные типы из числа встроенных или пользовательских. Например, функцию *Max* можно было бы описать следующим шаблоном с параметром-типом T:

```
template<typename T>
T Max(T value1, T value2)
{
   return value1 > value2 ? value1 : value2;
}
```

И далее — применять его для переменных различных типов (см. *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));
   ...
}
```

В данном случае компилятор автоматически сгенерирует варианты функции *Max* для типов *double* и *datetime*.

Шаблон сам по себе не генерирует исходный код. Для этого необходимо тем или иным образом создать экземпляр шаблона: вызвать шаблонную функцию или упомянуть имя шаблонного класса с конкретными типами для создания объекта или класса-наследника.

До тех пор, пока это не сделано, весь шаблон игнорируется компилятором. Например, мы можем написать следующую якобы шаблонную функцию, которая на самом деле содержит синтаксически некорректный код. Вместе с тем, компиляция модуля с этой функцией будет проходить успешно, пока она нигде не вызывается.

```
template<typename T>
void function()
{
   это не комментарий, но и не исходный код
   !%^&*
}
```

Для каждого случая использования шаблона компилятор выявляет реальные типы, соответствующие формальным параметрам шаблона. На основе этой информации, для каждого уникального сочетания параметров автоматически генерируется исходный код по шаблону. Это и есть экземпляр.

Так, в приведенном примере функции *Max*, мы вызвали шаблонную функцию два раза: для пары переменных типа *double*, и для пары переменных типа *datetime*. В результате было получено два экземпляра функции *Max* с исходным кодом для соответствий *T=double* и *T=datetime*. Разумеется, если этот же шаблон будет вызываться в других частях кода для тех же типов, новые экземпляры

генерироваться не будут. Новый экземпляр шаблона потребуется только в случае применения шаблона для другого типа (или набора типов, если параметров больше 1).

Обратите внимание, что параметр у шаблона *Max* один, и он задает тип сразу для двух входных параметров функции и её возвращаемого значения. Иными словами, описание шаблона способно накладывать определенные ограничение на типы допустимых аргументов.

Если бы мы вызвали *Max* для переменных разных типов, компилятор не смог бы выявить тип для создания экземпляра шаблона и выдал бы ошибку "неоднозначные параметры шаблона, должно быть 'double' или 'datetime'":

```
Print(Max(d1, t1)); // template parameter ambiguous,
                    // could be 'double' or 'datetime'
```

Данный процесс выявления реальных типов для параметров шаблона на основании контекста использования шаблона называется выведением (deduction) типов. В MQL5 выведение типов доступно только для шаблонов функций и методов.

Для классов, структур и объединений используется другой способ привязки типов к параметрам шаблона: требуемые типы явно указываются при создании экземпляра шаблона в угловых скобках (если параметров несколько, то соответствующее количество типов указывается через запятую). Подробнее об этом — в разделе про шаблоны объектных типов.

Этот же явный способ можно применять и для функций, как альтернативу для автоматического выведения типов.

Например, мы можем сгенерировать и вызвать экземпляр *Max* для типа *ulong*:

```
Print(Max<ulong>(1000, 10000000));
```

В данном случае, если бы не явное указание, шаблонная функция была бы связана с типом *int* (на основании значений целочисленных констант).

3.3.3 Шаблоны vs макросы препроцессора

Вероятно, может возникнуть вопрос, нельзя ли для целей кодогенерации использовать макроподстановки? В принципе, действительно, можно. Например, тот же набор функций *Max* легко представить в виде макроса:

```
#define MAX(V1,V2) ((V1) > (V2) ? (V1) : (V2))
```

Однако макросы обладают более ограниченными возможностями (это не более чем подстановка текста) и потому их реально использовать только в простых случаях (вроде приведенного выше).

При сравнении макросов и шаблонов следует отметить следующие различия.

Макросы "раскрываются" и подменяются в исходном тексте препроцессором, до начала компиляции. При этом отсутствует информация о типах параметров и контексте, в котором подставляется содержимое макроса. В частности, макрос MAX не может обеспечить проверку того, что типы параметров V1 и V2 совпадают, а также то, что для них определен оператор сравнения '>'. Кроме того, если в тексте программы повстречается переменная с именем MAX, препроцессор попытается подставить на её место "вызов" макроса MAX и будет "недоволен" отсутствием аргументов. Что еще более страшно, подобные подстановки игнорируют, в каких пространствах имен или классах, повстречался токен MAX — для них подойдет любой.

В отличие от макросов, шаблоны обрабатываются компилятором с учетом конкретных типов аргументов и места использования, поэтому для них обеспечиваются проверки на совместимость (и в целом применимость) типов для всех выражений в шаблоне, а также привязка к контексту. Например, мы можем определить шаблон метода внутри конкретного класса.

Шаблон с одним и тем же именем можно при необходимости по-разному определить для разных типов, в то время как макрос с заданным именем подменяется всегда одной и той же "реализацией". Например, в случае функции вроде MAX мы могли бы определить для строк регистронезависимое сравнение.

Ошибки компиляции из-за проблем в макросах трудно диагностировать, особенно если макрос состоит из нескольких строк, так как проблемная строка с "вызовом" макроса подсвечивается "как есть", без развернутого варианта текста, каким он поступил из препроцессора в компилятор.

В то же время, шаблоны представляют собой элементы исходного кода в уже готовом виде, какими они попадают в компилятор, и потому любая ошибка в них имеет конкретный номер строки и позицию в строке.

В макросах возможны побочные эффекты, которые мы рассматривали в разделе Форма #define в виде псевдо-функции: если аргументами макроса MAX будут выражения с инкрементами/декрементами, то они выполнятся два раза.

Вместе с тем, у макросов есть и кое-какие преимущества. Макросы способны генерировать любой текст, а не только правильные языковые конструкции. Например, с помощью нескольких макросов можно сымитировать инструкцию *switch* для строк (хотя такой подход и не рекомендуется).

В стандартной библиотеке макросы используются, в частности, для организации обработки событий на графиках (см. *MQL5/Include/Controls/Defines.mqh*: EVENT_MAP_BEGIN, EVENT_MAP_END, ON_EVENT и т.д.). Сделать так на шаблонах не получится, но способ компоновки карты событий на макросах, разумеется, далеко не единственный и не самый удобный для отладки. Отлаживать пошаговое (построчное) исполнение кода в макросах затруднительно. Шаблоны же поддерживают отладку в полном объеме.

3.3.4 Особенности встроенных и объектных типов в шаблонах

Следует иметь в виду, что ограничения на применимость типов в шаблоне накладывают 3 важных аспекта:

- является ли тип встроенным или пользовательским (пользовательские требуют передачи параметров по ссылке, а встроенные не позволят передать по ссылке литерал);
- является ли объектный тип классом (только для классов поддерживаются указателя);
- набор операций, выполняемых над данными соответствующих типов в алгоритме шаблона.

Допустим, у нас описана структура Dummy (см. скрипт *TemplatesMax.mq5*):

```
struct Dummy
{
   int x;
};
```

Если мы попробуем вызвать функцию Max для двух экземпляров структуры, то получим кучу ошибок, определяющие из которых следующие: "объекты можно передавать только по ссылке" и "нельзя применить шаблон".

```
   // ОШИБКИ:
   // 'object1' - objects are passed by reference only
   // 'Max' - cannot apply template
   Dummy object1, object2;
   Max(object1, object2);
```

Суть проблемы заключается в передаче параметров шаблонной функции по значению, а такой способ несовместим с любыми объектными типами. Чтобы её решить, можно поменять тип параметров на ссылки:

```
template<typename T>
T Max(T &value1, T &value2)
{
   return value1 > value2 ? value1 : value2;
}
```

Прежняя ошибка уйдет, но тогда мы получим новую ошибку: "'>' - неправильное использование оператора" ("'>' - illegal operation use"). Дело в том, что в шаблоне Max есть выражение с операцией сравнения '>'. Следовательно, если в шаблон подставляется пользовательский тип, в нем должен быть перегружен оператор '>' (а в структуре *Dummy* его нет: скоро мы этим займемся). Для более сложных функций, скорее всего, потребуется перегрузить гораздо большее число операторов. К счастью, компилятор подсказывает, чего именно не хватает.

Однако смена способа передачи параметров функции по ссылке дополнительно привела к неработоспособности предыдущего вызова:

```
Print(Max<ulong>(1000, 10000000));
```

Теперь он генерирует ошибки: "параметр передается по ссылке, ожидается переменная" ("parameter passed as reference, variable expected"). Таким образом, наш шаблон функции перестал работать с литералами и прочими временными значениями (в частности, в него нельзя напрямую передать выражение или результат вызова другой функции).

Можно было бы подумать, что универсальным выходом из ситуации стала бы перегрузка шаблонной функции, то есть определение обоих вариантов, отличающихся лишь амперсандом в параметрах:

```
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;
}
```

Но это не сработает. Теперь компилятор выдает ошибку "неоднозначная перегрузка функции с одинаковыми параметрами":

```
'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)
```

Окончательный, рабочий вариант перегрузки потребует добавить модификатор *const* для ссылок. Попутно мы добавили в шаблон *Max* оператор *Print*, чтобы видеть в журнале, какая из перегруженных версий вызывается и какому типу параметров соответствует 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;
   }
};
```

Также мы реализовали перегрузку оператора '>' в структуре Dummy. Поэтому все вызовы функции *Max* в тестовом скрипте завершаются успешно: как для встроенных типов, так и пользовательских, а также для литералов и переменных. В журнал выводится:

```
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
```

Внимательный читатель заметит, что у нас теперь две одинаковых функции, которые отличаются только способом передачи параметров (по значению и по ссылке), а это как раз та ситуация, против которой направлено использование шаблонов. Подобное дублирование может стать накладным, если тело функции не такое простое, как у нас. Решить это можно привычными методами: выделить реализацию в отдельную функцию и вызывать её из обеих "перегрузок", или вызывать одну "перегрузку" из другой (необязательный параметр потребовался, чтобы избежать вызова первой версией *Max* самой себя и, как следствие, переполнения стека):

```
template<typename T>
T Max(T value1, T value2)
{
   // вызываем функцию с параметрами по ссылке
   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);
}
```

Нам осталось рассмотреть еще один нюанс, связанный с пользовательскими типами, а именно использование в шаблонах указателей (напомним, они применимы только к объектам классов). Создадим простой класс *Data* и попробуем вызвать шаблонную функцию *Max* для указателей на его объекты.

```
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;
}
```

Мы увидим в журнале, что 'T=Data*', то есть атрибут указателя попадает в подставленный тип. Это наводит на мысль, что при необходимости можно описать еще одну перегрузку шаблонной функции, которая будет ответственна только за указатели.

```
template<typename T>
T *Max(T *value1, T *value2)
{
   Print(__FUNCSIG__, " T=", typename(T));
   return value1 > value2 ? value1 : value2;
}
```

В этом случае атрибут указателя '*' уже присутствует в параметрах шаблона, и потому выведение типа приводит к 'T=Data'. Такой подход позволяет предоставлять отдельную реализацию шаблона для указателей.

При наличии нескольких шаблонов, которые подходят для генерации экземпляра с конкретными типами, выбирается наиболее специализированная версия шаблона. В частности, при вызове функции *Max* с аргументами-указателями подойдут два шаблона с параметрами T (*T=Data**) и T* (*T=Data*), но поскольку первый способен принимать и значения, и указатели, то он является более общим, чем второй, который работает только с указателями. Поэтому для указателей будет выбран второй. Иными словами, чем меньше модификаторов в актуальном типе, который подставляется на место T, тем предпочтительнее вариант шаблона. Помимо атрибута указателя '*' сюда относится и модификатор *const*. Параметры *const T** или *const T* — более специализированы, чем просто *T** или *T*, соответственно.

3.3.5 Шаблоны функций

Шаблон функции состоит из заголовка с параметрами шаблона (синтаксис был описан ранее) и определения функции, в котором параметры шаблона обозначают произвольные типы.

В качестве первого примера рассмотрим функцию *Swap* для обмена местами двух элементов массива (*TemplatesSorting.mq5*). Параметр шаблона T используется как тип входной переменной-массива, а также как тип локальной переменной *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;
}
```

Все инструкции и выражения в теле функции должны быть применимы к реальным типам, для которых затем будут создаваться экземпляры шаблона. В данном случае используется оператор присваивания '='. Если для встроенных типов он существует всегда, то для пользовательских типов может потребоваться явно его перегрузить.

Компилятор по умолчанию генерирует реализацию оператора копирования для классов и структур, но она может быть удалена неявно или явно (см. ключевое слово *delete*). В частности, как мы видели в разделе Приведение объектных типов, наличие в классе константного поля приводит к тому, что компилятор удаляет свой неявный вариант копирования. Тогда вышеприведенная шаблонная функция *Swap* не сможет быть использована для объектов такого класса: компилятор выдаст ошибку.

Для классов/структур, с которыми работает функция *Swap*, желательно иметь не только оператор присваивания, но и конструктор копирования, потому что описание переменной *temp* на самом деле является конструированием с инициализацией, а не присваиванием. С конструктором копирования первая строка функции выполняется за один прием (*temp* создается на основе *array[i]*), в то время как в его отсутствие сначала будет вызван конструктор по умолчанию, а затем для *temp* выполнится оператор '='.

Посмотрим, как шаблонную функцию *Swap* можно использовать в алгоритме быстрой сортировки: его реализует другая шаблонная функция *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);
   }
}
```

Обратите внимание, что параметр T шаблона *QuickSort* определяет тип входного параметра *array*, и этот массив затем передается в шаблон *Swap*. Таким образом, выведение типа T для шаблона *QuickSort* автоматически определит и тип T для шаблона *Swap*.

Встроенная функция *ArraySize* (как и многие другие) способна работать с массивами произвольных типов: в некотором смысле она также является шаблонной, хотя и реализована непосредственно в терминале.

Сортировка выполняется благодаря оператору сравнения '>' в инструкции *if*. Как мы уже отметили ранее, данный оператор должен быть определен для любого типа T, для которого выполняется сортировка, потому что он применяется к элементам массива типа T.

Проверим, как сортировка работает для массивов встроенных типов.

```
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"
}
```

Два вызова шаблонной функции *QuickSort* приводят к автоматическому выведению типа T на основе типов передаваемых массивов. В результате мы получим два экземпляра *QuickSort* для типов *double* и *string*.

Для проверки сортировки пользовательского типа создадим структуру ABC с целочисленным полем *x*, и будем его заполнять случайными числами в конструкторе. В структуре также важно перегрузить оператор '>'.

```
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);
   /* Пример вывода:
            [x]
      [0]  1210
      [1]  2458
      [2] 10816
      [3] 13148
      [4] 15393
      [5] 20788
      [6] 24225
      [7] 29919
      [8] 32309
      [9] 32589
   */
}
```

Поскольку значения структур генерируются случайно, мы будем получать разные результаты, но они всегда будут отсортированы по возрастанию.

В данном случае тип T также выводится автоматически. Однако в некоторых случаях явное указание является единственным способом передать тип в шаблон функции. Так, если шаблонная функция должна вернуть значение уникального типа (отличного от типов её параметров) или если параметров нет, то задать его можно только явно.

Например, следующая шаблонная функция *createInstance* требует явного указания типа в инструкции вызова, так как автоматически "вычислить" тип T по возвращаемому значению нельзя. Если этого не сделать, компилятор генерирует ошибку "несоответствие шаблона" ("template mismatch").

```
class Base
{
   ...
};

template<typename T>
T *createInstance()
{
   T *object = new T(); // вызов конструктора
   ...                  // настройка объекта
   return object;
}

void OnStart()
{
   Base *p1 = createInstance();       // ошибка: template mismatch
   Base *p2 = createInstance<Base>(); // ok, явное указание
   ...
}
```

Если параметров шаблона несколько и тип возвращаемого значения не привязан ни к одному из входных параметров функции, то тоже требуется задать конкретный тип при вызове:

```
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);
}
```

Обратите внимание, что если типы для шаблона указываются в явном виде, то это требуется делать для всех параметров, несмотря на то, что второй параметр U мог бы быть выведен по передаваемому аргументу.

После того как компилятор сгенерирует все экземпляры шаблонной функции, они участвуют в стандартной процедуре выбора наилучшего кандидата из всех перегрузок функций с аналогичным именем и подходящим числом параметров. Из всех вариантов перегрузки (включая созданные экземпляры шаблонов) выбирается наиболее близкий по типам (с наименьшим количеством конвертаций).

Если в шаблонной функции некоторые входные параметры имеют конкретные типы, то она рассматривается как кандидат только при полном совпадении этих типов с аргументами: любая потребность в конвертации приведет к тому, что шаблон будет "отброшен" как неподходящий.

Нешаблонные перегрузки имеют приоритет перед шаблонными, из шаблонных "выигрывают" более специализированные ("узконаправленные").

Если аргумент шаблона (тип) задан явно, то для соответствующего аргумента функции (передаваемого значения), при необходимости, применяются правила неявного приведения типов, если эти типы отличаются.

Если несколько вариантов функции подходит в равной степени, мы получим ошибку "неоднозначный вызов перегруженной функции с одинаковыми параметрами" ("ambiguous call to overloaded function with the same parameters").

Например, если в дополнение к шаблону *MyCast* определена функция для преобразования строки в логический тип:

```
bool MyCast(const string u)
{
   return u == "true";
}
```

то вызов *MyCast<double,string>("123.0")* начнет порождать указанную ошибку, потому что две функции различаются только возвращаемым значением:

```
'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)
```

При описании шаблонных функций рекомендуется задействовать все параметры шаблона в параметрах функции. Типы могут быть выведены только из аргументов, но не из возвращаемого значения.

Если функция имеет параметр шаблонизированного типа T со значением по умолчанию, и при вызове соответствующий аргумент будет опущен, то компилятор также не сможет вывести тип T и выдаст ошибку "нельзя применить шаблон" ("cannot to apply template").

```
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();    // ошибка: cannot to apply template
   Base *p2 = createInstanceFrom(Base::type); // ok, авто-определение из аргумента
   Base *p3 = createInstanceFrom<Base>();     // ok, явное указание, аргумент опущен
}
```

3.3.6 Шаблоны объектных типов

Определение шаблона объектного типа начинается с заголовка, содержащего типизированные параметры (см. раздел Заголовок шаблона), и привычного определения класса, структуры или объединения.

```
template < typename T [, typename Ti ...] >
class имя_класса
{
   ...
};
```

Единственное отличие от стандартного определения заключается в том, что параметры шаблона могут встречаться в блоке кода, во всех синтаксических конструкциях языка, где допустимо использовать название типа.

После того как шаблон определен, его рабочие экземпляры создаются при описании в коде переменных шаблонного типа с указанием конкретных типов в угловых скобках:

```
ClassName<Type1,Type2> object;
StructName<Type1,Type2,Type3> struct;
ClassName<Type1,Type2> *pointer = new ClassName<Type1,Type2>();
ClassName1<ClassName2<Type>> object;
```

В отличие от вызова шаблонных функций, компилятор не способен вывести актуальные типы для шаблонов объектов самостоятельно.

Описание переменной шаблонного класса/структуры — не единственный способ создать экземпляр шаблона. Экземпляр также генерируется компилятором, если шаблонный тип используется в качестве базового для другого — конкретного (нешаблонного) класса или структуры.

Например, следующий класс *Worker*, даже будучи пустым, представляет собой реализацию *Base* для типа *double*:

```
class Worker : Base<double>
{
};
```

Такого минимального определения достаточно (с поправкой на добавление конструкторов, если их требует класс *Base*), чтобы запустить компиляцию и проверку кода шаблона.

В разделе Динамическое создание объектов мы познакомились с понятием динамического указателя на объект, получаемого с помощью оператора *new*. Этот гибкий механизм имеет один недостаток: за указателями нужно следить и не забывать удалить "вручную", когда они становятся не нужны. В частности, при выходе из функции или блока кода все локальные указатели нужно очистить с помощью вызова *delete*.

Чтобы упростить решение данной задачи, создадим шаблонный класс *AutoPtr* (*TemplatesAutoPtr.mq5*, *AutoPtr.mqh*). Его параметр T используем для описания поля *ptr*, хранящего указатель на объект произвольного класса. Значение указателя будем получать через параметр конструктора (*T *p*) или в перегруженном операторе '='. Основную работу поручим деструктору: в нем указатель будет удаляться вместе с объектом *AutoPtr* (для этого выделен статический вспомогательный метод *free*).

Принцип работы *AutoPtr* простой: локальный объект данного класса будет автоматически уничтожаться при выходе из блока, где он описан, и если ему до этого было поручено "следить" за каким-либо указателем, то *AutoPtr* освободит и его.

```
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;
   }
};
```

Дополнительно в классе *AutoPtr* реализован конструктор копирования (а точнее — перемещения, т.к. хозяином указателя становится текущий объект), что позволяет возвращать экземпляр *AutoPtr* вместе с контролируемым указателем из какой-либо функции.

Для проверки работоспособности *AutoPtr* опишем фиктивный класс *Dummy*.

```
class Dummy
{
   int x;
public:
   Dummy(int i) : x(i)
   {
      Print(__FUNCSIG__, " ", &this);
   }
   ...
   int value() const
   {
      return x;
   }
};
```

В скрипте, в функции *OnStart* введем переменную *AutoPtr<Dummy>* и получим для неё значение из функции *generator*. В самой функции *generator* также опишем объект *AutoPtr<Dummy>* и последовательно создадим и "привяжем" к нему два динамических объекта *Dummy* (чтобы проверить корректное освобождение памяти от "старого" объекта).

```
AutoPtr<Dummy> generator()
{
   AutoPtr<Dummy> ptr(new Dummy(1));
   // указатель на 1 будет освобожден при выполнении '='
   ptr = new Dummy(2);
   return ptr;
}

void OnStart()
{
   AutoPtr<Dummy> ptr = generator();
   Print(ptr[].value());             // 2
}
```

Поскольку во всех основных методах стоит вывод в журнал дескрипторов объектов (как самих *AutoPtr*, так и контролируемых указателей *ptr*), мы можем отследить все "превращения" указателей (для удобства все строки пронумерованы).

```
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
```

Отвлечемся ненадолго от шаблонов и подробно опишем работу утилиты, потому что подобный класс может пригодиться многим.

Сразу после запуска *OnStart* происходит вызов функции generator. Она должна вернуть значение для инициализации объекта *AutoPtr* в *OnStart*, и потому его конструктор пока не вызывается. В строке 02 создается объект AutoPtr#2097152 внутри функции *generator* и получает указатель на первый Dummy#3145728. Далее создается второй экземпляр Dummy#4194304 (строка 03), который заменяет в AutoPtr#2097152 прежнюю копию с дескриптором 3145728 (строка 04), причем прежняя копия удаляется (строка 05). В строке 06 создается временный объект AutoPtr#5242880 для возврата значения из *generator*, а локальный — удаляется (07). В строке 08 наконец используется конструктор копирования для объекта AutoPtr#1048576 в функции *OnStart*, и в него переносится указатель из временного объекта (который тут же удаляется в строке 09). Далее мы вызываем *Print* с содержимым указателя. По завершении *OnStart* автоматически срабатывает деструктор *AutoPtr* (11), в результате чего мы также удаляем рабочий объект *Dummy* (12).

Технология шаблонов делает класс *AutoPtr* параметризованным менеджером динамически распределяемых объектов. Но поскольку *AutoPtr* имеет поле *T *ptr*, он применим только для классов (точнее, указателей на объекты классов). Например, попытка инстанцировать шаблон для строки (*AutoPtr<string> s*) приведет к множеству ошибок в тексте шаблона, смысл которых в том, что тип *string* не поддерживает указатели.

Здесь это не является проблемой, поскольку назначение данного шаблона ограничено классами, однако для более универсальных шаблонов этот нюанс следует иметь в виду (см. врезку).

Указатели и ссылки

Обратите внимание, что конструкция *T ** не может встречаться в шаблонах, которые планируется применять, в том числе, для встроенных типов или структур. Дело в том, что указатели в MQL5 разрешены только для классов. Это не означает, что шаблон в принципе не может быть написан так, чтобы применяться и для встроенных, и для пользовательских типов, однако это может потребовать некоторых ухищрений. Вероятно, потребуется либо отказаться от части функционала, либо поступиться уровнем универсальности шаблона (сделать несколько шаблонов вместо одного, перегрузить функции и т.д.).

Наиболее прямолинейный способ "внедрения" в шаблон типа-указателя — включать модификатор '*' вместе с актуальным типом при создании экземпляра шаблона (то есть должно выполняться соответствие *T=Type**). Вместе с тем, некоторые функции (такие как *CheckPointer*), операторы (например, *delete*) и синтаксические конструкции (например, приведение типа *((T)variable)*), чувствительны к тому, являются ли их аргументы/операнды указателями или нет. В связи с этим, один и тот же текст шаблона не всегда синтаксически корректен одновременно и для указателей, и для значений простых типов.

Еще одно существенное различие в типах, которое следует иметь в виду: объекты передаются в методы только по ссылке, но литералы (константы) простых типов не могут передаваться по ссылке. Из-за этого наличие или отсутствие амперсанда может расцениваться компилятором, как ошибка, в зависимости от выведенного типа T. В качестве одного из "обходных маневров" можно, при необходимости, "оборачивать" константы-аргументы в объекты или переменные.

Другой прием связан с использованием шаблонных методов: он показан в следующем разделе.

Следует отметить, что объектно-ориентированные методики хорошо сочетаются с шаблонами. Поскольку указатель на базовый класс можно использовать для хранения объекта производного класса, *AutoPtr* применим для объектов любых классов-наследников *Dummy*.

В принципе, данный "гибридный" подход широко используется в классах-контейнерах (вектор, очередь, карта, список и т.д.), которые, как правило, являются шаблонными. Классы-контейнеры могут, в зависимости от реализации, накладывать на параметр шаблона дополнительные требования, в частности, чтобы подставляемый тип имел конструктор копирования и оператор присваивания (копирования).

В стандартной библиотеке MQL5, поставляемой вместе с MetaTrader 5, имеется множество готовых шаблонов из этой серии: *Stack.mqh*, *Queue.mqh*, *HashMap.mqh*, *LinkedList.mqh*, *RedBlackTree.mqh* и другие — все они находятся в каталоге MQL5/Include/Generic. Правда, они не обеспечивают контроль за динамическими объектами (указателями).

Мы рассмотрим собственный пример простого класса-контейнера в разделе Шаблоны методов.

3.3.7 Шаблоны методов

Шаблоном может быть не только объектный тип целиком, но и его метод в отдельности — простой или статический. Исключение составляют виртуальные методы: их нельзя делать шаблонами. Отсюда следует, что шаблонные методы нельзя описывать внутри интерфейсов. Вместе с тем, сами интерфейсы можно делать шаблонами, и виртуальные методы могут присутствовать в шаблонах классов.

Когда шаблон метода содержится в шаблоне класса/структуры, параметры обоих шаблонов должны отличаться. Если шаблонных методов несколько, их параметры никак не связаны и могут иметь одинаковые имена.

Шаблон метода описывается аналогично шаблону функции, но только в контексте класса, структуры или объединения (которые могут быть или не быть шаблонами).

```
[ template < typename T[, typename Ti ...] > ]
class имя_класса
{
   ...
   template < typename U [, typename Ui ...] >
   тип имя_метода(параметры_с_типами_T_и_U)
   {
   }
};
```

В параметрах, возвращаемом значении и теле метода могут использоваться типы T (общие для класса) и U (специфические для метода).

Экземпляр метода для конкретного сочетания параметров генерируется только при обращении к нему в коде программы.

В предыдущем разделе мы описали шаблонный класс *AutoPtr* для хранения и освобождения одного указателя. Когда однотипных указателей много, их удобно складывать в объект-контейнер. Создадим простой шаблон с подобным функционалом — класс *SimpleArray* (*SimpleArray.mqh*). Чтобы не дублировать функционал по контролю за освобождением динамической памяти, заложим в контракт класса, что он предназначен для хранения значений и объектов, но не указателей. Для хранения указателей будем помещать их в объекты *AutoPtr*, а уже те — в контейнер.

Это имеет еще один положительный эффект: поскольку объект *AutoPtr* маленький, его легко копировать (без расхода ресурсов), что часто происходит при обмене данными между функциями. А объекты тех прикладных классов, на которые *AutoPtr* указывает, могут быть большими, и в них даже не обязательно реализовывать свой конструктор копирования.

Конечно, дешевле всего возвращать из функций сами указатели, но тогда потребуется заново изобретать средства контроля за освобождением памяти. Поэтому проще воспользоваться готовым решением в виде *AutoPtr*.

Для объектов внутри контейнера заведем массив *data* шаблонизированного типа T.

```
template<typename T>
class SimpleArray
{
protected:
   T data[];
   ...
```

Поскольку одна из основных операций для контейнера — добавление элемента, предусмотрим вспомогательную функцию для расширения массива.

```
   int expand()
   {
      const int n = ArraySize(data);
      ArrayResize(data, n + 1);
      return n;
   }
```

Непосредственно добавление элементов будем делать через перегруженный оператор '<<'. Он использует общий параметр T шаблона.

```
public:
   SimpleArray *operator<<(const T &r)
   {
      data[expand()] = (T)r;
      return &this;
   }
```

Данный вариант принимает значение по ссылке, то есть переменную или объект. Следует пока обратить на это внимание, а почему это важно — станет ясно через пару моментов.

Чтение элементов осуществим путем перегрузки оператора '[]' (он имеет наивысший приоритет и потому не потребует использовать круглые скобки в выражениях).

```
   T operator[](int i) const
   {
      return data[i];
   }
```

Сперва убедимся в работоспособности класса на примере структуры.

```
struct Properties
{
   int x;
   string s;
};
```

Для этого опишем в функции *OnStart* контейнер для структуры и поместим в него один объект (*TemplatesSimpleArray.mq5*).

```
void OnStart()
{
   SimpleArray<Properties> arrayStructs;
   Properties prop = {12345, "abc"};
   arrayStructs << prop;
   Print(arrayStructs[0].x, " ", arrayStructs[0].s);
   ...
}
```

Отладочный вывод в журнал позволяет убедиться, что структура находится в контейнере.

Теперь попробуем сохранить в контейнер несколько чисел.

```
   SimpleArray<double> arrayNumbers;
   arrayNumbers << 1.0 << 2.0 << 3.0;
```

К сожалению, мы получим ошибки "параметр передается по ссылке, требуется переменная" ("parameter passed as reference, variable expected"), которые возникают как раз в перегруженном операторе '<<'.

Нам требуется перегрузка с передачей параметра по значению. Однако мы не можем просто написать похожий метод, который отличается отсутствием *const* и '&':

```
   SimpleArray *operator<<(T r)
   {
      data[expand()] = (T)r;
      return &this;
   }
```

Если так сделать, новый вариант приведет к некомпилируемости шаблона для объектных типов: ведь объекты нужно передавать только по ссылке. Даже если функция не используется для объектов, она все равно присутствует в классе. Поэтому мы определим новый метод как шаблон с собственным параметром.

```
template<typename T>
class SimpleArray
{
   ...
   template<typename U>
   SimpleArray *operator<<(U u)
   {
      data[expand()] = (T)u;
      return &this;
   }
```

Он появится в классе только в том случае, если в оператор '<<' передается что-то по значению, а значит это точно не объект. Правда, мы не можем гарантировать, что T и U совпадают, поэтому выполняется явное приведение типа *(T)u*. Для встроенных типов (если два типа не совпадут) в некоторых сочетаниях возможна конверсия с потерей точности, но код точно скомпилируется. Единственное исключение — запрет на преобразование строки в логический тип, но вряд ли контейнер станут использовать для массива *bool*, поэтому данное ограничение не существенно. Желающие могут решить эту проблему.

С новым шаблонным методом контейнер *SimpleArray<double>* работает как ожидалось и не конфликтует с *SimpleArray<Properties>*, так как у этих двух экземпляров шаблонов есть отличия в сгенерированном исходном коде.

Наконец, проверим контейнер с объектами *AutoPtr*. Для этого подготовим простой класс *Dummy*, который будет "поставлять" объекты для указателей внутри *AutoPtr*.

```
class Dummy
{
   int x;
public:
   Dummy(int i) : x(i) { }
   int value() const
   {
      return x;
   }
};
```

В функции *OnStart* создадим контейнер *SimpleArray<AutoPtr<Dummy>>* и заполним его.

```
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());
}
```

Напомним, что в *AutoPtr* оператор '[]' используется для возврата хранимого указателя, поэтому запись *arrayObjects[0][]* означает: вернуть 0-й элемент массива *data* в *SimpleArray*, то есть объект *AutoPtr*, и затем к тому применяется вторая пара квадратных скобок, что в результате дает указатель Dummy*. Далее мы можем работать со всеми свойствами этого объекта: в данном случае, мы извлекаем текущее значение поля x.

Поскольку в *Dummy* нет конструктора копирования, нельзя использовать контейнер для хранения этих объектов напрямую, без *AutoPtr*.

```
// ОШИБКА:
// object of 'Dummy' cannot be returned,
// copy constructor 'Dummy::Dummy(const Dummy &)' not found
SimpleArray<Dummy> bad;
```

Но находчивый пользователь может догадаться, как это обойти.

```
SimpleArray<Dummy*> bad;
bad << new Dummy(0);
```

Такой код будет компилироваться и работать. Однако в этом "решении" содержится проблема: *SimpleArray* не умеет контролировать указатели, и потому при выходе из программы обнаружится утечка памяти.

```
1 undeleted objects left
1 object of type Dummy left
24 bytes of leaked memory
```

Мы, как разработчики *SimpleArray*, обязаны прикрыть эту лазейку. Для этого добавим в класс еще один шаблонный метод с перегрузкой оператора '<<' — на этот раз для указателей. Поскольку это шаблон, он также включается в результирующий исходный код только "по требованию": когда программист пытается использовать данную перегрузку, то есть записать в контейнер указатель. В остальных случаях метод игнорируется.

```
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;
   }
```

Данная специализация вызывает ошибку компиляции ("object pointer expected") при создании экземпляра шаблона с типом указателя. Тем самым мы сообщаем пользователю, что такой режим не поддерживается.

```
SimpleArray<Dummy*> bad; // ОШИБКА генерируется в SimpleArray.mqh
```

Кроме того, она выполняет еще одно защитное действие. Если в клиентском классе всё же будет конструктор копирования, то сохранение динамически распределенных объектов в контейнере перестанет приводить к утечке памяти: копия объекта по переданному указателю *P *p* остается в контейнере, а оригинал — удаляется. При уничтожении контейнера в конце функции *OnStart*, его внутренний массив *data* автоматически вызовет деструкторы для своих элементов.

```
void OnStart()
{
   ...
   SimpleArray<Dummy> good;
   good << new Dummy(0);
} // SimpleArray "чистит" свои элементы
 // нет забытых объектов в памяти
```

Шаблоны методов и "простые" методы могут быть определены вне основного блока класса (или шаблона класса), по аналогии с тем, что мы рассматривали в разделе Разнесение объявления и определения класса. При этом они все предваряются заголовком шаблона (*TemplatesExtended.mq5*):

```
template<typename T>
class ClassType
{
   ClassType() // закрытый конструктор
   {
      s = &this;
   }
   static ClassType *s; // указатель на объект (если был create)
public:
   static ClassType *create() // создание (только при первом вызове)
   {
      static ClassType single; // паттерн "одиночка" для каждого T
      return single;
   }

   static ClassType *check() // проверка указателя без создания
   {
      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;
```

Здесь также показана инициализация шаблонизированной статической переменной, обозначающий паттерн проектирования "одиночка" (singleton).

В функции *OnStart* создадим экземпляр шаблона и протестируем его:

```
void OnStart()
{
   ClassType<string> *object = ClassType<string>::create();
   double d = 5.0;
   object.method(d);
   // ВЫВОД:
   // void ClassType<string>::method<double>(const double&) string double

   Print(ClassType<string>::check()); // 1048576 (пример id экземпляра)
   Print(ClassType<long>::check());   // 0 (нет экземпляра для T=long)
}
```

3.3.8 Вложенные шаблоны

Шаблоны могут быть вложенными в классы/структуры или в другие шаблоны классов/структур. Это же касается и объединений.

В разделе, посвященном Объединениям мы познакомились с возможностью "конвертировать" значения *long* в *double* и обратно без потери точности.

Теперь мы можем использовать шаблоны для написания универсального "конвертера" (*TemplatesConverter.mq5*). Шаблонный класс *Converter* имеет два параметра T1 и T2, обозначающих типы, между которыми будет производиться конвертация. Для записи значения по правилам одного типа и чтения по правилам другого нам снова потребуется объединение. Мы сделаем его также шаблоном (*DataOverlay*) с параметрами U1 и U2, и определим внутри класса.

Класс обеспечивает удобное преобразование за счет перегрузки операторов [], в реализации которых и производится запись и чтение полей объединения.

```
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;
   }
};
```

Объединение используется для описания поля *DataOverlay<T1,T2> data* внутри класса. Мы могли бы напрямую использовать T1 и T2 в *DataOverlay* и не делать это объединение шаблоном. Но для демонстрации самого технического приема, параметры внешнего шаблона передаются во внутренний шаблон при генерации поля *data*. Внутри *DataOverlay* та же пара типов будет известна как U1 и U2 (в дополнение к T1 и T2).

Проверим шаблон в действии.

```
#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 Специализация шаблонов, которой нет

В некоторых случаях бывает необходимо предоставить реализацию шаблона для конкретного типа (или набора типов), так чтобы она отличалась от общего варианта. Например, обычно имеет смысл подготовить особую версию функции swap (обмена значениями) для указателей или массивов. В подобных случаях C++ позволяет сделать специализацию шаблона, то есть определить его версию, в которой обобщенный параметр типа T заменен на требуемый конкретный тип.

При специализации шаблонов функций и методов должны быть заданы конкретные типы для всех параметров. Это называется полной специализацией.

В случае шаблонов объектных типов C++ специализация может быть не только полной, но и частичной: при ней уточняется тип лишь некоторых из параметров (а остальные будут выводиться или указываться при создании экземпляра шаблона). Частичных специализаций может быть несколько: единственное условие для этого — каждая специализация должна описывать уникальное сочетание типов.

К сожалению, в MQL5 нет специализации в полном смысле этого слова.

Специализация шаблонной функции ничем не отличается от перегрузки. Например, при наличии следующего шаблона *func*:

```
template<typename T>
void func(T t) { ... }
```

допускается предоставить его особую реализацию для заданного типа (такого как *string*) в одной из форм:

```
// явная специализация
template<>
void func(string t) { ... }
```

или:

```
// обычная перегрузка
void func(string t) { ... }
```

Должна быть выбрана только одна из форм. Иначе получим ошибку компиляции "'func' - функция уже определена и имеет тело" ("'func' - function already defined and has body").

Что касается специализации классов, то неким эквивалентом их частичной специализации можно считать наследование от шаблонов с указанием конкретных типов для части шаблонных параметров. В производном классе методы шаблона можно переопределить.

В следующем примере (*TemplatesExtended.mq5*) показано несколько вариантов использования параметров шаблонов в качестве родительских типов, в том числе и случаи, когда один из них прописан конкретным.

```
#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>                 // полная "специализация"
class Derived1 : public Base1<Base> // задан 1 из 1 параметра
{
public:
   Derived1() { RTTI; }
};

template<typename T,typename E>
class Base2 : public T
{
public:
   Base2() { RTTI; }
};

template<typename T>                       // частичная "специализация"
class Derived2 : public Base2<T,string> // задан 1 из 2 параметров
{
public:
   Derived2() { RTTI; }
};
```

Инстанцирование объекта по шаблону обеспечим с помощью переменной:

```
Derived2<Derived1<Base>> derived2;
```

Отладочный вывод типов в журнал с помощью макроса RTTI дает следующий результат:

```
Base
Derived<Base>
Base1<Base>
Derived1<Base>
Base2<Derived1<Base>,string>
Derived2<Derived1<Base>>
```

При разработке библиотек, которые поставляются в виде закрытого двоичного кода, необходимо обеспечить явное создание экземпляров шаблонов для всех типов, с которыми предполагается работа будущими пользователями библиотеки. Сделать это можно, вызвав явным образом шаблоны функций и создав объекты с указанием параметров-типов в какой-либо вспомогательной функции, например, привязанной к инициализации глобальной переменной.

Часть 4. Общеупотребительные MQL5 API

В предыдущих частях книги мы изучили основные понятия, синтаксис и правила использования языковых конструкций MQL5. Но это лишь фундамент для написания реальных программ, отвечающих требованиям трейдера, таким как аналитическая обработка данных и автоматическая торговля. Решение подобных задач было бы невозможно без широкого набора встроенных функций и средств взаимодействия с терминалом MetaTrader 5, составляющих MQL5 API.

В этой части мы приступим к освоению MQL5 API и продолжим это делать до конца книги, постепенно знакомясь со всеми специализированными подсистемами.

Перечень технологий и возможностей, предоставляемых любой MQL-программе ядром (средой исполнения MQL-программ внутри терминала), очень велик. Поэтому начать имеет смысл с наиболее простых вещей, которые могут пригодиться в большинстве программ. В частности, здесь мы рассмотрим функции для работы с массивами, строками, файлами, преобразования данных, взаимодействия с пользователем, математические функции и контроль за состоянием среды.

Ранее мы научились описывать в MQL5 свои собственные функции и вызывать их. Встроенные функции MQL5 API доступны из исходного кода, как говорится, "сразу из коробки", то есть без всякого предварительного описания.

Важно отметить, что в отличие от C++, не требуется никаких дополнительных директив препроцессора, чтобы подключить к программе специфический набор встроенных функций. Имена всех функций MQL5 API присутствуют в глобальном контексте (пространстве имен), всегда и безусловно.

С одной стороны, это удобно, но с другой — требует помнить о возможном конфликте имен. Если вы случайно попробуете использовать одно из имен встроенных функций, оно перекроет стандартную реализацию, что может привести к неожиданным последствиям: в лучшем случае Вы получите ошибку компилятора о неоднозначной перегрузке, а в худшем — все привычные вызовы будут переадресованы "новой" реализации, без всяких предупреждений.

В принципе, аналогичные имена можно использовать и в других контекстах, например, в качестве имени метода класса или в выделенном (пользовательском) пространстве имен. В таких случаях, вызвать глобальную функцию можно с помощью оператора разрешения контекста: мы рассматривали эту ситуацию в разделе Вложенные типы, пространства имен и оператор контекста '::'.

Программирование на MQL5 для трейдеров — исходные коды из книги: Часть 4

Примеры из книги также доступны в публичном проекте \MQL5\Shared Projects\MQL5Book

4.1 Преобразование данных встроенных типов

В программах часто возникает необходимость оперировать данными разных типов. В разделе Приведение типов мы уже сталкивались с механизмами явного и неявного приведения встроенных типов. Они предоставляют универсальные способы конвертации, которые не всегда подходят по тем или иным причинам. Для того чтобы программист мог управлять

преобразованием данных разных типов друг в друга и настраивать его результат, MQL5 API включает набор функций для конвертации.

Наиболее заметное место среди них занимают функции конвертации различных типов в строки и обратно. В частности, сюда входят преобразования для чисел, даты и времени, цветов, структур и перечислений. Для некоторых типов имеются дополнительные специфические операции.

В этом разделе рассматриваются различные методы преобразования данных, обеспечивая программистов необходимыми инструментами для работы с разнообразными типами данных в торговых роботах. Включает в себя следующие подразделы:

Числа в строки и обратно:

- Этот подраздел исследует методы преобразования числовых значений в строки и обратно. Он покрывает важные аспекты, такие как форматирование чисел и обработка различных систем исчисления.

Нормализация чисел double:

- Нормализация чисел double — это важный шаг при работе с финансовыми данными. Раздел рассматривает методы нормализации, способы избежать потери точности и обработки значений с плавающей точкой.

Дата и время:

- Преобразование даты и времени играет ключевую роль в торговых стратегиях. Здесь обсуждаются методы работы с датами, временными интервалами и специальными типами данных, такими как datetime.

Цвет:

- В MQL5 цвета представлены специальным типом данных. Подраздел рассматривает преобразование цветовых значений, их представление и использование в графических элементах торговых роботов.

Структуры:

- Преобразование данных в структурах — важная тема при работе с сложными структурированными данными. Здесь изучаются методы взаимодействия с структурами и их элементами.

Перечисления:

- Перечисления предоставляют именованные константы и улучшают читаемость кода. Этот подраздел рассматривает, как преобразовывать значения перечислений и эффективно использовать их в программе.

Тип complex:

- Тип complex предназначен для работы с комплексными числами. Раздел рассматривает методы преобразования и использования комплексных чисел.

В данной главе мы изучим все функции данного толка.

4.1.1 Числа в строки и обратно

Как мы знаем, для преобразования чисел в строки и обратно — строк в числа, можно пользоваться оператором явного приведения типов. Например, для типов *double* и *string*, это условно выглядит так:

```
double number = (double)text;
string text = (string)number;
```

Разумеется, возможно приведение к другим числовым типам: *float*, *long*, *int* и т.д.

Обратите внимание, что вариант приведения к вещественному типу *(float)* обеспечивает меньшее количество значащих цифр, что в некоторых задачах может считаться преимуществом, поскольку дает более компактное и легко читаемое представление.

Более того, строго говоря, приведение не обязательно, так как даже в отсутствии оператора явного приведения типа компилятор сделает приведение в неявном виде, хотя и выдаст об этом предупреждение (в связи с чем рекомендуется всегда делать приведения типов явными).

В дополнение к этому в MQL5 API имеется несколько полезных функций. После их описания приведен общий пример.

double StringToDouble(string text)

Функция *StringToDouble* преобразует строку в число типа *double*.

Она является полным аналогом приведения типа к (*double*). Её имеет смысл использовать только в целях обратной совместимости с унаследованными исходными кодами. Приведения типа — более предпочтительный способ, как более краткий и в рамках синтаксиса языка.

Процесс конвертации подразумевает, что в строке находится последовательность символов, отвечающих правилам записи литералов числовых типов (как вещественных, так и целых). В частности, строка может начинаться со знака '+' или '-', и цифры, а также состоять далее из цифр.

Для случая вещественных чисел допускается наличие одного символа точки '.', отделяющей дробную часть, и опциональной показательной степени в формате: символ 'e' или 'E', за которым следует последовательность цифр для степени (она также может предваряться знаком '+' или '-').

Для целых чисел поддерживается шестнадцатеричная запись, то есть после префикса "0x" могут следовать не только десятичные цифры, но и 'A', 'B', 'C', 'D', 'E', 'F' (в любом регистре).

Когда в строке встречается любой символ, отличный от ожидаемых (например, буква, знак препинания, вторая по счету точка или промежуточный пробел) преобразование заканчивается. При этом, если до данной позиции были разрешенные символы, они интерпретируются как число, а если нет — результатом будет 0.

Начальные пустотелые символы (пробелы, табуляции, переводы строк) пропускаются и не влияют на конвертацию. Если после них идут цифры и прочие отвечающие правилам символы, число будет получено корректно.

В следующей таблице приведены некоторые примеры правильных преобразований с пояснениями.

string	double	Результат
"123.45"	123.45	Одна десятичная точка
"\t 123"	123.0	Пробельные символы в начале игнорируются
"-12345"	-12345.0	Число со знаком
"123e-5"	0.00123	Научный формат с показателем степени
"0x12345"	74565.0	Шестнадцатеричная запись

В следующей таблице приведены примеры неправильных преобразований.

string	double	Результат
"x12345"	0.0	Начинается с неразрешенного символа (буквы)
"123x45"	123.0	Буква после 123 прерывает конвертацию
" 12 3"	12.0	Пробел после 12 прерывает конвертацию
"123.4.5"	123.4	Вторая десятичная точка после 123.4 прерывает конвертацию
"1,234.50"	1.0	Запятая после 1 прерывает конвертацию
"-+12345"	0.0	Два знака — "перебор"

string DoubleToString(double number, int digits = 8)

Функция *DoubleToString* преобразует число в строку с заданной точностью (количеством цифр от −16 до 16).

Она выполняет работу, схожую с приведением числа к (*string*), но позволяет выбрать с помощью второго параметра точность представления числа в результирующей строке.

Оператор *(string)*, примененный к *double*, выводит 16 значащих цифр (совокупно в мантиссе и дробной части). Полного эквивалента этому добиться с помощью функции нельзя.

Когда параметр *digits* больше или равен 0, он означает количество знаков после "запятой". При этом количество знаков до "запятой" определяется самим числом (тем, насколько оно большое), и если суммарное количество знаков в мантиссе и указанное в *digits* окажется больше 16, то младшие значащие разряды будут содержать "мусор" (из-за особенностей хранения вещественных чисел). Напомним, что 16 знаков — средняя максимальная точность для типа *double*, то есть установка *digits* в 16 (максимум) обеспечит точное представление только значениям меньше 10.

Когда параметр *digits* меньше 0, он задает количество значащих цифр, и число выводится в научном формате с экспонентой. С точки зрения точности (но не формата записи), установка *digits=-16* в функции близка по результату к приведению *(string)*.

Функцию, как правило, используют для однообразного форматирования наборов данных (включая выравнивание по правому краю столбца некоторой таблицы), причем данные эти имеют, в силу прикладной специфики, равную точность представления дробной части

(например, количество знаков после "запятой" в цене финансового инструмента или в размере лота).

> Следует иметь в виду, что в ходе математических вычислений могут возникать ошибки, приводящие к тому, что результат не является валидным числом, хотя и имеет тип *double* (или *float*). Например, переменная может содержать результат вычисления квадратного корня из отрицательного числа.
>
> Такие значения называются "не числами" (Not a Number, NaN), и отображаются в случае приведения к *(string)* как краткая подсказка типа ошибки, например, -nan(ind) (ind — не определено), nan(inf) (inf — бесконечность). В случае использования функции *DoubleToString*, вы получите некое большое число, не соответствующее действительности.
>
> Особенно важно, что все последующие вычисления с NaN будут также давать NaN. Для проверки таких значений существует функция *MathIsValidNumber*.

long StringToInteger(string text)

Функция преобразует строку в число типа *long*. Обратите внимание, что тип результата именно *long*, а не *int* (несмотря на название) и не *ulong*.

Альтернативный способ заключается в приведении типа с помощью оператора *(long)*. Более того, для приведения можно использовать любой другой целочисленный тип на выбор: *(int)*, *(uint)*, *(ulong)* и т.д.

Правила конвертации аналогичны типу *double*, но исключают символ точки и показатель степени из числа разрешенных символов.

string IntegerToString(long number, int length = 0, ushort filling = ' ')

Функция *IntegerToString* преобразует целое число типа *long* в строку указанной длины. Если представление числа занимает меньше символов, оно дополняется слева символом *filling* (по умолчанию, пробелом). В противном случае число выводится целиком, без ограничения. Вызов функции с параметрами по умолчанию эквивалентен приведению к *(string)*.

Разумеется, целые типы меньшего размера (например, *int*, *short*) будут обработаны функцией без проблем.

Примеры использования всех вышеописанных функций приведены в скрипте *ConversionNumbers.mq5*.

```
void OnStart()
{
   const string text = "123.4567890123456789";
   const string message = "-123e-5 buckazoid";
   const double number = 123.4567890123456789;
   const double exponent = 1.234567890123456789e-5;

   // приведение типов
   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

   // конверсии функциями
   Print(StringToDouble(text)); // 123.4567890123457
   Print(StringToDouble(message)); // -0.00123

   // по умолчанию, 8 цифр в дробной части
   Print(DoubleToString(number)); // 123.45678901

   // настраиваемая точность
   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
   //                                    2 последних цифры неточные!
   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 Нормализация чисел double

MQL5 API предоставляет функцию для округления чисел с плавающей точкой до указанной точности (количества значащих цифр в дробной части).

double NormalizeDouble(double number, int digits)

Округление требуется в торговых алгоритмах для задания объема и цен в приказах. Округление производится по стандартным правилам: последний видимый разряд увеличивается на 1, если следующий (отбрасываемый) разряд больше или равен 5.

Допустимые значения параметра *digits*: от 0 до 8.

Примеры использования функции доступны в файле *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
   ...
```

Из-за того, что любое вещественное число имеет ограниченную точность внутреннего представления, даже будучи нормализованным оно может отображаться приближенно:

```
   ...
   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
}
```

Это нормальное, неизбежное явление. Для более компактного форматирования используйте функции *DoubleToString*, *StringFormat* или промежуточное приведение к *(float)*.

Для округления числа до ближайшего целого вверх или вниз используйте функции *MathRound*, *MathCeil*, *MathFloor* (см. раздел Функции округления).

4.1.3 Дата и время

Значения типа *datetime*, предназначенные для хранения даты и/или времени, как правило, подвергаются нескольким видам конвертации:

- в строки и обратно для отображения пользователю и чтения из внешних источников данных;
- в специальные структуры *MqlDateTime* (см. ниже) для работы с отдельными компонентами даты и времени;
- в количество секунд, прошедшее с 01.01.1970, что соответствует внутреннему представлению *datetime*, и эквивалентно целому типу *long*.

Для последнего пункта используйте приведение *datetime* к *(long)* или обратно *long* к *(datetime)*, но учтите, что поддерживаемый диапазон дат — от 1-го января 1970 года (значение 0) до 31 декабря 3000 года (32535215999 секунд).

Для первых двух пунктов MQL5 API предоставляет нижеследующие функции.

string TimeToString(datetime value, int mode = TIME_DATE | TIME_MINUTES)

Функция *TimeToString* преобразует значение типа *datetime* в строку с компонентами даты и времени, в соответствии с параметром *mode*, в котором можно задать произвольное сочетание флагов:

- TIME_DATE — дата в формате "YYYY.MM.DD";
- TIME_MINUTES — время в формате "hh:mm", то есть с часами и минутами;
- TIME_SECONDS — время в формате "hh:mm:ss", то есть с часами, минутами и секундами.

Для полного вывода даты и времени можно задать *mode* равный TIME_DATE | TIME_SECONDS (вариант TIME_DATE | TIME_MINUTES | TIME_SECONDS тоже сработает, но является избыточным). Это эквивалентно приведению значения типа *datetime* к *(string)*.

Примеры использования сведены в файл *ConversionTime.mq5*.

```
#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));
}
```

Скрипт выведет в журнал:

```
(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)

Функция *StringToTime* преобразует строку, содержащую дату и/или время, в значение типа *datetime*. Строка может содержать только дату, только время или дату и время вместе.

Для дат распознаются следующие форматы:

- "YYYY.MM.DD"
- "YYYYMMDD"
- "YYYY/MM/DD"
- "YYYY-MM-DD"
- "DD.MM.YYYY"
- "DD/MM/YYYY"
- "DD-MM-YYYY"

Для времени поддерживаются следующие форматы:

- "hh:mm"
- "hh:mm:ss"

- "hhmmss"

Между датой и временем должен быть как минимум один пробел.

Если в строке присутствует только время, в результат будет подставлена текущая дата. Если в строке присутствует только дата, время будет равно 00:00:00.

Если поддерживаемый синтаксис в строке нарушен, результатом будет текущая дата.

Примеры использования функции приведены в скрипте *ConversionTime.mq5*.

```
void OnStart()
{
   string timeonly = "21:01";    // только время
   PRT(timeonly);
   PRT((datetime)timeonly);
   PRT(StringToTime(timeonly));

   string date = "2000-10-10";  // только дата
   PRT((datetime)date);
   PRT(StringToTime(date));
   PRT((long)(datetime)date);
   long seconds = 60;
   PRT((datetime)seconds); // 1 минута с начала 1970

   string ddmmyy = "15/01/2012 01:02:03"; // дата и время, причем дата в
   PRT(StringToTime(ddmmyy));             // "прямом" порядке, все равно ok

   string wrong = "January 2-nd";
   PRT(StringToTime(wrong));
}
```

В журнале мы увидим примерно следующее (####.##.## — текущая дата запуска скрипта):

```
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
```

Помимо *StringToTime* для преобразования строк в дату и время можно пользоваться оператором приведения к типу *(datetime)*. Однако плюс функции в том, что при обнаружении некорректной исходной строки она устанавливает внутреннюю переменную с кодом ошибки *_LastError* (доступную также через функцию *GetLastError*). В зависимости от того, какая часть строки содержит неинтерпретируемые данные, код ошибки может быть ERR_WRONG_STRING_DATE (5031), ERR_WRONG_STRING_TIME (5032) или другой из списка относящихся к получению даты и времени из строки.

bool TimeToStruct(datetime value, MqlDateTime &struct)

Для анализа компонентов даты и времени по отдельности MQL5 API предоставляет функцию *TimeToStruct*, которая преобразует значение типа *datetime* в структуру *MqlDateTime*:

```
struct MqlDateTime
{
   int year;           // год
   int mon;            // месяц
   int day;            // день
   int hour;           // час
   int min;            // минуты
   int sec;            // секунды
   int day_of_week;    // день недели
   int day_of_year;    // номер дня в году (1-е января имеет номер 0)
};
```

Дни недели нумеруются на американский манер: 0 — воскресенье, 1 — понедельник, и так далее вплоть до 6 — суббота. Для их идентификации существует встроенное перечисление ENUM_DAY_OF_WEEK.

Функция возвращает *true* в случае успеха, и *false* — в случае ошибки, в частности, если передана некорректная дата.

Проверим работу функции с помощью скрипта *ConversionTimeStruct.mq5*. Для этого создадим массив *time* типа *datetime* с тестовыми значениями, и в цикле вызовем *TimeToStruct* для каждого из них.

Результаты будем складывать в массив структур *MqlDateTime mdt[]*. Предварительно мы его проинициализируем нулями, но поскольку встроенная функция *ArrayInitialize* не умеет обрабатывать структуры, нам придется написать её перегрузку (в будущем мы познакомимся с более простым способом заполнить массив нулями: в разделе Обнуление объектов и массивов будет представлена функция *ZeroMemory*).

```
int ArrayInitialize(MqlDateTime &mdt[], MqlDateTime &init)
{
   const int n = ArraySize(mdt);
   for(int i = 0; i < n; ++i)
   {
      mdt[i] = init;
   }
   return n;
}
```

После процесса мы выведем массив структур в журнал с помощью встроенной функции *ArrayPrint* — это самый простой способ красиво отформатировать данные (его можно использовать даже если структура одна: достаточно положить её в массив размером 1).

```
void OnStart()
{
   // заполним массив с тестами
   datetime time[] =
   {
      D'2021.01.28 23:00:15', // корректное значение datetime
      D'3000.12.31 23:59:59', // максимальная поддерживаемая дата и время
      LONG_MAX // неверная дата: вызовет ошибку ERR_INVALID_DATETIME (4010)
   };

   // вычислим размер массива во время компиляции
   const int n = sizeof(time) / sizeof(datetime);

   MqlDateTime null = {}; // образец с нулями
   MqlDateTime mdt[];

   // выделим память под массив структур с результатами
   ArrayResize(mdt, n);

   // вызовем нашу перегрузку ArrayInitialize
   ArrayInitialize(mdt, null);

   // запустим тесты
   for(int i = 0; i < n; ++i)
   {
      PRT(time[i]); // выводим исходные данные

      if(!TimeToStruct(time[i], mdt[i])) // при ошибке выводим её код
      {
         Print("error: ", _LastError);
         mdt[i].year = _LastError;
      }
   }

   // выведем в лог результаты
   ArrayPrint(mdt);
   ...
}
```

В результате получим следующие строки в журнале:

```
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
```

Можно убедиться, что все поля получили соответствующие значения. Для некорректных исходных дат мы сохраняем код ошибки в поле *year* (в данном случае, такая ошибка одна: 4010, ERR_INVALID_DATETIME).

Напомним, что для максимального значения даты в MQL5 заведена константа DATETIME_MAX, равная целочисленному значению 0x793406fff, что соответствует 23:59:59 31 декабря 3000.

Наиболее часто встречающаяся задача, которую решают при помощи функции *TimeToStruct*, — это получение значения конкретной компоненты даты/времени. Поэтому имеет смысл подготовить вспомогательный заголовочный файл (*MQL5Book/DateTime.mqh*) с готовым вариантом реализации. В файле имеется класс *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;
```

К классу прилагается несколько макросов, упрощающих вызов его методов.

```
#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
```

В классе есть поле *mdtstruct* типа структуры *MqlDateTime*. Именно оно используется во всех внутренних преобразованиях. Поля структуры читаются с помощью методов-"геттеров": для каждого поля выделен соответствующий метод.

Внутри класса определен единственный статический экземпляр *_DateTime* (одного объекта достаточно, потому что все MQL-программы однопоточные). Конструктор является закрытым, поэтому создать другие объекты *DateTime* не удастся.

С помощью макросов можно удобно выделить из *datetime*, например, год (*TimeYear(T)*), месяц (*TimeMonth(T)*), число (*TimeDay(T)*) или день недели (*TimeDayOfWeek(T)*).

Если из одного значения *datetime* необходимо выделить несколько полей, то более экономичный способ предполагает во всех вызовах кроме первого использовать аналогичные макросы без параметра и начинающиеся с символа подчеркивания: они читают нужное поле из структуры без повторной установки даты/времени и вызова функции *TimeToStruct*. Например:

```
// используем класс DateTime из MQL5Book/DateTime.mqh:
// сначала получим день недели для заданного значения datetime
PRT(EnumToString(TimeDayOfWeek(time[0])));
// затем прочитаем год, месяц и число для того же значения
PRT(_TimeYear());
PRT(_TimeMonth());
PRT(_TimeDay());
```

В журнале должны появиться такие строки.

```
EnumToString(DateTime::_DateTime.assign(time[0]).__TimeDayOfWeek())=THURSDAY
DateTime::_DateTime.__TimeYear()=2021
DateTime::_DateTime.__TimeMonth()=1
DateTime::_DateTime.__TimeDay()=28
```

Встроенная функция *EnumToString* преобразует элемент любого перечисления в строку и будет описана в отдельном разделе.

datetime StructToTime(MqlDateTime &struct)

Функция *StructToTime* выполняет конвертацию из структуры *MqlDateTime* (см. выше описание функции *TimeToStruct*), содержащей компоненты даты и времени, в значение типа *datetime*. Поля *day_of_week* и *day_of_year* в процессе не участвуют.

Если состояние остальных полей некорректное (соответствует несуществующей или неподдерживаемой дате), функция может вернуть, в зависимости от серьезности проблемы, либо некоторое исправленное значение, либо значение WRONG_VALUE (-1 в представлении типа *long*). Поэтому следует проверять наличие ошибки по состоянию глобальной переменной *_LastError*. Успешная конвертация заканчивается с кодом 0. Перед конвертацией следует сбросить возможное ошибочное состояние в *_LastError* (сохранившееся как артефакт исполнения каких-то предыдущих инструкций) с помощью функции *ResetLastError*.

Проверка функции *StructToTime* также приведена в скрипте *ConversionTimeStruct.mq5*. Массив структур *parts* конвертируется в *datetime* в цикле.

```
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("");

// преобразуем все элементы в цикле
for(int i = 0; i < sizeof(parts) / sizeof(MqlDateTime); ++i)
{
   ResetLastError();
   datetime result = StructToTime(parts[i]);
   Print("[", i, "] ", (long)result, " ", result, " ", _LastError);
}
```

Для каждого элемента выводится получившееся значение и код ошибки.

```
    [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
```

Обратите внимание, что некоторые корректировки функция выполняет, не взводя флаг ошибки. Так, в элементе под номером 2 мы передавали в функцию 30 февраля 2021 года, которое было преобразовано во 2-е марта 2021, и *_LastError = 0*.

4.1.4 Цвет

Для работы с цветом в MQL5 API существует 3 встроенных функции: две из них — для преобразования типа *color* в строку и обратно, и одна — для получения специального представления цвета с прозрачностью (ARGB).

string ColorToString(color value, bool showName = false)

Функция *ColorToString* преобразует переданное значение цвета *value* в строку вида "R,G,B" (где R, G, B — это числа от 0 до 255, соответствующие интенсивности красной, зеленой, синей составляющей в цвете) или в название цвета из перечня предопределенных web-цветов, если параметр *showName* равен *true*. Название цвета возвращается только в том случае, если значение цвета точно соответствует одному из web-набора.

Примеры использования функции приведены в скрипте *ConversionColor.mq5*.

```
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 (нет имени для этого цвета)
   Print(ColorToString(0x34AB6821, true)); // 33,104,171 (0x21,0x68,0xAB)
}
```

color StringToColor(string text)

Функция *StringToColor* преобразует строку вида "R,G,B" или строку, содержащую наименование стандартного web-цвета, в значение типа *color*. Если в строке не содержится правильно отформатированной тройки чисел или названия цвета, функция вернет 0 (*clrBlack*).

Примеры можно увидеть в скрипте *ConversionColor.mq5*.

```
void OnStart()
{
   Print(StringToColor("0,0,255")); // clrBlue
   Print(StringToColor("clrBlue")); // clrBlue
   Print(StringToColor("Blue"));    // clrBlack (нет цвета с таким именем)
   // лишний текст игнорируется
   Print(StringToColor("255,255,255 more text"));      // clrWhite
   Print(StringToColor("This is color: 128,128,128")); // clrGray
}
```

uint ColorToARGB(color value, uchar alpha = 255)

Функция *ColorToARGB* преобразует значение типа *color* и однобайтовую величину *alpha* (задающую прозрачность) в ARGB-представление цвета (значение типа *uint*). ARGB-формат цвета используется при создании графических ресурсов и рисовании текста на графиках.

Величина *alpha* может меняться от 0 до 255. 0 соответствует полной прозрачности цвета (при отображении пикселя такого цвета он оставляет имеющееся изображение графика в этой точке без изменения), 255 означает применение полной плотности цвета (при отображении пикселя такого цвета он полностью заменяет цвет графика в соответствующей точке). Значение 128 (0x80) соответствует полупрозрачности.

> Как мы знаем, тип *color* описывает цвет с помощью трех компонент цвета: красной (Red), зеленой (Green) и синей (Blue), которые хранятся в формате 0x00BBGGRR в 4-байтовом целом (*uint*). Каждая компонента — это один байт, задающий насыщенность данного цвета в

интервале от 0 до 255 (от 0x00 до 0xFF в шестнадцатеричном формате). Старший байт — пустой. Например, белый цвет содержит в себе все цвета и потому имеет значение *color*, равное 0xFFFFFF.

Но в ряде задач требуется указание прозрачности цвета, чтобы описать, каким образом будет выглядеть изображение при наложении на некоторый фон (другое, уже существующее изображение). Для таких случаев вводится понятие альфа-канала, который кодируется дополнительным байтом.

Представление цвета в формате ARGB вместе с альфа-каналом (обозначен AA) имеет вид: 0xAARRGGBB. Например, значение 0x80FFFF00 означает желтый (за счет красной и зеленой компонент) полупрозрачный цвет.

При наложении изображения с альфа-каналом на некоторый фон получается результирующий цвет:

```
Cresult = (Cforeground * alpha + Cbackground * (255 - alpha)) / 255
```

где C - каждая из компонент R, G, B. Данная формула приведена для справки. При использовании встроенных функций с ARGB-цветами прозрачность применяется автоматически.

Пример использования *ColorToARGB* приведен в *ConversionColor.mq5*. Для удобства анализа цветовых компонент в скрипт добавлены вспомогательная структура *Argb* и объединение ColorARGB.

```
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) { }
};
```

Структура используется в качестве типа поля *split* в объединении и обеспечивает доступ к компонентам ARGB по именам. Также в объединении имеется байтовый массив *channels*, который позволяет обращаться к компонентам по индексу.

```
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
   */
}
```

Функцию *PrintFormat* мы рассмотрим чуть позже в соответствующем разделе.

Для обратного преобразования ARGB в *color* встроенной функции нет (поскольку обычно оно не требуется), но желающие могут воспользоваться следующим макросом:

```
#define ARGBToColor(U) (color) \
   ((((U) & 0xFF) << 16) | ((U) & 0xFF00) | (((U) >> 16) & 0xFF))
```

4.1.5 Структуры

При интеграции MQL-программ с внешними системами, в частности, при отправке или получении данных через Интернет, возникает необходимость преобразовать структуры данных в массивы байтов. Для этих целей MQL5 API предоставляет две функции: *StructToCharArray* и *CharArrayToStruct*.

В обоих случаях предполагается, что структура содержит только простые встроенные типы, то есть все встроенные типы кроме строк и динамических массивов. Также структура может содержать другие простые структуры. Объекты классов и указатели запрещены. Подобные структуры называются еще POD (Plain Old Data).

bool StructToCharArray(const void &object, uchar &array[], uint pos = 0)

Функция *StructToCharArray* копирует POD-структуру *object* в массив *array* типа *uchar*. Опционально можно с помощью параметра *pos* указать позицию в массиве, начиная с которой будут помещаться байты. По умолчанию копирование идет в начало массива, причем динамический массив будет автоматически увеличен в размере, если его текущего размера недостаточно для всей структуры.

Функция возвращает признак успеха (*true*) или ошибки (*false*).

Проверим её работу с помощью скрипта *ConversionStruct.mq5*. Создадим новый тип структуры *DateTimeMsc*, которая включает стандартную структуру *MqlDateTime* (поле *mdt*) и дополнительное поле *msc* типа *int* для хранения миллисекунд.

```
struct DateTimeMsc
{
   MqlDateTime mdt;
   int msc;
   DateTimeMsc(MqlDateTime &init, int m = 0) : msc(m)
   {
      mdt = init;
   }
};
```

В функции *OnStart* преобразуем некое тестовое значение *datetime* в нашу структуру и затем — в байтовый массив.

```
MqlDateTime TimeToStructInplace(datetime dt)
{
   static MqlDateTime m;
   if(!TimeToStruct(dt, m))
   {
      // можно вывести код ошибки _LastError,
      // но здесь просто возвращаем нулевое время
      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);
}
```

Получим в журнале следующий результат (массив переформатирован с дополнительными переносами строк, чтобы подчеркнуть соответствие байтов каждому из полей):

```
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)

Функция *CharArrayToStruct* копирует массив *array* типа *uchar* в POD-структуру *object*. С помощью параметра *pos* можно указать позицию в массиве, начиная с которой начнется чтение байтов.

Функция возвращает признак успеха (*true*) или ошибки (*false*).

Продолжая тот же пример (*ConversionStruct.mq5*), мы можем восстановить исходную дату и время из байтового массива.

```
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 Перечисления

MQL5 API предоставляет возможность преобразовать значение перечисления в строку с помощью функции *EnumToString*. Готового обратного преобразования нет.

string EnumToString(enum value)

Функция преобразует значение (т.е. идентификатор переданного элемента) перечисления любого типа в строку.

Воспользуемся ею, чтобы решить одну из востребованных задач: узнать размер перечисления (сколько в нем элементов) и какие именно значения соответствуют всем элементам. Для этой цели реализуем в заголовочном файле *EnumToArray.mqh* специальную шаблонную функцию (благодаря шаблонному типу E она подойдет для любого перечисления):

```
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;
}
```

Принцип её действия основан на следующем. Поскольку перечисления в MQL5 хранятся в виде целых чисел типа *int*, поддерживается неявное приведение любого перечисления к *(int)*, а также допускается явное приведение *int* обратно к любому типу перечисления. При этом, если значение соответствует одному из элементов перечисления, функция *EnumToString* возвращает строку с идентификатором этого элемента. В противном случае функция возвращает строку вида ENUM_TYPE::значение.

Таким образом, делая цикл по целым числам в приемлемом диапазоне и приводя их явно к типу перечисления, можно затем анализировать строку на выходе *EnumToString* на наличие '::', чтобы определить, является ли данное целое элементом перечисления или нет.

Использованная здесь функция *StringFind* будет представлена в следующей главе, как и другие функции работы со строками.

Для проверки концепции создадим скрипт *ConversionEnum.mq5*. В нем реализуем вспомогательную функцию *process*, которая будет вызвать шаблонную *EnumToArray*, сообщать количество элементов в перечислении и распечатывать полученный массив с соответствиями между элементами перечисления и их значениями.

```
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]);
  }
}
```

В качестве перечисления для исследования возьмем встроенное перечисление с типами цен ENUM_APPLIED_PRICE. В функции *OnStart* для начала убедимся, что *EnumToString* производит строки по описанному выше принципу. Так, для элемента PRICE_CLOSE функция вернет строку "PRICE_CLOSE", а для значения (ENUM_APPLIED_PRICE)10, которое заведомо лежит вне диапазона, — "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);
}
```

Далее вызовем функцию *process* для любого значения, приведенного к ENUM_APPLIED_PRICE (или переменной этого типа), и получим следующий результат:

```
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
```

Здесь мы видим, что в перечислении определено 7 элементов, причем нумерация начинается не с 0, как обычно, а с 1 (PRICE_CLOSE). Знание значений, сопоставленных с элементами, позволяет в некоторых случаях оптимизировать написание алгоритмов.

4.1.7 Тип complex

Встроенный тип *complex* представляет из себя структуру с двумя полями типа *double*:

```
struct complex
{
   double      real;   // вещественная часть
   double      imag;   // мнимая часть
};
```

Мы описываем её в разделе о конвертации типов, потому что она "превращает" два числа *double* в новую сущность, в чем-то по похожему принципу, как структуры "превращаются" в массивы байтов и обратно. Кроме того, было бы в принципе затруднительно представлять этот тип без предварительного описания структур.

Структура *complex* не имеет конструктора, поэтому создавать комплексные числа следует с помощью списка инициализации.

```
complex c = {re, im};
```

Для комплексных чисел на данный момент доступны только простые арифметические операции и операции сравнения: =, +, -, *, /, +=, -=, *=, /=, ==, !=. Поддержка математических функций будет добавлена позднее.

> Внимание! Комплексные переменные не могут быть описаны как входные (с помощью ключевого слова *input*) для MQL-программы.

Для описания комплексных (мнимых частей) констант используется суффикс 'i', например:

```
const complex x = 1 - 2i;
const complex y = 0.5i;
```

В следующем примере (скрипт *Complex.mq5*) создается комплексное число и возводится в квадрат.

```
input double r = 1;
input double i = 2;

complex c = {r, i};

complex mirror(const complex z)
{
   complex result = {z.imag, z.real}; // обмен местами реальной и мнимой частей
   return result;
}

complex square(const complex z)
{
   return (z * z);
}

void OnStart()
{
   Print(c);
   Print(square(c));
   Print(square(mirror(c)));
}
```

С параметрами по умолчанию скрипт выведет строки:

```
c=(1,2) / ok
square(c)=(-3,4) / ok
square(mirror(c))=(3,4) / ok
```

Здесь пары чисел в круглых скобках — это строковое представление комплексного числа.

Тип *complex* может передаваться по значению в качестве параметра MQL-функций (в отличие от обычных структур, которые передаются только по ссылке). Для функций, импортируемых из DLL, тип *complex* должен передаваться только по ссылке.

4.2 Работа со строками и символами

Хотя компьютеры получили свое название от глагола "вычислять" ("compute"), они в равной степени успешно применяются не только для обработки чисел, но и неструктурированной информации, наиболее известным примером которой выступает текст. В MQL-программах текст также используется повсеместно, начиная от названий самих программ и заканчивая комментариями в торговых приказах. Для работы с текстом в MQL5 имеется встроенный строковый тип, позволяющий оперировать символьными последовательностями произвольной длины.

Для выполнения типичных действий со строками MQL5 API предоставляет широкий набор функций, которые можно условно поделить на группы по назначению, такие как инициализация строк, их сложение, поиск и замена фрагментов внутри строк, преобразование строк в массивы символов, обращение к отдельным символам, а также форматирование.

Большинство функций данной главы возвращает признак статуса выполнения: успех или ошибку. Для функций с типом результата *bool* — как правило, *true* — это успех, а *false* — ошибка. Для функций с типом результата *int* ошибкой может считаться значение 0 или -1: об этом говорится в описании каждой функции. Во всех этих случаях разработчик имеет возможность уточнить суть проблемы. Для этого следует вызвать функцию *GetLastError* и получить код конкретной ошибки: список всех кодов с пояснениями имеется в документации. Важно вызывать *GetLastError* сразу после получения флага ошибки, потому что вызов каждой следующей инструкции в алгоритме способен привести к другой ошибке.

4.2.1 Инициализация и измерение строк

Как мы знаем из раздела о строковом типе, достаточно описать в коде переменную типа *string*, и она будет готова к работе.

Под любую переменную типа *string* выделяется 12 байтов на служебную структуру — внутреннее представление строки. Структура содержит адрес памяти (указатель), где хранится текст, и кое-какую другую мета-информацию. Сам текст также требует памяти достаточного размера, но выделяется этот буфер с применением некоторых не столь очевидных оптимизаций.

В частности, мы можем описать строку вместе с явной инициализацией, в том числе и пустым литералом:

```
string s = ""; // указатель на литерал, содержащий '\0'
```

В таком случае указатель установится непосредственно на литерал, и память под буфер не выделяется (даже если литерал — длинный). Очевидно, что под литерал уже выделена статическая память и её можно использовать напрямую. Память под буфер будет распределена,

только если какая-либо инструкция в программе изменит содержимое строки. Например (напомним, что для строк определена операция сложения '+'):

```
int n = 1;
s += (string)n;     // указатель на память, содержащую "1"'\0'[плюс резерв]
```

С этого момента строка фактически содержит текст "1" и требует, строго говоря, памяти для двух символов: цифры "1" и неявного терминального нуля '\0' (признака конца строки). Однако система выделит буфер большего размера, с запасом.

Когда мы описываем переменную без начального значения, она все равно неявно инициализируется компилятором, правда в этом случае — специальным значением NULL:

```
string z; // память для указателя не выделена, указатель = NULL
```

Такая строка требует только 12 байтов на структуру, а указатель никуда не ведет: это и обозначает NULL.

> В будущих версиях компилятора MQL5 данное поведение может измениться, и под пустую строку всегда будет изначально выделяться небольшой участок памяти, так сказать, "на вырост".

Кроме этих внутренних особенностей, переменные типа *string* ничем не отличаются от переменных других типов. Однако, в связи с тем, что строки могут быть переменной длинны и, что более важно, менять свою длину в ходе алгоритма, это способно отрицательно сказываться на эффективности распределения памяти и производительности.

Например, если в какой-то момент программе потребуется добавить в строку новое слово, то может оказаться, что выделенной под строку памяти недостаточно. Тогда среда исполнения MQL-программы незаметно для пользователя найдет новый свободный блок памяти увеличенного размера и скопирует туда прежнее значение вместе с добавляемым словом. После этого старый адрес подменяется новым в служебной структуре строки.

Если подобных операций производится много, замедление из-за копирования может стать заметным, а кроме того, память программы подвергается фрагментации: старые мелкие участки памяти, освобожденные во время копирования, образуют пустоты, которые не подходят по размеру для больших строк, и потому приводят к непродуктивному расходу памяти. Конечно, терминал способен контролировать подобные ситуации и проводить реорганизацию памяти, но это также требует затрат.

Наиболее действенным способом решения данной проблемы является явное заблаговременное указание размера буфера под строку и её инициализация с помощью встроенных функций MQL5 API, которые мы рассмотрим далее.

Основой для данной оптимизации как раз служит то, что размер выделенной памяти может превышать текущую (и потенциальную будущую) длину строки, которая определяется по первому нулевому символу в тексте. Таким образом, мы можем распределить буфер под 100 символов, но изначально поставить в самое начало '\0', что даст строку нулевой длины ("").

Разумеется, предполагается, что в подобных случаях программист заранее может примерно подсчитать ожидаемую длину строки или темпы её роста.

Поскольку строки в MQL5 строятся на двухбайтовых символах (что обеспечивает поддержку Unicode), размер строки и буфера в символах следует умножать на 2, чтобы получить объем занимаемой и выделенной памяти в байтах.

Общий пример использования всех функций (*StringInit.mq5*) будет приведен в конце раздела.

bool StringInit(string &variable, int capacity = 0, ushort character = 0)

Функция *StringInit* применяется дли инициализации (распределения и заполнения памяти) и деинициализации (освобождения памяти) строк. Обрабатываемая переменная передается в первом параметре.

Если параметр *capacity* больше 0, то под строку выделяется буфер (область памяти) указанного размера и заполняется символом *character*. Если *character* равен 0, то длина строки будет нулевой, потому что первый же символ является терминальным.

Если параметр *capacity* равен 0, ранее выделенная память освобождается. Состояние переменной становится идентично тому, как будто её только что описали без инициализации (указатель на буфер равен NULL). Более просто то же самое можно сделать путем присваивания строковой переменной значения NULL.

Функция возвращает признак успеха (*true*) или ошибки (*false*).

bool StringReserve(string &variable, uint capacity)

Функция *StringReserve* увеличивает или уменьшает размер буфера строки *variable* как минимум до количества символов, указанных в параметре *capacity*. Если значение *capacity* меньше текущей длины строки, функция ничего не делает. По факту размер буфера может оказаться больше, чем запрошенный: среда делает так из соображений эффективности будущих манипуляций со строкой. Таким образом, если функция вызвана с уменьшенным значением для буфера, она может проигнорировать запрос и при этом вернёт *true* ("нет ошибок").

Актуальный размер буфера можно получить с помощью функции *StringBufferLen* (см. ниже).

В случае успеха функция возвращает *true*, иначе — *false*.

В отличие от *StringInit* функция *StringReserve* не изменяет содержимое строки и не заполняет её символами.

bool StringFill(string &variable, ushort character)

Функция *StringFill* заполняет указанную строку *variable* символом *character*, на всю её текущую длину (до первого нуля). Если для строки выделен буфер, модификация производится по месту, без промежуточных операций создания новой строки и копирования.

Функция возвращает признак успеха (*true*) или ошибки (*false*).

int StringBufferLen(const string &variable)

Функция возвращает размер буфера, распределенного для строки *variable*.

Обратите внимание, что для строки с инициализацией литералом буфер изначально не выделяется, поскольку указатель ведет на литерал. Поэтому функция вернет 0, хотя длина строки *StringLen* (см. ниже) может быть и больше.

Значение -1 означает, что строка принадлежит клиентскому терминалу и изменять её нельзя.

bool StringSetLength(string &variable, uint length)

Устанавливает для строки *variable* указанную длину в символах *length*. Значение *length* должно быть не больше текущей длины строки. Иными словами, функция позволяет только укоротить

строку, но не удлинить. Увеличение длины строки происходит автоматически при вызове функции *StringAdd* или выполнении операции сложения '+'.

Эквивалентом функции *StringSetLength* является вызов: *StringSetCharacter(variable, length, 0)* (см. раздел Работа с символами и кодовыми страницами).

Если перед вызовом функции для строки уже был выделен буфер, то функция его не меняет. Если строка не имела буфера (указывала на литерал), уменьшение длины приведен к выделению нового буфера и копированию в него укороченной строки.

Функция возвращает *true* или *false* в случае, соответственно, успеха или ошибки.

int StringLen(const string text)

Функция возвращает число символов в строке *text*. Терминальный ноль не учитывается.

Обратите внимание, что параметр передается по значению, поэтому можно вычислять длину строк не только в переменных, но и для любых других промежуточных значений: результатов вычислений или литералов.

Для демонстрации вышеописанных функций создан скрипт *StringInit.mq5*. В нем используется особая версия макроса PRT — PRTE, которая анализирует результат выражения на *true* или *false*, и в последнем случае дополнительно выводит код ошибки:

```
#define PRTE(A) Print(#A, "=", (A) ? "true" : "false:" + (string)GetLastError())
```

Для отладочного вывода в журнал строки и её текущих метрик (длина строки и размер буфера) реализована функция *StrOut*:

```
void StrOut(const string &s)
{
   Print("'", s, "' [", StringLen(s), "] ", StringBufferLen(s));
}
```

Она использует встроенные функции *StringLen* и *StringBufferLen*.

Тестовый скрипт выполняет ряд действий над строкой в *OnStart*:

```
void OnStart()
{
   string s = "message";
   StrOut(s);
   PRTE(StringReserve(s, 100)); // ok, но получим буфер больше запрошенного: 260
   StrOut(s);
   PRTE(StringReserve(s, 500)); // ok, буфер увеличен до 500
   StrOut(s);
   PRTE(StringSetLength(s, 4)); // ok: строка укорочена
   StrOut(s);
   s += "age";
   PRTE(StringReserve(s, 100)); // ok: буфер остался равным 500
   StrOut(s);
   PRTE(StringSetLength(s, 8)); // no: удлиннение строк не поддерживается
   StrOut(s);                   //     через StringSetLength
   PRTE(StringInit(s, 8, '$')); // ok: строка увеличена за счет заполнения
   StrOut(s);                   //     буфер остался прежним
   PRTE(StringFill(s, 0));      // ok: строка схлопнулась до пустой, потому что
   StrOut(s);                   //     была заполнена 0-ми, буфер прежний
   PRTE(StringInit(s, 0));      // ok: строка обнулена, включая буфер
                                // можно было написать просто s = NULL;
   StrOut(s);
}
```

Скрипт выведет в журнал следующие сообщения:

```
'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
```

Обратите внимание, что вызов *StringSetLength* с увеличенной длиной строки закончился ошибкой 5035 (ERR_STRING_SMALL_LEN).

4.2.2 Суммирование (соединение) строк

Суммирование строк является, пожалуй, самой частой операцией со строками. В MQL5 её можно выполнить с помощью операторов '+' или '+='. Первый состыковывает две строки (операнды слева и справа от '+') и создает временную объединенную строку, которую можно присвоить

целевой переменной или передать в другую часть выражения (например, в вызов функции). Второй присоединяет строку справа от оператора '+=' к строке (переменной), стоящей слева от этого оператора.

В дополнение к этому MQL5 API предоставляет пару функций для составления строк из других строк или элементов других типов.

Примеры использования функций приведены в скрипте *StringAdd.mq5*, который рассматривается после их описания.

bool StringAdd(string &variable, const string addition)

Функция присоединяет к концу строковой переменной *variable* указанную строку *addition*. По возможности система использует имеющийся буфер строки *variable* (если его размера хватает для объединенного результата) без перевыделения памяти и копирования строк.

Функция эквивалента оператору *variable += addition*. Временные затраты и расход памяти примерно одинаковы.

Функция возвращает *true* в случае успеха, и *false* в случае ошибки.

int StringConcatenate(string &varibale, void argument1, void argument2 [, void argumentI...])

Функция преобразует в строковое представление два или более аргументов встроенных типов и объединяет их в строке *variable*. Аргументы для слияния передаются, начиная со второго параметра функции. В качестве аргументов не поддерживаются массивы, структуры, объекты, указатели.

Количество аргументов должно быть от 2 до 63.

Аргументы-строки добавляются в результирующую переменную как есть.

Аргументы типа *double* преобразуются с максимальной точностью (до 16 значащих знаков), при этом может быть выбрана научная нотация с показателем степени, если она получается более компактной. Аргументы типа *float* отображаются с 5 знаками.

Значения типа *datetime* конвертируются в строку со всеми полями даты и времени ("YYYY.MM.DD hh:mm:ss").

Перечисления, однобайтовые и двухбайтовые символы выводятся как целые числа.

Значения типа *color* отображаются в виде тройки компонент "R,G,B" или названия цвета (если он имеется в списке стандартных web-цветов).

При конвертации типа *bool* используются строки "true" или "false".

Функция *StringConcatenate* возвращает длину получившейся строки.

> *StringConcatenate* предназначена для построения строки из других источников (переменных, выражений), отличных от приемной переменной. Использовать *StringConcatenate* для пристыковки новых порций данных к одной и той же строке путем вызова *StringConcatenate(variable, variable, ...)* не рекомендуется. Такой вызов функции не оптимизирован и выполняется крайне медленно по сравнению с опертором '+' и StringAdd.

Функции *StringAdd* и *StringConcatenate* тестируются в скрипте *StringAdd.mq5*, который использует макрос PRTE и вспомогательную функцию *StrOut* из предыдущего раздела.

```
void OnStart()
{
   string s = "message";
   StrOut(s);
   PRTE(StringAdd(s, "r"));
   StrOut(s);
   PRTE(StringConcatenate(s, M_PI * 100, " ", clrBlue, PRICE_CLOSE));
   StrOut(s);
}
```

В результате его выполнения в лог выводятся следующие строки:

```
'message' [7] 0
StringAdd(s,r)=true
'messager' [8] 260
StringConcatenate(s,M_PI*100, ,clrBlue,PRICE_CLOSE)=true
'314.1592653589793 clrBlue1' [26] 260
```

В скрипте также подключен заголовочный файл *StringBenchmark.mqh* с классом *Benchmark*. Он представляет основу для классов-наследников, реализованных в скрипте для измерения быстродействия различных способов сложения строк. В частности, они позволяют убедиться, что сложение строк при помощи оператора '+' и функции *StringAdd* сопоставимы. Этот материал оставлен для самостоятельного изучения.

Дополнительно с книгой поставляется скрипт *StringReserve.mq5*: он проводит наглядное сравнение скорости сложения строк в зависимости от использования или неиспользования буфера (*StringReserve*).

4.2.3 Сравнение строк

Для сравнения строк в MQL5 можно использовать стандартные операторы сравнения, в частности '==', '!=', '>', '<'. Все такие операторы производят сравнение посимвольно, с учетом регистра.

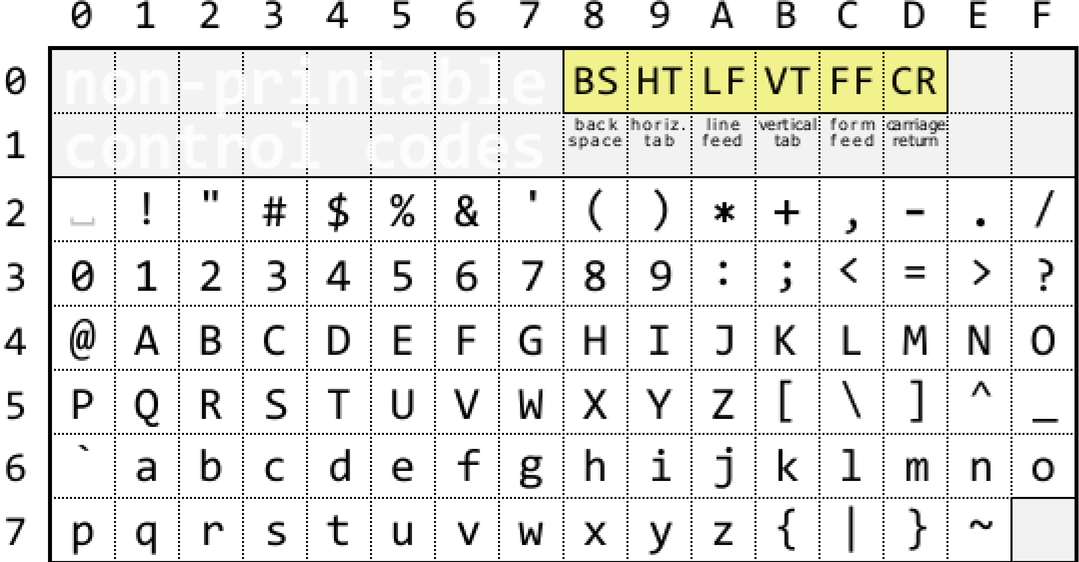
У каждого символа имеется код Unicode, представляющий собой целое типа *ushort*. Соответственно, сперва сравниваются коды первых символов двух строк, потом коды вторых, и так далее до первого несовпадения или конца одной из строк.

Например, строка "ABC" меньше чем "abc", потому что в таблице символов коды заглавных букв меньше, чем коды соответствующих строчных букв (уже на первом символе получим, что "A" < "a"). Если строки имеют совпадающие символы в начале, но одна из них длиннее другой, то более длинная считается большей ("ABCD" > "ABC").

Подобные отношения строк образуют лексикографический порядок, когда строка "A" меньше строки "Б" ("А" < "Б"), говорят что "А" предшествует "Б".

> Для ознакомления с кодами символов вы можете использовать стандартное приложение Windows "Таблица символов". В ней символы расположены в порядке увеличения кодов. Помимо общей таблицы Unicode, включающей множество национальных языков, существуют кодовые страницы: таблицы стандарта ANSI с однобайтными кодами символов — они различаются для каждого языка или группы языков. Мы более подробно изучим данный вопрос в разделе Работа с символами и кодовыми страницами.

Начальная часть таблиц символов с кодами от 0 до 127 является одинаковой для всех языков и приведена в следующей таблице.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	non-printable								BS	HT	LF	VT	FF	CR		
1	control codes								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	{	|	}	~	

Таблица кодов символов ASCII

Для получения кода символа возьмите шестнадцатеричную цифру слева (номер строки, в которой расположен символ) и дополните её цифрой сверху (номер колонки, в которой расположен символ): в результате получится шестнадцатеричное число. Например, для '!' слева стоит 2, а сверху 1, значит код символа 0x21, или в десятичной системе 33.

Коды до 32 являются управляющими. Среди них, в частности, табуляция (код 0x9), перевод строки (line feed, 0xA) и возврат каретки (carriage return, 0xD).

Пара символов 0xD 0xA, следующих один за другим, используется в текстовых файлах Windows для перехода на новую строку. Мы знакомились с соответствующими литералами MQL5 в разделе Символьные типы: 0xA можно обозначать как '\n', а 0xD — как '\r'. Табуляция 0x9 также имеет своё представление: '\t'.

MQL5 API предоставляет функцию *StringCompare*, которая позволяет отключать учет регистра при сравнении строк.

int StringCompare(const string &string1, const string &string2, const bool case_sensitive = true)

Функция сравнивает две строки и возвращает одно из трех значений: +1, если первая строка "больше" второй; 0 — если строки "равны"; -1 — если первая строка "меньше" второй. Понятия "больше", "меньше" и "равно" зависят от параметра *case_sensitive*.

Когда параметр *case_sensitive* равен *true* (что соответствует умолчанию), сравнение производится с учетом регистра, причем заглавные буквы считаются больше аналогичных строчных. Это является обратным порядком по отношению к стандартному лексикографическому порядку согласно кодам символов.

> При учете регистра функция *StringCompare* использует порядок заглавных и строчных букв, отличный от лексикографического. Например, мы знаем, что истинно отношение "A" < "a", в котором оператор '<' руководствуется кодами символов. Поэтому слова с заглавной буквы должны идти в гипотетическом словаре (массиве) раньше, чем слова с той же строчной буквы. Однако при сравнении "A" и "a" с помощью функции *StringCompare("A", "a")* мы получим +1, что означает "A" больше "a". Таким образом, в отсортированном словаре

спереди будут идти слова, начинающиеся со строчных букв, и только после них — с заглавных.

Иными словами, функция ранжирует строки в алфавитном порядке. Но кроме того, в режиме включенной чувствительности к регистру, дополнительно действует правило: при наличии строк, отличающихся лишь регистром, те из них, что имеют заглавные буквы, следуют после своих аналогов со строчными буквами (на тех же позициях в слове).

Когда параметр *case_sensitive* равен *false*, регистр букв не учитывается, поэтому строки "A" и "a" равны, а функция возвращает 0.

Убедиться в разных результатах сравнения оператором и функцией *StringCompare* можно в помощью скрипта *StringCompare.mq5*.

```
void OnStart()
{
   PRT(StringCompare("A", "a"));        // 1, что означает "A" > "a" (!)
   PRT(StringCompare("A", "a", false)); // 0, что означает "A" == "a"
   PRT("A" > "a");                      // false,   "A" < "a"

   PRT(StringCompare("x","y"));         // -1, что означает "x" < "y"
   PRT("x" > "y");                      // false,    "x" < "y"
   ...
}
```

В разделе Шаблоны функций мы создали шаблонизированный алгоритм быстрой сортировки. Преобразуем его в шаблонный класс и используем для нескольких вариантов сортировки: с помощью операторов сравнения, а также функции *StringCompare* как с включенным, так и выключенным режимом учета регистра. Новый класс *QuickSortT* поместим в заголовочный файл *QuickSortT.mqh* и подключим его в тестовый скрипт *StringCompare.mq5*.

Программный интерфейс сортировки остался почти без изменений.

```
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++);
            }
         }
      ...
   }
};
```

Основное отличие в том, что мы добавили виртуальный метод *Compare*, который по умолчанию содержит сравнение с помощью операторов '>' и '<', и возвращает +1, -1 или 0 по тому же принципу, что и *StringCompare*. Метод *Compare* используется теперь в методе *QuickSort* вместо простого сравнения и должен быть переопределен в классах-наследниках, для того чтобы задействовать функцию *StringCompare* или любой другой принцип сравнения величин.

В частности, в файле *StringCompare.mq5* мы реализуем следующий класс-"компаратор", производный от *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);
   }
};
```

В конструктор передается 1 параметр, задающий признак сравнения строк с учетом (*true*) или без учета (*false*) регистра. Само сравнение строк производится в переопределенном виртуальном

методе *Compare*, который вызывает функцию *StringCompare* с заданными аргументами и настройкой.

Для проверки работы сортировки нам нужен набор строк, сочетающий заглавные и строчные буквы. Мы можем его сгенерировать сами: достаточно разработать класс, который выполняет перестановки (с повторением) символов из предопределенного набора (алфавита) для заданной длины набора (строки). Например, можно ограничиться маленьким алфавитом "abcABC", то есть три начальных английских буквы в обоих регистрах, и генерировать из них все возможные строки длиной 2 символа.

Класс *PermutationGenerator* поставляется в файле *PermutationGenerator.mqh* и оставлен для самостоятельного изучения. Здесь приведем лишь его публичный интерфейс.

```
class PermutationGenerator
{
public:
   struct Result
   {
      int indices[]; // индексы элементов в каждой позиции набора, т.е.
   };                // например, номера букв "алфавита" в каждой позиции строки
   PermutationGenerator(const int length, const int elements);
   SimpleArray<Result> *run();
};
```

При создании объекта генератора необходимо задать длину генерируемых наборов *length* (в нашем случае это будет длина строк, то есть 2) и количество разных элементов, из которых будут составляться наборы (в нашем случае это количество уникальных букв, то есть 6). При таких входных данных должно получиться 6*6=36 вариантов строк.

Сам процесс выполняется методом *run*. Для возврата массива с результатами используется шаблонный класс *SimpleArray*, который мы рассматривали в разделе Шаблоны методов. В данном случае он параметризуется типом структуры *Result*.

Вызов генератора и фактическое создание строк в соответствии с полученным от него массивом перестановок (в виде индексов букв на каждой позиции для всех возможных строк) производится во вспомогательной функции *GenerateStringList*.

```
void GenerateStringList(const string symbols, const int len, string &result[])
{
   const int n = StringLen(symbols); // длина алфавита, уникальные символы
   PermutationGenerator g(len, n);
   SimpleArray<PermutationGenerator::Result> *r = g.run();
   ArrayResize(result, r.size());
   // цикл по всем полученным перестановкам символов
   for(int i = 0; i < r.size(); ++i)
   {
      string element;
      // цикл по всем знакоместам в строке
      for(int j = 0; j < len; ++j)
      {
         // добавляем букву из алфавита (по её индексу) к строке
         element += ShortToString(symbols[r[i].indices[j]]);
      }
      result[i] = element;
   }
}
```

Здесь используется несколько ещё незнакомых нам функций (*ArrayResize*, *ShortToString*), но скоро мы до них доберемся. Пока достаточно будет знать, что функция *ShortToString* по коду символа типа *ushort* возвращает строку, состоящую из этого одного символа. С помощью оператора '+=' мы состыковываем каждую результирующую строку из таких односимвольных строк. Напомним, что для строк определен оператор [], поэтому выражение *symbols[k]* вернет *k*-й символ строки *symbols*. Разумеется, *k* может быть в свою очередь целочисленным выражением, и здесь оно — *r[i].indices[j]* — обращается к *i*-му элементу массива *r*, из которого читается индекс символа "алфавита" для *j*-ой позиции строки.

Каждая полученная строка сохраняется в массив-параметр *result*.

Настало время применить эти наработки в функции *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);
}
```

Скрипт сначала получает все варианты строк в массив messages, а затем сортирует его в 3 режимах: с помощью встроенных операторов сравнения, с помощью функции *StringCompare* без учета регистра, и с помощью неё же, но уже с учетом регистра.

Мы получим следующий вывод в журнал:

```
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"
```

Легко увидеть, каким именно образом отличаются все 3 режима.

4.2.4 Изменение регистра символов и обрезка пробелов

При работе с текстами часто возникает необходимость в некоторых стандартных операциях: привести все символы к верхнему или нижнему регистру, а также убрать лишние пустые символы (например, пробелы) в начале или конце строки. Для этих целей MQL5 API предоставляет 4 соответствующие функции. Все они модифицируют строку по месту, то есть непосредственно в имеющемся буфере (если он уже распределен).

Входным параметром всех функций является ссылка на строку, то есть в них можно передать только переменные (не выражения), причем не константные переменные, поскольку функции предполагают модификацию аргумента.

Тестовый скрипт для всех функций приведен после их описаний.

bool StringToLower(string &variable)
bool StringToUpper(string &variable)

Функции преобразуют все символы указанной строки в соответствующий регистр: *StringToLower* — в строчные (маленькие) буквы, а *StringToUpper* — в прописные (большие). В том числе поддерживаются национальные языки, доступные на уровне системы Windows.

В случае успеха возвращается *true*, в случае ошибки — *false*.

int StringTrimLeft(string &variable)
int StringTrimRight(string &variable)

Функция удаляет символы перевода каретки ('\r'), перехода на новую строку ('\n'), пробелы (' '), табуляции ('\t') и некоторые другие неотображаемые символы в начале (для *StringTrimLeft*) или конце (для *StringTrimRight*) строки. Если внутри строки (между отображаемыми символами) есть еще пустые знакоместа, они сохранятся.

Функция возвращает количество удаленных символов.

В файле *StringModify.mq5* демонстрируется работа вышеперечисленных функций.

```
void OnStart()
{
   string text = "  \tAbCdE F1  ";
               // ↑        ↑  ↑
               // |        |  └2 пробела
               // |        └пробел
               // └2 пробела и табуляция
   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'  (больше нечего удалять)
   PRT(text);                  // 'ABCDE F1'
                               //       ↑
                               //       └пробел внутри остается

   string russian = "Русский Текст";
   PRT(StringToUpper(russian));  // 'true'
   PRT(russian);                 // 'РУССКИЙ ТЕКСТ'
   string german = "straßenführung";
   PRT(StringToUpper(german));   // 'true'
   PRT(german);                  // 'STRAßENFÜHRUNG'
}
```

4.2.5 Поиск, замена и извлечение фрагментов строк

Пожалуй, наиболее востребованными операциями при работе со строками являются поиск и замена фрагментов, а также их извлечение. В данном разделе мы изучим функции MQL5 API, которые помогут решить эти задачи. Примеры их использования сведены в файл *StringFindReplace.mq5*.

int StringFind(string value, string wanted, int start = 0)

Функция выполняет поиск подстроки *wanted* в строке *value*, начиная с позиции *start*. Если подстрока найдена, функция вернет позицию, где она начинается, причем символы в строке нумеруются с 0. В противном случае функция вернет -1. Оба параметра передаются по значению, что позволяет обрабатывать не только переменные, но и промежуточные результаты вычислений (выражения, вызовы функций).

Поиск выполняется на строгое соответствие символов, то есть с учетом регистра. Если требуется искать без учета регистра, необходимо предварительно привести исходную строку к единому регистру посредством *StringToLower или StringToUpper*.

Попробуем подсчитать количество вхождений искомой подстроки в тексте с помощью *StringFind*. Для этого напишем вспомогательную функцию *CountSubstring*, которая будет в цикле вызывать *StringFind*, постепенно смещая стартовую позицию поиска в последнем параметре *start*. Цикл продолжается пока находятся новые вхождения подстроки.

```
int CountSubstring(const string value, const string wanted)
{
   // делаем отступ назад из-за инкремента в начале цикла
   int cursor = -1;
   int count = -1;
   do
   {
      ++count;
      ++cursor; // поиск продолжаем со следующей позиции
      // получаем позицию следующей подстроки или -1, если совпадений нет
      cursor = StringFind(value, wanted, cursor);
   }
   while(cursor > -1);
   return count;
}
```

Важно отметить, что представленная реализация ищет подстроки с возможностью наложения. Это происходит из-за того, что текущая позиция изменяется на 1 (++*cursor*) перед поиском следующего вхождения. В результате, при поиске, например, подстроки "AAA" в строке "AAAAA" будет найдено 3 совпадения. Технические требования на поиск могут отличаться от данного поведения. В частности, существует практика продолжать поиск после той позиции, где закончился предыдущий найденный фрагмент. В таком случае потребуется модифицировать алгоритм, чтобы курсор сдвигался с шагом, равным *StringLen(wanted)*.

В функции *OnStart* вызовем *CountSubstring* для разных аргументов.

```
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)

Функция заменяет в строке *variable* все найденные подстроки *wanted* на другую подстроку *replacement*.

Функция возвращает количество сделанных замен или -1 в случае ошибки. Код ошибки можно получить, вызвав функцию *GetLastError*. В частности, возможны ошибки нехватки памяти или использования в качестве аргумента неинициализированной строки (NULL). Параметры *variable* и *wanted* должны быть строками ненулевой длины.

Когда в качестве аргумента *replacement* задается пустая строка "", все вхождения *wanted* просто вырезаются из исходной строки.

Если замен не было, результат функции равен 0.

Продолжая пример *StringFindReplace.mq5*, проверим *StringReplace* в действии.

```
   string abracadabra = "ABRACADABRA";
   ...
   PRT(StringReplace(abracadabra, "ABRA", "-ABRA-")); // 2
   PRT(StringReplace(abracadabra, "CAD", "-"));      // 1
   PRT(StringReplace(abracadabra, "", "XYZ"));      // -1, ошибка
   PRT(GetLastError());      // 5040, ERR_WRONG_STRING_PARAMETER
   PRT(abracadabra);                               // '-ABRA---ABRA-'
   ...
```

Далее попробуем выполнить с помощью функции *StringReplace* одну из задач, встречающуюся при обработке произвольных текстов, а именно: требуется обеспечить, чтобы некоторый символ разделитель всегда использовался одиночным, то есть последовательности нескольких таких символов должны быть заменены на один. Обычно речь идет о пробелах между словами, но в технических данных могут быть и другие разделители. Мы для наглядности будем тестировать свою программу на разделителе '-'.

Реализуем алгоритм в виде отдельной функции *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;
}
```

В цикле *do-while* программа пытается заменить последовательность из двух разделителей на один, и цикл продолжается, пока функция *StringReplace* возвращает значения больше 0 (то есть, заменять еще есть что). Функция возвращает общее количество сделанных замен.

В функции *OnStart* "очистим" нашу надпись от множественных символов '-'.

```
   ...
   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[])

Функция разбивает переданную строку *value* на подстроки по заданному разделителю и помещает их в массив *result*. Функция возвращает количество полученных подстрок или -1 в случае ошибки.

Если разделителя в строке нет, в массиве будет один элемент, равный всей строке целиком.

Если исходная строка пуста или равна NULL, функция вернет 0.

Для демонстрации работы данной функции решим предыдущую задачу новым способом — с помощью *StringSplit*. Для этого напишем функцию *NormalizeSeparatorsBySplit*.

```
int NormalizeSeparatorsBySplit(string &value, const ushort separator = ' ')
{
   const string single = ShortToString(separator);
   
   string elements[];
   const int n = StringSplit(value, separator, elements);
   ArrayPrint(elements); // отладка
   
   StringFill(value, 0); // результат заменит исходную строку
   
   for(int i = 0; i < n; ++i)
   {
      // пустые строки означают разделители, и мы их должны добавить только
      // в том случае, если предыдущая строка не пуста (т.е. не разделитель тоже)
      if(elements[i] == "" && (i == 0 || elements[i - 1] != ""))
      {
         value += single;
      }
      else // все прочие строки соединяются вместе "как есть"
      {
         value += elements[i];
      }
   }
   
   return n;
}
```

Когда разделители встречаются в исходном тексте один за другим, соответствующий элемент в выходном массиве *StringSplit* оказывается пустой строкой "". Также пустая строка будет в начале массива, если текст начинается с разделителя, и в конце массива, если текст заканчивается разделителем.

Чтобы получить "очищенный" текст, нужно сложить все непустые строки из массива, "склеив" их одиночными символами-разделителями. Причем в разделитель следует "превращать" только те пустые элементы, у которых предыдущий элемент массива не является также пустым.

Разумеется, это лишь один из возможных вариантов реализации данного функционала. Проверим его в функции *OnStart*.

```
...
string copy2 = "-" + abracadabra + "-";         // '--ABRA---ABRA--'
PRT(NormalizeSeparatorsBySplit(copy2, '-'));    // 8
// отладочный вывод сплит-массива (внутри функции):
// ""     ""     "ABRA" ""     ""     "ABRA" ""     ""
PRT(copy2);                                     // '-ABRA-ABRA-'
```

string StringSubstr(string value, int start, int length = -1)

Функция извлекает из переданного текста *value* подстроку, начинающуюся с указанной позиции *start* и длиной *length*. Начальная позиция может быть от 0 до длины строки минус 1. Если длина *length* равна -1 или превышает количество символов от *start* до конца строки, будет извлечена вся оставшаяся часть строки.

Функция возвращает подстроку или пустую строку, если параметры заданы неверно.

Посмотрим, как это работает.

```
PRT(StringSubstr("ABRACADABRA", 4, 3));        // 'CAD'
PRT(StringSubstr("ABRACADABRA", 4, 100));      // 'CADABRA'
PRT(StringSubstr("ABRACADABRA", 4));           // 'CADABRA'
PRT(StringSubstr("ABRACADABRA", 100));         // ''
```

4.2.6 Работа с символами и кодовыми страницами

Поскольку строки состоят из символов, иногда нужно или просто удобнее манипулировать отдельными символами или группами символов в строке на уровне их целочисленных кодов. Например, требуется читать или заменять символы по одиночке, преобразовывать в массивы целочисленных кодов для передачи по коммуникационным протоколам или в программные интерфейсы сторонних динамических библиотек DLL. Во всех таких случаях передача строк в виде текста может быть сопряжена с различными сложностями:

- обеспечение правильной кодировки (которых существует великое множество, и на выбор конкретной влияет локализация операционной системы, настройки программы, конфигурация серверов, с которыми осуществляется связь, и многое другое);
- конвертация символов национального языка из локальной текстовой кодировки в Unicode и обратно;
- выделение и освобождение памяти унифицированным образом.

Использование массивов с целочисленными кодами (что дает, фактически, двоичное, а не текстовое представление строки) упрощает решение этих проблем.

MQL5 API предоставляет набор функций, которые позволяют оперировать отдельными символами или их группами с учетом особенностей кодировки.

Напомним, что строки в MQL5 содержат символы в двухбайтовой кодировке Unicode. Это обеспечивает универсальную поддержку всего разнообразия национальных алфавитов в единой (но очень большой) таблице символов. Два байта позволяют закодировать 65535 элементов.

По умолчанию для символов используется тип *ushort*. Однако при необходимости строку можно конвертировать в последовательность однобайтовых символов *uchar* в конкретной языковой кодировке. Данная конвертация может сопровождаться потерей части информации (в частности, буквы, отсутствующие в локализованной таблице символов, могут "потерять" умляуты или вовсе "превратиться" в некий символ-заменитель: в зависимости от контекста он может отображаться по-разному, но обычно как '?' или "квадратик").

> Во избежание проблем с текстами, в которых могут встречаться произвольные символы, рекомендуется всегда использовать Unicode. Исключение можно сделать, если некие внешние сервисы или программы, с которыми требуется интегрировать вашу MQL-программу, не поддерживают Unicode, или если текст заведомо предназначен для хранения ограниченного набора символов (например, только числа и латинские буквы).

При конвертации в/из однобайтовые символы MQL5 API по умолчанию применяет ANSI-кодировку, зависящую от текущих настроек Windows. Однако разработчик может задать другую кодовую таблицу (см. далее функции *CharArrayToString*, *StringToCharArray*).

Примеры использования описываемых далее функций приведены в файле *StringSymbols.mq5*.

bool StringSetCharacter(string &variable, int position, ushort character)

Функция изменяет в переданной строке *variable* символ с номером *position* на значение *character*. Номер должен быть в диапазоне от 0 до длины строки (*StringLen*) минус 1.

Если записываемый символ равен 0, он задает новое окончание строки (выступает в роли терминального нуля), то есть длина строки становится равной *position*. Размер буфера, выделенный под строку, при этом не меняется.

Если параметр *position* равен длине строки и записываемый символ не равен 0, то символ добавляется к строке и её длина увеличивается на 1. Это эквивалентно выражению: *variable += ShortToString(character)*.

Функция возвращает *true* в случае успешного выполнения или *false* в случае ошибки.

```
void OnStart()
{
   string numbers = "0123456789";
   PRT(numbers);
   PRT(StringSetCharacter(numbers, 7, 0));    // обрезаем на 7-м символе
   PRT(numbers);                              // 0123456
   PRT(StringSetCharacter(numbers, StringLen(numbers), '*')); // добавляем '*'
   PRT(numbers);                              // 0123456*
   ...
}
```

ushort StringGetCharacter(string value, int position)

Функция возвращает код символа, расположенного в указанной позиции строки. Номер позиции должен лежать в пределах от 0 до длины строки (*StringLen*) минус 1. В случае ошибки функция вернет 0.

Функция эквивалентна записи с использованием оператора '[]': *value[position]*.

```
   string numbers = "0123456789";
   PRT(StringGetCharacter(numbers, 5));       // 53 = код '5'
   PRT(numbers[5]);                           // 53 - то же самое
```

string CharToString(uchar code)

Функция преобразует ANSI-код символа в односимвольную строку. В зависимости от установленной кодовой страницы Windows, верхняя половина кодов (старше 127) может генерировать отличные буквы (отличается начертание символа, код остается одним и тем же). Например, символ с кодом 0xB8 (184 в десятичной системе) обозначает седиль (нижний крючок) в западноевропейских языках, а в русском здесь располагается буква 'ё'. Или вот еще пример:

```
   PRT(CharToString(0xA9));    // "©"
   PRT(CharToString(0xE6));    // "æ", "ж", или другой знак
                               //      в зависимости от вашей локали Windows
```

string ShortToString(ushort code)

Функция преобразует Unicode-код символа в односимвольную строку. В качестве параметра *code* можно использовать литерал или целое число. Например, греческая заглавная буква "сигма" (знак суммы в математических формулах) может быть указана как 0x3A3 или 'Σ'.

```
PRT(ShortToString(0x3A3)); // "Σ"
PRT(ShortToString('Σ'));   // "Σ"
```

int StringToShortArray(const string text, ushort &array[], int start = 0, int count = -1)

Функция преобразует строку в последовательность *ushort*-кодов символов, которая копируется в указанное место массива: начиная с элемента под номером *start* (по умолчанию — 0, то есть начало массива) и в количестве *count*.

> Обратите внимание: параметр *start* относится к позиции в массиве, а не в строке. Если требуется сконвертировать часть строки, её необходимо предварительно извлечь с помощью функции *StringSubstr*.

Когда параметр *count* равен -1 (или WHOLE_ARRAY), копируются все символы до конца строки (включая терминальный ноль) или по размеру массива, если он фиксированного размера.

В случае динамического массива, он будет при необходимости автоматически увеличен в размере. Если размер динамического массива больше длины строки, то размер массива не уменьшается.

Чтобы скопировать символы без терминального нуля, следует явно указывать вызов *StringLen* в качестве аргумента *count*. В противном случае длина массива будет на 1 больше длины строки (и в последнем элементе — 0).

Функция возвращает количество скопированных символов.

```
...
ushort array1[], array2[]; // динамические массивы
ushort text[5];            // массив фиксированного размера
string alphabet = "ABCDEАБВГД";
// копируем с терминальным '0'
PRT(StringToShortArray(alphabet, array1)); // 11
ArrayPrint(array1); // 65   66   67   68   69 1040 1041 1042 1043 1044    0
// копируем без терминального '0'
PRT(StringToShortArray(alphabet, array2, 0, StringLen(alphabet))); // 10
ArrayPrint(array2); // 65   66   67   68   69 1040 1041 1042 1043 1044
// копируем в фиксированный массив
PRT(StringToShortArray(alphabet, text)); // 5
ArrayPrint(text); // 65 66 67 68 69
// копируем за прежние пределы массива
// (элементы [11-19] будут случайными)
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
*/
```

Обратите внимание, что если позиция для копирования выходит за пределы размера массива, то промежуточные элементы будут распределены, но не инициализированы. В результате в них могут оказаться случайные данные (подсвечены желтым выше).

string ShortArrayToString(const ushort &array[], int start = 0, int count = -1)

Функция преобразует часть массива с кодами символов в строку. Диапазон элементов массива задается параметрами *start* и *count*: соответственно начальной позицией и количеством. Параметр *start* должен быть в пределах от 0 до числа элементов в массиве минус 1. Когда *count* равно -1 (или WHOLE_ARRAY) копируются все элементы до конца массива или до первого нулевого.

Продолжая текущий пример из *StringSymbols.mq5*, попробуем преобразовать в строку массив *array2*, который имеет размер 30.

```
...
string s = ShortArrayToString(array2, 0, 30);
PRT(s); // "ABCDEАБВГД", здесь могут появиться дополнительные случайные символы
```

Поскольку в массив *array2* была дважды скопирована строка "ABCDEАБВГД", причем один раз — в самое начало, а второй раз — по смещению 20, промежуточные символы будут случайными и способны сформировать более длинную строку, чем получилось у нас.

int StringToCharArray(const string text, uchar &array[], int start = 0, int count = -1, uint codepage = CP_ACP)

Функция преобразует строку *text* в последовательность однобайтовых символов, которая копируется в указанное место массива: начиная с элемента под номером *start* (по умолчанию — 0, то есть начало массива) и в количестве *count*. В процессе копирования производится

конвертация символов из Unicode в выбранную кодовую страницу *codepage* — по умолчанию, CP_ACP, что означает язык операционной системы Windows (подробнее об этом — чуть ниже).

Когда параметр *count* равен -1 (или WHOLE_ARRAY), копируются все символы до конца строки (включая терминальный ноль) или по размеру массива, если он фиксированного размера.

В случае динамического массива, он будет при необходимости автоматически увеличен в размере. Если размер динамического массива больше длины строки, то размер массива не уменьшается.

Чтобы скопировать символы без терминального нуля, следует явно указывать вызов *StringLen* в качестве аргумента *count*.

Функция возвращает количество скопированных символов.

Перечень допустимых кодовых страниц для параметра *codepage* смотрите в документации. Вот некоторые из широко распространенных кодовых страниц стандарта ANSI:

Язык	**Код**
Центральноевропейский латинский	1250
Кириллица	1251
Западноевропейский латинский	1252
Греческий	1253
Турецкий	1254
Иврит	1255
Арабский	1256
Прибалтийский	1257

Таким образом, на компьютерах с западноевропейскими языками CP_ACP равна 1252, а, например, на русских — 1251.

В процессе конвертации некоторые символы могут быть преобразованы с потерей информации, поскольку таблица Unicode намного больше ANSI (в каждой таблице ANSI-кодов — 256 символов).

В связи с этим особую важность среди всех констант CP_*** имеет CP_UTF8. Она позволяет правильно сохранить национальные символы за счет кодирования с переменной длиной: результирующий массив по-прежнему хранит байты, но каждый национальный символ может занимать несколько байтов, записанных в особом формате. Из-за этого длина массива может быть существенно больше, чем длина строки. Кодировка UTF-8 широко используется в Интернете и различном программном обеспечении. Кстати говоря, UTF расшифровывается как Unicode Transformation Format, и существуют другие модификации, в частности UTF-16 и UTF-32.

Мы рассмотрим пример для *StringToCharArray* после того, как познакомимся с "обратной" функцией *CharArrayToString*: их работу необходимо демонстрировать в связке.

string CharArrayToString(const uchar &array[], int start = 0, int count = -1, uint codepage = CP_ACP)

Функция преобразует массив с байтами или его часть в строку. Массив должен содержать символы в определенной кодировке. Диапазон элементов массива задается параметрами *start* и *count*: соответственно начальной позицией и количеством. Параметр *start* должен быть в пределах от 0 до числа элементов в массиве. Когда *count* равно -1 (или WHOLE_ARRAY) копируются все элементы до конца массива или до первого нулевого.

Посмотрим, как функции *StringToCharArray* и *CharArrayToString* работают с разными национальными символами при разных настройках кодовых страниц. Для этого подготовлен тестовый скрипт *StringCodepages.mq5*.

В качестве подопытных будут использованы две строки — на русском и немецком языках:

```
void OnStart()
{
   Print("Locales");
   uchar bytes1[], bytes2[];

   string german = "straßenführung";
   string russian = "Русский Текст";
   ...
```

Мы будем их копировать в массивы *bytes1* и *bytes2*, а затем восстанавливать в строки.

Для начала преобразуем немецкий текст, применив европейскую кодовую страницу 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
```

На европейских копиях Windows это эквивалентно более простому вызову функции с параметрами по умолчанию, потому что там CP_ACP = 1252:

```
   StringToCharArray(german, bytes1);
```

Затем восстановим текст из массива с помощью следующего вызова и убедимся, что все совпадет с первоисточником:

```
   ...
   PRT(CharArrayToString(bytes1, 0, WHOLE_ARRAY, 1252));
   // CharArrayToString(bytes1,0,WHOLE_ARRAY,1252)='straßenführung'
```

Теперь попробуем преобразовать русский текст в той же европейской кодировке (или можно вызвать *StringToCharArray(russian, bytes2)* в среде Windows, где в качестве кодовой страницы по умолчанию CP_ACP стоит 1252):

```
   ...
   StringToCharArray(russian, bytes2, 0, WHOLE_ARRAY, 1252);
   ArrayPrint(bytes2);
   // 63 63 63 63 63 63 63 32 63 63 63 63 63  0
```

Здесь уже видно, что во время конвертации возникла проблема, потому что 1252 не имеет кириллицы. Восстановление строки из массива наглядно показывает суть:

```
...
PRT(CharArrayToString(bytes2, 0, WHOLE_ARRAY, 1252));
// CharArrayToString(bytes2,0,WHOLE_ARRAY,1252)='??????? ?????'
```

Повторим опыт в условной русской среде, то есть преобразуем туда и обратно обе строки с использованием кириллической кодовой страницы 1251.

```
...
StringToCharArray(russian, bytes2, 0, WHOLE_ARRAY, 1251);
// на русской Windows этот вызов эквивалентен более простому
// StringToCharArray(russian, bytes2);
// потому что CP_ACP = 1251
ArrayPrint(bytes2); // в этот раз коды символов осмысленные
// 208 243 241 241 234 232 233  32 210 229 234 241 242   0

// восстановим строку и убедимся, что она совпадает с исходной
PRT(CharArrayToString(bytes2, 0, WHOLE_ARRAY, 1251));
// CharArrayToString(bytes2,0,WHOLE_ARRAY,1251)='Русский Текст'

// и для немецкого текста...
StringToCharArray(german, bytes1, 0, WHOLE_ARRAY, 1251);
ArrayPrint(bytes1);
// 115 116 114  97  63 101 110 102 117 104 114 117 110 103   0
// если сравнить это содержимое bytes1 с предыдущим вариантом,
// легко заметить, что пара символов пострадала; вот что было:
// 115 116 114  97 223 101 110 102 252 104 114 117 110 103   0

// восстановим строку, чтобы увидеть отличия наглядно:
PRT(CharArrayToString(bytes1, 0, WHOLE_ARRAY, 1251));
// CharArrayToString(bytes1,0,WHOLE_ARRAY,1251)='stra?enfuhrung'
// испорченными оказались специфические немецкие символы
```

Таким образом, налицо хрупкость однобайтовых кодировок.

Наконец, задействуем кодировку CP_UTF8 для обеих тестовых строк. Эта часть примера будет стабильно работать вне зависимости от настроек 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'
```

Обратите внимание, что обе строки в кодировке UTF-8 потребовали более длинных массивов, чем в ANSI. Причем массив с русским текстом стал фактически в 2 раза длиннее, потому что все

буквы теперь занимают по 2 байта. Желающие могут найти в открытых источниках подробности о том, как именно устроена кодировка UTF-8. В контексте данной книги для нас важно, что MQL5 API предоставляет готовые функции для работы с ней.

4.2.7 Универсальный форматированный вывод данных в строку

При формировании строки для показа пользователю, сохранения в файл или передачи в Интернет может потребоваться включить в нее значения нескольких переменных разных типов. Эту задачу можно решить явным приведением всех переменных к типу *(string)* и сложением получившихся строк, но в этом случае инструкция MQL-кода окажется длинной и трудной для понимания. Вероятно, более удобным было бы применение функции *StringConcatenate*, но этот способ не решает проблему целиком.

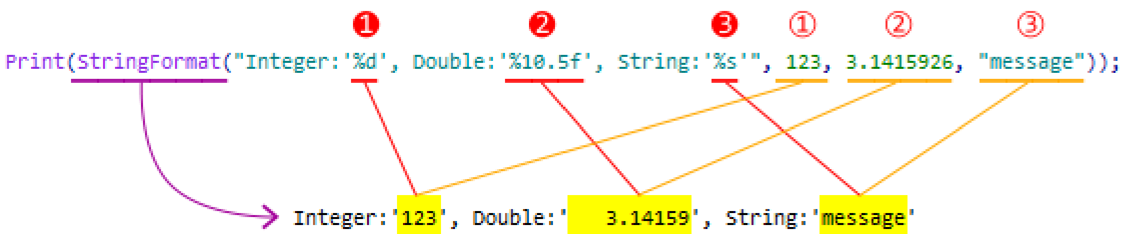
Дело в том, что строка обычно содержит не только переменные, но и некие текстовые вставки, выступающие связующими звеньями и обеспечивающие правильную структуру общего сообщения. Получается, что куски форматирующего текста перемешаны с переменными. Такой код сложно поддерживать — он противоречит одному из известных принципов программирования: отделяй представление от данных.

Для этой задачи существует специальное решение: функция *StringFormat*.

По такому же принципу действует другая функция MQL5 API: *PrintFormat*.

string StringFormat(const string format, ...)

Функция преобразует произвольные аргументы встроенных типов в строку в соответствии с заданным форматом. Первым параметром передается шаблон подготавливаемой строки, в котором особым способом указаны места вставки переменных и определен формат их вывода. Эти управляющие команды могут перемежаться с обычным текстом, который копируется в выходную строку без изменений. В последующих параметрах функции, разделенных запятыми, перечисляются все переменные в том порядке и тех типов, как для них зарезервированы места в шаблоне.



Взаимодействие форматной строки и аргументов StringFormat

Каждое место вставки переменной в строку отмечается форматным спецификатором: символом '%', после которого может указываться несколько настроек.

Строка формата анализируется слева направо. Когда в ней встречается первый спецификатор (если он есть), то значение первого параметра после строки формата преобразовывается и добавляется в результирующую строку согласно заданным настройкам. Второй спецификатор вызывает преобразование и вывод второго параметра, и так далее, до конца строки формата. Все прочие символы в шаблоне между спецификаторами копируются в результирующую строку без изменений.

Шаблон может не содержать ни одного спецификатора, то есть быть простой строкой. В таком случае в функцию нужно передать дополнительный фиктивный аргумент помимо самой строки (аргумент не будет помещен в строку).

Если в шаблоне требуется вывести знак процента, то следует написать его два раза подряд %%. Если знак % не задвоен, то несколько последующих символов после него всегда анализируется как спецификатор.

Обязательным атрибутом спецификатора является некий символ, обозначающий ожидаемый тип и интерпретацию очередного аргумента функции. Назовем этот символ условно T. Тогда в простейшем случае один спецификатор формата выглядит как %T.

В обобщенном виде спецификатор может состоять еще из нескольких полей (в квадратных скобках указаны опциональные поля):

%[Z][W][.P][M]T

Каждое поле выполняет свою функцию и принимает одно из разрешенных значений. Далее мы постепенно затронем все поля.

Тип T

Для целых чисел в качестве T могут применяться следующие символы, с пояснением как соответствующие им числа отображаются в строке:

- c — Unicode символ
- C — ANSI символ
- d, i — знаковое десятичное
- o — беззнаковое восьмеричное
- u — беззнаковое десятичное
- x — беззнаковое шестнадцатеричное (строчные буквы)
- X — беззнаковое шестнадцатеричное (заглавные буквы)

Напомним, что по способу внутреннего хранения данных к целым типам также относятся встроенные типы MQL5 *datetime*, *color*, *bool* и перечисления.

Для вещественных чисел в качестве T применимы следующие символы:

- e — научный формат с показателем (маленькая 'e')
- E — научный формат с показателем (большая 'E')
- f — обычный формат
- g — аналог f или e (выбирается наиболее компактный вид)
- G — аналог f или E (выбирается наиболее компактный вид)
- a — научный формат с показателем, шестнадцатеричное (строчные буквы)
- A — научный формат с показателем, шестнадцатеричное (заглавные буквы)

Наконец, для строк доступен всего один вариант символа T: s.

Размер целых чисел M

Для целых типов дополнительно можно явным образом указать размер переменной в байтах, если перед T поставить один из следующих знаков или их комбинаций (мы обобщили их под буквой M):

- h — 2 байта (short, ushort)
- l (маленькая L) — 4 байта (int, uint)
- I32 (большая i) — 4 байта (int, uint)
- ll (две мелких L) — 8 байт (long)
- I64 (большая i) — 8 байт (long, ulong)

Ширина W

Поле W — это неотрицательное десятичное число, которое задает минимальное количество знакомест, выделяемых под отформатированное значение. Если значение переменной укладывается в меньшее количество символов, то слева или справа добавляется соответствующее количество пробелов. Левая или правая сторона выбирается в зависимости от выравнивания (см. далее флаг '–' в поле Z). При наличии флага '0' перед выводимым значением добавляется соответствующее количество нулей. Если число выводимых символов больше заданной ширины, то настройка ширины игнорируется и выводимое значение не усекается.

Если в качестве ширины указана звездочка '*', то в списке передаваемых параметров, на предыдущей позиции перед форматируемой переменной должно находиться значение типа *int*, которое будет использовано в качестве ширины выводимого значения.

Точность P

Поле P тоже содержит неотрицательное десятичное число и всегда предваряется точкой '.'. Для целочисленных T данное поле задает минимальное количество значащих цифр. Если значение укладывается в меньшее количество цифр, оно предваряется нулями.

Для вещественных чисел P задает количество знаков в дробной части (по умолчанию — 6), за исключением спецификаторов g и G, для которых P — это общее количество значащих цифр (в мантиссе и дробной части).

Для строки P определяет количество отображаемых символов. Если длина строки превышает значение точности, то строка будет показана в усеченном виде.

Если в качестве точности указана звездочка '*', она обрабатывается по такому же принципу, как и для ширины, но управляет точностью.

Фиксированная ширина и/или точность, вместе с выравниванием по правому краю, позволяют выводить значения аккуратным столбиком.

Флаги Z

Наконец поле Z описывает флаги:

- - (минус) — выравнивание по левому краю в пределах заданной ширины (в отсутствие флага делается выравнивание по правому краю);
- + (плюс) — безусловный вывод знака '+' или '-' перед значением (без этого флага '-' отображается только для отрицательных значений);
- 0 — перед выводимым значением добавляются нули, если оно меньше заданной ширины;

- (пробел) — перед выводимым значением ставится пробел, если оно является знаковым и положительным;
- # — управляет отображением префиксов восьмеричной и шестнадцатеричной записи чисел в форматах *o*, *x* или *X* (например, для формата *x* перед выводимым числом добавляется префикс "0x", для формата *X* — префикс "0X"), десятичной точки в вещественных числах (форматы *e*, *E*, *a* или *A*) с нулевой дробной частью, и некоторыми другими нюансами.

Более подробно изучить возможности форматированного вывода в строку можно в документации.

Общее количество параметров функции не может превышать 64.

Если количество переданных в функцию аргументов больше, чем количество спецификаторов, то лишние аргументы опускаются.

Если количество спецификаторов в форматной строке больше аргументов, то система попытается вывести вместо отсутствующих данных нули, однако для строковых спецификаторв будет встроено текстовое предупреждение ("missing string parameter").

Если тип значения не совпадает с типом соответствующего спецификатора, система попытается прочитать данные из переменной в соответствии с форматом и отобразит получившуюся величину (она может выглядеть странно за счет неправильной интерпретации внутреннего битового представления реальных данных). В случае строк в результат может быть встроено предупреждение ("non-string passed").

Протестируем функцию с помощью скрипта *StringFormat.mq5*.

Сперва попробуем разные варианты спецификатора типов T и данных.

```
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));
```

Здесь представлены как правильные, так и неправильные спецификаторы (неправильные идут вторыми в каждой инструкции и помечены словом "overflow", так как передаваемое значение не умещается в типе формата).

Вот что получится в журнале (переносы длинных строк здесь и далее сделаны для публикации):

```
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'
```

Все следующие инструкции — правильные:

```
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"));
```

Результат их работы представлен ниже:

```
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'
```

Теперь рассмотрим различные модификаторы.

При выравнивании вправо (по умолчанию) и фиксированной ширине поля (количество знакомест) мы можем использовать разные варианты дополнения результирующей строки слева: пробелом или нулями. Кроме того, при любом выравнивании можно включать или отключать явное указание знака значения (чтобы выводился не только минус для отрицательных, но и плюс для положительных).

```
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));
```

Получим в журнале следующее:

```
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'
```

Для выравнивания влево необходимо применить флаг '-' (минус), дополнение строки до заданной ширины при этом происходит справа:

```
PRT(StringFormat("no sign (default): %-10i", SHORT_MAX));
PRT(StringFormat("with sign: %+-10i", SHORT_MAX));
```

Результат:

```
StringFormat(no sign (default): %-10i,SHORT_MAX)='no sign (default): 32767     '
StringFormat(with sign: %+-10i,SHORT_MAX)='with sign: +32767    '
```

При необходимости мы можем показывать или скрывать знак значения (по умолчанию выводится только минус у отрицательных значений), добавлять пробел для положительных значений и тем самым обеспечивать одинаковое форматирование, когда нужно вывести переменные столбиком:

```
PRT(StringFormat("default: %i", SHORT_MAX));  // стандарт
PRT(StringFormat("default: %i", SHORT_MIN));
PRT(StringFormat("space   : % i", SHORT_MAX)); // доп. пробел для положительного
PRT(StringFormat("space   : % i", SHORT_MIN));
PRT(StringFormat("sign    : %+i", SHORT_MAX)); // принудительно выводим знак
PRT(StringFormat("sign    : %+i", SHORT_MIN));
```

Вот как это выглядит в журнале:

```
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'
```

Теперь сравним, как ширина и точность влияют на вещественные числа.

```
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));

// точность по умолчанию = 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));
```

Результат:

```
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'
```

В отсутствие явно заданной ширины, значения выводятся без дополнения пробелами.

```
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));
```

Результат:

```
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'
```

Установление ширины и точности значений с помощью знака '*' и на основании дополнительных аргументов функции выполняется следующим образом:

```
PRT(StringFormat("double PI: %*.*f", 12, 5, M_PI));
PRT(StringFormat("string: %*s", 15, "ABCDEFGHIJ"));
PRT(StringFormat("string: %-*s", 15, "ABCDEFGHIJ"));
```

Обратите внимание, что перед выводимым значением передается 1 или 2 значения целого типа — по числу звездочек '*' в спецификаторе: вы можете управлять отдельно точностью, отдельно шириной, или тем и другим одновременно.

```
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     '
```

Наконец, рассмотрим несколько типичных ошибок форматирования.

```
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));
```

В первой инструкции спецификаторов больше, чем аргументов. В остальных случаях не совпадают типы спецификаторов и передаваемых значений. В результате получим следующий вывод:

```
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)'
```

Наличие единой форматной строки в каждом вызове функции *StringFormat* позволяет использовать её, в частности, для перевода внешнего интерфейса программ и сообщений на разные языки: достаточно загружать и подставлять в *StringFormat* различные форматные строки (подготовленные заранее) в зависимости от предпочтений пользователя или настроек терминала.

4.3 Работа с массивами

Любую программу, а в особенности — связанную с трейдингом, трудно представить без массивов. Мы уже изучили общие принципы описания и использования массивов в одноименной Главе. Они органично дополняются набором встроенных функций для работы с массивами.

Некоторые из них предоставляют готовые реализации наиболее востребованных операций над массивами, такие как поиск максимума и минимума, сортировка, вставка и удаление элементов.

Однако есть и целый ряд функций, без которых невозможно применение массивов конкретных типов. В частности, под динамический массив необходимо предварительно выделить память, прежде чем работать с ним, а массивы с данными для буферов индикаторов (этот тип MQL-программ мы изучим в 5-ой Части книги) используют особый порядок индексации элементов, устанавливаемый специальной функцией.

А начнем мы рассмотрение функций для работы с массивами с операции вывода в журнал. Она уже встречалась в предыдущих главах книги и пригодится во многих последующих.

> Поскольку массивы MQL5 могут быть многомерными (от 1 до 4 измерений), нам потребуется далее в тексте ссылаться на номера измерений. Мы будем их называть номерами, начиная с первого, что более привычно геометрически и подчеркивает тот факт, что массив обязан иметь хотя бы одну размерность (даже если он пустой). Однако элементы массивов по каждому измерению нумеруются, как принято в MQL5 (и во многих других языках программирования), с нуля. Таким образом, для массива, описанного как *array[5][10]*, первое измерение имеет размер 5, а второе — 10.

4.3.1 Вывод массивов в журнал

Вывод переменных, массивов и сообщений о статусе MQL-программы в журнал является наиболее простым средством информирования пользователя, отладки и диагностики проблем. Очевидно, что для массива можно организовать "печать" по элементам с помощью функции *Print*: она знакома нам по демонстрационным скриптам, но формально мы её опишем чуть позже — в разделе, посвященном взаимодействию программ с пользователем.

Однако удобнее переложить всю рутину по перебору элементов и их аккуратному форматированию на среду MQL5, потому что среди функций API имеется специальная для такого случая — *ArrayPrint*.

Примеры работы с этой функцией мы уже приводили в разделе Использование массивов. Теперь расскажем о её возможностях более подробно.

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)

Функция выводит в журнал массив, применяя указанные настройки. Массив должен иметь один из встроенных типов или тип простой структуры. Под простой структурой понимается структура с полями встроенных типов за исключением строк и динамических массивов. Наличие объектов классов и указателей в составе структуры выводят её из категории простой.

Массив должен иметь размерность 1 или 2. При этом форматирование автоматически подстраивается под конфигурацию массива и по возможности отображает его в наглядном виде (см. пример далее). Несмотря на то, что MQL5 поддерживает массивы размерностью вплоть до 4-х включительно, функция не отображает массивы с 3-мя и более размерностями, потому что их сложно представить в "плоском" виде. Происходит это без выдачи ошибок на стадии компиляции или выполнения программы.

Все параметры кроме первого могут быть опущены, для них определены значения по умолчанию.

Параметр *digits* применяется для массивов вещественных чисел и для числовых полей структур: он задает количество выводимых знаков в дробной части чисел. Значением по умолчанию является одна из предопределенных переменных графиков, а именно *_Digits* — это количество знаков после запятой в ценах финансового инструмента текущего графика.

Разделительный символ *separator* используется для обозначения колонок при выводе полей в массиве структур. При значении по умолчанию (NULL) функция применяет пробел в качестве разделителя.

Параметры *start* и *count* позволяют задать, соответственно, номер начального элемента и количество элементов для печати. По умолчанию функция выводит весь массив, но на результат может дополнительно оказать действие наличие флага ARRAYPRINT_LIMIT (см. ниже).

Параметр *flags* принимает сочетание флагов, которые управляют различными нюансами отображения. Вот некоторые из них:

- ARRAYPRINT_HEADER — вывод заголовка с названиями полей структуры перед массивом структур; не влияет на массивы не-структур;
- ARRAYPRINT_INDEX — вывод индексов элементов по измерениям (для одномерных массивов индексы выводятся слева, для двумерных — слева и сверху);
- ARRAYPRINT_LIMIT — для больших массивов вывод ограничен сотней первых и сотней последних записей (это ограничение включено по умолчанию);
- ARRAYPRINT_DATE — для значений типа *datetime* отображается дата;
- ARRAYPRINT_MINUTES — для значений типа *datetime* отображается время с точностью до минут;
- ARRAYPRINT_SECONDS — для значений типа *datetime* отображается время с точностью до секунд.

Значения типа *datetime* выводятся по умолчанию в формате ARRAYPRINT_DATE | ARRAYPRINT_SECONDS.

Значения типа *color* выводятся в шестнадцатеричном формате.

Значения перечислений отображаются как целые числа.

Функция не выводит вложенные массивы, структуры и указатели на объекты. Вместо них отображается троеточие.

Для демонстрации возможностей функции создан скрипт *ArrayPrint.mq5*.

В функции *OnStart* определено несколько массивов (одно-, двух- и трехмерный), которые выводятся с помощью *ArrayPrint* (с настройками по умолчанию).

```
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);
   ...
}
```

Мы получим в журнале следующие строки:

```
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
```

Массив *array1D* недостаточно большой (умещается на одном ряду), поэтому для него не показаны индексы.

Массив *array2D* имеет несколько строк (индексов), и потому их индексы отображаются (ARRAYPRINT_INDEX включен по умолчанию).

Обратите внимание, что поскольку скрипт запускался на графике EURUSD с пятизнаковыми ценами, *_Digits = 5*, и это сказывается на форматировании величин типа *double*.

Массив *array3D* проигнорирован: для него не выведено ни одной строки.

Дополнительно в скрипте определены структуры *Pair* и *SimpleStruct*:

```
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 содержит поля встроенных типов, указатель на *void*, а также поле типа *Pair*.

В функции *OnStart* создается массив типа *SimpleStruct* и выводится с помощью *ArrayPrint* в двух режимах: с настройками по умолчанию и с пользовательскими (количество цифр после "запятой" — 3, разделитель — ";", формат для *datetime* — только дата).

```
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);
}
```

Это производит следующий результат:

```
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"        ;  ...;   ...
```

Уточним, что журнал, который мы используем в данном случае и в предыдущих разделах, формируется в терминале и доступен пользователю на вкладке *Эксперты* окна *Инструменты*. Однако в дальнейшем мы познакомимся с тестером, который предоставляет для MQL-программ

некоторых типов (индикаторов и экспертов) такую же среду исполнения, как и сам терминал. В случае их запуска в тестере функция *ArrayPrint* и другие родственные функции, о которых рассказано в разделе Взаимодействие с пользователем, будут выводить сообщения в журнал агентов тестирования.

До сих пор мы работали и некоторое время продолжим работать только со скриптами, а они могут выполняться только в терминале.

4.3.2 Динамические массивы

Динамические массивы способны менять свой размер в процессе выполнения программы по запросу программиста. Напомним, что для описания динамического массива следует оставить пустой первую пару скобок после идентификатора массива. Все последующие измерения (если их больше одного) обязаны иметь фиксированный, заданный с помощью константы размер: таково требование MQL5.

Нарастить динамически количество элементов по любому измерению "старше" первого — нельзя. Кроме того, из-за "жесткого" описания размеров массивы имеют "квадратную" форму, то есть, например, невозможно сконструировать двумерный массив со столбцами или рядами разной длины. Если какое-либо из этих ограничений критично важно для реализации алгоритма, следует использовать не стандартные массивы MQL5, а собственные структуры или классы, написанные на MQL5.

Обратите внимание, что если у массива отсутствует размер по первому измерению, но при этом есть список инициализации, позволяющий определить размер, то такой массив является массивом с фиксированным размером, а не динамическим.

Например, в предыдущем разделе мы использовали массив *array1D*:

```
int array1D[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Из-за списка инициализации его размер известен компилятору, и потому массив является фиксированным.

В отличие от данного простого примера в блоках реальной программы не всегда просто определить, является ли конкретный массив динамическим. В частности, массив может быть передан как параметр внутрь функции. Вместе с тем, бывает важно знать, является ли массив динамическим, потому что только под такие массивы можно вручную выделять память, вызвав *ArrayResize*.

В подобных случаях определить тип массива позволяет функция *ArrayIsDynamic*.

Познакомимся с техническим описанием этой и других функций по работе с динамическими массивами, а затем проверим их с помощью тестового скрипта *ArrayDynamic.mq5*.

bool ArrayIsDynamic(const void &array[])

Функция проверяет, является ли переданный массив динамическим. Массив может быть любой разрешенной размерности от 1 до 4. Элементы массива могут быть любого типа.

Функция возвращает *true* для динамического массива или *false* в остальных случаях (фиксированный массив или массив с временным рядом, контролируемый самим терминалом или индикатором).

int ArrayResize(void &array[], int size, int reserve = 0)

Функция устанавливает новый размер *size* в первом измерении динамического массива *array*. Массив может быть любой разрешенной размерности от 1 до 4. Элементы массива могут быть любого типа.

Если параметр *reserve* больше нуля, память выделяется под массив с запасом на указанное количество элементов. Это имеет смысл для увеличения быстродействия программы при множестве последовательных вызовов функции. Пока новый запрашиваемый размер массива не превысит текущий с учетом резерва, физического перераспределения памяти происходить не будет, и новые элементы будут браться из резерва.

Функция возвращает новый размер массива, если его изменение прошло успешно, или -1 в случае ошибки.

Если функция применяется к фиксированному массиву или временному ряду, их размер не изменяется. В этих случаях, если запрошенный размер меньше или равен текущему размеру массива, функция вернет значение параметра *size*, а иначе — -1.

При увеличении размера уже существующего массива все данные его элементов сохраняются. Добавленные элементы ничем не инициализируются и могут содержать произвольные некорректные данные ("мусор").

Установка размера массива в 0 — *ArrayResize(array, 0)* — не освобождает реально выделенную под него память, включая и возможный резерв. Такой вызов лишь обнулит метаданные по массиву. Это сделано из соображений оптимизации будущих операций с массивом, которые нельзя исключить. Для принудительной "чистки" памяти следует использовать *ArrayFree* (см. далее).

Важно понимать, что параметр *reserve* используется не при каждом вызове функции, а только в те моменты, когда фактически выполняется перераспределение памяти, то есть когда запрашиваемый размер превышает текущую емкость массива с учетом запаса. Чтобы наглядно показать, как это работает, мы создадим макет внутреннего объекта массива (неполная копия) и реализуем для него функцию-двойник *ArrayResize*, а также, для полноты инструментария, аналоги *ArrayFree* и *ArraySize*.

```
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;
}
```

Преимущество функции *DynArrayResize* по сравнению со встроенной *ArrayResize* в том, что здесь мы вставили отладочную печать для тех моментов, когда внутренняя емкость массива перераспределяется.

Теперь мы можем взять стандартный пример для функции *ArrayResize* из документации MQL5 и заменить в ней вызовы встроенных функций на "самодельные" аналоги с префиксом "Dyn". Модифицированный результат представлен в скрипте *ArrayCapacity.mq5*.

```
void OnStart()
{
   ulong start = GetTickCount();
   ulong now;
   int   count = 0;

   DynArray<double> a;

   // быстрый вариант с резервированием памяти
   Print("--- Test Fast: ArrayResize(arr,100000,100000)");

   DynArrayResize(a, 100000, 100000);

   for(int i = 1; i <= 300000 && !IsStopped(); i++)
   {
      // установим новый размер и резерв на 100000 элементов
      DynArrayResize(a, i, 100000);
      // на круглых итерациях покажем размер массива и затраченное время
      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);

   // теперь медленный вариант без резервирования (с меньшим резервом)
   count = 0;
   start = GetTickCount();
   Print("---- Test Slow: ArrayResize(slow,100000)");

   DynArrayResize(a, 100000, 100000);

   for(int i = 1; i <= 300000 && !IsStopped(); i++)
   {
      // установим новый размер, но с резервом в 100 раз меньше: 1000
      DynArrayResize(a, i, 1000);
      // на круглых итерациях покажем размер массива и затраченное время
      if(DynArraySize(a) % 100000 == 0)
      {
         now = GetTickCount();
         count++;
         PrintFormat("%d. ArraySize(arr)=%d Time=%d ms",
            count, DynArraySize(a), (now - start));
         start = now;
      }
   }
}
```

Единственное существенное отличие: в медленном варианте вызов *ArrayResize(a, i)* заменен на более щадящий *DynArrayResize(a, i, 1000)*, то есть перераспределение запрашивается все-таки не на каждой итерации, а на каждой 1000-ой (иначе журнал "захлебнется" от сообщений).

Запустив скрипт, увидим в журнале примерно такой "тайминг" (абсолютные временные промежутки зависят от вашего компьютера, но для нас важно соотношение быстродействия с резервированием и без него):

```
--- 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
```

Выигрыш по времени существенный. Кроме того, мы видим, на каких итерациях и каким образом менялась реальная ёмкость массива (резерва).

void ArrayFree(void &array[])

Функция освобождает всю память переданного динамического массива (включая возможный резерв, установленный с помощью третьего параметра функции *ArrayResize*) и устанавливает размер его первого измерения в ноль.

В принципе, массивы в MQL5 освобождают память автоматически, когда исполнение алгоритма в текущем блоке завершается. Не важно, локально (внутри функций) или глобально описан массив, является ли он фиксированным или динамическим, — система в любом случае сама освободит память, не требуя явных действий программиста.

Таким образом, вызывать данную функцию не обязательно. Однако бывают ситуации, когда массив используется в алгоритме для повторного заполнения чем-либо с нуля, то есть требуется его освобождать перед каждым заполнением. Тогда и может пригодиться данная функция.

Имейте в виду, что если элементы массива содержат указатели на динамически распределенные объекты, функция их не удаляет: программист должен сам вызвать для них *delete* (см. пример далее).

Давайте протестируем изученные выше функции: *ArrayIsDynamic*, *ArrayResize*, *ArrayFree*.

В скрипте *ArrayDynamic.mq5* написана функция *ArrayExtend*, которая увеличивает размер динамического массива на 1 и записывает в новый элемент переданное значение.

```
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;
   }
}
```

Функция *ArrayIsDynamic* используется для проверки в условном операторе, чтобы массив модифицировался только в том случае, если он динамический. Функция *ArrayResize* позволяет изменить размер массива, а функция *ArraySize* — узнать текущий размер (она будет рассмотрена в следующем разделе).

В главной функции скрипта применим *ArrayExtend* для массивов разной категории: динамического и фиксированного.

```
void OnStart()
{
   int dynamic[];
   int fixed[10] = {}; // заполнение нулями

   PRT(ArrayResize(fixed, 0)); // предупреждение: неприменимо для фиксированного масс

   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); // предупреждение: неприменимо для фиксированного массива

   Print("Free Up");
   ArrayPrint(dynamic); // ничего не выводит
   ArrayPrint(fixed);
   ...
}
```

В строках кода, где вызываются функции, неприменимые для фиксированных массивов, компилятор выдает предупреждение "нельзя использовать для массива постоянного размера" ("cannot be used for static allocated array"). Важно отметить, что внутри функции *ArrayExtend* таких предупреждений нет, потому что в функцию может передаваться массив любой категории. Именно поэтому мы выполняем проверку с помощью *ArrayIsDynamic*.

После цикла в *OnStart* массив *dynamic* расширится до 10 и получит элементы, равные возведенным в квадрат индексам. Массив *fixed* останется заполнен нулями и не поменяет размер.

Освобождение фиксированного массива с помощью *ArrayFree* не будет иметь эффекта, а динамический массив будет фактически удален, и потому последняя попытка вывести его на печать не произведет в журнале никаких строк.

Посмотрим на результат выполнения скрипта.

```
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
```

Особый интерес вызывают динамические массивы с указателями на объекты. Определим простой класс-пустышку *Dummy* и создадим массив указателей на такие объекты.

```
class Dummy
{
};

void OnStart()
{
   ...
   Dummy *dummies[] = {};
   ArrayExtend(dummies, new Dummy());
   ArrayFree(dummies);
}
```

После расширения массива *dummy* новым указателем мы чистим его с помощью *ArrayFree*, но в журнале терминала появляются записи, свидетельствующие о том, что объект остался в памяти.

```
1 undeleted objects left
1 object of type Dummy left
24 bytes of leaked memory
```

Дело в том, что функция управляет только той памятью, которая выделена под массив. В данном случае, эта память хранила один указатель, но то, на что он указывает, массиву не принадлежит. Иными словами, если массив содержит указатели на "внешние" объекты, то о них нужно позаботиться самостоятельно. Например, так:

```
for(int i = 0; i < ArraySize(dummies); ++i)
{
   delete dummies[i];
}
```

Эту "чистку" нужно запускать до вызова *ArrayFree*.

Для сокращения записи можно использовать следующие макросы (цикл по элементам, вызов *delete* для каждого из них):

```
#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_]) }
```

Тогда удаление указателей упрощается до такой записи:

```
   ...
   CALLALL(dummies, FREE);
   ArrayFree(dummies);
```

В качестве альтернативного решения вы можете использовать класс-обертку для указателей, вроде *AutoPtr*, который мы рассмотрели в разделе Шаблоны объектных типов. Тогда массив следует описать с типом *AutoPtr*. Поскольку в массиве будут храниться объекты-обертки, а не указатели, при очистке массива автоматически вызовутся деструкторы для каждой "обертки", а из них в свою очередь освободится память указателей.

4.3.3 Измерение массивов

Одна из основных характеристик массива — это его размер, то есть общее количество элементов в нем. Важно отметить, что для многомерных массивов размер является произведением длин всех его измерений.

Для фиксированных массивов можно рассчитать их размер на стадии компиляции с помощью языковой конструкции на основе оператора sizeof:

```
sizeof(array) / sizeof(type)
```

где *array* — идентификатор, а *type* — тип массива.

Например, если в коде определен массив *fixed*:

```
int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};
```

то его размер равен:

```
int n = sizeof(fixed) / sizeof(int); // 8
```

Для динамических массивов это правило не работает, так как оператор *sizeof* всегда выдает один и тот же размер внутреннего объекта динамического массива: 52 байта.

Обратите внимание, что в функциях все параметры-массивы представлены на внутреннем уровне как объекты-обертки динамических массивов. Это сделано для того, чтобы в функцию можно было передать массив с любым способом распределения памяти, включая и фиксированный. Поэтому *sizeof(array)* будет выдавать 52 для массива-параметра, даже если через него был передан массив фиксированного размера.

Наличие "оберток" сказывается только на *sizeof*. Функция *ArrayIsDynamic* всегда правильно определяет категорию фактического аргумента, переданного через массив-параметр.

Для получения размера любого массива на стадии выполнении программы следует использовать функцию *ArraySize*.

int ArraySize(const void &array[])

Функция возвращает общее количество элементов в массиве. Размерность и типа массива может быть любой. Для одномерного массива вызов функции аналогичен *ArrayRange(array, 0)* (см. далее).

Если массив распределялся с резервом (третий параметр функции *ArrayResize*), его величина не берется в расчет.

Пока под динамический массив не выделена память с помощью *ArrayResize*, функция *ArraySize* вернет 0. Также нулевым размер становится после вызова для массива *ArrayFree*.

int ArrayRange(const void &array[], int dimension)

Функция *ArrayRange* возвращает количество элементов в указанном измерении массива. Размерность и типа массива может быть любой. Параметр *dimension* должен находиться в пределах от 0 до количества измерений массива минус 1. Индекс 0 соответствует первому измерению, индекс 1 — второму, и так далее.

Произведение всех значений *ArrayRange(array, i)* с *i*, пробегающим по всем измерениям, дает *ArraySize(array)*.

Приведем примеры для вышеописанных функций (см. файл *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 (некорректно)
   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 Инициализация и заполнение массивов

Описание массива со списком инициализации возможно только для массивов фиксированного размера. Динамические массивы можно заполнить только после выделения под них памяти функцией *ArrayResize*. Заполнение делается с помощью функций *ArrayInitialize* или *ArrayFill*. Они также пригодятся в программе, когда требуется массово заменить значения в фиксированных массивах или временных рядах.

Примеры использования функций приведены после их описания.

int ArrayInitialize(type &array[], type value)

Функция устанавливает всем элементам массива указанное значение. Поддерживаются только массивы встроенных числовых типов (*char, uchar, short, ushort, int, uint, long, ulong, bool, color, datetime, float, double*). Массивы строк, структур, указателей так заполнить нельзя: для них потребуется реализовать собственные функции инициализации. Массив может быть многомерным.

Функция возвращает количество элементов.

Если динамический массив распределен с резервом (третий параметр функции *ArrayResize*), то резерв не инициализируется.

Если после инициализации массива его размер будет увеличен с помощью *ArrayResize*, добавленные элементы не установятся автоматически в значение *value*. Их можно заполнить с помощью функции *ArrayFill*.

void ArrayFill(type &array[], int start, int count, type value)

Функция заполняет числовой массив или его часть указанным значением. Часть массива задается параметрами *start* и *count*, которые обозначают, соответственно, начальный номер элемента и количество элементов, подлежащих заполнению.

Для функции не имеет значения, установлен ли порядок нумерации элементов массива как в таймсерии или нет: данное свойство игнорируется. Иными словами, элементы массива всегда считаются от его начала к концу.

Для многомерного массива параметр *start* может быть получен путем пересчета координат по измерениям в сквозной индекс для эквивалентного одномерного массива. Так, для двумерного массива в памяти сперва располагаются элементы с 0-м индексом по первому измерению, потом элементы с индексом 1 по первому измерению, и так далее. Формула для вычисления *start* выглядит так:

```
start = D1 * N2 + D2
```

где D1 и D2 — соответственно, индексы по первому и второму измерению, N2 — количество элементов по второму измерению. D2 меняется от 0 до (N2-1), D1 — от 0 до (N1-1). Например, в массиве *array[3][4]* элемент с индексами [1][3] является седьмым по счету, и потому вызов *ArrayFill(array, 7, 2, ...)* заполнит два элемента: *array[1][3]* и следующий за ним *array[2][0]*. На схеме это можно изобразить следующим образом (в каждой ячейке приведен сквозной индекс элемента):

```
      [][0]  [][1]  [][2]  [][3]
[0][]    0      1      2      3
[1][]    4      5      6      7
[2][]    8      9     10     11
```

В скрипте *ArrayFill.mq5* представлены примеры использования функций.

```
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);
}
```

Вот как выглядит возможный результат (случайные данные в неинициализированных элементах динамического массива будут отличаться):

```
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
```

4.3.5 Копирование и редактирование массивов

В данном разделе мы узнаем, как с помощью встроенных функций можно вставлять и удалять элементы массивов, менять их порядок, а также копировать массивы целиком.

bool ArrayInsert(void &target[], const void &source[], uint to, uint from = 0, uint count = WHOLE_ARRAY)

Функция вставляет в массив-приемник *target* указанное количество элементов из массива-источника source. Место вставки в массив *target* задается индексом в параметре *to*. Начальный индекс элемента, с которого начинается копирование из массива *source*, задается индексом *from*. Константа WHOLE_ARRAY ((uint)-1) в параметре *count* задает перенос всех элементов массива-источника.

Все индексы и количество относятся к первому измерению массивов. Иными словами, для многомерных массивов вставка производится не отдельными элементами, а всей конфигурацией, описанной "высшими" размерностями. Например, для двумерного массива значение 1 в параметре *count* означает вставку вектора длиной, равной второй размерности (см. пример).

Из этого следует, что конфигурации массива-приемника и массива-источника должны быть одинаковыми (иначе возникнет ошибка, и копирования не произойдет). Для одномерных массивов это не является ограничением, но для многомерных необходимо соблюдать равенство размеров по измерениям выше первого. В частности, элементы из массива [][4] нельзя вставить в массив [][5] и наоборот.

Функция применима только для массивов фиксированного или динамического размера. Изменить таймсерии (массивы с временными рядами) с помощью данной функции нельзя. Также не допускается указывать в параметрах *target* и *source* один и тот же массив.

При вставке в фиксированный массив новые элементы сдвигают имевшиеся элементы вправо и вытесняют *count* самых правых элементов за пределы массива. Параметр *to* должен быть в диапазоне от 0 до размера массива за вычетом 1.

При вставке в динамический массив также происходит сдвиг вправо старых элементов, но они не пропадают, потому что сам массив расширяется на *count* элементов. Параметр *to* должен лежать в пределах от 0 до размера массива. Если он равен размеру массива, новые элементы добавляются в конец массива.

Указанные элементы копируются из одного массива в другой, то есть они остаются в исходном массиве без изменений, а их "двойники" в новом массиве становятся самостоятельными экземплярами, никак не связанными с "оригиналами".

Функция возвращает *true* в случае успеха или *false* в случае ошибки.

Рассмотрим некоторые примеры (*ArrayInsert.mq5*). В функции *OnStart* описано несколько массивов разной конфигурации, фиксированных и динамических.

```
#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};
   ...
```

Предварительно для удобства введен макрос, выводящий код ошибки (получаемый через функцию *GetLastError*) сразу после вызова тестируемой инструкции — PRTS. Это слегка модифицированная версия знакомого нам макроса PRT.

Попытки копирования элементов между массивами разной конфигурации заканчиваются ошибкой 4006 (ERR_INVALID_ARRAY).

```
   // нельзя смешивать массивы 1D и 2D
   PRTS(ArrayInsert(dynamic, fixed, 0)); // false:4006, ERR_INVALID_ARRAY
   ArrayPrint(dynamic); // пусто
   // нельзя смешивать 2D массивы разных конфигураций по второй размерности
   PRTS(ArrayInsert(dynamic2Dx5, fixed, 0)); // false:4006, ERR_INVALID_ARRAY
   ArrayPrint(dynamic2Dx5); // пусто
   // даже если оба массива фиксированные (или оба динамические),
   // размер по "высшим" размерностям должен совпадать
   PRTS(ArrayInsert(fixed, insert, 0)); // false:4006, ERR_INVALID_ARRAY
   ArrayPrint(fixed); // не изменен
   ...
```

Целевой индекс должен находиться в пределах массива.

```
   // целевой индекс 10 выходит за пределы массива 'insert',
   // мог быть 0, 1, 2, т.к. его размер = 3
   PRTS(ArrayInsert(insert, array, 10)); // false:5052, ERR_SMALL_ARRAY
   ArrayPrint(insert); // не изменен
   ...
```

Далее идут успешные модификации массивов:

```
    // копируется второй ряд из 'fixed', 'dynamic2Dx4' распределяется
    PRTS(ArrayInsert(dynamic2Dx4, fixed, 0, 1, 1)); // true
    ArrayPrint(dynamic2Dx4);
    // оба ряда из 'fixed' добавляются в конец 'dynamic2Dx4', он расширяется
    PRTS(ArrayInsert(dynamic2Dx4, fixed, 1)); // true
    ArrayPrint(dynamic2Dx4);
    // под 'dynamic' выделяется память для всех элементов 'insert'
    PRTS(ArrayInsert(dynamic, insert, 0)); // true
    ArrayPrint(dynamic);
    // 'dynamic' расширяется на 1 элемент
    PRTS(ArrayInsert(dynamic, array, 1)); // true
    ArrayPrint(dynamic);
    // новый элемент вытолкнет последний из 'insert'
    PRTS(ArrayInsert(insert, array, 1)); // true
    ArrayPrint(insert);
}
```

Вот, что появится в журнале:

```
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)

Функция копирует часть или весь массив *source* в другой массив *target*. Место в массиве *target*, куда записываются элементы, задается индексом в параметре *to*. Начальный индекс элемента, с которого начинается копирование из массива *source*, задается индексом *from*. Константа WHOLE_ARRAY (-1) в параметре *count* задает перенос всех элементов массива-источника. Если *count* меньше нуля или больше числа элементов, оставшихся от позиции *from* до конца массива *source*, копируется весь остаток массива.

В отличие от функции *ArrayInsert*, функция *ArrayCopy* не сдвигает имеющиеся элементы приемного массива, а записывает новые элементы в указанные позиции поверх старых.

Все индексы и количество элементов задаются с учетом сквозной нумерацией элементов вне зависимости от количества измерений в массивах и их конфигурации. Иными словами, элементы можно копировать из многомерных массивов в одномерные и обратно или между многомерными массивами с разными размерами по "высшим" размерностям (см. пример).

Функция работает с фиксированными и динамическими массивами, а также массивами временных рядов, назначенными в качестве индикаторных буферов.

Допустимо копировать элементы из массива в самого себя. Но если области *target* и *source* перекрываются, нужно иметь в виду, что обход делается слева направо.

Динамический массив-приемник при необходимости автоматически расширяется. Фиксированные массивы сохраняют свои размеры, причем копируемое должно уместиться в массиве, иначе возникнет ошибка.

Поддерживаются массивы встроенных типов и массивы структур с полями простых типов. При этом для числовых типов функция попытается выполнить конвертацию, если типы источника и приемника различаются.

Строковый массив можно скопировать только в строковый массив.

Объекты классов запрещены, однако можно копировать указатели на объекты.

Функция возвращает количество скопированных элементов (0 в случае ошибки).

В скрипте *ArrayCopy.mq5* приведено несколько примеров использования функции.

```
class Dummy
{
   int x;
};

void OnStart()
{
   Dummy objects1[5], objects2[5];
   // ошибка: структуры или классы с объектами не разрешены
   PRTS(ArrayCopy(objects1, objects2));
   ...
```

Массивы с объектами генерируют ошибку компиляции о том, что такие элементы не поддерживаются ("structures or classes containing objects are not allowed"). Зато можно копировать указатели.

```
   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]);
   }
   // выведет некие попарно одинаковые дескрипторы объектов
   /*
   0 1048576 1048576
   1 2097152 2097152
   2 3145728 3145728
   3 4194304 4194304
   4 5242880 5242880
   */
```

Массивы структур с полями простых типов также копируются без проблем.

```
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
   */
   ...
```

Для дальнейшей демонстрации работы с массивами разных типов и конфигураций определены следующие массивы (среди них есть фиксированные, динамические, с разным числом измерений):

```
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"};
...
```

При копировании одного элемента из массива *fixed* из позиции 1 (число 2) в приемном динамическом массиве *dynamic2Dx4* выделяется целый ряд из 4 элементов, и поскольку копируется только 1 элемент, три оставшихся будут содержать случайный "мусор" (подсвечен желтым).

```
PRTS(ArrayCopy(dynamic2Dx4, fixed, 0, 1, 1)); // 1 / status:0
ArrayPrint(dynamic2Dx4);
/*
    [,0][,1][,2][,3]
[0,]    2    1    2    3
*/
...
```

Далее копируем все элементы из массива *fixed*, начиная с третьего, в тот же массив *dynamic2Dx4*, но уже начиная с позиции 1. Поскольку копируется 5 элементов (общее количество в массиве *fixed* 8 минус начальная позиция 3), а помещаются они по индексу 1, всего в приемном массиве будет занято 1 + 5, итого 6 элементов. И поскольку массив *dynamic2Dx4* имеет 4 элемента в каждом ряду (во втором измерении), выделить память под него можно только для числа элементов кратного 4-м, то есть будет распределено еще 2 элемента, в которых останутся случайные данные.

```
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
*/
```

При копировании многомерного массива в одномерный, элементы будут представлены в "плоском" виде.

```
PRTS(ArrayCopy(dynamic, fixed)); // 8 / status:0
ArrayPrint(dynamic);
/*
1 2 3 4 5 6 7 8
*/
```

При копировании одномерного массива в многомерный, элементы "раскладываются" по измерениям приемного массива.

```
PRTS(ArrayCopy(dynamic2Dx5, insert)); // 3 / status:0
ArrayPrint(dynamic2Dx5);
/*
    [,0][,1][,2][,3][,4]
[0,]  10  11  12   4   5
*/
```

В данном случае копировалось 3 элемента и они уместились в один ряд длиной 5 элементов (согласно конфигурации приемного массива). Память под два оставшихся элемента ряда была выделена, но не заполнена (содержит "мусор").

Мы можем перезаписать массив *dynamic2Dx5* из другого источника, в том числе и из многомерного массива другой конфигурации. Поскольку в приемном массиве было выделено два ряда по 5 элементов, а в источнике было 2 ряда по 4 элемента, 2 дополнительных элемента осталось незаполненными.

```
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
*/
```

С помощью *ArrayCopy* можно менять элементы в фиксированных массивах-приемниках.

```
PRTS(ArrayCopy(fixed, insert)); // 3 / status:0
ArrayPrint(fixed);
/*
    [,0][,1][,2][,3]
[0,]  10  11  12   4
[1,]   5   6   7   8
*/
```

Здесь мы перезаписали первые три элемента массива *fixed*. А затем перезапишем 3 последних.

```
PRTS(ArrayCopy(fixed, insert, 5)); // 3 / status:0
ArrayPrint(fixed);
/*
    [,0][,1][,2][,3]
[0,]  10  11  12   4
[1,]   5  10  11  12
*/
```

Копировать в позицию, равную длине фиксированного массива, не получится (динамический массив-приемник при этом расширился бы).

```
PRTS(ArrayCopy(fixed, insert, 8)); // 4006, ERR_INVALID_ARRAY
ArrayPrint(fixed); // без изменений
```

Строковые массивы в сочетании с массивами других типов выдадут ошибку:

```
PRTS(ArrayCopy(texts, insert)); // 5050, ERR_INCOMPATIBLE_ARRAYS
ArrayPrint(texts); // пусто
```

Но между строковыми массивами копирование возможно:

```
PRTS(ArrayCopy(texts, message));
ArrayPrint(texts); // "ok"
```

Массивы разных числовых типов копируются с необходимой конвертацией.

```
PRTS(ArrayCopy(insert, array, 1)); // 1 / status:0
ArrayPrint(insert); // 10  3 12
```

Здесь мы записали число Пи в целочисленный массив, и потому получили значение 3 (оно заменило 11).

bool ArrayRemove(void &array[], uint start, uint count = WHOLE_ARRAY)

Функция удаляет из массива указанное количество элементов начиная с индекса *start*. Массив может быть многомерным и иметь любой встроенный тип или тип структуры с полями встроенных типов, за исключением строк.

Индекс *start* и количество *count* относятся к первому измерению массивов. Иными словами, для многомерных массивов удаление производится не отдельными элементами, а всей конфигурацией, описанной "высшими" размерностями. Например, для двумерного массива значение 1 в параметре *count* означает удаление целого ряда длиной, равной второй размерности (см. пример).

Значение *start* должно лежать в пределах от 0 до размера первого измерения минус 1.

Функцию нельзя применять к массивам с временными рядами (встроенным таймсериям или буферам индикаторов).

Для проверки работы функции подготовлен скрипт *ArrayRemove.mq5*. В нем, в частности, определены 2 структуры:

```
struct Simple
{
   int x;
};

struct NotSoSimple
{
   int x;
   string s; // поле типа string заставляет компилятор сделать неявный деструктор
};
```

Массивы с простой структурой могут обрабатываться функцией *ArrayRemove* успешно, в то время как массивы объектов с деструкторами (даже с неявными, как в *NotSoSimple*) вызывают ошибку:

```
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
   ...
```

Далее определены и проинициализированы массивы различной конфигурации.

```
   int dynamic[];
   int dynamic2Dx4[][4];
   int fixed[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};

   // делаем 2 копии
   ArrayCopy(dynamic, fixed);
   ArrayCopy(dynamic2Dx4, fixed);

   // показываем исходные данные
   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
   */
```

При удалении из фиксированного массива, все элементы, находящиеся после удаляемого фрагмента, сдвигаются влево. Важно, что размер массива при этом не меняется и потому копии сдвинутых элементов оказываются в двух экземплярах.

```
   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
   */
```

Здесь мы удалили один элемент первого измерения двумерного массива *fixed* по смещению 0, то есть начальный ряд. Элементы следующего ряда сдвинулись вверх и остались в прежнем ряду.

Если проделать ту же операцию с динамическим массивом (идентичным по содержимому массиву *fixed*), его размер будет автоматически уменьшен на количество удаленных элементов.

```
PRTS(ArrayRemove(dynamic2Dx4, 0, 1));
ArrayPrint(dynamic2Dx4);
/*
ArrayRemove(dynamic2Dx4,0,1)=true / status:0
    [,0][,1][,2][,3]
[0,]   5   6   7   8
*/
```

В одномерном массиве каждый удаляемый элемент соответствует одиночному значению. Например, в массиве *dynamic* при удалении трех элементов начиная с индекса 2, получим следующий результат:

```
PRTS(ArrayRemove(dynamic, 2, 3));
ArrayPrint(dynamic);
/*
ArrayRemove(dynamic,2,3)=true / status:0
1 2 6 7 8
*/
```

Значения 3, 4, 5 были удалены, размер массива сокращен на 3.

bool ArrayReverse(void &array[], uint start = 0, uint count = WHOLE_ARRAY)

Функция изменяет порядок следования указанных элементов в массиве на обратный. Переворачиваемые элементы определяются по начальной позиции *start* и количеству *count*. Если *start = 0*, а *count = WHOLE_ARRAY*, обращается весь массив целиком.

Поддерживаются массивы произвольной размерности и типов, как фиксированные, так и динамические (включая временные ряды в буферах индикаторов). Массив может содержать объекты, указатели или структуры.

Для многомерных массивов разворот производится только по первому измерению.

Значение *count* должно быть в диапазоне от 0 до количества элементов в первом измерении. Учтите, что *count* меньше 2 не даст заметного эффекта, но это можно использовать в целях унификации циклов в алгоритмах.

Функция возвращает *true* в случае успеха или *false* в случае ошибки.

Для проверки функции был написан скрипт *ArrayReverse.mq5*. В его начале определен класс для порождения объектов, хранимых в массиве. Наличие строк и прочих "сложных" полей — не помеха.

```
class Dummy
{
   static int counter;
   int x;
   string s; // поле типа string заставляет компилятор создать неявный деструктор
public:
   Dummy() { x = counter++; }
};

static int Dummy::counter;
```

Объекты идентифицируют по порядковому номеру (присваивается в момент создания).

```
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
   */
```

После применения *ArrayReverse* получим ожидаемый обратный порядок объектов.

```
   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
   */
```

Далее подготавливаются числовые массивы разной конфигурации и разворачиваются с разными параметрами.

```
    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)); // ничего не делает (count = 1)
    PRTS(ArrayReverse(dynamic2Dx4, 2, 1)); // false / status:5052, ERR_SMALL_ARRAY
}
```

В последнем случае значение *start* (2) превышает размер в первом измерении, поэтому возникает ошибка.

4.3.6 Перемещение (обмен) массивов

MQL5 позволяет экономным образом (без физического выделения памяти и копирования данных) обменивать содержимое двух массивов между собой. В некоторых других языках программирования похожая операция поддерживается не только для массивов, но и других переменных, и называется перемещением (move).

bool ArraySwap(void &array1[], void &array2[])

Функция обменивает между собой содержимое двух динамических массивов одного типа. Поддерживаются массивы любых типов. Однако функция неприменима к массивам таймсерий и буферам индикаторов, а также любым массивам с модификатором *const*.

Для многомерных массивов количество элементов во всех измерениях кроме первого должно совпадать.

Функция возвращает *true* в случае успеха или *false* в случае ошибки.

Основная область применения функции — ускорение работы программы за счет исключения физического копирования массива, когда он передается в функцию или возвращается из неё, причем известно, что массив-источник больше не потребуется. Дело в том, что обмен происходит практически моментально, за счет того, что прикладные данные никоим образом не перемещаются. Вместо этого происходит обмен мета-данными о массивах, хранящихся в служебных структурах, описывающих динамические массивы (а это всего лишь 52 байта).

Представим себе некий класс, предназначенный для обработки массива определенными алгоритмами. Один и тот же массив может быть подвергнут разным операциям и потому его имеет смысл сохранить в качестве члена класса. Но встает вопрос, как его передать в объект? Методы (как и вообще функции) в MQL5 позволяют передавать массивы только по ссылке. Если отставить в сторону всевозможные классы-обертки, содержащие массив и передаваемые по указателю, единственным простым решением кажется следующее: описать, например, в конструкторе класса массив-параметр и копировать его во внутренний массив с помощью *ArrayCopy*. Но более эффективным является как раз использование *ArraySwap*.

```
template<typename T>
class Worker
{
   T array[];

public:
   Worker(T &source[])
   {
      // ArrayCopy(array, source); // расход памяти и времени
      ArraySwap(source, array);
   }
   ...
};
```

Поскольку массив *array* до обмена был пустым, после операции пустым станет массив, использованный в качестве аргумента *source*. Зато *array* окажется заполненным входными данными практически без накладных расходов.

После того как "владельцем" массива становится объект класса, мы можем его модифицировать требуемыми алгоритмами, предположим через специальный метод *process*, который принимает в качестве параметра код запрашиваемого алгоритма. Это может быть сортировка, сглаживание, перемешивание, добавление шума и многое другое. Но для начала попробуем проверить идею на простой операции разворота массива функцией *ArrayReverse* (см. файл *ArraySwapSimple.mq5*).

```
bool process(const int mode)
{
   if(ArraySize(array) == 0) return false;
   switch(mode)
   {
   case -4:
      // например: перемешивание
      break;
   case -3:
      // например: логарифмирование
      break;
   case -2:
      // например: добавление шума
      break;
   case -1:
      ArrayReverse(array);
      break;
   ...
   }
   return true;
}
```

Предоставить доступ к результатам работы можно с помощью двух методов: поэлементно (перегрузив оператор '[]') или целиком массивом (в соответствующем методе *get* опять используем *ArraySwap*, но можно предусмотреть и метод копирования через *ArrayCopy*).

```
T operator[](int i)
{
   return array[i];
}

void get(T &result[])
{
   ArraySwap(array, result);
}
```

С целью универсальности класс сделан шаблонным. Это позволит адаптировать его в перспективе для массивов произвольных структур, а пока можно проверить обращение простого массива типа *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);  // обращение массива

   double res[];
   simple.get(res);
   ArrayPrint(res); // 3.00000 2.00000 1.00000
}
```

Более приближенной к реальности является задача сортировки, причем для массива структур может потребоваться сортировка по любому полю. В следующем разделе мы досконально изучим функцию *ArraySort*, которая позволяет сортировать по возрастанию массив любого встроенного типа, но не структур. Там мы и попробуем ликвидировать этот "пробел", оставив в действии *ArraySwap*.

4.3.7 Сравнение, сортировка и поиск в массивах

MQL5 API содержит несколько функций, позволяющих сравнивать и сортировать массивы, а также искать в них максимальное, минимальное или какое-либо конкретное значение.

int ArrayCompare(const void &array1[], const void &array2[], int start1 = 0, int start2 = 0, int count = WHOLE_ARRAY)

Функция возвращает результат сравнения двух массивов встроенных типов или структур с полями встроенных типов за исключением строк. Массивы объектов классов не поддерживаются. Также нельзя сравнивать массивы структур, в которых есть динамические массивы, объекты классов или указатели.

По умолчанию сравнение происходит для всех массивов целиком, но при необходимости можно указать фрагменты массивов, для чего предназначены параметры *start1* (начальная позиция в первом массиве), *start2* (начальная позиция во втором массиве) и *count*.

Массивы могут быть фиксированными или динамическими, а также многомерными. Многомерные массивы в процессе сравнения представляются как эквивалентные одномерные массивы (например, для двумерных массивов элементы второго ряда идут следом за элементами первого, элементы третьего ряда — за вторым и так далее). В связи с этим параметры *start1*, *start2* и *count* обозначают для многомерных массивов сквозную нумерацию элементов, а не индекс по первому измерению.

Пользуясь разными смещениями *start1* и *start2*, можно сравнивать разные части одного и того же массива.

Массивы сравниваются поэлементно, до первого расхождения или по достижении конца одного из массивов. Соотношение между двумя элементами (находящимися на одинаковых позициях в

обоих массивах) зависит от типа: для чисел используются операторы '>', '<', '==', а для строк — функция *StringCompare*. Структуры сравниваются побайтово, то есть это эквивалентно выполнению для каждой пары элементов следующего кода:

```
uchar bytes1[], bytes2[];
StructToCharArray(array1[i], bytes1);
StructToCharArray(array2[i], bytes2);
int cmp = ArrayCompare(bytes1, bytes2);
```

На основании соотношения первых различающихся элементов получается результат сравнения массивов *array1* и *array2* целиком. Если отличий не найдено и длина массивов равна, то массивы считаются одинаковыми. Если длина отличается, то более длинный массив считается больше.

Функция возвращает -1 — если *array1* "меньше" *array2*, +1 — если *array1* "больше" *array2*, 0 если они "равны".

В случае ошибки результат равен -2.

Рассмотрим некоторые примеры в скрипте *ArrayCompare.mq5*.

Прежде всего создадим простую структуру для наполнения массивов, подлежащих сравнению.

```
struct Dummy
{
   int x;
   int y;

   Dummy()
   {
      x = rand() / 10000;
      y = rand() / 5000;
   }
};
```

Поля класса заполняются случайными числами (при каждом запуске скрипта будем получать новые значения).

В функции *OnStart* опишем небольшой массив структур и сравним последовательные элементы друг с другом (как сдвигающиеся соседние фрагменты массива длиной 1 элемент).

```
#define LIMIT 10

void OnStart()
{
   Dummy a1[LIMIT];
   ArrayPrint(a1);

   // попарное сравнение соседних элементов
   // -1: [i] < [i + 1]
   // +1: [i] > [i + 1]
   for(int i = 0; i < LIMIT - 1; ++i)
   {
      PRT(ArrayCompare(a1, a1, i, i + 1, 1));
   }
   ...
```

Ниже приведены результаты для одного из вариантов массива (для удобства анализа колонка с признаками "больше"(+1)/"меньше"(-1) добавлена непосредственно справа от содержимого массива):

```
    [x] [y]   // результат
[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
```

Сравнение двух половин массива между собой дает -1:

```
// сравнение первой и второй половины
PRT(ArrayCompare(a1, a1, 0, LIMIT / 2, LIMIT / 2)); // -1
```

Далее проведем сравнение массивов строк с предопределенными данными.

```
string s[] = {"abc","456","$"};
string s0[][3] = {{"abc","456","$"}};
string s1[][3] = {{"abc","456",""}};
string s2[][3] = {{"abc","456"}}; // последний элемент опущен: он равен null
string s3[][2] = {{"abc","456"}};
string s4[][2] = {{"aBc","456"}};

PRT(ArrayCompare(s0, s));  // s0 == s, 1D и 2D массивы содержат одинаковые данные
PRT(ArrayCompare(s0, s1)); // s0 > s1, т.к. "$" > ""
PRT(ArrayCompare(s1, s2)); // s1 > s2, т.к. "" > null
PRT(ArrayCompare(s2, s3)); // s2 > s3, из-за разной длины: [3] > [2]
PRT(ArrayCompare(s3, s4)); // s3 < s4, т.к. "abc" < "aBc"
```

Наконец, проверим соотношение фрагментов массивов:

```
PRT(ArrayCompare(s0, s1, 1, 1, 1)); // вторые элементы (с индексом 1) равны
PRT(ArrayCompare(s1, s2, 0, 0, 2)); // два первых элемента равны
```

bool ArraySort(void &array[])

Функция сортирует числовой массив (в том числе, возможно многомерный) по первому измерению. Порядок сортировки — по возрастанию. Если требуется сортировка по убыванию, примените к результирующему массиву функцию *ArrayReverse* или обходите его в обратном порядке.

Функция не поддерживает массивы строк, структур или классов.

Функция возвращает *true* в случае успеха или *false* в случае ошибки.

Если для массива установлено свойство "таймсерии" (временного ряда), то индексация элементов в нем ведется в обратном порядке (см. подробности в разделе Направление индексации массивов как в таймсерии), и это оказывает "внешний" разворотный эффект на порядок сортировки: при прямом обходе такого массива вы получите убывающие значения. На физическом уровне массив всегда сортируется по возрастанию значений, и именно так и хранится.

В скрипте *ArraySort.mq5* генерируется двумерный массив размером 10 на 3 и затем сортируется с помощью *ArraySort*:

```
#define LIMIT 10
#define SUBLIMIT 3

void OnStart()
{
   // генерируем случайные данные
   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);     // исходный массив

   PRTS(ArraySort(array));

   Print("After sort");
   ArrayPrint(array);     // упорядоченный массив
   ...
}
```

По выводу в журнал легко убедиться, что первая колонка отсортирована по возрастанию (конкретные числа будут меняться из-за случайной генерации):

```
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
```

Значения в следующих колонках переместились синхронно с "ведущими" показателями в первой колонке. Иными словами, перестановке подвергаются ряды целиком, несмотря на то, что критерием сортировки выступает только первая колонка.

Что делать, если требуется сортировать двумерный массив по колонке, отличной от первой? Можно написать алгоритм самостоятельно. Один из вариантов включен в файл *ArraySort.mq5* как шаблонная функция:

```
template<typename T>
bool ArraySort(T &array[][], const int column)
{
   if(!ArrayIsDynamic(array)) return false;

   if(column == 0)
   {
      return ArraySort(array); // стандартная функция
   }

   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);
}
```

Данная функция работает только с динамическими массивами, поскольку размер *array* удваивается для сборки промежуточных результатов во второй половине массива, а в завершении первая половина (исходная) удаляется с помощью *ArrayRemove*. Именно поэтому исходный тестовый массив в функции *OnStart* распределялся через *ArrayResize*.

С принципом сортировки предлагается разобраться самостоятельно (или перевернуть пару страниц).

Для массивов с большим числом измерений (например, *array[][][]*) потребуется реализовать что-то аналогичное.

Теперь вспомним, что в предыдущем разделе мы поднимали вопрос о том, чтобы отсортировать массив структур по произвольному полю. Как мы знаем, стандартная *ArraySort* этого не умеет.

Попробуем придумать "обходной маневр". За основу возьмем класс из файла *ArraySwapSimple.mq5* из предыдущего раздела. Скопируем его в *ArrayWorker.mq5* и кое-что добавим.

В методе *Worker::process* предусмотрим вызов вспомогательного метода сортировки *arrayStructSort*, причем сортируемое поле будем задавать по номеру (как это возможно, расскажем чуть ниже):

```
   ...
   bool process(const int mode)
   {
      ...
      switch(mode)
      {
      ...
      case -1:
         ArrayReverse(array);
         break;
      default: // сортировка по полю номер 'mode'
         arrayStructSort(mode);
         break;
      }
      return true;
   }

private:
   bool arrayStructSort(const int field)
   {
      ...
   }
```

Теперь становится понятно, почему все предыдущие режимы (значения параметра *mode*) в методе *process* были отрицательными: нулевое и положительные значения зарезервированы для сортировки — они соответствуют номеру "колонки".

Идея сортировки массива структур взята из сортировки двумерного массива. Необходимо лишь каким-то образом отобразить одиночную структуру на одномерный массив (олицетворяющий ряд двумерного массива). Для этого сперва нужно определиться, какого типа должен быть массив.

Поскольку класс *Worker* уже является шаблонным, логично добавить еще один параметр в шаблон, чтобы тип массива можно было гибко задавать.

```
template<typename T,typename R>
class Worker
{
   T array[];
   ...
```

После этого уместно вспомнить про объединения, которые позволяют наложить переменные разных типов друг на друга. Таким образом, мы приходим к такой хитрой конструкции:

```
union Overlay
{
   T r;
   R d[sizeof(T) / sizeof(R)];
};
```

В этом объединении тип структуры совмещен с массивом типа R, причем его размер автоматически рассчитывается компилятором на основе соотношения размеров двух типов T и R.

Внутри метода *arrayStructSort* теперь можно частично продублировать код из сортировки двумерного массива.

```
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;
   }
   ...
```

Вместо массива с исходными структурами мы подготавливаем массив *temp[][2]* типа R, расширяем его до количества записей в *array*, и в цикле записываем по 0-му индексу каждого ряда "отображение" требуемого поля *field* из структуры, а по 1-му индексу — исходный индекс этого элемента.

"Отображение" основывается на том, что поля в структурах обычно каким-то образом выровнены, поскольку используют стандартные типы. Поэтому при правильно подобранном типе R можно обеспечить полное или частичное попадание полей в элементы массива в "оверлее".

Например, в стандартной структуре *MqlRates* первые 6 полей имеют размер 8 байтов и потому корректно накладываются на массив *double* или *long* (это кандидаты на шаблонный тип R).

```
struct MqlRates
{
   datetime time;
   double   open;
   double   high;
   double   low;
   double   close;
   long     tick_volume;
   int      spread;
   long     real_volume;
};
```

С двумя последними полями дело обстоит сложнее. Если до поля *spread* еще можно добраться, используя тип *int* в качестве R, то поле *real_volume* оказывается по смещению, некратному его

собственному размеру (из-за поля типа *int*, то есть 4 байта, перед ним). Это издержки конкретного метода. Его можно улучшить или изобрести другой.

Но вернемся к алгоритму сортировки. После заполнения массива *temp* его можно сортировать обычной функцией *ArraySort*, а потом использовать исходные индексы, чтобы сформировать новый массив с правильным порядком структур.

```
      ...
      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);
   }
```

Перед выходом из функции мы снова используем *ArraySwap*, чтобы экономным способом подменить содержимое внутриобъектного массива *array* на новое и упорядоченное — полученное в локальном массиве *result*.

Проверим класс *Worker* в действии: в функции *OnStart* определим массив структур *MqlRates* и запросим у терминала несколько тысяч записей.

```
#define LIMIT 5000

void OnStart()
{
   MqlRates rates[];
   int n = CopyRates(_Symbol, _Period, 0, LIMIT, rates);
   ...
```

Функция *CopyRates* будет описана в отдельном разделе. Пока нам достаточно знать, что она заполняет переданный массив *rates* котировками символа и таймфрейма текущего графика, на котором запущен скрипт. В макросе LIMIT указано количество запрашиваемых баров: необходимо удостовериться, что это значение не больше, чем настройка вашего терминала по количеству баров в каждом окне.

Для обработки полученных данных создадим объект *worker* с типами *T=MqlRates* и *R=double*:

```
Worker<MqlRates,double> worker(rates);
```

Запуск сортировки можно осуществить инструкцией вида:

```
worker.process(offsetof(MqlRates, open) / sizeof(double));
```

Здесь используется оператор *offsetof*, чтобы узнать байтовое смещение поля *open* внутри структуры. Далее оно делится на размер *double* и получается правильный номер "колонки" для сортировки по цене *open*. Результат сортировки можно прочитать поэлементно или получить весь массив целиком:

```
Print(worker[i].open);
...
worker.get(rates);
ArrayPrint(rates);
```

Обратите внимание, что получение массива методом *get* перемещает его из внутреннего массива *array* во внешний (переданный в качестве аргумента) с помощью *ArraySwap*. Поэтому после этого вызовы *worker.process()* бесполезны: в объекте *worker* уже нет данных.

Для упрощения запуска сортировки по разным полям была реализована вспомогательная функция *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]);
}
```

Она выводит в журнал некий заголовок и первый и последний элементы отсортированного массива. С её помощью тестирование в *OnStart* для трех полей выглядит следующим образом:

```
void OnStart()
{
   ...
   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 vo
   sort(worker, offsetof(MqlRates, time) / sizeof(double), "Sorting by time...");
}
```

К сожалению, стандартная функция *Print* не поддерживает печать одиночных структур, и в MQL5 нет встроенной функции *StructPrint*. Поэтому нам пришлось самим её написать, базируясь на *ArrayPrint*: фактически достаточно положить структуру в массив размером 1.

```
template<typename S>
void StructPrint(const S &s)
{
   S temp[1];
   temp[0] = s;
   ArrayPrint(temp);
}
```

В результате запуска скрипта можем получить примерно следующее (зависит от настроек терминала, а именно на каком символе/таймфрейме выполняется):

```
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
```

Здесь представлены данные для EURUSD,M15.

Приведенная реализация сортировки потенциально одна из самых быстрых, потому что использует встроенную *ArraySort*.

Если же сложности с выравниванием полей структуры или скептическое отношение к самому подходу с "отображением" структуры в массив вынуждают отказаться от данного метода (и тем самым, от функции *ArraySort*), остается проверенный способ "сделай всё сам".

Существует большое количество алгоритмов сортировки, которые легко адаптировать под MQL5. Один из варианто быстрой сортировки представлен в прилагаемом к книге файле *QuickSortStructT.mqh*. Это усовершенствованная версия *QuickSortT.mqh*, который мы использовали в разделе Сравнение строк. В нем метод *Compare* шаблонного класса *QuickSortStructT* сделан чисто виртуальным и должен быть переопределен в классе наследнике, чтобы возвращать аналог операции сравнения '>' для требуемого типа и его полей. Для удобства пользователей в заголовочном файле создан макрос:

```
#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);                                              \
}
```

С помощью него для сортировки массива структур по заданном полю достаточно написать одну инструкцию. Например:

```
   MqlRates rates[];
   CopyRates(_Symbol, _Period, 0, 10000, rates);
   SORT_STRUCT(MqlRates, rates, high);
```

Здесь выполняется сортировка массива *rates* типа *MqlRates* по цене *high*.

int ArrayBsearch(const type &array[], type value)

Функция ищет в числовом массиве заданное значение. Поддерживаются массивы всех встроенных числовых типов. Массив должен быть отсортирован по возрастанию по первому измерению, в противном случае результат будет некорректным.

Функция возвращает индекс совпадающего элемента (если их несколько, то индекс первого из них) или индекс ближайшего по значению элемента (если точного совпадения нет), то есть это может быть элемент как с большим, так и с меньшим значением, чем искомое. Если искомое значение меньше первого (минимального), то возвращается 0. Если искомое значения больше последнего (максимального), возвращается его индекс.

Индекс зависит от направления нумерации элементов в массиве: прямой (он начала к концу) или обратный (от конца к началу). Его можно узнать и изменить с помощью функций, описанных в разделе Направление индексации массивов как в таймсерии.

В случае ошибки выдается -1.

Для многомерных массивов поиск ограничивается первым измерением.

В скрипте *ArraySearch.mq5* приведены примеры использования функции *ArrayBsearch*.

```
void OnStart()
{
   int array[] = {1, 5, 11, 17, 23, 23, 37};
     // индексы   0  1   2   3   4   5   6
   int data[][2] = {{1, 3}, {3, 2}, {5, 10}, {14, 10}, {21, 8}};
     // индексы      0       1       2        3         4
   int empty[];
   ...
```

Для трех предопределенных массивов (один из них пустой) выполняются следующие инструкции:

```
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
...
```

Далее во вспомогательной функции *populateSortedArray* производится заполнение массива *numbers* случайными значениями, причем массив постоянно поддерживается в отсортированном состоянии с помощью *ArrayBsearch*.

```
void populateSortedArray(const int limit)
{
   double numbers[];  // массив для заполнения
   double element[1]; // новое значение для вставки
   
   ArrayResize(numbers, 0, limit); // выделяем память заранее
   
   for(int i = 0; i < limit; ++i)
   {
      // генерируем случайное число
      element[0] = NormalizeDouble(rand() * 1.0 / 32767, 3);
      // находим, где его место в массиве
      int cursor = ArrayBsearch(numbers, element[0]);
      if(cursor == -1)
      {
         if(_LastError == 5053) // пустой массив
         {
            ArrayInsert(numbers, element, 0);
         }
         else break; // ошибка
      }
      else
      if(numbers[cursor] > element[0]) // вставка в позицию 'cursor'
      {
         ArrayInsert(numbers, element, cursor);
      }
      else // (numbers[cursor] <= value) // вставка после 'cursor'
      {
         ArrayInsert(numbers, element, cursor + 1);
      }
   }
   ArrayPrint(numbers, 3);
}
```

Каждое новое значение попадает вначале в одноэлементный массив *element*, потому что так его проще вставлять в результирующий массив *numbers* с помощью функции *ArrayInsert*.

ArrayBsearch позволяет определить, в какое место следует вставить новое значение.

Результат работы функции выводится в журнал:

```
void OnStart()
{
   ...
   populateSortedArray(80);
   /*
     пример (будет отличаться при каждом запуске из-за рандомизации)
   [ 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)

Функции *ArrayMaximum* и *ArrayMinimum* ищут в числовом массиве элементы с максимальным и минимальным значением, соответственно. Диапазон индексов для поиска задается параметрами *start* и *count*: со значениями по умолчанию выполняется поиск по всему массиву.

Функция возвращает позицию найденного элемента.

Если для массива установлено свойство "серийности" ("таймсерии", временного ряда), индексация элементов в нем ведется в обратном порядке, и это сказывается на результате данной функции (см. пример). Встроенные функции для работы со свойством "серийности" рассматриваются в следующем разделе. Более подробно о "серийных" массивах будет рассказано в главах про таймсерии и индикаторы.

В многомерных массивах поиск ведется по первому измерению.

Если в массиве несколько одинаковых элементов с максимальным или минимальным значением, функция вернет индекс первого из них.

Пример использования функций приведен в файле *ArrayMaxMin.mq5*.

```
#define LIMIT 10

void OnStart()
{
   // генерируем случайные данные
   int array[];
   ArrayResize(array, LIMIT);
   for(int i = 0; i < LIMIT; ++i)
   {
      array[i] = rand();
   }

   ArrayPrint(array);
   // по умолчанию новый массив не является таймсерией
   PRTS(ArrayMaximum(array));
   PRTS(ArrayMinimum(array));
   // включаем свойство "серийности"
   PRTS(ArraySetAsSeries(array, true));
   PRTS(ArrayMaximum(array));
   PRTS(ArrayMinimum(array));
}
```

Скрипт выведет в журнал примерно следующее (из-за случайной генерации данных каждый запуск будет отличаться):

```
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 Направление индексации массивов как в таймсерии

В силу прикладной трейдерской специфики MQL5 привносит в работу с массивами дополнительные нюансы. Один из них заключается в том, что элементы массивов могут содержать данные, соответствующие отсчетам во времени. К их числу относятся, например, массивы с котировками финансовых инструментов, их тиков, показаниям технических индикаторов. Хронологический порядок данных означает, что новые элементы постоянно добавляются в конец массива и их индексы увеличиваются.

Однако, с точки зрения торговли удобнее вести отсчет от настоящего в прошлое. Тогда элемент под номером 0 всегда содержит самое свежее, актуальное значение, элемент под номером 1 — предыдущее значение, и так далее.

MQL5 позволяет выбирать и на лету переключать направление индексации массивов. Массив с нумерацией от настоящего в прошлое называется временным рядом или таймсерией. Если же увеличение индексов происходит от прошлого к настоящему — это обычный массив. В таймсериях время уменьшается с ростом индексов. В обычных массивах время увеличивается, как в реальной жизни.

Важно отметить, что массив, в принципе, не обязан содержать величины, связанные со временем, чтобы для него можно было переключить порядок адресации. Просто эта возможность наиболее востребована и, собственно, появилась для работы с историческими данными.

Данный атрибут массива никак не влияет на размещение данных в памяти. Меняется лишь порядок нумерации. В частности, мы могли бы сами реализовать его аналог на MQL5 за счет обхода массива в цикле "задом наперед". Но MQL5 предоставляет готовые функции, чтобы скрыть всю эту рутину от прикладных программистов.

Таймсерией можно сделать любой одномерный динамический массив, описанный в MQL-программе, а также внешние массивы, передаваемые в MQL-программу из ядра MetaTrader 5 как параметры служебных функций. Например, специальный тип MQL-программ — индикаторы — получает массивы с ценовыми данными текущего графика в обработчике события *OnCalculate*. Мы изучим все особенности прикладного использования таймсерий позднее, в пятой Части книги.

Массивы, определенные в MQL-программе, по умолчанию не являются таймсерией.

А пока познакомимся с набором функций для определения и изменения атрибута "серийности" массива, а также его "принадлежности" терминалу. Общий скрипт с примерами *ArrayAsSeries.mq5* будет приведен после описания.

bool ArrayIsSeries(const void &array[])

Функция возвращает признак того, является ли указанный массив "настоящей" таймсерией, то есть управляется и предоставляется самим терминалом. Изменить эту характеристику массива нельзя. Подобные массивы доступны MQL-программе в режиме "только чтение".

> В документации MQL5 термины "таймсерия" и "серийность" используются для описания и обратной индексации массива, и того факта, что массив может "принадлежать" терминалу (он выделяет под него память и наполняет данными). В книге мы постараемся избегать этой двусмысленности и называть "таймсериями" или временными рядами массивы с обратной индексацией. А массивы терминала так и будут — *собственными* массивами терминала.

Индексацию любого собственного массива терминала можно менять по своему усмотрению, переключая его в режим таймсерии или обратно в стандартный. Это делается с помощью функции *ArraySetAsSeries*, которая применима не только к собственными, но и пользовательским динамическим массивам (см. ниже).

bool ArrayGetAsSeries(const void &array[])

Функция возвращает признак того, включен ли для указанного массива режим индексации "как в таймсерии", то есть с увеличением индексов в направлении от настоящего в прошлое. Изменить направление индексации можно с помощью функции *ArraySetAsSeries*.

Направление индексации влияет на значения, возвращаемые функциями *ArrayBsearch*, *ArrayMaximum*, *ArrayMinimum* (см. раздел Сравнение, сортировка и поиск в массивах).

bool ArraySetAsSeries(const void &array[], bool as_series)

Функция устанавливает направление индексации в массиве согласно параметру *as_series*: значение *true* означает обратный порядок нумерации элементов "как в таймсерии", *false* — обычный порядок.

Функция возвращает *true* при успешной установки атрибута или *false* в случае ошибки.

Поддерживаются массивы любых типов, но менять направление индексации запрещено у многомерных массивов и массивов фиксированного размера.

В скрипте *ArrayAsSeries.mq5* описано несколько небольших массивов для экспериментов с участием вышеописанных функций.

```
#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); // выделяем память
   // заполняем пару массивов номерами: 1, 2, 3,...
   indexArray(fixed);
   indexArray(dynamic);
   ...
```

Это двумерный массив *array2D*, фиксированный и динамический массив — все типа *double*, а также массивы структур и объектов класса. Массивы *fixed* и *dynamic* заполняются для наглядности последовательными целыми числами (используется вспомогательная функция *indexArray*). Для остальных типов массивов мы лишь проверим применимость "серийного" режима, поскольку суть эффекта обращения индексации станет понятна на примере заполненных массивов.

Сперва убедимся, что ни один из массивов не является собственным массивом терминала:

```
   PRTS(ArrayIsSeries(array2D)); // false
   PRTS(ArrayIsSeries(fixed));   // false
   PRTS(ArrayIsSeries(dynamic)); // false
   PRTS(ArrayIsSeries(rates));   // false
```

Все вызовы *ArrayIsSeries* возвращают *false*, поскольку все массивы определены нами в MQL-программе. Мы увидим значение *true* у массивов-параметров функции *OnCalculate* в индикаторах (в пятой Части).

Далее проверим начальное направление индексации массивов:

```
PRTS(ArrayGetAsSeries(array2D)); // false, не может быть true
PRTS(ArrayGetAsSeries(fixed));   // false
PRTS(ArrayGetAsSeries(dynamic)); // false
PRTS(ArrayGetAsSeries(rates));   // false
PRTS(ArrayGetAsSeries(dummies)); // false
```

И опять получим везде *false*.

Выведем массивы *fixed* и *dynamic* в журнал, чтобы видеть исходный порядок элементов.

```
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
*/
```

Теперь пробуем изменить порядок индексации:

```
// ошибка: parameter conversion not allowed
// PRTS(ArraySetAsSeries(array2D, true));

// предупреждение: cannot be used for static allocated array
PRTS(ArraySetAsSeries(fixed, true));    // false

// далее все штатно
PRTS(ArraySetAsSeries(dynamic, true)); // true
PRTS(ArraySetAsSeries(rates, true));    // true
PRTS(ArraySetAsSeries(dummies, true)); // true
```

Инструкция для массива *array2D* вызывает ошибку компиляции, и потому закомментирована.

Инструкция для массива *fixed* выдает предупреждение компилятора о невозможности её применения к массиву постоянного размера. На стадии выполнения все 3 последних инструкции вернули признак успеха (*true*). Посмотрим, как поменялись атрибуты у массивов:

```
// проверки на атрибуты:
// во-первых, стали ли они собственными для терминала
PRTS(ArrayIsSeries(fixed));             // false
PRTS(ArrayIsSeries(dynamic));           // false
PRTS(ArrayIsSeries(rates));             // false
PRTS(ArrayIsSeries(dummies));           // false

// во-вторых, направление индексации
PRTS(ArrayGetAsSeries(fixed));          // false
PRTS(ArrayGetAsSeries(dynamic));        // true
PRTS(ArrayGetAsSeries(rates));          // true
PRTS(ArrayGetAsSeries(dummies));        // true
```

Как и ожидалось, массивы не превратились в собственные массивы терминала. Однако три массива из четырех поменяли индексацию на режим таймсерий, включая массив структур и объектов. Для демонстрации результата в журнал снова выводятся массивы *fixed* и *dynamic*.

```
ArrayPrint(fixed, 1);    // без изменений
ArrayPrint(dynamic, 1);  // обратный порядок
/*
    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
*/
```

Поскольку к массиву постоянного размера режим не применился, он остался без изменений. Массив *dynamic* отображается теперь в обратном порядке.

Если перевести массив в режим обратной индексации, изменить его размер, а затем вернуть прежнюю индексацию, то добавленные элементы окажутся вставленными в начало массива.

4.3.9 Обнуление объектов и массивов

Обычно инициализация или заполнение переменных и массивов не вызывает проблем. Так, для простых переменных действительно довольно просто использовать оператор '=' в инструкции определения вместе с инициализацией или присвоить требуемое значение в любой более поздний момент.

Для структур доступна агрегатная инициализация вида (см. раздел Определение структур):

```
Struct struct = {value1, value2, ...};
```

Но она возможна, только если в структуре нет динамических массивов и строк. Более того, синтаксис агрегатной инициализации нельзя использовать для повторной очистки структуры. Вместо этого необходимо либо присваивать значения каждому полю по отдельности, либо зарезервировать в программе экземпляр пустой структуры и копировать его в очищаемые экземпляры.

Если речь идет еще и о массиве структур, то исходный код быстро разрастется за счет вспомогательных, но необходимых инструкций.

Для массивов, как мы знаем, существуют функции *ArrayInitialize и ArrayFill*, но они поддерживают только числовые типы: массив строк или структур ими заполнить не получится.

В таких случаях может пригодиться функция *ZeroMemory*. Она не является панацеей, поскольку имеет существенные ограничения по области применения, но, тем не менее, её стоит взять на вооружение.

void ZeroMemory(void &entity)

Функция может применяться к широкому набору различных сущностей: переменным простых или объектных типов, а также их массивам (фиксированным, динамическим, многомерным).

Переменные получают значение 0 (для чисел) или его эквивалент (NULL для строк и указателей).

В случае массива обнулению подвергаются все его элементы, причем элементы могут быть объектами и в свою очередь содержать объекты. Иными словами, функция *ZeroMemory* выполняет глубокую очистку памяти за один вызов.

Однако для допустимых объектов есть ограничения. Заполняться нулями могут лишь объекты структур и классов, которые:

- содержат только открытые поля (то есть в них нет данных с типом доступа *private* или *protected*);
- не содержат полей с модификатором *const*;
- не содержат указателей.

Первые два ограничения заложены в компилятор: попытка обнулить объекты с полями, несоответствующими указанным требованиям, вызовет ошибки (см. пример далее).

Третье ограничение относится к разряду рекомендаций: внешнее обнуление указателя затруднит контроль за целостностью данных, что, вероятно, приведет к потере связанного объекта и утечке памяти.

Строго говоря, требование публичности полей в обнуляемых объектах нарушает принцип инкапсуляции, свойственный объектам класса, и потому *ZeroMemory* преимущественно используется с объектами простых структур и их массивами.

Примеры работы с *ZeroMemory* приведены в скрипте *ZeroMemory.mq5*.

Проблемы со списком агрегатной инициализации демонстрируются с помощью структуры *Simple*:

```
#define LIMIT 5

struct Simple
{
   MqlDateTime data[]; // динамический массив запрещает список инициализации,
   // string s;        // и строковое поле тоже запретило бы,
   // ClassType *ptr;  // и указатель тоже
   Simple()
   {
      // выделяем память, она будет содержать произвольные данные
      ArrayResize(data, LIMIT);
   }
};
```

В функции *OnStart* или в глобальном контексте мы не можем определить и тут же обнулить объект такой структуры:

```
void OnStart()
{
   Simple simple = {}; // ошибка: cannot be initialized with initializer list
   ...
```

Компилятор выдает ошибку "нельзя использовать список инициализации". Она специфична для полей вроде динамических массивов, строковых переменных и указателей. В частности, если бы массив *data* был фиксированного размера, ошибки бы не возникло.

Поэтому вместо списка инициализации используем *ZeroMemory*:

```
void OnStart()
{
   Simple simple;
   ZeroMemory(simple);
   ...
```

Начальное заполнение нулями можно было бы сделать и в конструкторе структуры, но последующие очистки удобнее делать снаружи (или предоставить для этого метод с той же самой *ZeroMemory*).

Далее в коде определен класс *Base*.

```
class Base
{
public: // public обязателен для ZeroMemory
   // const у любого поля вызовет ошибку компиляции при вызове ZeroMemory:
   // "not allowed for objects with protected members or inheritance"
   /* const */ int x;
   Simple t;   // используем вложенную структуру: она тоже будет обнулена
   Base()
   {
      x = rand();
   }
   virtual void print() const
   {
      PrintFormat("%d %d", &this, x);
      ArrayPrint(t.data);
   }
};
```

Поскольку класс далее используется в массивах объектов, обнуляемых с помощью *ZeroMemory*, мы вынуждены прописать для его полей секцию доступа *public* (что, в принципе, не типично для классов и сделано для иллюстрации навязываемых требований со стороны *ZeroMemory*). Также обратите внимание, что поля не могут иметь модификатор *const*. В противном случае мы получим ошибку компиляции с текстом, который, к сожалению, не особо соответствует проблеме: "запрещено для объектов с защищенными членами или наследованием".

Конструктор класса заполняет поле *x* случайным числом, чтобы потом можно было наглядно увидеть его очистку функцией *ZeroMemory*. Метод *print* выводит содержимое всех полей для анализа, включая и уникальный номер (дескриптор) объекта — *&this*.

MQL5 не запрещает "натравить" *ZeroMemory* на переменную указатель:

```
   Base *base = new Base();
   ZeroMemory(base); // установит указатель в NULL, но оставит объект
```

Однако так делать не следует, потому что функция обнуляет только саму переменную *base* и, если она ссылалась на объект, тот останется "висеть" в памяти, недоступным из программы из-за утраты указателя.

Обнулять указатель можно только после того, как указываемый экземпляр освобожден с помощью оператора *delete*. Причем отдельный указатель из вышеприведенного примера, как и любую другую простую переменную (несоставную), проще сбросить оператором присваивания. *ZeroMemory* имеет смысл применять для составных объектов и массивов.

Функция позволяет работать с объектами иерархии классов. Например, мы можем описать производный от *Base* класс *Dummy*:

```
class Dummy : public Base
{
public:
   double data[]; // мог бы быть и многомерным: ZeroMemory сработает
   string s;
   Base *pointer; // публичный указатель (опасно)

public:
   Dummy()
   {
      ArrayResize(data, LIMIT);

      // из-за последующего применения ZeroMemory к объекту
      // мы потеряем указатель 'pointer'
      // и получим предупреждения при завершении скрипта
      // о неудаленных объектах типа Base
      pointer = new Base();
   }

   ~Dummy()
   {
      // из-за применения ZeroMemory, этот указатель пропадет
      // и не будет освобожден
      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();
   }
};
```

Он включает поля с динамическим массивом типа *double*, строку и указатель типа *Base* (это тот же тип, от которого класс наследован, но он здесь использован только для демонстрации проблем с указателем, чтобы не описывать еще один фиктивный класс). Когда функция *ZeroMemory* обнуляет объект *Dummy*, объект по указателю *pointer* теряется и не может быть освобожден в деструкторе. В результате, это приводит после завершения скрипта к появлению предупреждений об утечке памяти в оставшихся объектах.

ZeroMemory используется в *OnStart* для очистки массива объектов *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();
   }
```

В журнал будет выведено примерно следующее (начальное состояние будет отличаться, т.к. оно выводит содержимое "грязной", только что выделенной памяти; здесь приведен небольшой фрагмент):

```
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
```

Для сравнения состояния объектов до и после очистки пользуйтесь дескрипторами.

Таким образом, одиночный вызов *ZeroMemory* способен сбросить состояние произвольной разветвленной структуры данных (массивов, структур, массивов структур с вложенными полями-структурами и массивами).

Наконец, посмотрим, как *ZeroMemory* может решить проблему инициализации массива строк. Функции *ArrayInitialize* и *ArrayFill* не работают со строками.

```
string text[LIMIT] = {};
// некий алгоритм заполняет и использует 'text'
// ...
// затем нужно повторно использовать массив
// вызовы функций дают ошибки:
// 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
```

В закомментированных инструкциях компилятор выдал бы ошибки, смысл которых в том, что тип *string* не поддерживается в данных функциях.

Выходом является функция *ZeroMemory*.

4.4 Математические функции

Наиболее востребованные математические функции, как правило, встроены тем или иным образом во все современные языки программирования, и MQL5 — не исключение. В этой главе мы познакомимся с несколькими группами функций, доступных, что называется, "сразу из коробки". К ним относятся функции округления, тригонометрические, гиперболические, показательные, логарифмические, степенные функции, а также несколько специальных, таких как генерация случайных чисел и проверка вещественных чисел на нормальность.

Большинство функций имеет два названия: полное (с приставкой "Math", с капитализацией слов) и сокращенное (без приставки, строчными буквами). Мы укажем оба варианта: они работают одинаково. Выбор можно сделать, исходя из стиля оформления исходных кодов.

Поскольку математические функции выполняют некие вычисления и возвращают результат в виде вещественного числа, потенциальные ошибки способны приводить к ситуации, когда результат не определен. Например, нельзя получить квадратный корень из отрицательного числа или логарифм от нуля. В таких случаях функции возвращают специальные величины, не являющиеся числами (NaN, Not A Number). Мы уже знакомились с ними в разделах Вещественные числа, Арифметические операции и Числа в строки и обратно. Проанализировать корректность числа и отсутствие ошибок позволяют функции *MathIsValidNumber* и *MathClassify* (см. раздел Проверка вещественных чисел на нормальность).

Наличие хотя бы одного операнда со значением NaN приведет к тому, что любые последующие вычисления с ним, включая вызовы функций, также будут давать результат NaN.

> В качестве задания для самостоятельного изучения и наглядного материала к данной главе прилагается скрипт *MathPlot.mq5*, который позволяет отображать графики математических функций с одним аргументом из числа описываемых. Скрипт использует в своей работе стандартную библиотеку для рисования *Graphic.mqh*, поставляемую с MetaTrader 5 (она выходит за рамки данной книги). Ниже приведен образец того, как может выглядеть кривая гиперболического синуса в окне MetaTrader 5.

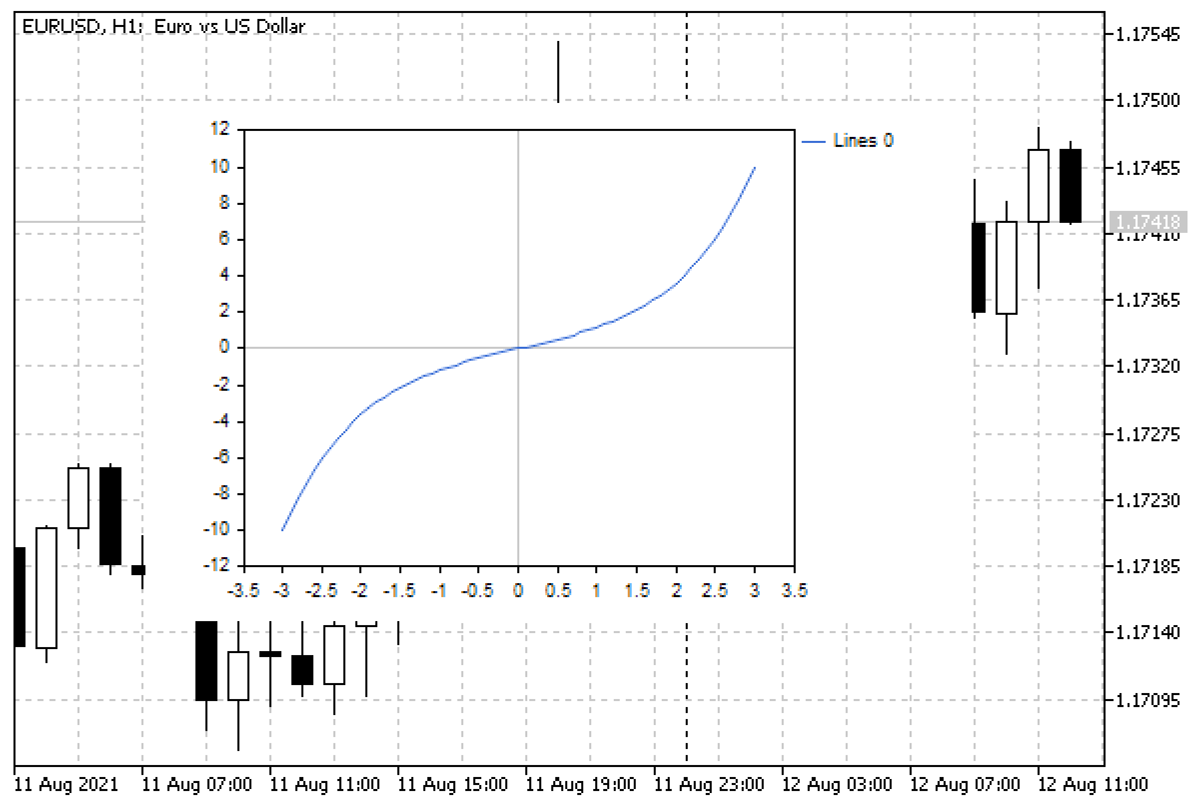


График гиперболического синуса в окне MetaTrader 5

4.4.1 Абсолютное значение числа

В MQL5 API имеется функция *MathAbs*, позволяющая убрать у числа знак минус, если он есть. Она избавляет нас от ручного кодирования более длинных эквивалентов, например, такого:

```
if(x < 0) x = -x;
```

numeric MathAbs(numeric value) ≡ numeric fabs(numeric value)

Функция возвращает абсолютное значение переданного ей числа, то есть его модуль. Аргументом может быть число любого типа. Иными словами, функция перегружена для *char/uchar*, *short/ushort*, *int/uint*, *long/ulong*, *float* и *double*, хотя для беззнаковых типов — значения и так всегда неотрицательные.

При передаче строки, она будет неявно преобразована в число *double*, о чем компилятор выдаст предупреждение.

Тип возвращаемого значения всегда совпадает с типом аргумента и потому компилятору может потребоваться выполнить приведение к типу приемной переменной, если типы отличаются.

Примеры использования функции приведены в файле *MathAbs.mq5*.

```
void OnStart()
{
   double x = 123.45;
   double y = -123.45;
   int i = -1;

   PRT(MathAbs(x)); // 123.45, число осталось "как есть"
   PRT(MathAbs(y)); // 123.45, знак минус ушел
   PRT(MathAbs(i)); // 1, int обрабатывается естественным образом

   int k = MathAbs(i);  // без предупреждения: тип int у параметра и результата

   // ситуации с предупреждениями:
   // требуется конвертация double в long
   long j = MathAbs(x); // possible loss of data due to type conversion

   // требуется конвертация из большого типа (4 байта) в маленький (2 байта)
   short c = MathAbs(i); // possible loss of data due to type conversion
   ...
```

Важно отметить, что приведение знакового целого к беззнаковому не эквивалентно получению модуля числа:

```
   uint u_cast = i;
   uint u_abs = MathAbs(i);
   PRT(u_cast);              // 4294967295, 0xFFFFFFFF
   PRT(u_abs);               // 1
```

Также обратите внимание, что число 0 может иметь знак:

```
   ...
   double n = 0;
   double z = i * n;
   PRT(z);                // -0.0
   PRT(MathAbs(z));       //  0.0
   PRT(z == MathAbs(z)); // true
}
```

Наиболее показательным примером использования *MathAbs* является проверка двух вещественных чисел на равенство. Как известно, вещественные числа имеют ограниченную точность представления значений, которая может дополнительно деградировать в ходе длительных вычислений (например, сумма десяти значений 0.1 не дает в точности 1.0). Строгое условие *value1* == *value2* может давать *false* в большинстве случаев, когда чисто умозрительно должно бы выполняться равенство.

Поэтому для сравнения вещественных величин обычно используют запись:

```
MathAbs(value1 - value2) < EPS
```

где EPS — некое малое положительное значение, заданная точность (см. пример в разделе Операции сравнения).

4.4.2 Максимальное и минимальное из двух чисел

Для нахождения наибольшего или наименьшего числа из двух MQL5 предлагает функции *MathMax* и *MathMin*. Их краткие псевдонимы, соответственно — *fmax* и *fmin*.

numeric MathMax(numeric value1, numeric value2) ≡ numeric fmax(numeric value1, numeric value2)
numeric MathMin(numeric value1, numeric value2) ≡ numeric fmin(numeric value1, numeric value2)

Функции возвращают максимальное или минимальное из двух переданных значений. Функции перегружены для всех встроенных типов.

Если в функции передаются параметры разных типов, то параметр "младшего" типа автоматически приводится к "старшему" типу, например, в паре типов *int* и *double*, *int* будет приведен к *double*. Подробнее о неявном приведении типов см. раздел Арифметические преобразования типов. Тип возвращаемого значения соответствует "старшему" типу.

При наличии параметра типа *string*, он будет "старшим", то есть всё приводится к строке. Строки сравниваются лексикографически, как в функции *StringCompare*.

Скрипт *MathMaxMin.mq5* демонстрирует функции в действии.

```
void OnStart()
{
   int i = 10, j = 11;
   double x = 5.5, y = -5.5;
   string s = "abc";

   // числа
   PRT(MathMax(i, j)); // 11
   PRT(MathMax(i, x)); // 10
   PRT(MathMax(x, y)); // 5.5
   PRT(MathMax(i, s)); // abc

   // преобразования типов
   PRT(typename(MathMax(i, j))); // int, как есть
   PRT(typename(MathMax(i, x))); // double
   PRT(typename(MathMax(i, s))); // string
}
```

4.4.3 Функции округления

MQL5 API включает несколько функций для округления чисел до ближайшего целого (в ту или иную сторону). Несмотря на операцию округления, все функции возвращают число типа *double* (с пустой дробной частью).

С технической точки зрения они принимают аргументы любого числового типа, но округляются только вещественные, а целые числа лишь приводятся к *double*.

Если требуется делать округление с точностью до конкретного знака, воспользуйтесь функцией *NormalizeDouble* (см. раздел Нормализация чисел double).

Примеры работы с функциями приведены в файле *MathRound.mq5*.

double MathRound(numeric value) ≡ double round(numeric value)

Функция округляет число до ближайшего целого вверх или вниз.

```
PRT((MathRound(5.5)));  // 6.0
PRT((MathRound(-5.5))); // -6.0
PRT((MathRound(11)));   // 11.0
PRT((MathRound(-11)));  // -11.0
```

Если значение дробной части больше или равно 0.5, мантисса увеличивается на единицу (вне зависимости от знака числа).

double MathCeil(numeric value) ≡ double ceil(numeric value)
double MathFloor(numeric value) ≡ double floor(numeric value)

Функции возвращают целое значение, ближайшее сверху (для *ceil*) или снизу (для *floor*) к переданной величине *value*. Если *value* уже равно целому числу (имеет нулевую дробную часть), возвращается оно само.

```
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 Деление чисел по модулю

Для деления целых чисел по модулю MQL5 имеет встроенный оператор '%', описанный в разделе Арифметические операции. Однако этот оператор неприменим для вещественных чисел. В случае, когда делитель, делимое или оба операнда являются вещественными, следует использовать функцию *MathMod* (или в краткой форме *fmod*).

double MathMod(double dividend, double divider) ≡ double fmod(double dividend, double divider)

Функция возвращает вещественный остаток от деления первого переданного числа (*dividend*) на второе (*divider*).

Если какой-либо аргумент является отрицательным, знак результата определяется по правилам, описанным в вышеупомянутом разделе.

Примеры работы функции доступны в скрипте *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, знак ушел
PRT(MathMod(-10.0, -1.5));  // -1.0
```

4.4.5 Степени и корни

MQL5 API предоставляет универсальную функцию *MathPow* для возведения числа в произвольную степень, а также функцию для частного случая со степенью 0.5, более привычную в виде извлечения квадратного корня *MathSqrt*.

Протестировать функции можно со скриптом *MathPowSqrt.mq5*.

double MathPow(double base, double exponent) ≡ double pow(double base, double exponent)

Функция возводит основание *base* в указанную степень *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)

Функция возвращает квадратный корень числа.

```
PRT(MathSqrt(2.0));      // 1.414213562373095
PRT(MathSqrt(-2.0));     // -nan(ind)
```

В MQL5 определено несколько констант, содержащих готовые значения вычислений с участием *sqrt*.

Константа	Описание	Значение
M_SQRT2	sqrt(2.0)	1.41421356237309504880
M_SQRT1_2	1 / sqrt(2.0)	0.707106781186547524401
M_2_SQRTPI	2.0 / sqrt(M_PI)	1.12837916709551257390

Здесь M_PI — число Пи (π=3.14159265358979323846, см. далее раздел Тригонометрические функции).

Все встроенные константы перечислены в документации.

4.4.6 Показательные и логарифмические функции

Вычисление показательных и логарифмических функций доступно в MQL5 с помощью соответствующего раздела API.

Отсутствие среди API двоичного логарифма, который часто требуется в информатике и комбинаторике, не является проблемой, поскольку его легко вычислить, на выбор, через имеющиеся функции натурального или десятичного логарифма.

```
log2(x) = log(x) / log(2) = log(x) / M_LN2
```

```
log2(x) = log10(x) / log10(2)
```

Здесь *log* и *log10* — имеющиеся функции логарифмов (по основанию *e* и 10, соответственно), M_LN2 — встроенная константа, равная *log(2)*.

В следующей таблице перечислены все константы, могущие быть полезными при логарифмических вычислениях.

Константа	Описание	Значение
M_E	e	2.71828182845904523536
M_LOG2E	log2(e)	1.44269504088896340736
M_LOG10E	log10(e)	0.434294481903251827651
M_LN2	ln(2)	0.693147180559945309417
M_LN10	ln(10)	2.30258509299404568402

Примеры описываемых далее функций собраны в файле *MathExp.mq5*.

double MathExp(double value) ≡ double exp(double value)

Функция возвращает экспоненту, то есть число *e* (доступно как предопределенная константа M_E), возведенное в указанную степень *value*. При переполнении функция возвращает *inf* (разновидность NaN для бесконечности).

```
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)

Функция возвращает натуральный логарифм переданного числа. Если значение *value* отрицательно, функция возвращает *-nan(ind)* (NaN "неопределенное значение"). Если *value* равно 0, функция возвращает *inf* (NaN "бесконечность").

```
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
```

В последней строке использована реализация двоичного логарифма через *MathLog*:

```
double Log2(double value)
{
   return MathLog(value) / M_LN2;
}
```

double MathLog10(double value) ≡ double log10(double value)

Функция возвращает десятичный логарифм числа.

```
PRT(MathLog10(10.0)); // 1.0
PRT(MathLog10(10000.0)); // 4.0
```

double MathExpm1(double value) ≡ double expm1(double value)

Функция возвращает значение выражения *(MathExp(value) - 1)*. В экономических расчетах функция используется для вычисления эффективного процента (дохода или платежа) в единицу времени в схеме сложных процентов, когда количество периодов стремится к бесконечности.

```
PRT(MathExpm1(0.1)); // 0.1051709180756476
```

double MathLog1p(double value) ≡ double log1p(double value)

Функция возвращает значение выражения *MathLog(1 + value)*, то есть производит обратное действие к функции *MathExpm1*.

```
PRT(MathLog1p(0.1)); // 0.09531017980432487
```

4.4.7 Тригонометрические функции

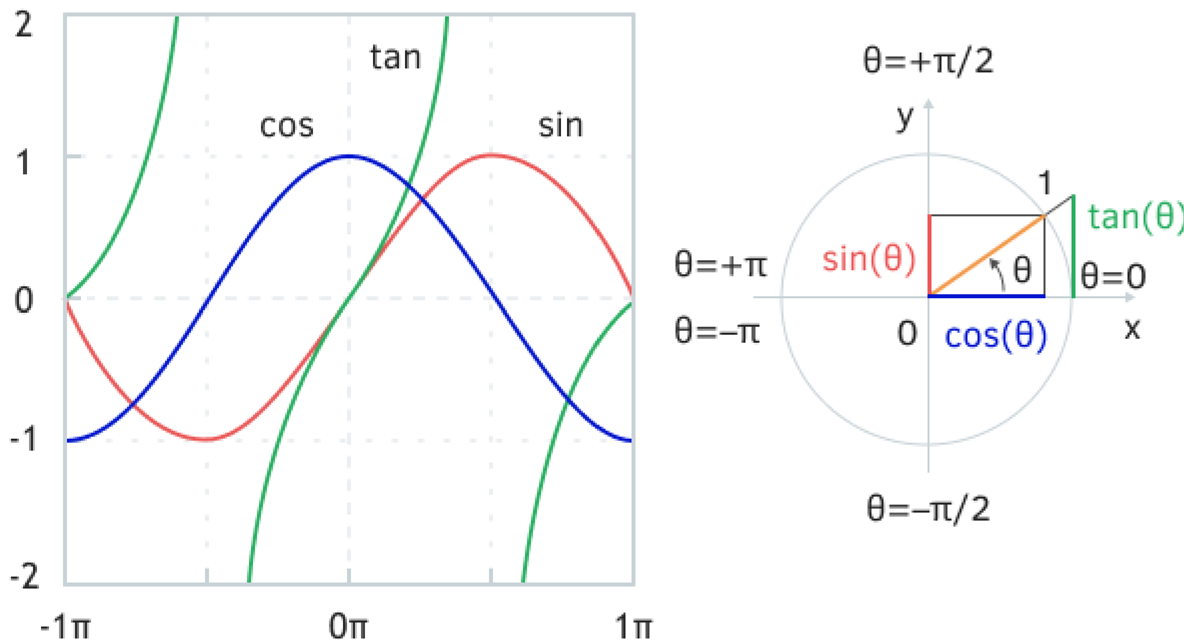
В MQL5 представлены 3 основных тригонометрических функции (*MathCos*, *MathSin*, *MathTan*) и обратные им (*MathArccos*, *MathArcsin*, *MathArctan*). Все они работают с углами в радианах. Для углов в градусах используйте известную формулу:

```
radians = degrees * M_PI / 180
```

Здесь M_PI — одна из нескольких констант с тригонометрическими величинами (число Пи и производные от него), встроенных в язык.

Константа	Описание	Значение
M_PI	π	3.14159265358979323846
M_PI_2	π/2	1.57079632679489661923
M_PI_4	π/4	0.785398163397448309616
M_1_PI	1/π	0.318309886183790671538
M_2_PI	2/π	0.636619772367581343076

Арктангенс можно также рассчитать для величины, представленной соотношением двух координат *y* и *x*: этот расширенный вариант называется *MathArctan2*, он способен восстанавливать углы в полном диапазоне окружности от -M_PI до +M_PI, в отличие от *MathArctan*, который ограничен диапазоном от -M_PI_2 до +M_PI_2.



Тригонометрические функции и квадранты единичного круга

Примеры расчетов приведены в скрипте *MathTrig.mq5* (см. после описаний).

double MathCos(double value) ≡ double cos(double value)
double MathSin(double value) ≡ double sin(double value)

Функции возвращают, соответственно, косинус и синус переданного числа (угла в радианах).

double MathTan(double value) ≡ double tan(double value)

Функция возвращает тангенс переданного числа (угла в радианах).

double MathArccos(double value) ≡ double acos(double value)
double MathArcsin(double value) ≡ double asin(double value)

Функции возвращают значение, соответственно, арккосинуса и арксинуса переданного числа, то есть угол в радианах. Если *x = MathCos(t)*, то *t = MathArccos(x)*. Для синуса и арксинуса — аналогично. Если *y = MathSin(t)*, то *t = MathArcsin(y)*.

Параметр должен находиться в пределах от -1 до +1. В противном случае функция вернет NaN.

Результат арккосинуса лежит в диапазоне от 0 до M_PI, а арксинуса — от -M_PI_2 до +M_PI_2. Указанные диапазоны называются главными, поскольку функции являются многозначными, то есть их значения периодически повторяются. Выбранные полупериоды полностью покрывают область определения от -1 до +1.

Полученный угол для косинуса лежит в верхнем полукруге, и симметричное решение в нижнем полукруге может быть получено добавлением знака, то есть *t* = -*t*. Для синуса полученный угол находится в правом полукруге, и второе решение в левом полукруге равно *M_PI - t* (если для отрицательного *t* требуется получить также отрицательный дополнительный угол, то -*M_PI - t*).

double MathArctan(double value) ≡ double atan(double value)

Функция возвращает для переданного числа значение арктангенса, то есть угол в радианах, в диапазоне от -M_PI_2 до +M_PI_2.

Функция является обратной по отношению к *MathTan*, но с одним нюансом.

Обратите внимание, что период тангенса в 2 раза меньше полного периода (длины окружности) за счет того, что отношение синуса и косинуса повторяются в противоположных квадрантах (четвертях окружности) из-за наложения знаков. В результате этого, самого по себе значения тангенса недостаточно, чтобы однозначно определить исходный угол в полном диапазоне от -M_PI до +M_PI. Это можно сделать с помощью функции *MathArctan2*, в которой тангенс представлен двумя раздельными компонентами.

double MathArctan2(double y, double x) ≡ double atan2(double y, double x)

Функция возвращает в радианах значение угла, тангенс которого равен отношению двух указанных чисел: координат по оси *y* и по оси *x*.

Результат (обозначим его как *r*) лежит в диапазоне от -M_PI до +M_PI, причем для него выполняется условие *MathTan(r) = y / x*.

Функция принимает во внимание знак обоих аргументов, чтобы определить правильный квадрант (с учетом граничных условий, когда либо *x*, либо *y* равны 0, то есть находятся на границе квадрантов).

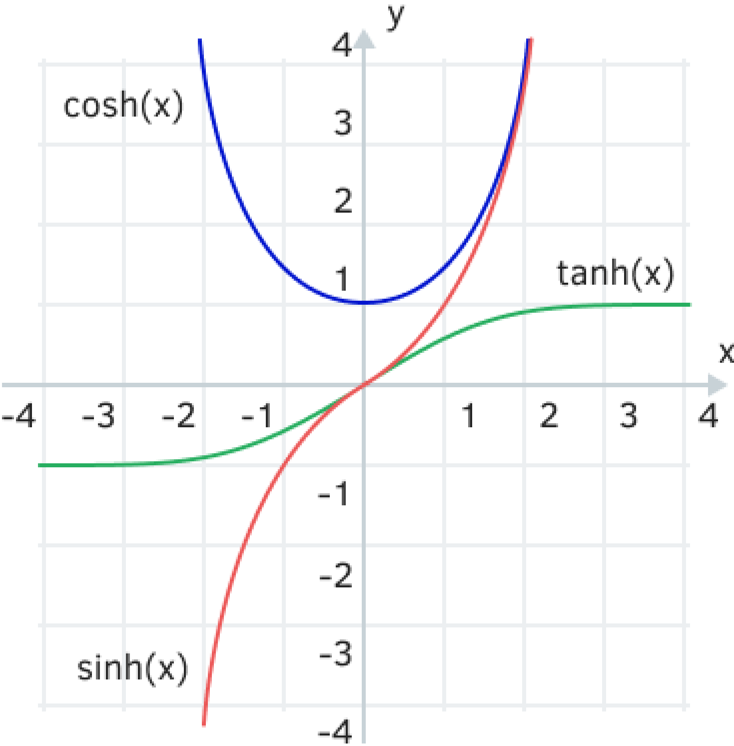
- 1 — x >= 0, y >= 0, 0 <= r <= M_PI_2
- 2 — x < 0, y >= 0, M_PI_2 < r <= M_PI
- 3 — x < 0, y < 0, -M_PI < r < -M_PI_2
- 4 — x >= 0, y < 0, -M_PI_2 <= r < 0

Ниже представлены результаты вызовов тригонометрических функций в скрипте *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 Гиперболические функции

MQL5 API включает набор прямых и обратных гиперболических функций.



Гиперболические функции

double MathCosh(double value) ≡ double cosh(double value)
double MathSinh(double value) ≡ double sinh(double value)
double MathTanh(double value) ≡ double tanh(double value)

Тройка основных функций вычисляет гиперболический косинус, синус и тангенс.

double MathArccosh(double value) ≡ double acosh(double value)
double MathArcsinh(double value) ≡ double asinh(double value)
double MathArctanh(double value) ≡ double atanh(double value)

Тройка обратных функций вычисляет гиперболический арккосинус, арксинус и арктангенс.

Для арккосинуса аргумент должен быть больше или равен +1. В противном случае функция вернет NaN.

Область определения арктангенса: от -1 до +1. При выходе аргумента за эти пределы функция вернет NaN.

Примеры гиперболических функций показаны в скрипте *MathHyper.mq5*.

```
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 Проверка вещественных чисел на нормальность

Поскольку вычисления с вещественными числами допускают возникновение нештатных ситуаций, таких как выход за пределы области определения функции, получение математической бесконечности, потеря порядка и других, результат может содержать не число, а некое специальное значение, фактически описывающее характер проблемы. Все такие специальные значения имеют обобщающее название "не число" (Not A Number, NaN).

В предыдущих разделах книги мы их уже встречали. В частности, при выводе в журнал (см. раздел Числа в строки и обратно) они отображаются в виде текстовых меток (например, *nan(ind)*, *+inf* и т.д.). Еще одной особенностью является то, что единственного значения NaN среди операндов какого-либо выражения достаточно, чтобы всё выражение перестало правильно рассчитываться и начало давать результат NaN. Исключение составляют только "не числа", представляющие плюс/минус бесконечности: если на них что-то разделить, получится ноль. Однако и здесь есть ожидаемое исключение: если разделить бесконечность на бесконечность, опять получим NaN.

Поэтому в программах важно определять момент, когда в расчетах появляется NaN, и обрабатывать ситуацию особым образом: сигнализировать об ошибке, подставлять некое приемлемое значение по умолчанию или повторять расчет с другими параметрами (например, уменьшить точность/шаг итеративного алгоритма).

В MQL5 есть 2 функции, которые позволяют проанализировать вещественное число на нормальность: *MathIsValidNumber* выдает простой ответ — да (*true*) или нет (*false*), а *MathClassify* производит более подробную категоризацию.

На физическом уровне все специальные значения кодируются в числе особым сочетанием битов, которое не используется для представления обычных чисел. Для типов *double* и *float* эти кодировки, разумеется, отличаются. Попробуем "заглянуть за кулисы" *double* (как более востребованного, чем *float*).

В разделе Вложенные шаблоны мы создали класс *Converter*, который позволял переключать представление за счет совмещения двух разных типов в объединении. Воспользуемся этим классом, чтобы изучить битовое устройство NaN.

Для удобства перенесем класс в отдельный заголовочный файл *ConverterT.mqh*. В тестовом скрипте *MathInvalid.mq5* подключим этот mqh-файл и создадим экземпляр конвертера для связки типов *double/ulong* (порядок следования не важен, конвертер способен работать в обе стороны).

```
static Converter<ulong,double> NaNs;
```

Сочетание битов в NaN стандартизовано, поэтому возьмем несколько общеупотребительных значений, представленных константами *ulong*, и посмотрим, как на них реагируют встроенные функции.

```
// основные NaN
#define NAN_INF_PLUS  0x7FF0000000000000
#define NAN_INF_MINUS 0xFFF0000000000000
#define NAN_QUIET     0x7FF8000000000000
#define NAN_IND_MINUS 0xFFF8000000000000

// примеры пользовательских NaN
#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
   ...
}
```

Как и ожидалось, результаты совпали.

Давайте познакомимся с формальным описанием функций *MathIsValidNumber* и *MathClassify*, а затем продолжим тесты.

bool MathIsValidNumber(double value)

Функция проверяет корректность действительного числа. Параметр может быть типа *double* или *float*. Результат *true* означает правильное число, а *false* — "не число" (одна из разновидностей NaN).

ENUM_FP_CLASS MathClassify(double value)

Функция возвращает категорию вещественного числа (типа *double* или *float*) — одно из значений перечисления ENUM_FP_CLASS:

- FP_NORMAL — нормальное число;
- FP_SUBNORMAL — число меньше, чем минимально представимое в нормализованном виде (например, для типа *double* это значения меньше DBL_MIN, 2.2250738585072014e-308); потеря порядка (точности);

* FP_ZERO — ноль (положительный или отрицательный);
* FP_INFINITE — бесконечность (положительная или отрицательная);
* FP_NAN — все прочие виды "не чисел" (подразделяются на семейства "тихих" и "сигнальных" NaN).

"Сигнальные" NaN отсутствуют в MQL5. Они используются в механизме исключений (exceptions), который позволяет перехватывать и реагировать на критические ошибки внутри программы. В MQL5 такого механизма нет, поэтому, если случается, например, деление на 0, MQL-программа просто завершает свою работу (экстренно выгружается с графика).

"Тихих" NaN может быть много, и вы можете их конструировать с помощью конвертера, чтобы дифференцированно обозначать и обрабатывать нестандартные состояния в своих вычислительных алгоритмах.

Выполним несколько вычислений в *MathInvalid.mq5*, чтобы наглядно увидеть, каким образом могут получиться числа различных категорий.

```
// вычисления с 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

// вычисления с 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
```

Мы можем использовать конвертер в обратную сторону: по значению *double* получать его битовое представление и тем самым детектировать "не числа":

```
PrintFormat("%I64X", NaNs[MathSqrt(-1.0)]);      // FFF8000000000000
PRT(NaNs[MathSqrt(-1.0)] == NAN_IND_MINUS);      // true, nind
```

Функция *PrintFormat* аналогична *StringFormat*, единственное отличие в том, что результат сразу выводится в журнал, а не в строку.

Наконец убедимся, что "не числа" всегда не равны:

```
// NaN != NaN всегда true
PRT(MathSqrt(-1.0) != MathSqrt(-1.0)); // true
PRT(MathSqrt(-1.0) == MathSqrt(-1.0)); // false
```

Для получения NaN или бесконечности в MQL5 существует способ, основанный на приведении строк "nan" и "inf" к *double*.

```
double nan = (double)"nan";
double infinity = (double)"inf";
```

4.4.10 Генерация случайных чисел

Многие алгоритмы в трейдинге требуют генерации случайных чисел. MQL5 предоставляет 2 функции, позволяющие инициализировать и затем опрашивать генератор псевдослучайных целых чисел.

Для получения более качественной "случайности" можно использовать поставляемую с MetaTrader 5 библиотеку Alglib (см. *MQL5/Include/Math/Alglib/alglib.mqh*).

void MathSrand(int seed) ≡ void srand(int seed)

Функция устанавливает некое начальное состояние генератора псевдослучайных целых чисел. Её следует вызывать однократно перед началом алгоритма. Сами случайные значения следует получать с помощью последовательного вызова функции *MathRand*.

Инициализация генератора одним и тем же значением *seed* позволяет получать воспроизводимые последовательности чисел. Значение *seed* не является первым случайным числом, полученным из *MathRand*.

Генератор поддерживает некое внутреннее состояние, которое в каждый момент времени (между обращениями к нему за новым случайным числом), характеризуется целочисленным значением: оно доступно из программы как встроенная переменная *uint _RandomSeed*. Именно это начальное значение состояния и устанавливает вызов *MathSrand*.

Работа генератора при каждом вызове *MathRand* описывается двумя формулами:

```
Xn = Tf(Xp)
R = Gf(Xn)
```

Функция *Tf* называется переходной (transition) функцией: она вычисляет новое внутреннее состояние генератора *Xn*, основываясь на предыдущем состоянии *Xp*.

Функция *Gf* генерирует очередное "случайное" значение, которое вернет функция *MathRand*, пользуясь для этого новым внутренним состоянием.

В MQL5 эти формулы реализованы следующим образом (псевдокод):

```
Tf: _RandomSeed = _RandomSeed * 214013 + 2531011
Gf: MathRand = (_RandomSeed >> 16) & 0x7FFF
```

Рекомендуется передавать в качестве *seed* значение функции *GetTickCount* или *TimeLocal*.

int MathRand() ≡ int rand()

Функция возвращает псевдослучайное целое число в диапазоне от 0 до 32767. Последовательность генерируемых чисел меняется в зависимости от начальной инициализации, выполненной с помощью вызова *MathSrand*.

Пример работы с генератором приведен в файле *MathRand.mq5*. В нем производится подсчет статистики по распределению генерируемых чисел по заданному количеству поддиапазонов (корзинок). В идеале мы должны получить равномерное распределение.

```
#define LIMIT 1000 // количество попыток (генерируемых чисел)
#define STATS 10   // количество корзинок

int stats[STATS] = {}; // подсчет статистики попаданий в корзинки

void OnStart()
{
   const int bucket = 32767 / STATS;
   // сброс генератора
   MathSrand((int)TimeLocal());
   // повторяем эксперимент в цикле
   for(int i = 0; i < LIMIT; ++i)
   {
      // получение нового случайного числа и обновление статистики
      stats[MathRand() / bucket]++;
   }
   ArrayPrint(stats);
}
```

Пример результатов для трех запусков (каждый раз будем получать новую последовательность):

```
 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 Управление порядком байтов в целых числах

Исторически так сложилось, что в различных информационных системах, на аппаратном уровне, применяется разный порядок байтов при представлении чисел в памяти. Поэтому в задачах интеграции MQL-программ с "внешним миром", в частности, при реализации сетевых коммуникационных протоколов или чтении/записи файлов распространенных форматов может потребоваться изменить порядок байтов.

В компьютерах, работающих под управлением Windows, применяется порядок "от младшего к старшему" (little-endian), то есть первым в ячейке памяти, выделенной под переменную, лежит младший байт, после него — байт с более старшими разрядами и так далее. Альтернативный порядок "от старшего к младшему" (big-endian) широко используется в сети Интернет. В этом случае первым байтом в ячейке памяти идет байт со старшими разрядами, а последним располагается байт с младшими. Именно этот порядок похож на то, как мы записываем числа "слева-направо" в обычной жизни. Например, значение 1234 начинается с цифры 1 и обозначает тысячи, цифра 2 следом обозначает сотни, цифра 3 — десятки, и последняя 4 — это просто четыре (младший разряд).

Посмотрим, каков по умолчанию порядок байтов в MQL5. Для этого обратимся к скрипту *MathSwap.mq5*.

В нем описан шаблон объединения, который позволяет преобразовать целое число в массив байтов:

```
template<typename T>
union ByteOverlay
{
   T value;
   uchar bytes[sizeof(T)];
   ByteOverlay(const T v) : value(v) { }
   void operator=(const T v) { value = v; }
};
```

Этот код позволяет наглядно разделить число на байты и пронумеровать их индексами из массива.

Опишем в *OnStart* переменную *uint* со значением 0x12345678 (учтите, что цифры шестнадцатеричные — в такой записи они точно соответствуют границам байтов: каждые 2 цифры это отдельный байт). Преобразуем число в массив и выведем его в журнал.

```
void OnStart()
{
   const uint ui = 0x12345678;
   ByteOverlay<uint> bo(ui);
   ArrayPrint(bo.bytes); // 120  86  52  18 <==> 0x78 0x56 0x34 0x12
   ...
```

Функция *ArrayPrint* не умеет печатать числа в шестнадцатеричном формате, поэтому мы видим их десятичное представление, однако их нетрудно перевести в основание 16 и убедиться, что они соответствуют исходным байтам. Визуально они идут в обратном порядке: то есть под 0-м индексом в массиве находится 0x78, и далее 0x56, 0x34 и 0x12. Очевидно, что это порядок "от младшего к старшему" (мы действительно на Windows).

Теперь познакомимся с функцией *MathSwap*, которую MQL5 предоставляется для изменения порядка байтов.

integer MathSwap(integer value)

Функция возвращает целое число, в котором порядок байтов из переданного аргумента изменен на обратный. Функция принимает параметры типа *ushort/uint/ulong* (т.е. размером 2, 4, 8 байтов).

Попробуем функцию в действии:

```
const uint ui = 0x12345678;
PrintFormat("%I32X -> %I32X", ui, MathSwap(ui));
const ulong ul = 0x0123456789ABCDEF;
PrintFormat("%I64X -> %I64X", ul, MathSwap(ul));
```

Вот каков результат:

```
12345678 -> 78563412
123456789ABCDEF -> EFCDAB8967452301
```

Попробуем вывести в журнал массив байтов после преобразования величины 0x12345678 с помощью *MathSwap*:

```
bo = MathSwap(ui);     // записали в ByteOverlay результат MathSwap
ArrayPrint(bo.bytes); //  18  52  86 120 <==> 0x12 0x34 0x56 0x78
```

В байте с индексом 0, где раньше было 0x78, теперь находится 0x12, и в элементах с остальными номерами значения также обменялись.

4.5 Работа с файлами

Редко какая программа обходится без ввода-вывода данных. Мы уже знаем, что MQL-программы могут получать настройки через входные переменные и выводить информацию в журнал — последним мы пользовались практически во всех тестовых скриптах. Но этого недостаточно в большинстве случаев.

Например, довольно часть настройки программы включают объемы данных, которые нельзя уместить во входных параметрах. Или программу необходимо интегрировать с некими внешними аналитическими средствами, то есть выгружать рыночную информацию в стандартном или специализированном формате, обрабатывать и затем загружать в терминал в новом виде, в частности, как торговые сигналы, набор весов нейронной сети или коэффициенты дерева решений. Да и журнал бывает удобно вести для MQL-программы отдельный, собственный.

Наиболее универсальные возможности для подобных задач предоставляет файловая подсистема. И MQL5 API содержит широкий спектр функций по работе с файлами, включая их создание, удаление, поиск, запись и считывание. Всё это мы изучим в данной главе.

Все операции с файлами в MQL5 ограничены особой областью на диске, которая называется "песочницей". Это сделано из соображений безопасности, чтобы ни одна MQL-программа не могла использоваться в злонамеренных целях и навредить вашему компьютеру и операционной системе.

> Продвинутые пользователи имеют возможность обойти это ограничение с помощью специальных мер, о которых мы расскажем чуть ниже, но делать это следует только в исключительных случаях, соблюдая меры предосторожности и принимая на себя всю ответственность.

У каждого экземпляра терминала, установленного на компьютере, корневой каталог "песочницы" располагается по пути: *<папка_данных_терминала>/MQL5/Files/*. Из редактора MetaEditor легко найти папку данных с помощью команды *Файл -> Открыть каталог данных*. При наличии достаточных прав доступа на компьютере этот каталог, как правило, совпадает с тем местом, куда установлен терминал. Если прав недостаточно, путь будет иметь вид:

```
X:/Users/<user_name>/AppData/Roaming/MetaQuotes/Terminal/<instance_id>/MQL5/Files/
```

Здесь *X* — литера диска, где установлена система, *<user_name>* — логин пользователя Windows, *<instance_id>* — уникальный идентификатор экземпляра терминала. Папка *Users* также имеет алиас "*Documents and Settings*".

> Обратите внимание, что если подключение к компьютеру осуществляется удаленно через RDP (Remote Desktop Protocol), то в любом случае используется каталог *Roaming* и его подкаталоги, даже если у вас есть административные права.

Напомним, что папка MQL5 в каталоге данных — это то место, где хранятся все MQL-программы: как их исходные коды, так и скомпилированные ex5-файлы. Каждый тип MQL-программ — индикаторы, эксперты, скрипты и прочие — имеет в папке MQL5 выделенную вложенную папку. Таким образом, папка *Files* для рабочих файлов находится с ними по соседству.

Помимо этой "персональной" "песочницы" каждой копии терминала на компьютере имеется общая, разделяемая "песочница" для всех терминалов: через неё они могут "общаться". Путь к ней "пролегает" через домашнюю папку пользователя Windows и может отличаться в зависимости от версии операционной системы. Например, в стандартных установках Windows 7, 8, 10 он имеет вид:

```
X:/Users/<user_name>/AppData/Roaming/MetaQuotes/Terminal/Common/Files/
```

Редактор MetaTrader и в данном случае облегчает поиск папки: достаточно выполнить команду *Файл -> Открыть общую папку данных*, и вы окажетесь внутри папки Common.

Следует напомнить, что некоторые типы MQL-программ (эксперты и индикаторы) способны выполняться не только в терминале, но и тестере. При работе в нем общая "песочница" остается доступной, а вместо "песочницы" отдельного экземпляра используется папка внутри агента тестирования. Как правило, она имеет вид:

```
X:/<путь_к_терминалу>/Tester/Agent-IP-port/MQL5/Files/
```

Из самой MQL-программы это обстоятельство незаметно, то есть все файловые функции работают совершенно одинаково, однако с точки зрения пользователя может сложиться впечатление, что существует какая-то проблема. Например, если программа сохранит результаты своей работы в файл, тот будет удален в папке агента тестера после окончания прогона (как будто файл и не создавался). Такое поведение задумано с целью предотвратить утечку потенциально ценных данных одной программы в другую программу, которая может тестироваться на том же агенте спустя какое-то время (тем более что агенты могут быть в общем доступе). Для передачи файлов на агенты и возврата результатов с агентов в терминал предусмотрены другие технологии, о которых мы поговорим в пятой части книги.

Для того чтобы обойти ограничение на "песочницу", можно воспользоваться способностью Windows назначать символические ссылки на объекты файловой системы. В нашем случае, для перенаправления доступа к папкам на локальном компьютере лучше всего подходят так называемые соединения (junction). Они создаются с помощью следующей команды (имеется в виду командная строка Windows):

```
mklink /J new_name existing_target
```

Параметр *new_name* — это имя новой виртуальной "папки", которая будет указывать на реальную папку *existing_target*.

Для создания соединений к внешним папками вне "песочницы" рекомендуется завести внутри MQL5/Files выделенную папку, например, *Links*. Затем зайдя в неё можно выполнить вышеуказанную команду, выбрав *new_name* и подставив реальный путь вне "песочницы" в качестве *existing_target*. Например, следующая команда создаст в папке *Links* новую ссылку с именем *Settings*, которая обеспечит обращение к папке MQL5/Presets:

```
mklink /J Settings "..\..\Presets\"
```

Относительный путь "..\..\" предполагает, что команда выполняется в указанной папке MQL5/Files/Links. Сочетание двух точек ".." обозначает переход из текущей папки в родительскую. Указанная дважды, эта комбинация предписывает два раза подняться наверх по иерархии пути. В результате целевая папка (*existing_target*) сформируется как MQL5/Presets. Но в параметре *existing_target* можно задавать и абсолютный путь.

Удалить символические ссылки можно как обычные файлы (но, разумеется, следует предварительно убедиться, что удаляется именно папка со значком стрелочки в её левом нижнем

углу, т.е. ссылка, а не оригинальная папка). Рекомендуется делать это сразу же после того, как необходимость в прорыве за границу "песочницы" отпала. Дело в том, что созданные виртуальные папки становятся доступны всем MQL-программам, а не только вашей, и не известно, как дополнительную свободу могу использовать чужие программы.

Во многих разделах главы речь пойдет об именах файлов. Они выступают в роли идентификаторов элементов файловой системы и для них существуют похожие правила, в том числе и кое-какие ограничения.

Напомним, что имя файла не может содержать некоторые символы, которые играют особые роли в файловой системе ('<', '>', '/', '\\', '"', ':', '|', '*', '?'), а также любые символы с кодами от 0 до 31 включительно.

Также в операционной системе зарезервированы для специального применения и не могут использоваться следующие имена файлов: CON, PRN, AUX, NUL, COM1, COM2, COM3, COM4, COM5, COM6, COM7, COM8, COM9, LPT1, LPT2, LPT3, LPT4, LPT5, LPT6, LPT7, LPT8, LPT9.

Следует учесть, что файловая система Windows не "видит" принципиальной разницы между буквами в разных регистрах, поэтому имена вроде "Name", "NAME", "name" ссылаются на один и тот же элемент.

В качестве символа разделителя между составными частями пути (вложенными папками и файлом) Windows позволяет использовать как обратную косую черту '\\', так и прямую '/'. Однако обратный слэш требуется экранировать (то есть, фактически писать его дважды) в строках MQL5, потому что сам символ '\' является специальным: с помощью него строятся последовательности управляющих символов, такие как '\r', '\n', '\t' и другие (см. раздел Символьные типы). Например, следующие пути эквивалентны: "MQL5Book/file.txt" и "MQL5Book\\file.txt".

Символ точки '.' служит разделителем между именем и расширением. Если элемент файловой системы имеет несколько точек в своем идентификаторе, то расширением считается фрагмент справа от самой правой точки, а все, что слева от неё, является именем. Название (перед точкой) или расширение (после точки) может быть пустым. Например, вот имя файла без расширения — "text", а вот файл без имени (только с расширением) — ".txt".

> Общая длина пути и имени файла в Windows имеет ограничения. При этом для управления файлами в MQL5 следует учитывать, что к их пути и имени будет спереди добавлен путь к самой "песочнице", то есть на названия файловых объектов в вызовах MQL-функций будет отведено еще меньше места. По умолчанию, общее ограничение длины составляет системную константу MAX_PATH, равную 260. Начиная с Windows 10 (сборки 1607) вы можете увеличить этот предел до 32767. Для этого необходимо сохранить следующий текст в reg-файл и запустить его, добавив в Реестр Windows.

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem]
"LongPathsEnabled"=dword:00000001
```

> Для других версий Windows можно воспользоваться "обходными маневрами" из командной строки. В частности, сократить путь можно с помощью рассмотренных выше соединений (создав виртуальную папку с кратким путем). Также можно применить команду оболочки — *subst*, например, *subst z: c:\very\long\path* (см. подробности в справке Windows).

4.5.1 Способы хранения информации: текстовый и двоичный

Как мы видели во многих предыдущих разделах, одна и та же информация может быть представлена в текстовом и двоичном виде. Например, числа форматов *int*, *long*, *double*, дата и время (*datetime*) или цвет (*color*) хранятся в памяти как последовательность байтов определенной длины. Этот способ компактный и хорошо подходит для интерпретации компьютером, но для человека удобнее анализировать информацию в текстовом виде, хотя он и длиннее. Поэтому мы уделили много внимания преобразованию чисел в строки и обратно и функциям для работы со строками.

На уровне файлов разделение на двоичное и текстовое представление данных тоже сохраняется. Двоичный файл предназначен для хранения данных в том же внутреннем представлении, какое используется в памяти. Текстовый файл содержит строковое представление.

Текстовые файлы обычно используются для стандартных форматов, таких как CSV (Comma Separated Values), JSON (JavaScript Object Notation), XML (Extensible Markup Language), HTML (HyperText Markup Language).

Двоичные (или по-другому, бинарные) файлы, конечно, тоже имеют стандартные форматы для многих применений, в частности, для изображений (PNG, GIF, JPG, BMP), звуков (WAV, MP3) или сжатых архивов (ZIP). Однако бинарный формат изначально предполагает большую закрытость и низкоуровневую работу с данными и потому чаще применяется для решения внутренних задач, когда важна лишь эффективность хранения и доступность данных для конкретной программы. Иными словами, объекты любых прикладных структур и классов могут легко сохранять и восстанавливать свое состояние в двоичном файле, фактически делая слепок памяти и не заботясь о совместимости с каким-либо стандартом.

В принципе, мы могли бы вручную конвертировать данные в строки при записи в двоичный файл и затем обратно преобразовывать их из строк в числа (или структуры, или массивы) при чтении файла. Это стало бы аналогом того, что для нас автоматически предоставляет режим текстовых файлов, но требовало бы дополнительных усилий. Режим текстового файла избавляет нас от подобной рутины. И более того, файловая подсистема MQL5 выполняет неявным образом несколько дополнительных, но важных операций, которые необходимы при работе с текстом.

Во-первых, понятие текста основывается на неких общеупотребительных правилах использования символов-разделителей. В частности, подразумевается, что все тексты состоят из строк — так их удобнее читать и анализировать алгоритмически. Следовательно, существуют специальные символы, отделяющие одну строку от другой.

Здесь нас ожидают первые сложности, связанные с тем, что в разных операционных системах приняты различные комбинации этих символов. В Windows разделителем строк является последовательность двух символов '\r\n' (или в виде шестнадцатеричных кодов: 0xD 0xA, или в виде названия CRLF, которое расшифровывается как возврат каретки ("Carriage Return") и перевод строки ("Line Feed")). В Unix и Linux стандартом является единичный символ '\n', а в некоторых версиях и программах под MacOS может использоваться единичный символ '\r'.

Хотя MetaTrader 5 работает под Windows, у нас нет гарантии, что какой-либо полученный текстовый файл не будет сохранен с "непривычными" разделителями. Если бы мы его читали в двоичном режиме и сами проверяли на наличие разделителей, чтобы сформировать строки, указанные разночтения потребовали бы специфической обработки. А текстовый режим работы файлов в MQL5 приходит здесь на помощь: он автоматически нормализует переводы строк при чтении и записи.

> Нужно помнить, что MQL5 обеспечивает исправление переносов строк не во всех случаях. В частности, одиночный символ '\r' не будет при чтении текстового файла интерпретирован как '\r\n'. В отличие от этого, одиночный '\n' правильно воспринимается как '\r\n'.

Во-вторых, строки могут храниться в памяти в нескольких представлениях. По-умолчанию строка (тип *string*) состоит в MQL5 из двухбайтовых символов. Это обеспечивает поддержку универсальной кодировки Unicode, которая хороша тем, что включает в себя все национальные алфавиты. Однако во многих случаях такой "интернационал" не требуется (например, при хранении чисел или англоязычных сообщений), и тогда экономнее использовать строки из однобайтовых символов в той или иной кодировке ANSI. Функции MQL5 API позволяют выбирать для файлов в текстовом режиме предпочтительный способ записи строк. Однако, если в своей MQL-программе мы сами контролируем запись и можем гарантировать обоснованность и надежность переключения с Unicode на однобайтовые символы, то при интеграции с неким внешним программным обеспечением или веб-сервисом, кодовая страница ANSI в его файлах может быть любой. В связи с этим, возникает следующий пункт.

В-третьих, из-за наличия множества национальных языков нужно быть готовым к текстам в различных ANSI-кодировках. Без правильной интерпретации кодировки текст может быть записан или прочитан с искажениями или вовсе стать нечитаемым. Мы видели это в разделе Работа с символами и кодовыми страницами. Поэтому файловые функции уже включают в себя средства для корректной обработки символов: достаточно лишь указать в параметрах желаемую или ожидаемую кодировку. О выборе кодировки более подробно рассказано в отдельном разделе.

Наконец, в-четвертых, текстовый режим имеет встроенную поддержку известного формата CSV. Поскольку в трейдинге часто приходится оперировать табличными данными, CSV для этого хорошо подходит. В текстовом файле в режиме CSV функции MQL5 API обрабатывают не только разделители для переноса строк текста, но и дополнительный разделитель для границы столбцов (полей в каждом ряду таблицы). Обычно это символ табуляции '\t', запятая ',' или точка с запятой ';'. Например, вот как выглядит CSV-файл с новостями Forex (разделитель — запятая, показан фрагмент):

```
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
```

И вот он же, для наглядности, в виде таблицы:

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

4.5.2 Запись и чтение файлов в упрощенном режиме

Среди файловых функций MQL5, которые предназначены для записи и чтения данных, существует разделение на 2 неравных группы. В первую из них входят две функции — *FileSave* и *FileLoad*, позволяющие записывать или считывать данные в двоичном режиме за один вызов функции. С одной стороны, данный подход имеет неоспоримое преимущество — простоту, но с другой — имеет некоторые ограничения (подробнее о них — чуть ниже). Во второй многочисленной группе все файловые функции используются иначе: требуется последовательно вызвать несколько из них, чтобы выполнить логически законченную операцию чтения или записи. Это кажется более сложным, но зато предоставляет гибкость и управление процессом. Функции второй групп оперируют особыми целыми числами — дескрипторами файлов, которые следует получить с помощью функции *FileOpen* (см. следующий раздел).

Познакомимся с формальным описанием этих двух функций, а затем рассмотрим их совместный пример (*FileSaveLoad.mq5*).

bool FileSave(const string filename, const void &data[], const int flag = 0)

Функция записывает в бинарный файл с именем *filename* все элементы переданного массива *data*. Параметр *filename* может содержать не только имя файла, но и имена папок нескольких уровней вложенности: функция создаст указанные папки, если их еще нет. Если файл существует, он будет перезаписан (если не занят другой программой).

В качестве параметра *data* может быть передан массив любых встроенных типов, кроме строк. Также это может быть массив простых структур, содержащих поля встроенных типов за исключением строк, динамических массивов и указателей. Классы также не поддерживаются.

Параметр *flag* может, при необходимости, содержать предопределенную константу FILE_COMMON, которая означает создание и запись файла в общий каталог данных всех терминалов (*Common/Files/*). Если флаг не указан (что соответствует значению по умолчанию 0), то файл

пишется в обычный каталог данных (если MQL-программа выполняется в терминале) или в каталог агента тестирования (если дело происходит в тестере). В двух последних случаях, как было описано в начале главы, внутри каталога используется "песочница" *MQL5/Files/*.

Функция возвращает признак успеха операции (*true*) или ошибки (*false*).

long FileLoad(const string filename, void &data[], const int flag = 0)

Функция считывает всё содержимое бинарного файла *filename* в указанный массив *data*. Имя файла может включать иерархию папок внутри "песочницы" *MQL5/Files* или *Common/Files*.

Массив *data* должен иметь любой встроенный тип кроме *string*, или тип простой структуры (см. выше).

Параметр *flag* управляет выбором каталога, где ищется и открывается файл: по умолчанию (при значении 0) — в стандартной "песочнице", а если задано значение FILE_COMMON, то — в общей для всех терминалов.

Функция возвращает количество прочитанных элементов или -1 в случае ошибки.

Обратите внимание, что данные из файла читаются блоками размером в один элемент массива. Если размер файла оказывается некратным размеру элемента, то оставшиеся данные пропускаются (не читаются). Например, при размере файла 10 байтов его чтение в массив типа *double* (*sizeof(double)=8*) приведет к тому, что фактически будет загружено только 8 байтов, то есть 1 элемент (и функция вернет 1). Оставшиеся 2 байта в конце файла будут проигнорированы.

В скрипте *FileSaveLoad.mq5* определим две структуры для тестов.

```
struct Pair
{
   short x, y;
};

struct Simple
{
   double d;
   int i;
   datetime t;
   color c;
   uchar a[10]; // массив фиксированного размера разрешен
   bool b;
   Pair p;      // составные поля (вложенные простые структуры) тоже разрешены

   // строки и динамические массивы вызовут ошибку компиляции при использовании
   // FileSave/FileLoad: structures or classes containing objects are not allowed
   // string s;
   // uchar a[];

   // указатели также не поддерживаются
   // void *ptr;
};
```

Структура *Simple* содержит поля большинства разрешенных типов, а также составное поле с типом структуры *Pair*. В функции *OnStart* заполним небольшой массив типа *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}},
   };
   ...
```

Файл для записи данных выберем вместе с вложенной папкой "MQL5Book", чтобы наши эксперименты не перемешивались с вашими рабочими файлами:

```
   const string filename = "MQL5Book/rawdata";
```

Запишем массив в файл, прочитаем его в другой массив и сравним их.

```
   PRT(FileSave(filename, write/*, FILE_COMMON*/)); // true

   Simple read[];
   PRT(FileLoad(filename, read/*, FILE_COMMON*/)); // 2

   PRT(ArrayCompare(write, read)); // 0
```

FileLoad вернула 2, то есть было прочитано 2 элемента (2 структуры). Результат сравнения 0 означает, что данные совпали. Вы можете в своем любимом файловом менеджере открыть папку *MQL5/Files/MQL5Book* и убедиться, что там есть файл "rawdata" (смотреть его внутренности

текстовым редактором не рекомендуется, используйте программу просмотра с поддержкой двоичного режима).

Далее в скрипте мы преобразуем прочтенный массив структур в байты и выводим их в журнал в виде шестнадцатеричных кодов: это своего рода дамп памяти — он позволяет понять, что из себя представляют бинарные файлы.

```
uchar bytes[];
for(int i = 0; i < ArraySize(read); ++i)
{
   uchar temp[];
   PRT(StructToCharArray(read[i], temp));
   ArrayCopy(bytes, temp, ArraySize(bytes));
}
ByteArrayPrint(bytes);
```

Результат:

```
[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 |
```

Поскольку встроенная функция *ArrayPrint* не умеет "печатать" в шестнадцатеричном формате, нам пришлось разработать собственную функцию *ByteArrayPrint* (здесь её исходный код не будем приводить, см. прилагаемый файл).

Далее, вспомним, что *FileLoad* способна загрузить данные в массив любого типа, поэтому прочтем тот же файл с помощью неё непосредственно в массив байтов.

```
uchar bytes2[];
PRT(FileLoad(filename, bytes2/*, FILE_COMMON*/)); // 78,  39 * 2
PRT(ArrayCompare(bytes, bytes2)); // 0, равенство
```

Успешное сравнение двух байтовых массивов показывает, что *FileLoad* "не стесняется" оперировать сырыми данными из файла произвольным образом: так, как ему "скажут" (в файле нет информации о том, что он хранит массив именно структур *Simple*).

Здесь важно отметить, что поскольку тип байт имеет минимальный размер (1), то ему кратен любой размер файла. Поэтому в байтовый массив любой файл всегда читается без остатка. Функция *FileLoad* вернула здесь число 78 (число элементов равно числу байтов). Это размер файла (две структуры по 39 байтов каждая).

В принципе, способность *FileLoad* интерпретировать данные под любой тип требует осторожности и проверок со стороны программиста. В частности, далее в скрипте мы читаем тот же файл в массив структур *MqlDateTime*. Это, конечно же, неправильно, но работает без ошибок.

```
MqlDateTime mdt[];
PRT(sizeof(MqlDateTime)); // 32
PRT(FileLoad(filename, mdt)); // 2
// внимание: 14 байтов осталось непрочитанными
ArrayPrint(mdt);
```

Результат содержит бессмысленный набор чисел:

```
          [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
```

Поскольку размер *MqlDateTime* равен 32, то в файле длиной 78 байтов укладывается только две таких структуры, причем лишними остаются еще 14 байтов. Само наличие остатка свидетельствует о проблеме. Но даже если его нет, это не гарантирует осмысленность выполненной операции, потому что два разных размера могут чисто случайно укладываться целое (но разное) число раз в длину файла. Более того, две разных по смыслу структуры могут иметь одинаковый размер, но это не значит, что их следует записывать и читать из одной в другую.

Неудивительно, что вывод в журнал массива структур *MqlDateTime* показывает "сумасшедшие" величины: ведь это был, на самом деле, совсем другой тип данных.

Чтобы сделать чтение в некоторой степени более "осторожным", в скрипте реализован аналог функции *FileLoad* — *MyFileLoad*. Детально мы разберем эту функцию, а также парную ей *MyFileSave*, в следующих разделах, когда изучим новые файловые функции и с их помощью смоделируем внутреннее устройство *FileSave/FileLoad*. А пока лишь отметим, что в своей версии мы можем проверять наличие непрочитанного остатка в файле и выводить предупреждение.

В заключение разберем еще пару потенциальных ошибок, продемонстрированных в скрипте.

```
/*
// ошибка компиляции, тип string не поддерживается здесь
string texts[];
FileSave("any", texts); // parameter conversion not allowed
*/

double data[];
PRT(FileLoad("any", data)); // -1
PRT(_LastError); // 5004, ERR_CANNOT_OPEN_FILE
```

Первая из них происходит во время компиляции (в связи с чем блок кода закомментирован), потому что строковые массивы запрещены.

Вторая заключается в чтении несуществующего файла, из-за чего *FileLoad* возвращает -1. Пояснительный код ошибки легко получить с помощью *GetLastError* (или *_LastError*).

4.5.3 Открытие и закрытие файлов

Для записи и чтения данных из файла большинство функций MQL5 требуют предварительно этот файл открыть. Для этой цели предназначена функция *FileOpen*. После выполнения требуемых операций открытый файл следует закрыть с помощью функции *FileClose*. Дело в том, что открытый файл может, в зависимости от примененных опций, быть заблокированным для доступа из других программ. Кроме того, операции с файлами буферизуются в памяти (кэше) из соображений повышения быстродействия, и без закрытия файла новые данные в него могут физически не выгружаться на протяжении какого-то времени. Это особенно критично, если записываемые данные ждет внешняя программа (например, при интеграции MQL-программы с другими системами). Про альтернативный способ "сброса" буфера на диск мы узнаем из описания функции *FileFlush*.

С открытым файлом в MQL-программе связывается специальное целое число — дескриптор. Его возвращает функция *FileOpen*. Все операции, связанные с доступом или модификацией внутреннего содержимого файла, требуют указания этого идентификатора в соответствующих функциях API. Те функции, которые оперируют файлом целиком (копируют, удаляют, перемещают, проверяют существование), не требуют дескриптора. Для выполнения этих действий открывать файл не надо.

```
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)
```

Функция открывает файл с указанным именем и в режиме, заданном параметром *flags*. Параметр *filename* может содержать вложенные папки перед непосредственным именем файла. В этом случае, если файл открывается на запись и требуемая иерархия папок еще не существует, она будет создана.

Параметр *flags* должен содержать комбинацию констант, описывающих требуемый режим работы с файлом. Комбинация выполняется с помощью операций побитового ИЛИ. Ниже приведена таблица доступных констант.

Идентификатор	Значение	Описание
FILE_READ	1	Файл открывается для чтения
FILE_WRITE	2	Файл открывается для записи
FILE_BIN	4	Двоичный режим чтения-записи, без преобразования данных из строки и в строку
FILE_CSV	8	Файл типа CSV; записываемые данные преобразуются в текст соответствующего типа (Unicode или ANSI, см. далее), а при чтении производится обратная конвертация из текста в требуемый тип (указывается в функции чтения); одна CSV-запись — это отдельная строка текста, ограниченная символами перевода строки (обычно CRLF); внутри CSV-записи элементы разделяются символом-разделителем (параметр *delimiter*);
FILE_TXT	16	Простой текстовый файл, аналогичный режиму CSV, но символ-разделитель не используется (значение параметра *delimiter* игнорируется)
FILE_ANSI	32	Строки типа ANSI (однобайтовые символы)
FILE_UNICODE	64	Строки типа Unicode (двухбайтовые символы)
FILE_SHARE_READ	128	Совместный доступ по чтению со стороны нескольких программ
FILE_SHARE_WRITE	256	Совместный доступ по записи со стороны нескольких программ
FILE_REWRITE	512	Разрешение перезаписать файл (если он уже существует) в функциях FileCopy и FileMove
FILE_COMMON	4096	Расположение файла в общей папке всех клиентских терминалов /Terminal/Common/Files (флаг используется при открытии файлов (FileOpen), копировании файлов (FileCopy, FileMove) и проверке существования файлов (FileIsExist))

При открытии файла обязательно должен быть указан один из флагов FILE_WRITE, FILE_READ или их комбинация.

Флаги FILE_SHARE_READ и FILE_SHARE_WRITE не заменяют и не отменяют необходимости указывать флаги FILE_READ и FILE_WRITE.

> Среда исполнения MQL-программ всегда буферизует файлы на чтение, что эквивалентно неявному добавлению флага FILE_READ. В связи с этим для нормальной работы с разделяемыми файлами всегда следует использовать FILE_SHARE_READ (даже если известно, что другой процесс открывает файл только на запись).

Если не указан ни один из флагов FILE_CSV, FILE_BIN, FILE_TXT, подразумевается FILE_CSV, как наиболее приоритетный. Если указано несколько из этих трех флагов, применяется наиболее приоритетный из переданных (выше они перечислены в порядке убывания приоритета).

Для текстовых файлов по умолчанию действует режим FILE_UNICODE.

Параметр *delimiter*, влияющий только на CSV, может быть типа *ushort* или *string*. Во втором случае, если длина строки больше 1, будет использован только первый её символ.

Параметр *codepage* влияет только на файлы, открываемые в текстовом режиме (FILE_TXT или FILE_CSV), причем только в том случае, если для строк выбран режим FILE_ANSI. Если строки хранятся в Unicode (FILE_UNICODE), кодовая страница не важна.

В случае успеха функция возвращает дескриптор файла — целое положительное число. Оно уникально только внутри конкретной MQL-программы, обмениваться им с другими программами бессмысленно. Для дальнейшей работы с файлом дескриптор передается в вызовы других функций.

В случае ошибки результат равен INVALID_HANDLE (-1). Суть ошибки следует выяснять из кода, возвращаемого функцией *GetLastError*.

Все настройки режима работы, сделанные в момент открытия файла, сохраняются без изменений все время, пока файл открыт. Если возникнет необходимость изменить режим, файл следует закрыть и открыть вновь с новыми параметрами.

Для каждого открытого файла среда выполнения MQL-программ поддерживает внутренний указатель, то есть текущую позицию внутри файла. Сразу после открытия файла указатель установлен на начало (позиция равна 0). В процессе записи или чтения позиция соответствующим образом сдвигается, согласно передаваемому или получаемому объему данных из различных файловых функций. Также существует возможность напрямую влиять на позицию (перемещать назад или вперед). Все эти возможности будут рассмотрены в последующих разделах.

FILE_READ и FILE_WRITE в разных сочетаниях позволяют добиться нескольких сценариев:

- FILE_READ — открытие файла, только если он существует; в противном случае функция вернет ошибку, а новый файл не создается;
- FILE_WRITE — создание нового файла, если его еще не было, или открытие существующего файла, причем его содержимое очищается, а размер обнуляется;
- FILE_READ|FILE_WRITE — открытие существующего файла вместе со всем его содержимым или создание нового файл, если его еще не было.

Как видно, некоторые сценарии недоступны только за счет флагов. В частности, открыть файл на запись, только если он уже существует, — нельзя. Этого можно добиться, используя дополнительные функции, например, *FileIsExist*. Также не получится "автоматом" обнулить файл, открываемый для комбинации чтения и записи: в этом случае MQL5 всегда оставляет содержимое.

Чтобы дописывать данные в файл нужно не только открыть файл в режиме FILE_READ|FILE_WRITE, но и переместить текущую позицию внутри файла в его конец с помощью вызова *FileSeek*.

Правильное описание разделяемого доступа к файлу — обязательное условие успешности выполнения *FileOpen*. Управляется данный аспект следующим образом.

- Если не указан ни один из флагов FILE_SHARE_READ и FILE_SHARE_WRITE, то текущая программа получает исключительный доступ к файлу, если открывает его первой. Если этот же файл уже был открыт кем-то раньше (другой или этой же программой), вызов функции завершится ошибкой.

- При установке флага FILE_SHARE_READ, программа разрешает последующие запросы на открытие того же файла на чтение. Если в момент вызова функции файл уже открыт другой или этой же программой на чтение, а данный флаг не установлен, функция завершится ошибкой.
- При установке флага FILE_SHARE_WRITE, программа разрешает последующие запросы на открытие того же файла на запись. Если в момент вызова функции файл уже открыт другой или этой же программой на запись, а данный флаг не установлен, функция завершится ошибкой.

Разделение доступа проверяется не только в отношении других MQL-программ или внешних по отношению к MetaTrader 5 процессов, но и к той же самой MQL-программе, если она открывает файл повторно.

Таким образом, наименее конфликтный режим подразумевает указание обоих флагов, но он все равно не гарантирует открытия файла, если кому-то уже выдан для него дескриптор с запретом на совместный доступ. Вместе с тем, следует придерживаться более строгих правил в зависимости от планируемых операций чтения или записи.

Например, при открытии файла на чтение имеет смысл оставить возможность его чтения другими. Дополнительно можно, вероятно, разрешить другим и писать в него, если речь о пополняемом файле (например, журнале). Однако при открытии файла на запись, вряд ли стоит оставлять право записи для других: это привело бы к непредсказуемому наложению данных.

void FileClose(int handle)

Функция закрывает ранее открытый файл по его дескриптору.

После закрытия файла его дескриптор в программе становится недействительным: попытка вызвать для него какую-либо файловую функцию закончится ошибкой. Однако вы можете использовать ту же переменную для хранения другого дескриптора, если вновь откроете этот же или другой файл.

При завершении программы открытые файлы принудительно закрываются, а буфер записи, если он не пустой, записывается на диск. Вместе с тем рекомендуется закрывать файлы явно.

Закрытие файла по завершении работы с ним — важное правило, которому необходимо следовать. Это связано не только с кэшированием записываемой информации, которая может какое-то время оставаться в оперативной памяти и не сохраненной на диск (о чем уже упоминалось выше), если не закрыть файл. Помимо этого открытый файл потребляет некий внутренний ресурс операционной системы, и речь здесь — не о месте на диске. Количество одновременно открытых файлов ограничено (может быть несколько сотен или тысяч в зависимости от настроек Windows). Если множество программ будет держать открытыми большое количество файлов, этот лимит может быть исчерпан, и попытки открыть новые файлы завершатся ошибками.

В связи с этим желательно обезопасить себя от возможной "потери" дескрипторов с помощью класса-обертки, который бы открывал файл и получал дескриптор при создании объекта, а освобождение дескриптора и закрытие файла происходило бы автоматически в деструкторе.

Мы создадим класс-обертку, после того как проверим работу функций *FileOpen* и *FileClose* в чистом виде.

Но прежде чем окунуться в файловую специфику, подготовим новую версию макроса для наглядного вывода в журнал вызовов наших функций. Новая версия потребовалась потому, что

до сих пор макросы типа PRT и PRTS (использовавшиеся в предыдущих разделах) "проглатывали" во время печати возвращаемые значения функций. Например, мы писали:

```
PRT(FileLoad(filename, read));
```

Здесь результат вызова *FileLoad* отправляется в журнал, но получить его в вызывающей строке кода невозможно. Правда, нам это было и не нужно. Но сейчас функция *FileOpen* будет возвращать дескриптор файла, и его следует сохранить в переменной для дальнейших манипуляций с файлом.

Проблем со старыми макросами целых две. Во-первых, они основаны на функции *Print*, которая "поглощает" переданные данные (отправляя их в журнал), но сама ничего не возвращает. Во-вторых, какое-либо значение для переменной с результатом можно получить только из выражения, а вызов *Print* невозможно сделать частью выражения в силу того, что она имеет тип *void*.

Для решения проблем нам потребуется вспомогательная функция печати, возвращающая печатаемое значение. А её вызов мы "упакуем" в новый макрос 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(); // очистка флага ошибки для следующего вызова
   return retval;
}
```

С помощью магического оператора преобразования в строку '#' мы получаем подробный описатель фрагмента кода (выражения A), который передается первым аргументом в *ResultPrint*. Само выражение (аргумент макроса) вычисляется (если там есть функция, она вызывается) и его результат передается вторым аргументом в *ResultPrint*. Далее уже в дело вступает привычная функция Print, а в заключение тот же результат возвращается в вызывающий код.

Для того чтобы не заглядывать в справку за расшифровкой кодов ошибок был подготовлен макрос E2S, который использует перечисление MQL_ERROR со всеми ошибками MQL5. Его можно найти в заголовочном файле *MQL5/Include/MQL5Book/MqlError.mqh*. Сам новый макрос и функция *ResultPrint* определены в файле PRTF.mqh, рядом с тестовыми скриптами.

В скрипте *FileOpenClose.mq5* попробуем открыть разные файлы, причем один и тот же файл будет открываться параллельно несколько раз. В реальных программах так обычно не делают. Одного дескриптора конкретного файла в экземпляре программы достаточно для большинства задач.

Один из файлов — *MQL5Book/rawdata* — должен уже существовать, поскольку был создан скриптом из раздела Запись и чтение файлов в упрощенном режиме. Другой файл создадим в ходе теста.

Тип файлов выберем FILE_BIN, но это не принципиально: с FILE_TXT или FILE_CSV работа на данном этапе ведется аналогично.

Под дескрипторы файлов зарезервируем массив, чтобы в конце скрипта закрыть все файлы сразу.

Первым откроем *MQL5Book/rawdata* в режиме чтения без разделения доступа. Предполагая, что файл не занят никаким сторонним приложением, можем ожидать успешное получение дескриптора.

```
void OnStart()
{
   int ha[4] = {}; // массив под дескрипторы тестовых файлов

   // этот файл должен существовать после запуска FileSaveLoad.mq5
   const string rawdata = "MQL5Book/rawdata";
   ha[0] = PRTF(FileOpen(rawdata, FILE_BIN | FILE_READ)); // 1 / ok
```

Если попытаемся открыть тот же файл еще раз, столкнемся с ошибкой, потому что ни первый, ни второй вызов не разрешает совместный доступ.

```
  ha[1] = PRTF(FileOpen(rawdata, FILE_BIN | FILE_READ)); // -1 / CANNOT_OPEN_FILE(5004
```

Закроем первый дескриптор, откроем файл заново, но уже с правами на совместное чтение, и убедимся, что повторное открытие теперь работает (правда в нем также требуется разрешить разделяемое чтение):

```
   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
```

Открыть файл на запись (FILE_WRITE) не получится, поскольку два предыдущих вызова *FileOpen* разрешают только FILE_SHARE_READ.

```
   ha[2] = PRTF(FileOpen(rawdata, FILE_BIN | FILE_READ | FILE_WRITE | FILE_SHARE_READ
   // -1 / CANNOT_OPEN_FILE(5004)
```

Теперь попробуем создать новый файл *MQL5Book/newdata*. Если открывать его только для чтения, файл не создастся.

```
   const string newdata = "MQL5Book/newdata";
   ha[3] = PRTF(FileOpen(newdata, FILE_BIN | FILE_READ));
   // -1 / CANNOT_OPEN_FILE(5004)
```

Для создания файла необходимо указать режим FILE_WRITE (наличие FILE_READ здесь некритично, но делает вызов более универсальным: как мы помним, в таком сочетании инструкция гарантирует, что откроется либо старый файл, если он есть, либо создастся новый).

```
   ha[3] = PRTF(FileOpen(newdata, FILE_BIN | FILE_READ | FILE_WRITE)); // 3 / ok
```

Попытаемся что-нибудь записать в новый файл с помощью известной нам функции *FileSave*. Она выступает в роли "внешнего игрока", поскольку работает с файлом в обход нашего дескриптора, примерно также как могла бы действовать другая MQL-программа или стороннее приложение.

```
   long x[1] = {0x123456789ABCDEF0};
   PRTF(FileSave(newdata, x)); // false
```

Этот вызов заканчивается неудачей, потому что дескриптор был открыт без прав совместного доступа. Закроем и вновь откроем файл с максимальными "разрешениями".

```
      FileClose(ha[3]);
      ha[3] = PRTF(FileOpen(newdata,
         FILE_BIN | FILE_READ | FILE_WRITE | FILE_SHARE_READ | FILE_SHARE_WRITE)); // 3
```

На этот раз *FileSave* работает, как ожидалось.

```
      PRTF(FileSave(newdata, x)); // true
```

Вы можете заглянуть в папку *MQL5/Files/MQL5Book/* и найти там файл *newdata* длиной 8 байтов.

Обратите внимание, что после того как мы закрываем файл, его дескриптор возвращается в пул свободных дескрипторов, и при следующем открытии файла (может быть, другого файла) тот же номер вновь вступает в дело.

Для аккуратного завершения работы закроем явным образом все открытые файлы.

```
      for(int i = 0; i < ArraySize(ha); ++i)
      {
         if(ha[i] != INVALID_HANDLE)
         {
           FileClose(ha[i]);
         }
      }
   }
```

4.5.4 Управление дескрипторами файлов

Необходимость постоянно помнить об открытых файлах и освобождать локальные дескрипторы при любом выходе из функций, приводит к мысли поручить всю рутину специальным объектам.

Данный принцип хорошо известен в программировании и называется инициализацией с захватом ресурса (RAII, Resource Acquisition Is Initialization). Использование RAII упрощает контроль за ресурсами и обеспечивает их корректное состояние. В частности, это особенно эффективно, если выход из функции, где открывается файл (и для него создается объект-владелец), производится из нескольких разных мест.

Область применения RAII не ограничивается файлами. В разделе Шаблоны объектных типов мы создали класс *AutoPtr*, который управляет указателем на объект и является другим примером данной концепции. Ведь указатель — это тоже ресурс (память), и его очень просто "потерять" и накладно освобождать в нескольких разных ветках алгоритма.

Класс-обертка для файлов может быть полезен и с другой точки зрения. Одно из упущений файлового API заключается в отсутствии функции, которая позволила бы по дескриптору получить имя файла (несмотря на то, что такая связь, безусловно, существует внутри). Вместе с тем, внутри объекта мы можем сохранить это имя и осуществить собственную его привязку к дескриптору.

В простейшем случае нам нужен некий класс, который хранит в себе файловый дескриптор и автоматически закрывает его в деструкторе. Пример реализации показан в файле *FileHandle.mqh*.

```
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;
   }
   ...
```

Два конструктора, а также перегруженный оператор присваивания обеспечивают привязку объекта к файлу (дескриптору). Второй конструктор позволяет передать ссылку на локальную переменную (из вызывающего кода), в которую дополнительно попадет новый дескриптор. Это будет своего рода внешний алиас того же дескриптора, который можно будет привычным образом использовать в других вызовах функций.

Но можно обойтись и без алиаса. Для этих случаев в классе определен оператор '~', возвращающий значение внутренней переменной *handle*.

```
   int operator~() const
   {
      return handle;
   }
```

Наконец, самое главное, ради чего класс и был задуман, — "умный" деструктор:

```
~FileHandle()
{
   if(handle != INVALID_HANDLE)
   {
      ResetLastError();
      // установит код внутренней ошибки, если handle недействителен
      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);
      }
   }
}
```

В нем, после нескольких проверок, вызывается *FileClose* для контролируемой переменной *handle*. Дело в том, что файл может быть закрыт явным образом в другом месте программы, хотя это больше и не требуется при наличии данного класса. В результате, дескриптор может стать недействительным к моменту вызова деструктора, когда исполнение алгоритма покинет блок, в котором определен объект *FileHandle*. Для выяснения этого обстоятельства используется фиктивный вызов функции *FileGetInteger*: фиктивный, потому что он ничего полезного не делает. Если после вызова код внутренней ошибки остался равным 0, дескриптор рабочий.

В принципе, всех этих проверок можно не делать и написать просто:

```
~FileHandle()
{
   if(handle != INVALID_HANDLE)
   {
      FileClose(handle);
   }
}
```

Если дескриптор испорчен, *FileClose* не станет "возмущаться". Но мы добавили проверки, чтобы иметь возможность вывести диагностическую информацию.

Попробуем класс *FileHandle* в действии. Тестовый скрипт для него называется *FileHandle.mq5*.

```
const string dummy = "MQL5Book/dummy";

void OnStart()
{
   // создаем новый файл или открываем существующий и обнуляем его
   FileHandle fh1(PRTF(FileOpen(dummy,
      FILE_TXT | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ))); // 1
   // другой вариант подключения дескриптора через '='
   int h = PRTF(FileOpen(dummy,
      FILE_TXT | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ)); // 2
   FileHandle fh2 = h;
   // и еще один поддерживаемый синтаксис:
   // int f;
   // FileHandle ff(f, FileOpen(dummy,
   //    FILE_TXT | FILE_WRITE | FILE_SHARE_WRITE | FILE_SHARE_READ));

   // здесь предполагается запись данных
   // ...

   // закрываем файл вручную (это не обязательно: сделано, чтобы продемонстрировать,
   // что FileHandle это обнаружит и не станет пытаться закрыть повторно)
   FileClose(~fh1); // оператор '~', примененный к объекту, возвращает дескриптор

   // дескриптор в переменной 'h', привязанный к объекту 'fh2', не закрыт вручную
   // и будет автоматически закрыт в деструкторе
}
```

Согласно выводу в журнал, всё действует по плану:

```
FileHandle::~FileHandle: Automatic close for handle: 2
FileHandle::~FileHandle: handle 1 is incorrect, INVALID_FILEHANDLE(5007)
```

Однако, если файлов много, создание под каждый из них собственного экземпляра следящего объекта может стать неудобством. Для таких ситуаций имеет смысл спроектировать единый объект, собирающий "под свое крыло" все дескрипторы в заданном контексте (например, внутри функции).

Такой класс реализован в файле *FileHolder.mqh* и демонстрируется в скрипте *FileHolder.mq5*. Один экземпляр *FileHolder* сам создает по запросу вспомогательные объекты-наблюдатели класса *FileOpener*, который имеет общие черты с *FileHandle*, в особенности, деструктор, а также поле *handle*.

Для открытия файла через *FileHolder* следует использовать его метод *FileOpen* (его сигнатура повторяет сигнатуру стандартной функции *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;
   }
```

Все объекты *FileOpener* складываются в массиве *files* для отслеживания их времени жизни. Там же нулевыми элементами отмечаются моменты регистрации локальных контекстов (блоков кода), в которых создаются объекты *FileHolder* — за это отвечает конструктор *FileHolder*.

```
   FileHolder()
   {
      const int n = expand();
      if(n > -1)
      {
         files[n] = NULL;
      }
   }
```

Как мы знаем, в процессе выполнения программы, она заходит во вложенные блоки кода (вызывает функции). Если в них требуется управление локальными дескрипторами файлов, там следует описать объекты *FileHolder* (по одному на блок или реже). Все такие описания по правилам стека (первым пришел, последним ушел) складываются в *files* и затем освобождаются в обратном порядке по мере того, как программа покидает контексты. В каждый такой момент вызывается деструктор.

```
   ~FileHolder()
   {
      for(int i = ArraySize(files) - 1; i >= 0; --i)
      {
         if(files[i] == NULL)
         {
            // уменьшаем массив и выходим
            ArrayResize(files, i);
            return;
         }
         
         delete files[i];
      }
   }
```

Его задача — удалить последние объекты *FileOpener* в массиве вплоть до первого встреченного нулевого элемента, который означает границу контекста (далее в массиве идут дескрипторы из другого, внешнего контекста).

В полном объеме данный класс оставлен для самостоятельного изучения.

Посмотрим на его применение в тестовом скрипте *FileHolder.mq5*. Помимо функции *OnStart* здесь имеется функция *SubFunc*, и в обоих контекстах ведется работа с файлами.

```
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));
   // используем h и f
   // ...
   // не нужно закрывать файлы вручную и отслеживать ранние выходы из функции
   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));
   // запись данных и другие действия над файлом по дескриптору
   // ...
   /*
   int a[] = {1, 2, 3};
   FileWriteArray(h, a);
   */

   SubFunc();
   SubFunc();

   if(rand() > 32000) // имитируем ветвление по условиям
   {
      // благодаря holder нам не нужен явный вызов
      // FileClose(h);
      Print(__FUNCTION__, " return");
      return; // может быть много выходов из функции
   }

   /*
      ... еще код
   */

   // благодаря holder нам не нужен явный вызов
   // FileClose(h);
   Print(__FUNCTION__, " exit");
}
```

Мы не закрыли ни один дескриптор вручную, экземпляры *FileHolder* сделают это автоматически в деструкторах.

Вот пример вывода в журнал:

```
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 Выбор кодировки для текстового режима

Для записываемых текстовых файлов кодировку следует выбирать исходя из особенностей текста или подстраиваясь под требования внешних программ, для которых предназначены генерируемые файлы. Если внешних требований нет, можно придерживаться правила всегда использовать ANSI для простых текстов с числами, английскими буквами и пунктуацией (таблица 128 таких международных символов приведена в разделе Сравнение строк). При работе с национальными языками или спецсимволами используйте UTF-8 или Unicode, т.е. соответственно:

```
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);
```

Например, эти настройки пригодятся для сохранения в файл названий финансовых инструментов, поскольку в них иногда используются спецсимволы, обозначающие валюты или режимы торговли.

Чтение собственных файлов не должно составлять проблему, потому что здесь достаточно указать при чтении те же настройки кодировки, что были сделаны при записи. Однако текстовые файлы могут поступать из разных источников. Их кодировка может быть неизвестна или меняться без предварительного уведомления. Поэтому встает вопрос, что делать, если часть файлов может поставляться в виде однобайтовых строк (ANSI), часть — в виде "широких" двухбайтовых (Unicode), а часть — в кодировке UTF-8.

Понятно, что выбор кодировки можно делать с помощью входных параметров программы, однако это действенно только для одного файла, а если в процессе работы приходится открывать много разных файлов, их кодировки могут не совпадать. Поэтому желательно поручить самой системе осуществлять правильный выбор режимов на лету (от файла к файлу).

MQL5 не позволяет полностью автоматически детектировать и применять правильные кодировки, однако существует один наиболее универсальный режим для считывать разнообразных текстовых файлов. Для этого нужно задать следующие входные параметры функции *FileOpen*:

```
int h = FileOpen(filename, FILE_READ | FILE_TXT | FILE_ANSI, 0, CP_UTF8);
```

Здесь на нас работает несколько факторов.

Во-первых, кодировка UTF-8 прозрачно пропускает упомянутые 128 символов в любой кодировке ANSI (т.е. они передаются "один в один").

Во-вторых, она является наиболее популярной в протоколах Интернет.

В-третьих, в MQL5 встроен дополнительный анализ на форматирование текста в двухбайтовом Unicode, который позволяет при необходимости автоматически переключить режим работы файла в FILE_UNICODE, невзирая на заданные параметры. Дело в том, что файлы в формате Unicode обычно предваряются специальной парой символов: 0xFFFE или наоборот 0xFEFF. Эта последовательность называется меткой порядка байтов (Byte Order Mark, BOM). Она нужна из-за того, что, как мы знаем, байты могут храниться внутри чисел в разном порядке на разных платформах (об этом говорилось в разделе Управление порядком байтов в целых числах).

Формат FILE_UNICODE использует 2-байтовое целое число (код) на каждый символ, поэтому порядок байтов становится важен, в отличие от прочих кодировок. BOM, соответствующий порядку байт для Windows, — это последовательность 0xFFFE. Если ядро MQL5 находит эту метку в начале текстового файла, его чтение автоматически переключается в режиме Unicode.

Давайте посмотрим, как различные настройки режимов работают с текстовыми файлами разных кодировок. Для этого подготовлен скрипт *FileText.mq5* и несколько текстовых файлов с одним и тем же содержимым, но в разных кодировках (в скобках указан размер в байтах):

- ansi1252.txt (50) — европейская кодировка 1252 (будет отображаться целиком без искажений в Windows с европейским языком);
- unicode1.txt (102) — двухбайтовый Unicode, в начале идет присущий Windows BOM 0xFFFE;
- unicode2.txt (100) — двухбайтовый Unicode без BOM (в принципе, BOM является опциональным);
- unicode3.txt (102) — двухбайтовый Unicode, в начале идет BOM, присущий Unix, 0xFEFF;
- utf8.txt (54) — кодировка UTF-8.

В функции *OnStart* мы считываем эти файлы в циклах при разных настройках *FileOpen*. Обратите внимание, что за счет использования *FileHandle* (рассмотренного в предыдущем разделе) нам не приходится заботиться о закрытии файлов: все происходит автоматически внутри каждой итерации.

```
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));
   }
}
```

Функция *FileReadString* читает из файла строку, мы рассмотрим её в разделе о записи и чтении переменных.

Вот пример журнала с результатами выполнения скрипта:

```
=====> 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 -> 桔獩椠⁳⁡整瑸眠瑩⁨灳捥慩档牡捡整獲›㾱⼠ꌠ⼠ꔠ
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 -> 吀栀椀猀�椀猀�愀�琀攀砀琀�眀椀琀栀�猀瀀攀挀椀愀氀�挀栀愀
MQL5Book/utf8.txt -> 桔獩椠⁳⁡整瑸眠瑩⁨灳捥慩档牡捡整獲›ꇂ쒹⼠술⼠슥
=====> 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¥
```

Файл unicode1.txt всегда читается правильно, потому что в нем есть BOM 0xFFFE, и система игнорирует настройки в исходном коде. Однако, если метка отсутствует или имеет обратный порядок байтов, это автоопределение не работает. Также при установке FILE_UNICODE мы теряем способность читать однобайтовые тексты и UTF-8.

В результате, более устойчивым к вариациям форматирования следует считать вышеупомянутое сочетание FILE_ANSI и CP_UTF8. Выбор специфической национальной кодовой страницы рекомендуется только при явном требовании.

Несмотря на существенную помощь для программиста со стороны API при работе с файлами в текстовом режиме, мы можем при необходимости отказаться от режима FILE_TXT или FILE_CSV, и открыть по сути текстовый файл в двоичном режиме FILE_BINARY. Это переложит все сложности по парсингу текста и определению кодировки на плечи программиста, зато позволит поддержать другие нестандартные форматы. Но основной момент здесь в том, что текст можно считывать и записывать в файл, открытый в двоичном режиме. А вот обратное, в общем случае, невозможно. Бинарный файл с произвольными данными (то есть, если в него не были записаны исключительно строки), открытый в текстовом режиме, будет, скорее всего, интерпретироваться как текстовая "абракадабра". Если вам требуется записать бинарные данные в текстовый файл, воспользуйтесь предварительно функцией *CryptEncode* и кодировкой CRYPT_BASE64.

4.5.6 Запись и чтение массивов

Две функции MQL5 предназначены для записи и чтения массивов: *FileWriteArray* и *FileReadArray*. С двоичными файлами они позволяют обрабатывать массивы любых встроенных типов кроме строк, а также массивы простых структур, в которых нет строковых полей, объектов, указателей и динамических массивов. Данные ограничения связаны с оптимизацией процессов записи и чтения, которая возможна за счет исключения типов с переменной длиной. А таковыми как раз и являются строки, объекты и динамические массивы.

Вместе с тем, при работе с текстовыми файлами данные функции способны оперировать массивами типа *string* (прочие типы массивов в файлах с режимом FILE_TXT/FILE_CSV не допускаются этими функциями). Такие массивы хранятся в файле в формате: по одному элементу на каждой строке.

Если необходимо хранить в файле структуры или классы без ограничений по типам, используйте специализированные по типам функции, обрабатывающие за один вызов одно значение. Они описаны в двух разделах о записи и чтении переменных встроенных типов: для двоичных и текстовых файлов.

Кроме того, поддержку структур со строками можно организовать за счет внутренней оптимизации хранения информации. Например, вместо строковых полей можно применить целочисленные, которые будут содержать индексы соответствующих строк в отдельном массиве со строками. Учитывая возможность переопределить средствами ООП многие операции (в частности, присваивание) и получение структурного элемента массива по номеру, внешний вид алгоритма практически не изменится. Зато при записи можно сначала открыть файл в двоичном режиме и вызвать *FileWriteArray* для массива с "упрощенным" типом структур, а затем переоткрыть файл в текстовом режиме и дописать в него массив всех строк с помощью второго вызова *FileWriteArray*. Для чтения такого файла следует предусмотреть в его начале некий заголовок, содержащий количество элементов в массивах, чтобы передать его в качестве параметра *count* в *FileReadArray* (см. далее).

Если требуется сохранить или прочитать не массив структур, а одну структуру, воспользуйтесь функциями *FileWriteStruct* и *FileReadStruct*, описанными в следующем разделе.

Изучим сигнатуры функций, а потом рассмотрим общий пример (*FileArray.mq5*).

uint FileWriteArray(int handle, const void &array[], int start = 0, int count = WHOLE_ARRAY)

Функция записывает в файл с дескриптором *handle* массив *array*, который может быть многомерным. Параметры *start* и *count* позволяют задать диапазон элементов, по умолчанию он равен всему массиву. В случае многомерных массивов индекс *start* и количество элементов *count* относятся к сквозной нумерации по всем измерениям, а не к первому измерению массива. Например, если массив имеет конфигурацию [][5], то значение *start*, равное 7, укажет на элемент с индексами [1][2], а *count = 2* добавит к нему элемент [1][3].

Функция возвращает количество записанных элементов. В случае ошибки это будет 0.

Если *handle* получен в двоичном режиме, массивы могут быть любых встроенных типов, кроме строк, или типов простых структур. Если *handle* открыт в любом из текстовых режимов, массив должен быть типа *string*.

uint FileReadArray(int handle, const void &array[], int start = 0, int count = WHOLE_ARRAY)

Функция читает из файла с дескриптором *handle* данные в массив. Массив может быть многомерным и динамическим. Для многомерных массивов параметры *start* и *count* работают по принципу сквозной нумерации элементов по всем измерениям, описанному выше. Динамический массив при необходимости автоматически увеличивается в размере под читаемые данные. Если *start* больше исходной длины массива, эти промежуточные элементы после выделения памяти будут содержать случайные данные (см. пример).

> Обратите внимание, что функция не может контролировать соответствие конфигурации массива, который использовался при записи файла, и приемного массива при чтении. В принципе, нет гарантии, что читаемый файл записывался с помощью *FileWriteArray*.
>
> Для проверки правильности структуры данных обычно используют некие предопределенные форматы начальных заголовков или другие описатели внутри файлов. Сами функции прочтут любое содержимое файла в пределах его размера и поместят в указанный массив.

Если *handle* получен в двоичном режиме, массивы могут быть любых встроенных нестроковых типов или типов простых структур. Если *handle* открыт в текстовом режиме, массив должен быть типа *string*.

В скрипте *FileArray.mq5* проверим работу как в бинарном, так и в текстовом режиме. Для этого зарезервируем два имени файлов.

```
const string raw = "MQL5Book/array.raw";
const string txt = "MQL5Book/array.txt";
```

В функции *OnStart* описано три массива типа *long* и два массива типа *string*. Только первый массив каждого типа заполнен данными, а во все остальные будет производиться проверочное чтение после записи файлов.

```
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];
   ...
```

Кроме того, для тестирования работы со структурами определены 3 следующих типа:

```
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;
};
```

Структуру типа *TT* мы не сможем использовать в описываемых функциях, поскольку она содержит строковые поля. Она нужна, что продемонстрировать потенциальную ошибку компиляции в закомментированной инструкции (см. далее). Наследование между структурами *B* и *XYZ*, а также наличие закрытого поля не являются препятствием для функций *FileWriteArray* и *FileReadArray*.

Структуры используются для декларации пары массивов:

```
   TT tt[]; // пустой, т.к. данные не важны
   XYZ xyz[1];
   xyz[0].setB(-1);
   xyz[0].x = xyz[0].y = xyz[0].z = clrRed;
```

Начнем с бинарного режима. Создадим новый или откроем существующий файл, сбросив его содержимое. Затем в трех вызовах *FileWriteArray* попытаемся записать массивы *numbers1*, *text1* и *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);
```

Массивы *numbers1* и *xyz* записываются успешно, о чем говорит количество записанных элементов. Массив *text1* получает отказ с ошибкой FILE_NOTTXT(5012) из-за того, что строковые массивы требуют, чтобы файл был открыт в текстовом режиме. Поэтому содержимое *xyz* будет расположено в файле непосредственно после всех элементов *numbers1*.

Обратите внимание, каждая функция записи (или чтения) начинает записывать (или считывать) данные в текущую позицию внутри файла и сдвигает её на размер записанных или прочитанных данных. Если этот указатель находится в конце файла перед операцией записи, размер файла увеличивается. Если конец файла достигается в процессе чтения, указатель больше не двигается, а система взводит специальный внутренний код ошибки 5027 (FILE_ENDOFFILE). В новом файле нулевого размера начало и конец совпадают.

Из массива *text1* было записано 0 элементов, поэтому ничто в файле не напоминает о том, что между двумя удачными вызовами *FileWriteArray* был один неудачный.

В тестовом скрипте мы просто выводим результат функции и статус (код ошибки) в журнал, а в реальной программе следует "на лету" проводить анализ проблем и предпринимать некие действия: что-то исправлять в параметрах, в настройках файла или прерывать процесс с сообщением пользователю.

Прочитаем файл в массив *numbers2*.

```
int reader = PRTF(FileOpen(raw, FILE_BIN | FILE_READ)); // 1 / ok
PRTF(FileReadArray(reader, numbers2)); // 8 / ok
ArrayPrint(numbers2);
```

Поскольку в файл было записано два разных массива (не только *numbers1*, но и *xyz*), в приемный массив прочиталось 8 элементов (т.е. весь файл до конца, т.к. иное не было задано с помощью параметров).

Действительно, размер структуры *XYZ* равен 16 байтам (4 поля по 4 байта: один *int* и три *color*), что соответствует одному ряду в массиве *numbers2* (2 элемента типа *long*). В данном случае, это совпадение. Как уже отмечалось выше, функции не имеют представления о конфигурации и размере сырых данных, и могут прочитать все что угодно в какой угодно массив: за корректностью операции должен следить программист.

Сравним исходное и полученное состояния. Исходный массив *numbers1*:

```
    [,0][,1]
[0,]   1   4
[1,]   2   5
[2,]   3   6
```

Полученный массив *numbers2*:

```
                [,0]          [,1]
[0,]               1             4
[1,]               2             5
[2,]               3             6
[3,] 1099511627775 1095216660735
```

Начало массива *numbers2* полностью совпадает с исходным массивом *numbers1*, то есть запись и чтение через файл работают исправно.

Последний ряд целиком занимает одна структура *XYZ* (с правильными значениями, но неверным их представлением в виде двух чисел *long*).

Теперь "перемотаем" файл на начало (это делается с помощью функции *FileSeek*, которую мы рассмотрим позднее, в разделе Управление позицией внутри файла) и вызовем *FileReadArray* с указанием номера и числа элементов, то есть выполним частичное чтение.

```
PRTF(FileSeek(reader, 0, SEEK_SET)); // true
PRTF(FileReadArray(reader, numbers3, 10, 3));
FileClose(reader);
ArrayPrint(numbers3);
```

Из файла считываются 3 элемента и помещаются, начиная с индекса 10, в приемный массив *numbers3*. Поскольку файл читается с начала, этими элементами являются значения 1, 4, 2. А поскольку двумерный массив имеет конфигурацию [][2], сквозной индекс 10 указывает на элемент [5,0]. Вот как это выглядит в памяти:

```
    [,0][,1]
[0,]   1   4
[1,]   1   4
[2,]   2   6
[3,]   0   0
[4,]   0   0
[5,]   1   4
[6,]   2   0
```

Элементы, помеченные желтым цветом, являются случайными (могут меняться от запуска к запуску скрипта). Вполне может быть, что они все окажутся нулевыми, но это не гарантировано. Изначально массив *numbers3* был пустым, и вызов *FileReadArray* инициировал распределение памяти, достаточной для приема 3 элементов по смещению 10 (итого 13). Выделенный блок ничем не заполняется, а из файла читается только 3 числа. Поэтому элементы со сквозными индексами от 0 до 9 (т.е. 5 первых рядов), а также последний, с индексом 13, содержат "мусор".

Напомним, что многомерные массивы "масштабируются" по первому измерению, и потому прирост в 1 номер означает добавление всей конфигурации по высшим измерениям. В данном случае, распределение касается ряда из двух чисел ([][2]). Иными словами, запрошенный размер 13 округлен в большую сторону до кратного двум, то есть 14.

Наконец, протестируем работу функций со строковыми массивами. Создадим новый или откроем существующий файл, сбросив его содержимое. Затем в двух вызовах *FileWriteArray* попытаемся записать массивы *text1* и *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);
```

Строковый массив сохраняется успешно. Числовой — игнорируется с ошибкой FILE_NOTBIN(5011), поскольку для него файл должен быть открыт в двоичном режиме.

При попытке записать массив структур *tt* мы получим ошибку компиляции с пространным сообщением "структуры или классы с объектами не разрешены". На самом деле компилятор имеет в виду, что ему не нравятся поля типа *string* (подразумевается, что строки и динамические массивы имеют внутреннее представление неких служебных объектов). Таким образом, несмотря на то, что файл открыт в текстовом режиме, и в структуре — только текстовые поля, такое сочетание не поддерживается MQL5.

```
// ОШИБКА КОМПИЛЯЦИИ: structures or classes containing objects are not allowed
FileWriteArray(writer, tt);
```

Наличие строковых полей делает структуру "сложной" и непригодной для работы с функциями *FileWriteArray/FileReadArray* в любом режиме.

После запуска скрипта вы можете перейти в каталог *MQL5/Files/MQL5Book* и изучить содержимое сгенерированных файлов.

Ранее в разделе Запись и чтение файлов в упрощенном режиме мы рассмотрели функции *FileSave* и *FileLoad*. В тестовом скрипте (*FileSaveLoad.mq5*) для них были реализованы, но детально не представлены эквивалентные варианты этих функций, использующие *FileWriteArray* и *FileReadArray*. Поскольку теперь мы познакомились с этими новыми функциями, то можем изучить исходный код:

```
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)));
   // эта версия дополнена по сравнению со стандартной FileLoad следующей проверкой:
   // если размер файла не кратен размеру структуры, выводим предупреждение
   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;
}
```

Как видно, *MyFileSave* строится на единственном вызове *FileWriteArray*, а *MyFileLoad* — на вызове *FileReadArray*, между парой вызовов *FileOpen/FileClose*. В обоих случаях пишутся и читаются все имеющиеся данные. Благодаря шаблонам наши функции также способны принимать массивы произвольных типов. Но если в качестве мета-параметра T будет выведен какой-либо неподдерживаемый тип (например, класс), то возникнет ошибка компиляции, как и при некорректном обращении ко встроенным функциям.

4.5.7 Запись и чтение структур (бинарные файлы)

В предыдущем разделе мы научились выполнять операции ввода-вывода над массивами структур. Когда чтение или запись касается отдельной структуры, удобнее воспользоваться парой функций *FileWriteStruct* и *FileReadStruct*.

uint FileWriteStruct(int handle, const void &data, int size = -1)

Функция записывает в бинарный файл с дескриптором *handle* содержимое "простой" структуры *data*. Как мы знаем, такие структуры могут содержать только поля встроенных нестроковых типов и вложенные "простые" структуры.

Основная "фишка" функции заключается в параметре *size*. С помощью него задается количество записываемых байтов, что позволяет отбросить некоторую часть структуры (её окончание). По умолчанию параметр равен -1, что означает сохранение всей структуры целиком. Если *size*

больше размера структуры, превышение игнорируется, то есть записывается только структура, *sizeof(data)* байтов.

При успешном выполнении функция возвращает количество записанных байтов, в случае ошибки — 0.

uint FileReadStruct(int handle, void &data, int size = -1)

Функция считывает из бинарного файла с дескриптором *handle* содержимое в структуру *data*. Параметр *size* позволяет указать количество байтов, подлежащих чтению. Если оно не указано или превышает размер структуры, то используется точный размер указанной структуры.

При успешном выполнении функция возвращает количество прочитанных байтов, в случае ошибки — 0.

Опция с отсечением концовки структуры присутствует только в функциях *FileWriteStruct* и *FileReadStruct*. Поэтому их использование в цикле становится наиболее подходящей альтернативой для сохранения и чтения массива "урезанных" структур: функции *FileWriteArray* и *FileReadArray* такой возможностью не обладают, а запись и чтение по отдельным полям более накладно (мы рассмотрим соответствующие функции в следующих разделах).

Следует отметить, что для эксплуатации данной возможности следует проектировать свои структуры таким образом, чтобы все временные, расчетные, не подлежащие сохранению поля располагались в конце структуры.

Рассмотрим примеры использования этих двух функций в скрипте *FileStruct.mq5*.

Предположим, мы хотим периодически архивировать последние котировки, чтобы иметь возможность проверять их неизменность в дальнейшем или сравнивать с аналогичными периодами у других поставщиков. В принципе, это можно сделать вручную через диалог "Символы" (закладка "Бары") в MetaTrader 5, но потребовало бы лишних усилий и соблюдения расписания. Гораздо проще делать это автоматически из программы. Кроме того, ручной экспорт котировок делается в текстовом формате CSV, а нам может потребоваться отправлять файлы на внешний сервер и потому желательно сохранять их в компактном двоичном виде. В дополнение к этому допустим, что информация о тиках, спреде и реальных объемах (которые всегда пусты для символов Forex) нас не интересует.

В разделе Сравнение, сортировка и поиск в массивах мы познакомились со структурой *MqlRates* и функцией *CopyRates*. Подробно они будут описаны позднее, а сейчас еще раз используем их в качестве тестовой площадки для файловых операций.

С помощью параметра *size* в *FileWriteStruct* мы сможем сохранять лишь часть структуры *MqlRates*, без последних полей.

В начале скрипта определим макросы и имя тестового файла.

```
#define BARLIMIT 10 // количество баров для записи
#define HEADSIZE 10 // размер заголовка нашего формата
const string filename = "MQL5Book/struct.raw";
```

Особый интерес представляет константа HEADSIZE. Как уже упоминалось ранее, файловые функции не отвечают сами по себе за согласованность данных в файле и типов структур, куда эти данные считываются. Программист должен обеспечить такой контроль в своем коде. Поэтому обычно в начале файла пишется некий заголовок, с помощью которого можно, во-первых,

убедиться, что это файл "нашего" формата, а во-вторых, сохранить в нем мета-информацию, необходимую для правильного чтения.

В частности, в заголовке может быть указано количество записей. Строго говоря, последнее не всегда обязательно, потому что мы можем вычитывать файл постепенно, пока он не закончится. Однако более эффективно — выделить память сразу под все ожидаемые записи, на основе счетчика в заголовке.

Для наших целей была разработана простая структура *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);
   }
};
```

Она начинается с текстовой сигнатуры "CANDLES" (в поле *signature*), номера версии "1.0" (там же) и количества записей (в поле *n*). Поскольку мы не можем использовать строковое поле для сигнатуры (тогда структура перестала бы быть простой и отвечающей требованиям файловых функций), текст фактически упакован в массив *uchar* фиксированного размера HEADSIZE. Его инициализацию в экземпляре делает конструктор на основе локальной статической копии.

В функции *OnStart* запрашиваем BARLIMIT последних баров, открываем файл в режиме FILE_WRITE и записываем в него сперва заголовок, а потом полученные котировки в усеченном виде.

```
void OnStart()
{
   MqlRates rates[], candles[];
   int n = PRTF(CopyRates(_Symbol, _Period, 0, BARLIMIT, rates)); // 10 / ok
   if(n < 1) return;

   // создаем новый файл или перезаписываем с нуля старый
   int handle = PRTF(FileOpen(filename, FILE_BIN | FILE_WRITE)); // 1 / ok

   FileHeader fh(n);// заголовок с актуальным количеством записей

   // сначала записываем заголовок
   PRTF(FileWriteStruct(handle, fh)); // 14 / ok

   // потом записываем данные
   for(int i = 0; i < n; ++i)
   {
      FileWriteStruct(handle, rates[i], offsetof(MqlRates, tick_volume));
   }
   FileClose(handle);
   ArrayPrint(rates);
   ...
```

В качестве значения параметра *size* функции *FileWriteStruct* используется выражение со знакомым нам оператором *offsetof*: *offsetof(MqlRates, tick_volume)*, то есть все поля начиная с *tick_volume* отбрасываются при записи в файл.

Для проверочного чтения данных откроем тот же файл в режиме FILE_READ и прочитаем структуру *FileHeader*.

```
   handle = PRTF(FileOpen(filename, FILE_BIN | FILE_READ)); // 1 / ok
   FileHeader reference, reader;
   PRTF(FileReadStruct(handle, reader)); // 14 / ok
   // если заголовки не совпадают, это не наши данные
   if(ArrayCompare(reader.signature, reference.signature))
   {
      Print("Wrong file format; 'CANDLES' header is missing");
      return;
   }
```

В структуре *reference* содержится неизмененный заголовок по умолчанию (сигнатура). В структуру *reader* попало 14 байтов из файла. Если две сигнатуры совпали, мы можем продолжать работу, так как формат файла оказался правильным, и в поле *reader.n* находится прочитанное из файла количество записей. Мы выделяем и обнуляем память требуемого размера под приемный массив *candles*, а затем читаем в него все записи.

```
   PrintFormat("Reading %d candles...", reader.n);
   ArrayResize(candles, reader.n); // распределяем память под ожидаемые данные заране
   ZeroMemory(candles);

   for(int i = 0; i < reader.n; ++i)
   {
      FileReadStruct(handle, candles[i], offsetof(MqlRates, tick_volume));
   }
   FileClose(handle);
   ArrayPrint(candles);
}
```

Обнуление потребовалось потому, что чтение структур *MqlRates* также производится частично, и без обнуления оставшиеся поля содержали бы "мусор".

Посмотрим в журнале, какими были исходные данные (целиком) для XAUUSD,H1.

```
                 [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_
[0] 2021.08.16 03:00:00 1778.86 1780.58 1778.12 1780.56          3049        5
[1] 2021.08.16 04:00:00 1780.61 1782.58 1777.10 1777.13          4633        5
[2] 2021.08.16 05:00:00 1777.13 1780.25 1776.99 1779.21          3592        5
[3] 2021.08.16 06:00:00 1779.26 1779.26 1776.67 1776.79          2535        5
[4] 2021.08.16 07:00:00 1776.79 1777.59 1775.50 1777.05          2052        6
[5] 2021.08.16 08:00:00 1777.03 1777.19 1772.93 1774.35          3213        5
[6] 2021.08.16 09:00:00 1774.38 1775.41 1771.84 1773.33          4527        5
[7] 2021.08.16 10:00:00 1773.26 1777.42 1772.84 1774.57          4514        5
[8] 2021.08.16 11:00:00 1774.61 1776.67 1773.69 1775.95          3500        5
[9] 2021.08.16 12:00:00 1775.96 1776.12 1773.68 1774.44          2425        5
```

А теперь посмотрим, что было прочитано.

```
                 [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_
[0] 2021.08.16 03:00:00 1778.86 1780.58 1778.12 1780.56             0        0
[1] 2021.08.16 04:00:00 1780.61 1782.58 1777.10 1777.13             0        0
[2] 2021.08.16 05:00:00 1777.13 1780.25 1776.99 1779.21             0        0
[3] 2021.08.16 06:00:00 1779.26 1779.26 1776.67 1776.79             0        0
[4] 2021.08.16 07:00:00 1776.79 1777.59 1775.50 1777.05             0        0
[5] 2021.08.16 08:00:00 1777.03 1777.19 1772.93 1774.35             0        0
[6] 2021.08.16 09:00:00 1774.38 1775.41 1771.84 1773.33             0        0
[7] 2021.08.16 10:00:00 1773.26 1777.42 1772.84 1774.57             0        0
[8] 2021.08.16 11:00:00 1774.61 1776.67 1773.69 1775.95             0        0
[9] 2021.08.16 12:00:00 1775.96 1776.12 1773.68 1774.44             0        0
```

Котировки совпадают, но последние три поля в каждой структуре пусты.

Желающие могут перейти в папку *MQL5/Files/MQL5Book* и изучить внутреннее представление файла *struct.raw* (используйте программу просмотра, поддерживающую двоичный режим; пример показан ниже).

```
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    =·ЧЈрЛ›@ё·
0020:  85 EB 51 D2  9B 40 14 AE  47 E1 7A C8  9B 40 0A D7  …лQТ›@·®GбzИ›@·Ч
0030:  A3 70 3D D2  9B 40 C0 E2  19 61 00 00  00 00 3D 0A  Јр=Т›@Ав·a    =·
0040:  D7 A3 70 D2  9B 40 B8 1E  85 EB 51 DA  9B 40 66 66  ЧЈрТ›@ё·…лQЪ›@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    ЧЈ
0090:  70 3D 0A CD  9B 40 D7 A3  70 3D 0A CD  9B 40 48 E1  р=·Н›@ЧЈр=·Н›@Hб
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›@      ѕ›@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  ·ЧЈЅ›@ЏВх(\Ї›@ё·
0120:  85 EB 51 B5  9B 40 20 37  1A 61 00 00  00 00 D7 A3  …лQµ›@ 7·a    ЧЈ
0130:  70 3D 0A B5  9B 40 48 E1  7A 14 AE C5  9B 40 8F C2  р=·µ›@Hбz·®Е›@ЏВ
0140:  F5 28 5C B3  9B 40 E1 7A  14 AE 47 BA  9B 40 30 45  х(\і›@бz·®Gє›@0E
0150:  1A 61 00 00  00 00 3D 0A  D7 A3 70 BA  9B 40 48 E1  ·a    =·ЧЈрє›@Hб
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№›@
```

Варианты представления бинарного файла с архивом котировок во внешней программе просмотра

Здесь представлен типичный способ отображения бинарных файлов: слева расположена колонка с адресами (смещениями от начала файла), в середине — коды байтов, а справа — символьное представление соответствующих байтов. Первая и вторая колонки используют шестнадцатеричную запись чисел. Символы в правой колонке могут отличаться, в зависимости от выбранной кодовой страницы ANSI. На них имеет смысл обращать внимание только в тех фрагментах, где известно наличие текста. В нашем случае наглядно "проявляется" сигнатура "CANDLES1.0" в самом начале. Числа следует анализировать по средней колонке. В ней, например, после сигнатуры видно 4-байтовое значение 0x0A000000, то есть 0x0000000A в "перевернутом" виде (вспоминаем раздел Управление порядком байтов в целых числах): это 10 — количество записанных структур.

4.5.8 Запись и чтение переменных (бинарные файлы)

Если какая-либо структура содержит поля тех типов, которые запрещены для "простых" структур (строки, динамические массивы, указатели), то записать её в файл или прочитать из файла рассмотренными ранее функциями не получится. То же самое касается и объектов классов. Вместе с тем, такие сущности содержат обычно большую часть данных в программах и также требуют сохранения и восстановления своего состояния.

На примере структуры заголовка в предыдущем разделе было наглядно видно, что без строк (и прочих типов переменной длины) можно, в принципе, обойтись, но при этом приходится изобретать альтернативные, более громоздкие реализации алгоритмов (например, заменять строку на массив символов).

Для записи и чтения данных произвольной сложности MQL5 предоставляет наборы функций нижнего уровня, если так можно выразиться. Они оперируют одним значением конкретного типа: *double*, *float*, *int/uint*, *long/ulong* или *string*. Напомним, что все другие встроенные типы MQL5 эквиваленты целым разного размера: *char/uchar* — 1 байт, *short/ushort* — 2 байта, *color* — 4 байта, перечисления — 4 байта, *datetime* — 8 байтов. Такие функции можно назвать

атомарными (т.е. неделимыми), потому что функций чтения и записи в файлы на уровне битов уже не существует.

Разумеется, поэлементная запись или чтение снимает также и ограничение на файловые операции с динамическими массивами.

Что же касается указателей на объекты, в духе парадигмы ООП мы можем поручить сохранение и восстановление объектов им самим: достаточно в каждом классе реализовать некий интерфейс (набор методов), отвечающий за перенос важного содержимого в файлы и обратно, и использующий низкоуровневые функции. Тогда, если в составе объекта нам встретится поле-указатель на другой объект, мы просто делегируем сохранение или чтение ему, а он, в свою очередь, "разберется" со своими полями, среди которых могут быть другие указатели, и делегирование продолжится вглубь, пока не охватит все элементы.

> Обратите внимание, что в данном разделе мы рассмотрим атомарные функции для бинарных файлов. Их аналоги для текстовых файлов будут представлены в следующем разделе. Все функции данного раздела возвращают количество записанных байтов или 0 в случае ошибки.

uint FileWriteDouble(int handle, double value)
uint FileWriteFloat(int handle, float value)
uint FileWriteLong(int handle, long value)

Функции записывают в бинарный файл с дескриптором *handle* значение соответствующего типа, переданное в параметре *value* (*double*, *float*, *long*).

uint FileWriteInteger(int handle, int value, int size = INT_VALUE)

Функция записывает в бинарный файл с дескриптором *handle* целочисленное значение *value*. Размер значения в байтах задается параметром *size* и может быть одной из предопределенных констант: CHAR_VALUE (1), SHORT_VALUE (2), INT_VALUE (4, по умолчанию), что соответствует типам *char*, *short* и *int* (со знаком и без знака).

Функция поддерживает недокументированный режим записи целого числа размером 3 байта. Его использование не рекомендуется.

Файловый указатель перемещается на количество записанных байтов (а не на размер *int*).

uint FileWriteString(int handle, const string value, int length = -1)

Функция записывает в бинарный файл с дескриптором *handle* строку из параметра *value*. В параметре *length* можно указать количество символов для записи. Если оно меньше длины строки, в файл попадет лишь указанная часть строки. Если *length* равен -1 или не указан, в файл переносится вся строка без терминального нуля. Если *length* больше длины строки, лишние символы заполняются в файле нулями.

Следует иметь в виду, что при записи в файл, открытый с флагом FILE_UNICODE (либо без флага FILE_ANSI), строка сохраняется в формате Unicode (каждый символ занимает 2 байта). При записи в файл, открытый с флагом FILE_ANSI, каждый символ занимает 1 байт (возможно искажение национальных символов).

> Функция *FileWriteString* может работать и с текстовыми файлами. Такое её применение описано в следующем разделе.

double FileReadDouble(int handle)
float FileReadFloat(int handle)
long FileReadLong(int handle)

Функции читают число соответствующего типа — *double*, *float* или *long* — из бинарного файла с указанным дескриптором. При необходимости приведите результат к *ulong* (если в файле в данной позиции ожидается беззнаковое длинное целое).

int FileReadInteger(int handle, int size = INT_VALUE)

Функция читает из бинарного файла с дескриптором *handle* целочисленное значение, причем его размер в байтах указывается в параметре *size*.

Поскольку результат функции — типа *int*, его необходимо явно приводить к требуемому целевому типу, если он отличается от *int* (т.е. к *uint*, или *short/ushort*, или *char/uchar*). В противном случае вы, как минимум, получите предупреждение компилятора, а как максимум — потерю знака.

Дело в том, что при чтении CHAR_VALUE или SHORT_VALUE, результат по умолчанию всегда положительный (т.е. соответствует *uchar* и *ushort*, которые целиком "умещаются" в *int*). В этих случаях, если числа действительно типов *uchar* и *ushort*, предупреждения компилятора чисто номинальные, так как мы заведомо уверены в том, что внутри значения типа *int* заполнены лишь 1 или 2 младших байта, и у них нет знака. Это происходит без искажений.

Однако в случае хранения в файле значений со знаками (типов *char* и *short*) приведение становится необходимым, потому что без него отрицательные величины превратятся в "обратные" положительные с тем же битовым представлением (см. врезку "Знаковые и беззнаковые целые" в разделе Арифметические преобразования типов).

Но в любом случае от предупреждений лучше избавиться путем явного приведения типа.

Функция поддерживает режим чтения целого числа размером 3 байта. Его использование не рекомендуется.

Файловый указатель перемещается на количество прочитанных байтов (а не на размер *int*).

string FileReadString(int handle, int size = -1)

Функция читает из файла с дескриптором *handle* строку указанного размера в символах. Параметр *size* обязательно должен быть задан при работе с бинарным файлом (значение по умолчанию подходит только для текстовых файлов, в которых используются символы-разделители). В противном случае строка не читается (функция возвращает пустую строку), а внутренний код ошибки *_LastError* равен 5016 (FILE_BINSTRINGSIZE).

Таким образом, еще на стадии записи строки в бинарный файл необходимо подумать, каким образом строка будет читаться. Существует 3 основных варианта:

- записывать строки с нулевым терминальным символом на конце; в этом случае их придется анализировать посимвольно в цикле и объединять символы в строку, пока не встретится 0;
- всегда записывать строку фиксированной (предопределенной) длины; длину следует выбрать с запасом для большинства сценариев или по спецификации (техзадания, протокола, и т.д.), но это неэкономно и не дает 100% гарантии, что какая-нибудь редкая строка не окажется укороченной при записи в файл;
- записывать перед строкой её длину как целое число.

Функция *FileReadString* может работать и с текстовыми файлами. Такое её применение описано в следующем разделе.

Также обратите внимание, что если параметр *size* равен 0 (что может получиться в ходе каких-либо вычислений), то функция не производит чтение: указатель файла остается на прежнем месте и функция возвращает пустую строку.

В качестве примера для данного раздела мы усовершенствуем скрипт *FileStruct.mq5* из предыдущего раздела. Новое название программы *FileAtomic.mq5*.

Задача останется прежней: сохранить в двоичный файл заданное количество урезанных структур *MqlRates* с котировками. Но теперь структура *FileHeader* станет классом (причем сигнатура формата будет храниться в строке, а не в массиве символов), заголовок этого типа и массив котировок войдут в состав другого управляющего класса *Candles*, и оба класса будут унаследованы от интерфейса *Persistent* для записи произвольных объектов в файл и чтения из файла.

А вот и сам интерфейс:

```
interface Persistent
{
   bool write(int handle);
   bool read(int handle);
};
```

Для большего приближения к реальности предусмотрим в классе *FileHeader* сохранение и проверку при чтении не только сигнатуры формата (поменяем её на "CANDLES/1.1"), но и названия текущего инструмента и таймфрейма графика (подробнее про *_Symbol* и *_Period*).

Запись производится в реализации метода *write*, унаследованного от интерфейса.

```
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;
   }
```

Сигнатуру мы записываем точно по её длине, поскольку образец хранится в объекте и при чтении будет задана та же длина.

Для инструмента текущего графика мы предварительно сохраняем в файле длину его имени (достаточно 1 байта для длин вплоть до 255), и только после неё — саму строку.

Название таймфрейма всегда не превышает 3 символа, если из него исключить постоянный префикс "PERIOD_", поэтому для этой строки выбрана фиксированная длина. Получение имени таймфрейма без префикса поручено вспомогательной функции *PeriodToString*: она находится в отдельном заголовочном файле *Periods.mqh* (он будет более подробно рассмотрен в разделе Символы и таймфреймы).

Чтение выполняется в методе *read* в обратном порядке (разумеется, подразумевается, что чтение будет выполняться в другой, новый объект).

```
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;
}
```

Если какое-либо из свойств (сигнатура, символ, таймфрейм) не совпадает в файле и на текущем графике, функция возвращает признак ошибки: *false*.

Обратное преобразование имени таймфрейма в перечисление ENUM_TIMEFRAMES делает функция *StringToPeriod*, также из файла *Periods.mqh*.

Основной класс для запроса, сохранения и чтения архива котировок *Candles* выглядит следующим образом.

```
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; // инициализация не удалась
      }
      limit = n; // может быть меньше запрошенного
   }
```

Полями являются заголовок типа *FileHeader*, запрошенное количество баров *limit* и массив, принимающий структуры *MqlRates* от MetaTrader 5. Заполнение массива производится в конструкторе. В случае ошибки поле *limit* сбрасывается в ноль.

Будучи производным от интерфейса *Persistent*, класс *Candles* требует реализации методов *write* и *read*. В методе *write* мы сначала поручаем объекту заголовка сохранить себя, а затем дописываем в файл количество котировок, диапазон дат (для справки) и сам массив.

```
   bool write(int handle) override
   {
      if(!limit) return false; // нет данных
      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;
   }
```

Чтение производится в обратном порядке:

```
bool read(int handle) override
{
   if(!header.read(handle))
   {
      return false;
   }
   limit = PRTF(FileReadInteger(handle));
   ArrayResize(rates, limit);
   ZeroMemory(rates);
   // даты нужно прочитать: они не используются, но это сдвигает позицию в файле;
   // можно было явно изменить позицию, но эта функция нами пока не изучалась
   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;
}
```

В реальной программе архивирования котировок наличие диапазона дат позволило бы по заголовкам файлов выстроить их правильную последовательность на продолжительной истории и в некоторой степени защитило бы от произвольного переименования файлов.

Для контроля процесса имеется простой метод *print*:

```
void print() const
{
   ArrayPrint(rates);
}
```

В главной функции скрипта мы создаем два объекта *Candles*, и с помощью одного сначала сохраняем архив котировок, а с помощью другого затем восстанавливаем. Файлы управляются уже известной нам оберткой *FileHandle* (см. раздел Управление дескрипторами файлов).

```
const string filename = "MQL5Book/atomic.raw";

void OnStart()
{
   // создаем новый файл и обнуляем старый
   FileHandle handle(PRTF(FileOpen(filename,
      FILE_BIN | FILE_WRITE | FILE_ANSI | FILE_SHARE_READ)));
   // формируем данные
   Candles output(BARLIMIT);
   // записываем их в файл
   if(!output.write(~handle))
   {
      Print("Can't write file");
      return;
   }
   output.print();

   // открываем для проверки только что созданный файл
   handle = PRTF(FileOpen(filename,
      FILE_BIN | FILE_READ | FILE_ANSI | FILE_SHARE_READ | FILE_SHARE_WRITE));
   // создаем пустой объект для приема котировок
   Candles inputs;
   // читаем в него данные из файла
   if(!inputs.read(~handle))
   {
      Print("Can't read file");
   }
   else
   {
      inputs.print();
   }
```

Вот пример вывода в журнал исходных данных для 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
                 [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_
[0] 2021.08.17 15:00:00 1791.40 1794.57 1788.04 1789.46          8157        5
[1] 2021.08.17 16:00:00 1789.46 1792.99 1786.69 1789.69          9285        5
[2] 2021.08.17 17:00:00 1789.76 1790.45 1780.95 1783.30          8165        5
[3] 2021.08.17 18:00:00 1783.30 1783.98 1780.53 1782.73          5114        5
[4] 2021.08.17 19:00:00 1782.69 1784.16 1782.09 1782.49          3586        6
[5] 2021.08.17 20:00:00 1782.49 1786.23 1782.17 1784.23          3515        5
[6] 2021.08.17 21:00:00 1784.20 1784.85 1782.73 1783.12          2627        6
[7] 2021.08.17 22:00:00 1783.10 1785.52 1782.37 1785.16          2114        5
[8] 2021.08.17 23:00:00 1785.11 1785.84 1784.71 1785.80           922        5
[9] 2021.08.18 01:00:00 1786.30 1786.34 1786.18 1786.20            13        5
```

А вот пример восстановленных данных (напомним, что структуры сохраняются в урезанном виде по нашему гипотетическому техзаданию):

```
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
                 [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_
[0] 2021.08.17 15:00:00 1791.40 1794.57 1788.04 1789.46             0        0
[1] 2021.08.17 16:00:00 1789.46 1792.99 1786.69 1789.69             0        0
[2] 2021.08.17 17:00:00 1789.76 1790.45 1780.95 1783.30             0        0
[3] 2021.08.17 18:00:00 1783.30 1783.98 1780.53 1782.73             0        0
[4] 2021.08.17 19:00:00 1782.69 1784.16 1782.09 1782.49             0        0
[5] 2021.08.17 20:00:00 1782.49 1786.23 1782.17 1784.23             0        0
[6] 2021.08.17 21:00:00 1784.20 1784.85 1782.73 1783.12             0        0
[7] 2021.08.17 22:00:00 1783.10 1785.52 1782.37 1785.16             0        0
[8] 2021.08.17 23:00:00 1785.11 1785.84 1784.71 1785.80             0        0
[9] 2021.08.18 01:00:00 1786.30 1786.34 1786.18 1786.20             0        0
```

Легко убедиться, что данные сохранены и прочитаны правильно. А теперь посмотрим, как они выглядят внутри файла:

```
signature          symbol length  symbol
   timeframe count       first datetime
  last datetime
0000: 43 41 4E 44  4C 45 53 2F  31 2E 31 06  58 41 55 55  CANDLES/1.1·XAUU
0010: 53 44 48 31  00 0A 00 00  00 F0 CE 1B  61 00 00 00  SDH1 ·   pO·a
0020: 00 90 5B 1C  61 00 00 00  00 F0 CE 1B  61 00 00 00   ђ[·a    pO·a
0030: 00 9A 99 99  99 99 FD 9B  40 E1 7A 14  AE 47 0A 9C   љ™™™™э›@бz·®G·њ
0040: 40 5C 8F C2  F5 28 F0 9B  40 A4 70 3D  0A D7 F5 9B  @\ЏВх(р›@¤p=·Чх›
0050: 40 00 DD 1B  61 00 00 00  00 A4 70 3D  0A D7 F5 9B  @ Э·a    ¤p=·Чх›
0060: 40 29 5C 8F  C2 F5 03 9C  40 F6 28 5C  8F C2 EA 9B  @)\ЏВх·њ@ц(\ЏВк›
0070: 40 F6 28 5C  8F C2 F6 9B  40 10 EB 1B  61 00 00 00  @ц(\ЏВц›@·л·a
0080: 00 D7 A3 70  3D 0A F7 9B  40 CD CC CC  CC CC F9 9B   ЧЈp=·ч›@НММММщ›
0090: 40 CD CC CC  CC CC D3 9B  40 33 33 33  33 33 DD 9B  @НММММУ›@33333Э›
00A0: 40 20 F9 1B  61 00 00 00  00 33 33 33  33 33 DD 9B  @ щ·a    33333Э›
00B0: 40 52 B8 1E  85 EB DF 9B  40 85 EB 51  B8 1E D2 9B  @Rё·…лЯ›@…лQё·Т›
00C0: 40 52 B8 1E  85 EB DA 9B  40 30 07 1C  61 00 00 00  @Rё·…лЪ›@0··a
00D0: 00 F6 28 5C  8F C2 DA 9B  40 71 3D 0A  D7 A3 E0 9B   ц(\ЏВЪ›@q=·ЧЈа›
00E0: 40 8F C2 F5  28 5C D8 9B  40 29 5C 8F  C2 F5 D9 9B  @ЏВх(\Ш›@)\ЏВхЩ›
00F0: 40 40 15 1C  61 00 00 00  00 29 5C 8F  C2 F5 D9 9B  @@··a    )\ЏВхЩ›
0100: 40 52 B8 1E  85 EB E8 9B  40 48 E1 7A  14 AE D8 9B  @Rё·…ли›@Hбz·®Ш›
0110: 40 52 B8 1E  85 EB E0 9B  40 50 23 1C  61 00 00 00  @Rё·…ла›@P#·a
0120: 00 CD CC CC  CC CC E0 9B  40 66 66 66  66 66 E3 9B   ННММа›@fffffг›
0130: 40 52 B8 1E  85 EB DA 9B  40 14 AE 47  E1 7A DC 9B  @Rё·…лЪ›@·®GбzЬ›
0140: 40 60 31 1C  61 00 00 00  00 66 66 66  66 66 DC 9B  @`1·a    fffffЬ›
0150: 40 AE 47 E1  7A 14 E6 9B  40 14 AE 47  E1 7A D9 9B  @®Gбz·ж›@·®GбzЩ›
0160: 40 71 3D 0A  D7 A3 E4 9B  40 70 3F 1C  61 00 00 00  @q=·ЧЈд›@p?·a
0170: 00 3D 0A D7  A3 70 E4 9B  40 8F C2 F5  28 5C E7 9B   =·ЧЈpд›@ЏВх(\з›
0180: 40 A4 70 3D  0A D7 E2 9B  40 33 33 33  33 33 E7 9B  @¤p=·Чв›@33333з›
0190: 40 90 5B 1C  61 00 00 00  00 33 33 33  33 33 E9 9B  @ђ[·a    33333й›
01A0: 40 8F C2 F5  28 5C E9 9B  40 1F 85 EB  51 B8 E8 9B  @ЏВх(\й›@·…лQёи›
01B0: 40 CD CC CC  CC CC E8 9B  40                        @ННММи›@
```

Просмотр внутреннего устройства бинарного файла с архивом котировок во внешней программе

Здесь цветом подсвечены различные поля нашего заголовка: сигнатура, длина названия символа, название символа, название таймфейма и т.д.

4.5.9 Запись и чтение переменных (текстовые файлы)

Для текстовых файлов имеется свой набор функций атомарного (поэлементного) сохранения и чтения данных. Он несколько отличается от набора для бинарных файлов из предыдущего раздела. Также следует отметить отсутствие функций-аналогов для записи/чтения структуры или массива структур в текстовый файл. Если вы попытаетесь использовать какие-либо из этих функций с текстовым файлом, они не произведут никакого эффекта, но взведут внутренний код ошибки 5011 (FILE_NOTBIN).

Как мы уже знаем, у текстовых файлов в MQL5 существует две ипостаси: простой текст, и текст в формате CSV. Соответствующий режим — FILE_TXT или FILE_CSV — задается при открытии файла и не может быть изменен без закрытия и повторного получения дескриптора. Разница между ними проявляется только при чтении файлов. Запись обоих режимов производится одинаково.

В режиме TXT каждый вызов функции чтения (любой функции из тех, что мы рассмотрим в этом разделе) находит в файле следующий перевод строки (символ '\n' или пару '\r\n') и обрабатывает все данные вплоть до него. Суть обработки заключается в преобразовании текста из файла в значение конкретного типа, соответствующего вызванной функции. В простейшем случае, если вызвана функция *FileReadString*, обработка не производится (строка возвращается "как есть").

В режиме CSV при каждом вызове функции чтения текст в файле логически разбивается не только переводами строк, но и дополнительным разделителем, указанным при открытии файла. В

остальном обработка фрагмента от текущей позиции файла до ближайшего разделителя происходит аналогично.

Иными словами, чтение текста и перевод внутренней позиции внутри файла производится фрагментами от разделителя к разделителю, где под "разделителем" понимается не только символ *delimiter* в списке параметров *FileOpen*, но и перевод строки ('\n', '\r\n'), а также начало и конец файла.

На запись текста в файлы FILE_TXT и FILE_CSV дополнительный разделитель влияет одинаково, но только при использовании функции *FileWrite*: она этот символ автоматически вставляет между записываемыми элементами. Функция *FileWriteString* разделитель не учитывает.

Приведем формальное описание функций, а затем рассмотрим пример *FileTxtCsv.mq5*.

uint FileWrite(int handle, ...)

Функция относится к категории функций, которые принимают переменное количество параметров. Такие параметры обозначаются в прототипе функции многоточием. Поддерживаются только встроенные типы данных. Для записи структур или объектов классов необходимо разыменовывать их элементы и передавать по отдельности.

Функция записывает в текстовый файл с дескриптором *handle* все аргументы, переданные после первого. Аргументы разделяются запятыми, как в обычном списке аргументов. Количество выводимых в файл аргументов не может превышать 63.

При выводе числовые данные преобразуются в текстовый формат по правилам стандартного приведения к *(string)*. Значения типа *double* выводятся с точностью до 16 значащих цифр: либо в традиционном, либо в научном формате с показателем степени (выбирается более компактный вариант). Данные типа *float* выводятся с точностью до 7 значащих цифр. Для вывода вещественных чисел с другой точностью или в явно указанном формате необходимо использовать функцию *DoubleToString* (см. раздел Числа в строки и обратно).

Значения типа *datetime* выводятся в формате "YYYY.MM.DD hh:mm:ss" (см. раздел Дата и время).

Стандартный цвет (из списка web-цветов) выводится в виде названия, нестандартный — как тройка значений компонентов RGB (см. раздел Цвет), разделенных запятыми (внимание: запятая — наиболее частый символ-разделитель в CSV).

Для перечислений выводится целое число, обозначающее элемент, а не его идентификатор (название). Например, при записи FRIDAY (из ENUM_DAY_OF_WEEK, см. раздел Перечисления) получим в файле число 5.

Значения типа *bool* выводятся в виде строк "true" или "false".

Если при открытии файла был указан символ-разделитель, отличный от 0, он будет вставляться между двумя соседними строками, получившимися после конвертации соответствующих аргументов.

После записи всех аргументов в файл добавляется признак конца строки '\r\n'.

Функция возвращает количество записанных байтов или 0 в случае ошибки.

uint FileWriteString(int handle, const string text, int length = -1)

Функция записывает в текстовый файл с дескриптором *handle* строковый параметр *text*. Параметр *length* применим только для бинарных файлов и в данном контексте игнорируется (строка записывается полностью).

> Функция *FileWriteString* может работать и с бинарными файлами. Такое её применение описано в предыдущем разделе.

Любые разделители (между элементами в строке) и переводы строк программист должен вставить/добавить самостоятельно, если они требуются.

Функция возвращает количество записанных байтов (в режиме FILE_UNICODE это будет в 2 раза больше длины строки в символах) или 0 в случае ошибки.

string FileReadString(int handle, int length = -1)

Функция читает из файла с дескриптором *handle* строку вплоть до следующего разделителя (символа-разделителя в файле CSV, символа перевода строки в любом файле или до конца файла). Параметр *length* применим только для бинарных файлов и в данном контексте игнорируется.

Полученную строку можно преобразовать в значение требуемого типа по стандартным правилам приведения или с помощью функций конвертации. Альтернативно можно использовать специализированные функции чтения: *FileReadBool*, *FileReadDatetime*, *FileReadNumber*, описанные далее.

В случае ошибки будет возвращена пустая строка. Код ошибки можно узнать через переменную *_LastError* или функцию *GetLastError*. В частности, по достижении конца файла код ошибки будет равен 5027 (FILE_ENDOFFILE).

bool FileReadBool(int handle)

Функция читает из CSV-файла фрагмент до следующего разделителя или до конца строки и преобразует его в значение типа *bool*. Если фрагмент содержит текст "true" (в любом регистре, в том числе смешанном, например, "True") или ненулевое число, получим *true*. В остальных случаях получим *false*.

Слово "true" должно занимать весь прочитанный элемент. Даже если строка начинается на "true", но имеет продолжение (например "True Volume"), получим *false*.

datetime FileReadDatetime(int handle)

Функция читает из CSV-файла строку одного из форматов: "YYYY.MM.DD hh:mm:ss", "YYYY.MM.DD" или "hh:mm:ss" и преобразует ее в значение типа *datetime*. Если фрагмент не содержит корректного текстового представления даты и/или времени, функция вернет нулевое или "странное" время, в зависимости от того, какие символы её удастся интерпретировать как фрагменты даты и времени. Для пустых или нечисловых строк получим текущую дату с нулевым временем.

Более гибкого чтения даты и времени (с большим числом поддерживаемых форматов) можно добиться, скомбинировав две функции: *StringToTime(FileReadString(handle))*. Про *StringToTime* см. раздел Дата и время.

double FileReadNumber(int handle)

Функция читает из CSV-файла фрагмент до следующего разделителя или до конца строки и преобразует его в значение типа *double* по стандартным правилами приведения типов.

Обратите внимание, что тип *double* обладает свойством потери точности при достаточно больших значениях, что может сказаться на чтении больших чисел типов *long/ulong* (значение, после которого начинаются искажения целых внутри *double*, равно 9007199254740992: пример такого явления приведен в разделе Объединения).

Функции, рассмотренные в предыдущем разделе — *FileReadDouble*, *FileReadFloat*, *FileReadInteger*, *FileReadLong*, *FileReadStruct* — нельзя использовать с текстовыми файлами.

Для демонстрации работы с текстовыми файлами подготовлен скрипт *FileTxtCsv.mq5*. В прошлый раз мы выгружали котировки в двоичный файл. Теперь сделаем это в форматах TXT и CSV.

В принципе, MetaTrader 5 позволяет экспортировать и импортировать котировки в CSV-формате из диалога "Символы". Но в образовательных целях мы воспроизведем данный процесс. Кроме того, программная реализация позволяет отойти от точного формата, генерируемого по умолчанию. Фрагмент истории XAUUSD H1, экспортированный стандартным способом, приведен ниже.

```
<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
```

Здесь нас, в частности, может не устраивать используемый по умолчанию символ-разделитель (табуляция, обозначен как '»'), порядок колонок или то, что дата и время разделены на два поля.

В своем скрипте выберем запятую в качестве разделителя, а колонки будем генерировать в порядке полей структуры *MqlRates*. Выгрузку и последующее проверочное чтение произведем в режимах FILE_TXT и FILE_CSV.

```
const string txtfile = "MQL5Book/atomic.txt";
const string csvfile = "MQL5Book/atomic.csv";
const short delimiter = ',';
```

Котировки запросим в начале функции *OnStart* привычным способом:

```
void OnStart()
{
   MqlRates rates[];
   int n = PRTF(CopyRates(_Symbol, _Period, 0, 10, rates)); // 10
```

Названия колонок укажем в массиве по отдельности, а также "склеим" их вместе с помощью вспомогательной функции *StringCombine*. Раздельные названия требуются, поскольку мы объединяем их в общий заголовок, используя выбираемый символ-разделитель (альтернативное решение могло бы строиться на *StringReplace*). С исходным кодом *StringCombine* предлагается

ознакомиться самостоятельно: она делает обратную операцию по отношению к встроенной *StringSplit*.

```
const string columns[] = {"DateTime", "Open", "High", "Low", "Close",
                          "Ticks", "Spread", "True"};
const string caption = StringCombine(columns, delimiter) + "\r\n";
```

Последняя колонка должна была бы называться "Volume", но мы на её примере проверим работу функции *FileReadBool*. Вы можете считать, что текущее название подразумевает "True Volume" (но такая строка не интерпретировалась бы как *true*).

Далее открываем два файла в режимах FILE_TXT и FILE_CSV, и записываем в них подготовленный заголовок.

```
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
```

Поскольку функция *FileWriteString* не добавляет автоматически перевод строки, мы предварительно приплюсовали "\r\n" к переменной *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);
```

Запись полей структур из массива *rates* делается одинаково — вызовом в цикле *FileWrite* для каждого из двух файлов. Напомним, что функция *FileWrite* автоматически вставляет символ-разделитель между аргументами и добавляет "\r\n" в конце строки. Разумеется, можно было самостоятельно преобразовывать все выводимые значения в строки и отправлять в файл с помощью *FileWriteString*, но тогда мы сами должны были бы заботиться о разделителях и переводах строк. В некоторых случаях они бывают не нужны: например, если производится запись в формате JSON в компактном виде (фактически, в одну гигантскую строку).

Таким образом, на стадии записи оба файла управлялись одинаково и получились одинаковыми. Вот пример их содержимого для XAUUSD,H1 (ваши результаты могут быть другими):

```
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
```

Различия в работе с этими файлами начнут проявляться на стадии чтения.

Откроем текстовый файл для чтения и "просканируем" его функцией *FileReadString* в цикле, пока она не вернет пустую строку (т.е. до конца файла).

```
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);
```

В журнал будет выведено примерно следующее:

```
===== 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 / c
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)
```

Каждый вызов *FileReadString* читает в режиме FILE_TXT всю строку целиком (вплоть до '\r\n'). Чтобы разделить её на элементы, следовало бы позаботиться о дополнительной обработке. Или выбрать режим FILE_CSV.

Сделаем то же самое для CSV-файла.

```
   fh2 = PRTF(FileOpen(csvfile, FILE_CSV | FILE_ANSI | FILE_READ, delimiter)); // 2
   Print("===== Reading CSV");
   do
   {
      read = PRTF(FileReadString(fh2));
   }
   while(StringLen(read) > 0);
```

На этот раз записей в журнале появится гораздо больше:

```
===== 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)
```

Все дело в том, что функция *FileReadString* в режиме FILE_CSV учитывает символ-разделитель и дробит строки на элементы. Каждый вызов *FileReadString* возвращает отдельное значение (ячейку) из таблицы CSV. Очевидно, что полученные строки нужно впоследствии преобразовать к соответствующим типам.

Эту задачу можно в обобщенном виде решить с помощью специализированных функций *FileReadDatetime*, *FileReadNumber*, *FileReadBool*. Однако разработчик в любом случае должен сам отслеживать номер текущей читаемой колонки и определять её прикладной смысл. Пример такого алгоритма приведен на третьем шаге теста. Он использует тот же CSV-файл (для простоты мы его закрываем в конце каждого шага и открываем в начале следующего).

Чтобы упростить присвоение очередного поля в структуре *MqlRates* по номеру колонки, была создана структура-наследник *MqlRates*, содержащая один шаблонный метод *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;
      }
   }
};
```

В функции *OnStart* мы описали массив из одной такой структуры, куда будем складывать поступающие значения. Массив потребовался, чтобы упростить вывод в журнал с помощью *ArrayPrint* (для печати структуры самой по себе — готовой функции в MQL5 нет).

```
   Print("===== Reading CSV (alternative)");
   MqlRatesM r[1];
   int count = 0;
   int column = 0;
   const int maxColumn = ArraySize(columns);
```

Переменная *count* для подсчета записей потребовалась не только для статистики, но и как средство пропустить первую строку, которая содержит заголовки, а не данные. Номер текущей колонки отслеживается в переменной *column*. Её максимальное значение не должно превышать количества колонок *maxColumn*.

Теперь нам осталось лишь открыть файл и считывать из него элементы в цикле с помощью разных функций, пока не случится ошибка, в особенности такая ожидаемая ошибка, как 5027 (FILE_ENDOFFILE), то есть достижение конца файла.

Когда номер колонки равен 0, мы применяем функцию *FileReadDatetime*. Для остальных колонок используется *FileReadNumber*. Исключение составляет случай первой строки с заголовками: для него мы вызываем функцию *FileReadBool*, чтобы продемонстрировать, как она отреагирует на заголовок "True", который был специально сделан у последней колонки.

```
fh2 = PRTF(FileOpen(csvfile, FILE_CSV | FILE_ANSI | FILE_READ, delimiter)); // 1
do
{
   if(column)
   {
      if(count == 1) // демо для FileReadBool на 1-ой записи с заголовками
      {
         r[0].set(column, PRTF(FileReadBool(fh2)));
      }
      else
      {
         r[0].set(column, FileReadNumber(fh2));
      }
   }
   else // 0-ая колонка это дата и время
   {
      ++count;
      if(count > 1) // структура из предыдущей строки готова
      {
         ArrayPrint(r, _Digits, NULL, 0, 1, 0);
      }
      r[0].time = FileReadDatetime(fh2);
   }
   column = (column + 1) % maxColumn;
}
while(_LastError == 0); // выход по достижении конца файла 5027 (FILE_ENDOFFILE)

// печать последней структуры
if(column == maxColumn - 1)
{
   ArrayPrint(r, _Digits, NULL, 0, 1, 0);
}
```

Вот что выводится в журнал:

```
===== 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
```

Легко заметить, что из всех заголовков только последний преобразован в значение *true*, а предыдущие — *false*.

Содержимое прочитанных структур совпадает с исходными данными.

4.5.10 Управление позицией внутри файла

Как мы уже знаем, с каждым открытым файлом система ассоциирует некий указатель: он определяет то место в файле (смещение от его начала), куда будут записываться или откуда будут считываться данные при следующем вызове какой-либо функции ввода-вывода. После выполнения функции указатель сдвигается на размер записанных или прочитанных данных.

В некоторых случаях требуется изменить положение указателя без операций ввода-вывода. В частности, когда нужно дописать данные в конец файла, мы его открываем в "смешанном" режиме FILE_READ | FILE_WRITE, а потом должны каким-то образом очутиться в конце файла (иначе мы начнем перезаписывать данные с начала). Можно было бы вызывать функции чтения, пока есть что читать (тем самым сдвигая указатель), но это не эффективно. Лучше применить специальную функцию FileSeek. А получить фактическое значение указателя (позицию в файле) позволяет функция *FileTell*.

В данном разделе мы изучим эти и пару других функций, связанных с текущей позицией в файле. Некоторые из них работают одинаково для файлов в текстовом и бинарном режимах, а другие имеют особенности.

bool FileSeek(int handle, long offset, ENUM_FILE_POSITION origin)

Функция перемещает файловый указатель на количество *offset* байтов, используя в качестве точки отсчета *origin* — одно из предопределенных положений, описанных в перечислении ENUM_FILE_POSITION. Смещение *offset* может быть как положительным (движение к концу файла и далее за его пределы), так и отрицательным (движение к началу). Перечисление ENUM_FILE_POSITION имеет следующие элементы:

- SEEK_SET — начало файла
- SEEK_CUR — текущая позиция
- SEEK_END — конец файла

Если вычисление новой позиции относительно точки привязки дало отрицательное значение (то есть запрашивается смещение "левее" начала файла), то файловый указатель будет установлен на начало файла.

Если выставить позицию за конец файла (значение больше, чем размер файла), то последующая запись в файл будет произведена не с конца файла, а с выставленной позиции. При этом между предыдущим концом файла и заданной позицией будут записаны неопределенные значения (см. пример далее).

При успешном завершении функция возвращает *true*, а в случае ошибки — *false*.

ulong FileTell(int handle)

Для файла, открытого с дескриптором *handle*, функция возвращает текущее положение внутреннего указателя (смещение относительно начала файла). В случае ошибки будет получено значение ULONG_MAX ((ulong)-1). Код ошибки доступен в переменной *_LastError* или через функцию *GetLastError*.

bool FileIsEnding(int handle)

Функция возвращает признак того, находится ли указатель в конце файла *handle*. Если это так, результат равен *true*.

bool FileIsLineEnding(int handle)

Для текстового файла с дескриптором *handle* функция возвращает признак того, находится ли файловый указатель в конце строки (сразу после символов перевода строки '\n' или '\r\n'). Иными словами, возвращенное значение *true* означает, что текущая позиция находится на начале следующей строки (или в конце файла). Для бинарных файлов результат всегда равен *false*.

Тестовый скрипт для рассмотренных функций называется *FileCursor.mq5*. Работа в нем ведется с тремя файлами: двумя бинарными и одним текстовым.

```
const string fileraw = "MQL5Book/cursor.raw";
const string filetxt = "MQL5Book/cursor.csv";
const string file100 = "MQL5Book/k100.raw";
```

Для упрощения вывода в журнал текущей позиции и признаков конца файла (End-Of-File, EOF) и конца строки (End-Of-Line, EOL) создана вспомогательная функция *FileState*.

```
string FileState(int handle)
{
   return StringFormat("P:%I64d, F:%s, L:%s",
      FileTell(handle),
      (string)FileIsEnding(handle),
      (string)FileIsLineEnding(handle));
}
```

Сценарий проверки функций на бинарном файле включает следующие шаги.

Мы создаем новый или открываем существующий файл *fileraw* ("MQL5Book/cursor.raw") в режиме на чтение и запись. Сразу после открытия и далее после каждой операции выводим текущее состояние файла с помощью вызова *FileState*.

```
void OnStart()
{
   int handle;
   Print("\n * Phase I. Binary file");
   handle = PRTF(FileOpen(fileraw, FILE_BIN | FILE_WRITE | FILE_READ));
   Print(FileState(handle));
   ...
```

Перемещаем указатель в конец файла, что позволит при каждом запуске скрипта дописывать данные в этот файл (а не перезаписывать с начала). Самый очевидный способ сослаться на конец файла: нулевой *offset* относительно *origin=SEEK_END*.

```
   PRTF(FileSeek(handle, 0, SEEK_END));
   Print(FileState(handle));
```

Если файл уже не пустой (не новый), мы можем считывать существующие данные в его произвольной позиции (относительной или абсолютной). В частности, если параметр *origin* функции *FileSeek* равен SEEK_CUR, значит при отрицательном смещении *offset* текущая позиция сдвинется на соответствующее число байтов назад (влево), а при положительном — вперед (направо).

В данном примере мы пробуем отступить назад на размер одного значения типа *int*. Чуть позже мы увидим, что в этом месте должно находиться поле *day_of_year* (последнее поле) структуры *MqlDateTime*, потому что мы её записываем в файл в последующих инструкциях, и эти данные доступны из файла при следующем запуске. Прочитанное значение выводится в журнал для сверки с тем, что было ранее сохранено.

```
   if(PRTF(FileSeek(handle, -1 * sizeof(int), SEEK_CUR)))
   {
      Print(FileState(handle));
      PRTF(FileReadInteger(handle));
   }
```

В новом пустом файле вызов *FileSeek* закончится ошибкой 4003 (INVALID_PARAMETER), и блок инструкции *if* не выполнится.

Далее происходит пополнение файла данными. Сначала текущее локальное время компьютера (8 байтов *datetime*) пишется с помощью *FileWriteLong*.

```
   datetime now = TimeLocal();
   PRTF(FileWriteLong(handle, now));
   Print(FileState(handle));
```

Затем мы пытаемся отступить назад с текущего места на 4 байта (-4) и прочитать *long*.

```
   PRTF(FileSeek(handle, -4, SEEK_CUR));
   long x = PRTF(FileReadLong(handle));
   Print(FileState(handle));
```

Эта попытка закончится ошибкой 5015 (FILE_READERROR), потому что мы были в конце файла и после смещения на 4 байта влево не можем прочитать 8 байтов справа (размер *long*). Однако, как

мы увидим из журнала, в результате этой неудачной попытки указатель все же сместится снова в конец файла.

Если отступить назад на 8 байтов (-8), последующее чтение значения *long* окажется успешным, и оба времени — исходное и полученное из файла — должны совпасть.

```
PRTF(FileSeek(handle, -8, SEEK_CUR));
Print(FileState(handle));
x = PRTF(FileReadLong(handle));
PRTF((now == x));
```

Наконец, запишем в файл структуру *MqlDateTime*, заполненную тем же временем. Позиция в файле увеличится на 32 (размер структуры в байтах).

```
MqlDateTime mdt;
TimeToStruct(now, mdt);
StructPrint(mdt); // выводим дату/время в журнал наглядно
PRTF(FileWriteStruct(handle, mdt)); // 32 = sizeof(MqlDateTime)
Print(FileState(handle));
FileClose(handle);
```

После первого запуска скрипта для сценария с файлом *fileraw* ("MQL5Book/cursor.raw") получим примерно следующее (время будет отличаться):

```
первый запуск
 * 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
```

Согласно статусу, размер файла равен сначала нулю, поскольку позиция "P:0" после смещения на конец файла ("F:true"). После каждой записи (с помощью *FileWriteLong* и *FileWriteStruct*) позиция P увеличивается на размер записанных данных.

После второго запуска скрипта можно заметить в логе кое-какие изменения:

```
второй запуск
 * 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
```

Во-первых, размер файла после открытия равен 40 (судим по позиции "P:40" после смещения в конец файла). С каждым запуском скрипта файл будет увеличиваться на 40 байтов.

Во-вторых, поскольку файл не пустой, удается перемещаться в нем и читать "старые" данные. В частности, после отступа на *-1*sizeof(int)* от текущей позиции (которая по совместительству является концом файла), мы успешно читаем значение 234 — последнее поле структуры *MqlDateTime* (это номер дня в году — он будет у вас, скорее всего, другим).

Вторым тестовым сценарием является работа с текстовым csv-файлом *filetxt* ("MQL5Book/cursor.csv"). Его мы также открываем в "смешанном" режиме чтения и записи, однако не перемещаем указатель в конец файла. Из-за этого каждый запуск скрипта будет перезаписывать данные, начиная с начала файла. Чтобы легко заметить различия, числа в первой колонке CSV генерируются случайным образом. Во второй колонке всегда подставляются один и те же строки из шаблона в функции *StringFormat*.

```
Print(" * Phase II. Text file");
srand(GetTickCount());
// создаем новый или открываем существующий файл для записи/перезаписи
// с самого начала и последующего чтения; внутри CSV-данные (Unicode)
handle = PRTF(FileOpen(filetxt, FILE_CSV | FILE_WRITE | FILE_READ, ','));
// три ряда с данными (пара "число,строка" в каждом), разделенные '\n'
// заметьте, что последний элемент не заканчивается переводом строки '\n'
// это не обязательно, но допустимо
string content = StringFormat(
   "%02d,abc\n%02d,def\n%02d,ghi",
   rand() % 100, rand() % 100, rand() % 100);
// '\n' будет заменен на '\r\n' автоматически, благодаря FileWriteString
PRTF(FileWriteString(handle, content));
```

Вот пример генерируемых данных:

```
34,abc
20,def
02,ghi
```

Затем мы возвращаемся на начало файла и в цикле читаем его с помощью *FileReadString*, постоянно выводя статус в журнал.

```
   PRTF(FileSeek(handle, 0, SEEK_SET));
   Print(FileState(handle));
   // подсчитаем строки в файле с помощью признака FileIsLineEnding
   int lineCount = 0;
   while(!FileIsEnding(handle))
   {
      PRTF(FileReadString(handle));
      Print(FileState(handle));
      // FileIsLineEnding также равен true, когда FileIsEnding равен true,
      // даже если завершающий символ '\n' отсутствует
      if(FileIsLineEnding(handle)) lineCount++;
   }
   FileClose(handle);
   PRTF(lineCount);
```

Ниже представлены фрагменты журнала для файла *filetxt* после первого и второго запуска скрипта. Сначала первый:

```
первый запуск
 * 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
```

А вот второй:

```
второй запуск
 * 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
```

Нетрудно заметить, что файл не меняется в размерах, но по одним и тем же смещениям записываются различные числа. Поскольку данный CSV-файл имеет две колонки, после каждого второго прочитанного значения мы видим взведенный признак EOL ("L:true").

Количество обнаруженных строк равно 3, несмотря на то, что в файле есть только 2 символа перевода строк: последняя (третья) строка оканчивается вместе с файлом.

Наконец, последний тестовый сценарий использует файл *file100* ("MQL5Book/k100.raw") для перемещения в нем указателя за конец файла (на отметку 1000000 байтов) и тем самым увеличивает его размер (резервирует место на диске для потенциальных будущих операций записи).

```
Print(" * Phase III. Allocate large file");
handle = PRTF(FileOpen(file100, FILE_BIN | FILE_WRITE));
PRTF(FileSeek(handle, 1000000, SEEK_END));
// чтобы размер изменился, нужно хоть что-то записать
PRTF(FileWriteInteger(handle, 0xFF, 1));
PRTF(FileTell(handle));
FileClose(handle);
```

Вывод в журнал для этого сценария не меняется от запуска к запуску, однако случайные данные, которые оказываются в выделенном под файл месте, могут отличаться (здесь его содержимое не приводится: используйте внешнюю программу просмотра двоичных файлов).

```
 * 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 Получение свойств файла

В процессе работы с файлами помимо непосредственно записи и чтения данных часто возникает необходимость проанализировать их свойства. Одно из основных свойств — размер файла — можно получить с помощью функции *FileSize*. Но есть и еще несколько характеристик, которые можно запросить с помощью *FileGetInteger*.

Обратите внимание, что функция *FileSize* требует наличия дескриптора открытого файла. А у функции *FileGetInteger* есть часть свойств, которые можно узнать по имени файла, без необходимости его предварительно открывать (в том числе и размер).

ulong FileSize(int handle)

Функция возвращает размер открытого файла по его дескриптору. В случае ошибки результат равен 0, что является допустимым размером и при штатном выполнении функции, поэтому анализ потенциальных ошибок требуется всегда проводить с помощью *_LastError* (или *GetLastError*).

Размер файла можно также получить путем перемещения указателя в конец файла *FileSeek(handle, 0, SEEK_END)* и вызова *FileTell(handle)* — обе функции описаны в предыдущем разделе.

long FileGetInteger(int handle, ENUM_FILE_PROPERTY_INTEGER property)
long FileGetInteger(const string filename, ENUM_FILE_PROPERTY_INTEGER property, bool common = false)

Функция имеет 2 варианта: для работы через дескриптор открытого файла и по имени файла (в том числе закрытого).

Функция возвращает одно из свойств файла, указанное в параметре *property*, причем перечень допустимых свойств отличается для каждого из вариантов (см. ниже). Несмотря на то, что тип значения — *long*, он может содержать, в зависимости от запрошенного свойства, не только целое число, но и *datetime* или *bool*: выполните необходимое приведение типа явным образом.

При запросе свойства по имени файла можно дополнительно уточнить с помощью параметра common, в какой папке следует искать файл: текущего терминала *MQL5/Files* (*false*, по умолчанию) или общей *Users/<пользователь>...MetaQuotes/Terminal/Common/Files* (*true*). Если MQL-программа выполняется в тестере, рабочий каталог расположен внутри папки агента тестирования (*Tester/<агент>/MQL5/Files*), см. вводную часть главы Работа с файлами.

В следующей таблице перечислены все элементы ENUM_FILE_PROPERTY_INTEGER.

Свойство	Описание
FILE_EXISTS *	Проверка на существование (аналог FileIsExist)
FILE_CREATE_DATE *	Дата создания
FILE_MODIFY_DATE *	Дата последнего изменения
FILE_ACCESS_DATE *	Дата последнего доступа
FILE_SIZE *	Размер файла в байтах (аналог FileSize)
FILE_POSITION	Позиция указателя в файле (аналог FileTell)
FILE_END	Признак позиции в конце файла (аналог FileIsEnding)
FILE_LINE_END	Признак позиции в конце строки (аналог FileIsLineEnding)
FILE_IS_COMMON	Файл открыт в общей папке терминалов (FILE_COMMON)
FILE_IS_TEXT	Файл открыт как текстовый (FILE_TXT)
FILE_IS_BINARY	Файл открыт как бинарный (FILE_BIN)
FILE_IS_CSV	Файл открыт как CSV (FILE_CSV)
FILE_IS_ANSI	Файл открыт как ANSI (FILE_ANSI)
FILE_IS_READABLE	Файл открыт на чтение (FILE_READ)
FILE_IS_WRITABLE	Файл открыт на запись (FILE_WRITE)

Свойства, разрешенные для использования по имени файла, помечены звездочкой. При попытке получения других свойств второй вариант функции вернет ошибку 4003 (INVALID_PARAMETER).

Часть свойств может меняться в процессе работы с открытым файлом: FILE_MODIFY_DATE, FILE_ACCESS_DATE, FILE_SIZE, FILE_POSITION, FILE_END, FILE_LINE_END (только для текстовых файлов).

В случае ошибки результат вызова равен -1.

Второй вариант функции позволяет проверить, является ли указанное имя именем файла или каталога. Если при получении свойств по имени будет указан каталог, то функция выставит специальный код внутренней ошибки 5018 (ERR_MQL_FILE_IS_DIRECTORY), при этом возвращаемое значение будет корректным.

Функции данного раздела протестируем в скрипте *FileProperties.mq5*. Он будет работать с файлом с предопределенным именем.

```
const string fileprop = "MQL5Book/fileprop";
```

В начале *OnStart* попробуем запросить размер по ошибочному дескриптору (он не был получен через вызов *FileOpen*). После *FileSize* потребуется проверка переменной *_LastError*, а *FileGetInteger* сразу возвращает специальное значение — индикатор ошибки (-1).

```
void OnStart()
{
   int handle = 0;
   ulong size = FileSize(handle);
   if(_LastError)
   {
      Print("FileSize error=", E2S(_LastError) + "(" + (string)_LastError + ")");
      // Получим: FileSize 0, error=WRONG_FILEHANDLE(5008)
   }

   PRTF(FileGetInteger(handle, FILE_SIZE)); // -1 / WRONG_FILEHANDLE(5008)
```

Далее мы создаем новый или открываем существующий файл и обнуляем его, а затем записываем тестовый текст.

```
   handle = PRTF(FileOpen(fileprop, FILE_TXT | FILE_WRITE | FILE_ANSI)); // 1
   PRTF(FileWriteString(handle, "Test Text\n")); // 11
```

Выборочно запрашиваем некоторые из свойств.

```
   PRTF(FileGetInteger(fileprop, FILE_SIZE)); // 0, еще не записан на диск
   PRTF(FileGetInteger(handle, FILE_SIZE)); // 11
   PRTF(FileSize(handle)); // 11
   PRTF(FileGetInteger(handle, FILE_MODIFY_DATE)); // 1629730884, кол-во секунд с 197
   PRTF(FileGetInteger(handle, FILE_IS_TEXT)); // 1, bool true
   PRTF(FileGetInteger(handle, FILE_IS_BINARY)); // 0, bool false
```

Информация о длине файла по его дескриптору учитывает текущий буфер кэширования, а по имени файла актуальная длина станет доступна только после закрытия файла или если вызвать функцию *FileFlush* (см. раздел Принудительная запись кэша на диск).

Даты и время функция возвращает как количество секунд стандартной эпохи с 1 января 1970 года, что соответствует типу *datetime* и может быть приведено к нему.

Запрос флагов открытия файла (его режима) успешно проходит для варианта функции с дескриптором, в частности, мы получили ответ, что файл является текстовым и не двоичным. Однако следующий аналогичный запрос по имени файла завершится ошибкой, так как свойство поддерживается только при передаче валидного дескриптора. Это происходит несмотря на то, что имя указывает на тот же файл, который мы открыли.

```
   PRTF(FileGetInteger(fileprop, FILE_IS_TEXT)); // -1 / INVALID_PARAMETER(4003)
```

Выждем 1 секунду, закроем файл и снова проверим дату модификации (на этот раз по имени, т.к. дескриптор уже не действует).

```
   Sleep(1000);
   FileClose(handle);
   PRTF(FileGetInteger(fileprop, FILE_MODIFY_DATE)); // 1629730885 / ok
```

Здесь наглядно видно, что время увеличилось на 1.

Наконец, убедимся, что свойства доступны и для директорий (каталогов).

```
PRTF((datetime)FileGetInteger("MQL5Book", FILE_CREATE_DATE));
// Получим: 2021.08.09 22:38:00 / FILE_IS_DIRECTORY(5018)
```

Поскольку все примеры книги расположены в папке "MQL5Book", она должна уже существовать. Однако актуальное время создания у вас будет отличаться. Код ошибки FILE_IS_DIRECTORY в данном случае для нас выводит макрос PRTF. В рабочей программе вызов функции следует делать без макроса, а код затем прочитать в *_LastError*.

4.5.12 Принудительная запись кэша на диск

Запись и чтение файлов в MQL5 кэшируются. Это значит, что для данных поддерживается некий буфер в памяти, за счет которого повышается эффективность работы. Так, данные, переданные с помощью вызовов функций при записи, попадают в буфер вывода, и только по факту его наполнения происходит физическая запись на диск. При чтении, наоборот, с диска в буфер считывается больше данных, чем программа запрашивала с помощью функций (если это не конец файла), и последующие операции чтения (вероятность которых очень высока) выполняются быстрее.

Кэширование — стандартная технология, применяемая в большинстве приложений и на уровне самой операционной системы. Однако, кэширование имеет помимо плюсов и свои минусы.

В частности, если файлы используются как средство обмена данными между программами, отложенная запись может существенно замедлить коммуникацию и сделать её менее предсказуемой, поскольку размер буфера может быть довольно большим, а периодичность его "сброса" на диск — подстраиваться по неким алгоритмам.

Например, в MetaTrader 5 существует целая категория MQL-программ для копирования торговых сигналов из одного экземпляра терминала в другой. Они, как правило, используют файлы для передачи информации, а для них очень важно, чтобы кэширование не замедляло процесс. На этот случай в MQL5 имеется функция *FileFlush*.

void FileFlush(int handle)

Функция выполняет принудительный сброс на диск всех данных, оставшихся в файловом буфере ввода-вывода для файла с дескриптором *handle*.

Если не использовать данную функцию, то часть "отправленных" из программы данных может, в худшем случае, попасть на диск только при закрытии файла.

Функция обеспечивает бóльшие гарантии сохранности ценных данных для непредвиденных случаев (таких как зависание операционной системы или программы). Однако с другой стороны, частый вызов *FileFlush* при массированной записи не рекомендуется, поскольку может негативно сказаться на быстродействии.

Если файл открыт в "смешанном" режиме — одновременно записи и чтения — функцию *FileFlush* необходимо вызывать между операциями чтения и записи в файл.

В качестве примера рассмотрим скрипт *FileFlush.mq5*, в котором реализуем два режима, имитирующих работу копировщика сделок. Нам потребуется запустить два экземпляра скрипта на разных графиках, при этом один из них станет отправителем данных, а второй — получателем.

В скрипте предусмотрено два входных параметра: *EnableFlashing* позволяет сравнить действия программ с использованием функции *FileFlush* и без неё, а *UseCommonFolder* предписывает создавать файл, выступающий как средство передачи данных, на выбор — в папке текущего

экземпляра терминала или в общей папке (в последнем случае можно тестировать передачу данных между разными терминалами).

```
#property script_show_inputs
input bool EnableFlashing = false;
input bool UseCommonFolder = false;
```

Напомним, что для того, чтобы при запуске скрипта появился диалог с входными переменными, необходимо дополнительно задать свойство *script_show_inputs*.

Имя "транзитного" файла указано в переменной *dataport*. Опция *UseCommonFolder* управляет флагом FILE_COMMON, добавляемым в набор переключателей режимов открываемых файлов в функции *FileOpen*.

```
const string dataport = "MQL5Book/dataport";
const int flag = UseCommonFolder ? FILE_COMMON : 0;
```

Главная функция *OnStart* состоит фактически из двух частей: настройки открываемого файла и цикла с периодической отправкой или приемом данных.

Нам потребуется запустить два экземпляра скрипта, и в каждом из них будет свой дескриптор файла, указывающий на один и тот же файл на диске, но открытый в разных режимах.

```
void OnStart()
{
   bool modeWriter = true; // по умолчанию скрипт должен писать данные
   int count = 0;          // количество произведенных записей/чтений
   // создать новый или обнулить старый файл в режиме чтения, в роли "отправителя"
   int handle = PRTF(FileOpen(dataport,
      FILE_BIN | FILE_WRITE | FILE_SHARE_READ | flag));
   // если запись невозможна, скорее всего, другой экземпляр скрипта уже пишет в файл
   // поэтому пытаемся открыть его на чтение
   if(handle == INVALID_HANDLE)
   {
      // если открыть файл на чтение получится, продолжим работу в роли "получателя"
      handle = PRTF(FileOpen(dataport,
         FILE_BIN | FILE_READ | FILE_SHARE_WRITE | FILE_SHARE_READ | flag));
      if(handle == INVALID_HANDLE)
      {
         Print("Can't open file"); // такого быть не должно, что-то не так
         return;
      }
      modeWriter = false; // переключаем режим/роль
   }
```

В начале мы пытаемся открыть файл в режиме FILE_WRITE, без разрешения совместной записи (FILE_SHARE_WRITE), поэтому первый экземпляр запущенного скрипта "захватит" файл и не даст второму работать также в режиме записи. Второй экземпляр, получив ошибку и INVALID_HANDLE после первого вызова *FileOpen*, попытается открыть файл в режиме чтения (FILE_READ) с помощью второго вызова *FileOpen*, на этот раз с флагом, разрешающим параллельную запись FILE_SHARE_WRITE. В идеале это должно успешно произойти, и тогда переменная *modeWriter* устанавливается в *false* для индикации актуальной роли скрипта.

Основной рабочий цикл имеет следующую структуру:

```
while(!IsStopped())
{
   if(modeWriter)
   {
      // ...записываем тестовые данные
   }
   else
   {
      // ...считываем тестовые данные
   }
   Sleep(5000);
}
```

Цикл выполняется до тех пор, пока пользователь не удалит скрипт с графика вручную: об этом просигнализирует функция *IsStopped*. Внутри цикла работа активируется с периодичностью один раз в 5 секунд, за счет вызова функции *Sleep*, которая "замораживает" программу на указанное количество миллисекунд (5000 в данном случае). Это сделано для облегчения анализа происходящих изменений "на лету" и исключения слишком частой записи состояния в журнал. В реальной программе без подробных логов ничто не мешает пересылать данные каждые 100 миллисекунд или чаще.

В качестве пересылаемых данных будет выступать текущее время (одно значение *datetime*, 8 байтов). В первой ветви инструкции *if(modeWriter)*, где осуществляется запись в файл, вызовем *FileWriteLong* с последним отсчетом (получается из функции *TimeLocal*), увеличим на 1 счетчик операций (*count*++) и выведем текущее состояние в журнал.

```
      long temp = TimeLocal(); // получаем текущее локальное время datetime
      FileWriteLong(handle, temp); // дописываем его в файл (раз в 5 секунд)
      count++;
      if(EnableFlashing)
      {
         FileFlush(handle);
      }
      Print(StringFormat("Written[%d]: %I64d", count, temp));
```

Особо следует отметить, что вызов функции *FileFlush* после каждой записи делается только в том случае, если входной параметр *EnableFlashing* установлен в *true*.

Во второй ветви оператора *if*, где происходит чтение, первым делом сбросим внутренний признак ошибок, вызвав *ResetLastError*. Это нужно, потому что мы собираемся читать из файла данные, что называется "до упора", то есть пока они не закончатся, что сообщается программе специфическим кодом ошибки 5015 (ERR_FILE_READERROR).

Поскольку встроенные таймеры MQL5, включая и функцию *Sleep*, обладают ограниченной точностью (примерно 10 мс), мы не можем исключить ситуацию, когда между двумя последовательными попытками чтения файла произошло две последовательных записи. Например, одно чтение произошло в 10:00:00'200, а второе — в 10:00:05'210 (в нотации *"часов:минут:секунд'миллисекунд"*). При этом параллельно происходили две записи: одна в 10:00:00'205, а вторая — в 10:00:05'205, и обе попали в вышеприведенный период. Такая ситуация маловероятна, но возможна: даже при абсолютно точной выдержке временных интервалов система выполнения MQL5 может быть вынуждена выбирать между двумя выполняющимися скриптами, какой "пробудить" раньше, а какой позже, если общее количество программ велико и на них всех не хватает ядер процессора.

В принципе MQL5 предоставляет таймеры повышенной точности (микросекундной), но это не критично для текущей задачи.

Вложенный цикл нужен и еще по одной причине. Сразу после запуска скрипта в роли "получателя" данных он должен обработать все записи из файла, накопившиеся за время, прошедшее с запуска "отправителя" (запустить оба скрипта одновременно вряд ли получится). Вероятно, кто-нибудь предпочел бы другой алгоритм: пропустить все "старые" записи и отслеживать далее только вновь появляющиеся. Это можно сделать, но здесь реализован вариант "без потерь".

```
ResetLastError();
while(true) // цикл, пока есть данные и нет проблем
{
   bool reportedEndBeforeRead = FileIsEnding(handle);
   ulong reportedTellBeforeRead = FileTell(handle);

   temp = FileReadLong(handle);
   // если данных больше нет, получим ошибку 5015 (ERR_FILE_READERROR)
   if(_LastError) break; // выходим из цикла по любой ошибке

   // здесь данные получены без ошибок
   count++;
   Print(StringFormat("Read[%d]: %I64d\t"
      "(size=%I64d, before=%I64d(%s), after=%I64d)",
      count, temp,
      FileSize(handle), reportedTellBeforeRead,
      (string)reportedEndBeforeRead, FileTell(handle)));
}
```

Здесь следует отметить важный нюанс. Метаданные об открытом на чтение файле, такие как его размер, возвращаемый функцией *FileSize* (см. раздел Получение свойств файла), не изменяются после открытия файла. Если другая программа дописала что-то позднее в открытый нами на чтение файл, его "детектируемая" длина не обновится у нас, даже если вызвать *FileFlash* для дескриптора чтения. Можно было бы закрыть и вновь открыть файл (перед каждым чтением, но это не эффективно): тогда для нового дескриптора "проявилась" бы новая длина. Но мы обойдемся без этого, за счет другого трюка.

Прием заключается в том, чтобы продолжать читать данные с помощью функций чтения (в нашем случае, *FileReadLong*), пока они возвращают данные без ошибок. При этом важно не пользоваться другими функциями, оперирующими метаданными. В частности, из-за того, что признак конца файла, открытого только на чтение, остается постоянным, проверка функцией *FileIsEnding* (см. раздел Управление позицией внутри файла) будет давать *true* на старой позиции, несмотря на возможное пополнение файла из другого процесса. Более того, попытка переместить внутренний указатель файла в конец (*FileSeek(handle, 0, SEEK_END)*), про функцию *FileSeek* см. в том же разделе) приведет к переходу не на фактическое окончание данных, а на устаревшую позиции, где конец располагался в момент открытия.

Реальную позицию внутри файла сообщает нам функция *FileTell* (см. тот же раздел). По мере добавления в файл информации из другого экземпляра скрипта и чтения её в данном цикле, указатель будет сдвигаться все дальше и дальше вправо, превышая, как это ни странно, *FileSize*. Для наглядной демонстрации перемещения указателя за размер файла мы сохраняем его значения до вызова *FileReadLong* и после, а затем выводим в журнал вместе с размером.

Как только чтение с помощью *FileReadLong* сгенерирует какую-либо ошибку, внутренний цикл прервется. Штатный выход из цикла подразумевает ошибку 5015 (ERR_FILE_READERROR). Она, в частности, возникает при отсутствии данных, доступных для чтения в текущей позиции в файле.

Последние успешно прочитанные данные выводятся в журнал, и их легко сравнить с тем, что туда вывел скрипт-отправитель.

Давайте запустим дважды новый скрипт, и чтобы различать его копии, сделаем это на графиках разных инструментов.

> При запуске обоих скриптов важно соблюдать одинаковую настройку параметра *UseCommonFolder*. Оставим её в наших тестах равной *false*, поскольку мы будем все делать в одном терминале. Передачу данных между разными терминалами с установкой *UseCommonFolder* в *true* предлагается протестировать самостоятельно.

Сначала запустим первый экземпляр на графике EURUSD,H1, оставив все настройки по умолчанию, включая *EnableFlashing = false*. Затем запустим второй экземпляр на графике XAUUSD,H1 (тоже всё по умолчанию). В журнале увидим примерно такие записи (ваше время будет отличаться):

```
(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_C
(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
```

"Отправитель" благополучно открыл файл на запись и начал раз в 5 секунд отправлять данные, о чем свидетельствуют строки со словом "Written" и увеличивающимися значениями. Меньше чем через 5 секунд после запуска "отправителя" был также запущен "получатель". Он вывел сообщение об ошибке, так как ему не удалось открыть файл на запись. Зато потом он успешно открыл файл на чтение. Однако никаких записей о том, что ему удалось обнаружить передаваемые данные в файле, не появилось. Данные остались "висеть" в кэше "отправителя".

Остановим оба скрипта и запустим их снова в той же последовательности: сначала "отправитель" на EURUSD, затем "получатель" на XAUUSD. Только в этот раз в "отправителе" укажем опцию *EnableFlashing = true*.

Вот что получится в журнале:

```
(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
```

Один и тот же файл вновь успешно открыт в разных режимах в обоих скриптах, но на этот раз записанные значения регулярно читаются "получателем".

Интересно отметить, что перед каждым очередным чтением данных, кроме первого, функция *FileIsEnding* возвращает *true* (выводится в той же строке, что и полученные данные, в круглых скобках после строки "before"). Таким образом, налицо признак того, что мы находимся в конце файла, однако затем *FileReadLong* успешно читает значение якобы за пределом файла и сдвигает позицию вправо. Например, запись "size=8, before=8(true), after=16" означает, что размер файла сообщается MQL-программе равным 8, текущий указатель до вызова *FileReadLong* также равняется 8 и взведен признак конца файла, а после успешного вызова *FileReadLong* указатель перемещен на 16. Однако на следующей и всех остальных итерациях мы вновь видим "size=8", а указатель постепенно сдвигается все дальше и дальше за пределы файла.

Поскольку запись в "отправителе" и чтение в "получателе" происходят раз в 5 секунд, то в зависимости от фазы смещения их циклов мы можем наблюдать эффект различной задержки между двумя операциями, вплоть до "почти" 5 секунд в худшем случае. Однако это не значит, что сброс кэша происходит так медленно. На самом деле это практически моментальный процесс. Для обеспечения более оперативного обнаружения изменений можно уменьшить период "спячки" в циклах (но следует иметь в виду, что данный тест при слишком малой задержке быстро заполнит журнал, поскольку в отличие от реальной программы здесь "новые данные" генерируются всегда — ведь это текущее время "отправителя" с точностью до секунды).

Между прочим, вы можете запустить несколько "получателей", в отличие от "отправителя", который должен быть один. В журнале ниже показана работа "отправителя" на EURUSD и двух "получателей" на чартах XAUUSD и 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
```

К моменту, когда запустился третий скрипт на USDRUB, в файле было уже 2 записи по 8 байтов, поэтому внутренний цикл сразу проделал 2 итерации с *FileReadLong*, а размер файла "видится" равным 16.

4.5.13 Удаление и проверка на существование файла

Проверка файла на существование и его удаление — критически важные действия, относящиеся к файловой системе: внешней среде, в которой "живут" файлы. До сих пор мы рассматривали функции, которые манипулируют внутренним содержимым файлов. Начиная с этого раздела акцент сместится в сторону функций, которые управляют файлами как неделимыми элементами.

bool FileIsExist(const string filename, int flag = 0)

Функция проверяет, существует ли файл с именем *filename*, и возвращает *true*, если это так. Каталог поиска выбирается с помощью параметра *flag*: когда он равен 0 (значение по умолчанию), файл ищется в каталоге текущей копии терминала (*MQL5/Files*); если *flag* равен FILE_COMMON, проверяется общий каталог всех терминалов *Users/<пользователь>...MetaQuotes/Terminal/Common/Files*. Если MQL-программа выполняется в тестере, рабочий каталог расположен внутри папки агента тестирования (*Tester/<агент>/MQL5/Files*), см. вводную часть главы Работа с файлами.

Указанное имя может принадлежать не файлу, а каталогу. В этом случае функция *FileIsExist* возвратит *false*, а в переменную *_LastError* будет записана псевдо-ошибка 5018 (FILE_IS_DIRECTORY).

bool FileDelete(const string filename, int flag = 0)

Функция удаляет файл с указанным именем *filename*. Параметр *flag* определяет местоположение файла. При значении по умолчанию удаление производится в рабочем каталоге текущего экземпляра терминала (*MQL5/Files*) или агента тестирования (*Tester/<агент>/MQL5/Files*), если

программа выполняется в тестере. Если *flag* равен FILE_COMMON, файл должен находиться в общей папке всех терминалов (*/Terminal/Common/Files*).

Функция возвращает признак успешного выполнения (*true*) или ошибки (*false*).

Данная функция не позволяет удалять каталоги. Для этой цели воспользуйтесь функцией *FolderDelete* (см. раздел Работа с папками).

Упражнения с описанными функциями выполним в скрипте *FileExist.mq5*. В нем проведем несколько манипуляций с временным файлом.

```
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)
}
```

Файл изначально не существует, поэтому обе функции *FileIsExist* и *FileDelete* возвращают *false*, и код ошибки равен 5019 (FILE_NOT_EXIST).

Затем мы создаем файл, и функция *FileIsExist* сообщает о его наличии. Однако удалить его нельзя, потому что он открыт и занят нашим процессом (код ошибки 5006, CANNOT_DELETE_FILE).

После того как файл закрыт, его удается удалить.

В завершении скрипта проверке и попытке удаления подвергается каталог "MQL5Book". *FileIsExist* возвращает *false*, потому что это не файл, однако код ошибки 5018 (FILE_IS_DIRECTORY) уточняет, что это директория.

4.5.14 Копирование и перемещение файлов

Основные операции над файлами на уровне файловой системы — это копирование и перемещение. В MQL5 для этих целей реализованы две функции с одинаковыми прототипами.

bool FileCopy(const string source, int flag, const string destination, int mode)

Функция копирует исходный файл *source* в файл *destination*. Оба упомянутых параметра могут содержать исключительно имена файлов или имена вместе с предваряющими их путями

(иерархиями папок) в "песочницах" MQL5. Параметры *flag* и *mode* определяют, соответственно, в какой рабочей папке ищется исходный файл и какая рабочая папка является целевой: значение 0 — это папка локального текущего экземпляра терминала (или агента тестирования, если программа запущена в тестере), а значение FILE_COMMON — общая папка для всех терминалов.

Кроме того, в параметре *mode* можно дополнительно указать константу FILE_REWRITE (если необходимо скомбинировать FILE_REWRITE и FILE_COMMON, это делается с помощью оператора побитового ИЛИ (|)). В отсутствие FILE_REWRITE копирование поверх существующего файла запрещено. Иными словами, если файл с путем и именем, указанным в параметре *destination*, уже существует, необходимо подтвердить намерение его переписать, установив FILE_REWRITE. Если этого не сделать, вызов функции завершится ошибкой.

Функция возвращает *true* при успешном выполнении или *false* в случае ошибки.

Копирование может завершится ошибкой даже при разрешении, если исходный или целевой файл заняты (открыты) другим процессом.

> При копировании файлов в них обычно сохраняются метаданные файловой системы (время создания, права доступа, альтернативные потоки данных). Если необходимо выполнить "чистое" копирование только данных самого файла, можно воспользоваться последовательными вызовами *FileLoad* и *FileSave*, см. раздел Запись и чтение файлов в упрощенном режиме.

bool FileMove(const string source, int flag, const string destination, int mode)

Функция перемещает или переименовывает файл. Исходные путь и имя указываются в параметре *source*, а целевые — в параметре *destination*.

Список параметров и принцип их действия — такие же, как для функции *FileCopy*. Грубо говоря, функция *FileMove* производит ту же работу, что и *FileCopy*, но дополнительно удаляет исходный файл после успешного копирования.

Приемы работы с функциями изучим на практике с помощью скрипта *FileCopy.mq5*. В нем заведено 2 переменных с именами файлов. Оба файла не существуют на момент запуска скрипта.

```
const string source = "MQL5Book/source";
const string destination = "MQL5Book/destination";
```

В *OnStart* произведем последовательность действий по простому сценарию. Для начала попробуем скопировать файл *source* из локального рабочего каталога в файл *destination* общего каталога. Мы ожидаемо получим результат *false*, а код ошибки в *_LastError* будет равен 5019 (FILE_NOT_EXIST).

```
void OnStart()
{
   PRTF(FileCopy(source, 0, destination, FILE_COMMON)); // false / FILE_NOT_EXIST(501
   ...
```

Поэтому создадим исходный файл привычным способом, запишем некие данные и сбросим на диск.

```
   int handle = PRTF(FileOpen(source, FILE_TXT | FILE_WRITE)); // 1
   PRTF(FileWriteString(handle, "Test Text\n")); // 22
   FileFlush(handle);
```

Поскольку файл остался открытым и при открытии не было указано разрешение FILE_SHARE_READ, доступ к нему другими способами (минуя дескриптор) пока заблокирован. Поэтому следующая попытка копирования вновь завершится ошибкой.

```
   PRTF(FileCopy(source, 0, destination, FILE_COMMON)); // false / CANNOT_OPEN_FILE(5
```

Закроем файл и повторим попытку. Но предварительно выведем в журнал свойства получившегося файла: когда он создан и модифицирован. Оба свойства будут содержать текущую метку времени на вашем компьютере.

```
   FileClose(handle);
   PRTF(FileGetInteger(source, FILE_CREATE_DATE)); // 1629757115, пример
   PRTF(FileGetInteger(source, FILE_MODIFY_DATE)); // 1629757115, пример
```

Выждем 3 секунды прежде чем вызывать *FileCopy*. Это позволит увидеть разницу в свойствах исходного файла и его копии. Эта пауза никак не связана с предыдущей блокировкой файла: мы могли бы выполнять копирование сразу после того, как закрыли файл, или даже в процессе его записи, если была бы включена опция FILE_SHARE_READ.

```
   Sleep(3000);
```

Скопируем файл — на этот раз удачно — и выведем свойства со временем для копии.

```
   PRTF(FileCopy(source, 0, destination, FILE_COMMON)); // true
   PRTF(FileGetInteger(destination, FILE_CREATE_DATE, true)); // 1629757118, +3 секун
   PRTF(FileGetInteger(destination, FILE_MODIFY_DATE, true)); // 1629757115, пример
```

Нетрудно заметить, что время создания у каждого файла свое (у копии оно на 3 секунды больше, чем у оригинала), а вот время модификации одинаковое (копия унаследовала свойство оригинала).

Теперь попробуем переместить копию обратно в локальную папку. Без опции FILE_REWRITE "этот номер не пройдет", потому что нет разрешения перезаписать исходный файл.

```
   PRTF(FileMove(destination, FILE_COMMON, source, 0)); // false / FILE_CANNOT_REWRIT
```

Изменив значение параметра, добьемся успешного переноса файла.

```
   PRTF(FileMove(destination, FILE_COMMON, source, FILE_REWRITE)); // true
```

В завершение исходный файл также удаляется, чтобы оставить "чистой" обстановку для повторных экспериментов с этим скриптом.

```
   ...
   FileDelete(source);
}
```

4.5.15 Поиск файлов и папок

MQL5 позволяет осуществлять поиск файлов и папок в пределах "песочниц" терминала, агентов тестирования и общей для всех терминалов (подробнее про "песочницы" см. вводную часть этой

главы Работа с файлами). Если вы точно знаете требуемое имя и расположение файла/директории, используйте функцию *FileIsExist*.

long FileFindFirst(const string filter, string &found, int flag = 0)

Функция начинает поиск файлов и папок согласно переданному фильтру. Фильтр может содержать путь, состоящий из вложенных папок внутри "песочницы", и должен содержать точное имя или шаблон имени искомых элементов файловой системы. Параметр *filter* не может быть пустым.

Шаблоном является строка, в которой присутствует один или более подстановочных символов. Существует два типа таких символов: звездочка ('*') заменяет произвольное количество любых символов (включая и нулевое количество), а знак вопроса ('?') заменяет не более одного любого символа. Например, фильтр "*" найдет все файлы и папки, а "???.*" — только те из них, в которых имя не длиннее 3 символов, причем расширение может быть или не быть. Файлы с расширением "csv" можно найти фильтром "*.csv" (но учтите, что папка тоже может иметь расширение). Фильтр "*." находит элементы без расширения, а ".*" — элементы без имени. Однако здесь следует помнить один нюанс.

> Во многих версиях Windows для элементов файловой системы генерируется два вида имен: длинное (по умолчанию, до 260 символов) и короткое (в формате 8.3, унаследованном от MS-DOS). Второй вид создается автоматически из длинного имени, если оно превышает 8 символов или расширение длиннее 3. Подобную генерацию коротких имен можно отключить в системе, если никакое программное обеспечение ими не пользуется, но обычно их поддержка включена.
>
> Поиск файлов производится в обоих видах имен, из-за чего возвращаемый перечень может содержать неожиданные, на первый взгляд, элементы. В частности, короткое имя, если оно сгенерировано системой из длинного названия, всегда содержит начальную часть перед точкой, занимающую вплоть до 8 символов. В ней может случайно обнаружиться совпадение с искомым шаблоном.

Если требуется найти файлы нескольких расширений или с разными фрагментами в имени, которые невозможно обобщить одним шаблоном, придется повторить процесс поиска несколько раз с разными настройками.

Поиск производится только в конкретной папке (либо в корневой папке "песочницы", если в фильтре нет пути, либо в указанной вложенной папке, если фильтр содержит путь) и не заходит в подкаталоги.

Поиск не зависит от регистра. Например, запрос файлов "*.txt" выдаст также файлы с расширением "TXT", "Txt" и т.д.

Если файл или папка с подходящих именем найдена, это имя помещается в выходной параметр *found* (требуется переменная, поскольку результат передается по ссылке), а функция возвращает дескриптор поиска: его нужно будет передавать в функцию *FileFindNext*, чтобы продолжать перебор подходящих элементов, если их много.

В параметре *found* возвращается только имя и расширение без пути (иерархии папок), который мог быть указан в фильтре.

Если найденный элемент является папкой, справа к его имени добавляется символ '\' (обратная косая черта).

Параметр *flag* позволяет выбрать область поиска между локальной рабочей папкой текущей копии терминала (по значению 0) или общей папкой всех терминалов (по значению FILE_COMMON). Когда MQL-программа выполняется в тестере, её локальная "песочница" (0) находится в каталоге агента тестирования.

После завершения процедуры поиска полученный дескриптор следует освободить, вызвав *FileFindClose* (см. далее).

bool FileFindNext(long handle, string &found)

Функция продолжает поиск подходящих элементов файловой системы, начатый функцией *FileFindFirst*. Первым параметром передается дескриптор, полученный из *FileFindFirst*, за счет чего применяются все прежние условия поиска.

Если очередной элемент найден, его имя передается в вызывающий код через аргумент *found*, а функция возвращает *true*.

Если элементов больше нет, функция возвращает *false*.

void FileFindClose(long handle)

Функция закрывает дескриптор поиска, полученный в результате вызова *FileFindFirst*.

Функцию необходимо вызвать после завершения процедуры поиска, чтобы освободить системные ресурсы.

В качестве примера рассмотрим скрипт *FileFind.mq5*. В предыдущих разделах мы тестировали много других скриптов, создававших файлы в каталоге *MQL5/Files/MQL5Book*. Запросим перечень всех таких файлов.

```
void OnStart()
{
   string found; // приемная переменная
   // начинаем поиск и получаем дескриптор
   long handle = PRTF(FileFindFirst("MQL5Book/*", found));
   if(handle != INVALID_HANDLE)
   {
      do
      {
         Print(found);
      }
      while(FileFindNext(handle, found));
      FileFindClose(handle);
   }
}
```

Даже если вы очистили этот каталог, в него можно скопировать поставляемые с книгой примеры файлов в различных кодировках. Таким образом, скрипт *FileFind.mq5* должен будет вывести как минимум такой список (порядок перечисления может меняться):

```
ansi1252.txt
unicode1.txt
unicode2.txt
unicode3.txt
utf8.txt
```

Чтобы упростить процесс поиска, в скрипте реализована вспомогательная функция *DirList*. Она содержит в себе все необходимые вызовы встроенных функций и цикл для построения строкового массива со списком элементов, отвечающих фильтру.

```
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;
}
```

С помощью неё запросим перечень каталогов в локальной "песочнице". Для этого используем предположение, что каталоги обычно не имеют расширения (в принципе, это не всегда так, и потому более строгий запрос перечня вложенных папок желающим следует реализовать иначе). Фильтр для элементов без расширения, вообще говоря, таков: "*." (вы можете проверить его с помощью команды *dir* в оболочке Windows "dir *."). Однако в функциях MQL5 данный шаблон вызывает ошибку 5002 (WRONG_FILENAME). Поэтому укажем более "расплывчатый" шаблон "*.?": он означает элементы без расширения или с расширением в 1 символ.

```
void OnStart()
{
   ...
   string list[];
   // пробуем запросить элементы без расширения
   // (работает в командной строке Windows)
   PRTF(DirList("*.", list)); // false / WRONG_FILENAME(5002)
   
   // расширим условие: расширение должно состоять из не более чем 1 символа
   if(DirList("*.?", list))
   {
      ArrayPrint(list);
      // пример: "MQL5Book\" "Tester\"
   }
}
```

На моей копии MetaTrader 5 скрипт находит 2 папки "MQL5Book\" и "Tester\" — первая из них должна быть и у вас, если вы запускали предыдущие тестовые скрипты.

4.5.16 Работа с папками

Файловую систему трудно представить без способности структурировать хранимую информацию за счет произвольной иерархии каталогов — контейнеров для наборов логически связанных файлов. На уровне MQL5 данная возможность также поддерживается. При необходимости мы можем создавать, очищать и удалять папки с помощью встроенных функций *FolderCreate*, *FolderClean*, *FolderDelete*.

Ранее мы уже видели один способ создания папки, причем, возможно, даже не одной, а сразу всей требуемой иерархии вложенных папок: для этого достаточно при создании (открытии) файла с помощью *FileOpen* или при его копировании (*FileCopy*, *FileMove*) задать не просто имя, а предварить его требуемым путем. Например,

```
FileCopy("MQL5Book/unicode1.txt", 0, "ABC/DEF/code.txt", 0);
```

Эта инструкция создаст папку "ABC" в "песочнице", в ней — папку "DEF", и скопирует туда файл под новым именем (файл-источник должен существовать).

Если создавать заранее файл-источник не хочется, можно создать файл-пустышку на лету:

```
uchar dummy[];
FileSave("ABC/DEF/empty", dummy);
```

Здесь мы получим ту же иерархию папок, что и в предыдущем примере, но с файлом "empty" нулевого размера.

При подобных подходах создание папок становится как бы побочным продуктом работы с файлами. Однако иногда требуется оперировать папками, как самостоятельными сущностями и без побочных эффектов, в частности, просто создать пустую папку. Это позволяет сделать функция *FolderCreate*.

bool FolderCreate(const string folder, int flag = 0)

Функция создает папку с именем *folder*, которое может включать путь (несколько названий папок верхних уровней). В любом случае, одиночная папка или иерархия папок создается в "песочнице", определяемой параметром *flag*. По умолчанию, когда *flag* равен 0, используется локальная рабочая папка *MQL5/Files* терминала или агента тестирования (если программа запущена в тестере). Если *flag* равен FILE_COMMON, используется общая папка всех терминалов.

Функция возвращает *true* в случае успеха или если папка уже существует. В случае ошибки результат равен *false*.

bool FolderClean(const string folder, int flag = 0)

Функция удаляет все файлы и папки любого уровня вложенности (вместе со всем содержимым) в указанном каталоге *folder*. Параметр *flag* задает "песочницу" (локальную или общую), в которой производится действие.

> Осторожно пользуйтесь этой функцией, так как все файлы и вложенные папки (с их файлами) удаляются безвозвратно.

bool FolderDelete(const string folder, int flag = 0)

Функция удаляет указанную папку (*folder*). Перед вызовом функции папка должна быть пуста, иначе её не удастся удалить.

Приемы работы с данными тремя функциями продемонстрированы в скрипте *FileFolder.mq5*. Вы можете выполнять данный скрипт в режиме отладки по шагам (инструкция за инструкцией) и наблюдать в файловом менеджере, как появляются и исчезают папки и файлы. Однако учтите, что перед выполнением очередной инструкции следует выходить файловым менеджером из созданных папок наверх до уровня "MQL5Book", потому что в противном случае папки могут быть заняты файловым менеджером, и это нарушит работу скрипта.

В начале мы создаем несколько вложенных папок как побочный продукт записи в них пустого фиктивного файла.

```
void OnStart()
{
   const string filename = "MQL5Book/ABC/DEF/dummy";
   uchar dummy[];
   PRTF(FileSave(filename, dummy)); // true
```

Далее мы создаем еще одну папку на нижнем уровне вложенности с помощью *FolderCreate*: на этот раз папка появляется сама по себе, без вспомогательного файла.

```
   PRTF(FolderCreate("MQL5Book/ABC/GHI")); // true
```

Если попытаться удалить папку "DEF", это закончится ошибкой, потому что она не пуста (там есть файл).

```
   PRTF(FolderDelete("MQL5Book/ABC/DEF")); // false / CANNOT_DELETE_DIRECTORY(5024)
```

Для того чтобы её удалить, необходимо её сперва очистить, и проще всего это сделать с помощью *FolderClean*. Но мы попробуем сымитировать часто встречающуюся ситуацию, когда некоторые файл в очищаемых папках могут быть заблокированы другими MQL-программами, внешними приложениями или самим терминалом. Откроем файл на чтение и вызовем *FolderClean*.

```
   int handle = PRTF(FileOpen(filename, FILE_READ)); // 1
   PRTF(FolderClean("MQL5Book/ABC")); // false / CANNOT_CLEAN_DIRECTORY(5025)
```

Функция возвращает *false* и выставляет код ошибки 5025 (CANNOT_CLEAN_DIRECTORY). После того как мы закроем файл, очистка и удаление всей иерархии папок проходит успешно.

```
   FileClose(handle);
   PRTF(FolderClean("MQL5Book/ABC")); // true
   PRTF(FolderDelete("MQL5Book/ABC")); // true
}
```

Потенциальные блокировки более вероятны при использовании общего каталога терминалов, где на один файл или папку могут "претендовать" разные экземпляры программ. Но и в локальной песочнице не стоит забывать о возможных конфликтах (например, если csv-файл открыт в Excel). Снабжайте фрагменты кода, работающего с папками, подробной диагностикой и выводом ошибок, чтобы пользователь смог заметить и устранить проблему.

4.5.17 Диалог выбора файла или папки

В группе функций по работе с файлами и папками есть одна, позволяющая интерактивно запросить имя файла или папки, а также группу файлов у пользователя, чтобы передать эту

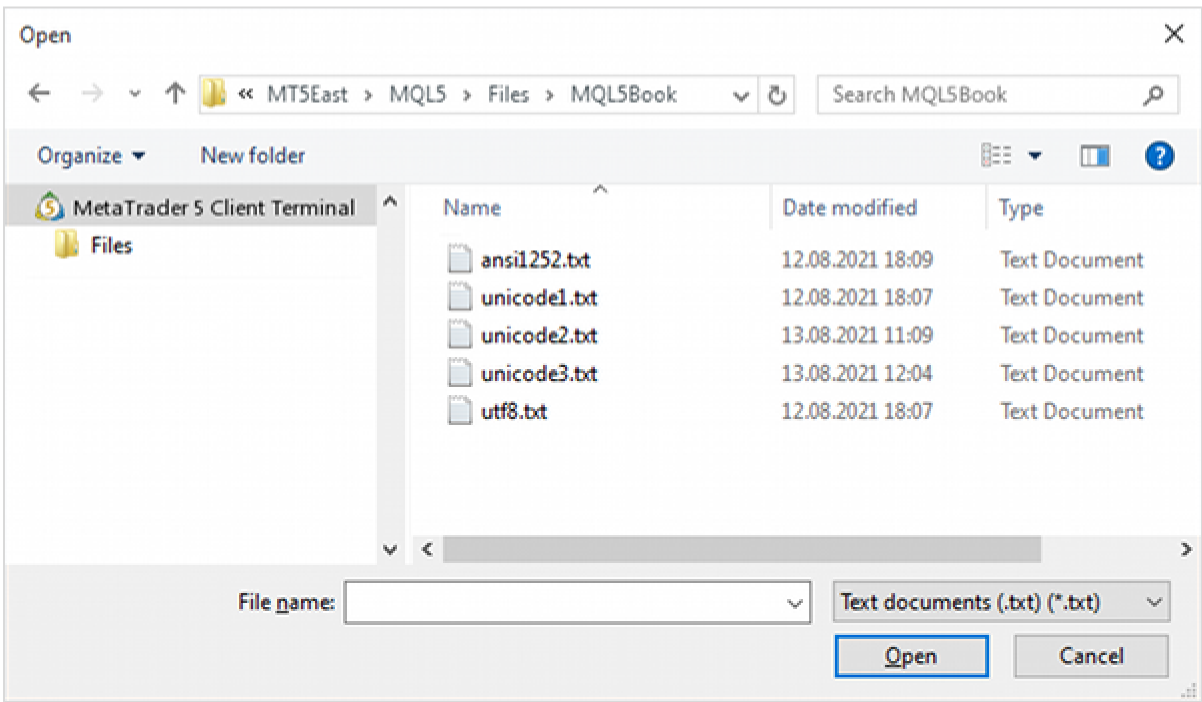
информацию в MQL-программу. Вызов данной функций — *FileSelectDialog* — приводит к появлению в терминале стандартного окна выбора файлов и папок системы Windows.

> Поскольку диалог прерывает выполнение MQL-программы до момента его закрытия, вызов функции разрешен только в двух типах MQL-программ, которые исполняются в отдельных потоках: экспертах и скриптах (см. раздел о типах MQL-программ). Пользоваться данной функцией запрещено в индикаторах и сервисах: первые исполняются в интерфейсном потоке терминала (и их остановка заморозила бы обновление графиков соответствующих инструментов), а вторые исполняются в фоновом режиме и не могут обращаться к пользовательскому интерфейсу.

Все элементы файловой системы, с которыми работает функция, находятся внутри "песочницы", то есть в каталоге текущей копии терминала или агента тестирования (если программа выполняется в тестере), в подпапке *MQL5/Files*.

При наличии флага FSD_COMMON_FOLDER в параметре *flags* (см. далее) используется общая "песочница" всех терминалов *Users/<пользователь>...MetaQuotes/Terminal/Common/Files*.

Внешний вид диалога зависит от операционной системы Windows. Ниже показан один из возможных вариантов интерфейса.



Диалог выбора файлов и папок Windows

```
int FileSelectDialog(const string caption, const string initDir, const string filter,
    uint flags, string &filenames[], const string defaultName)
```

Функция выводит стандартный диалог Windows для открытия или создания файла, или выбора папки. В параметре *caption* задается заголовок диалога: при значении NULL используется стандартный заголовок — "Открыть" для чтения или "Сохранить как" для записи файла, или же "Выбор папки", в зависимости от флагов в параметре *flags*.

Параметр *initDir* позволяет задать начальную папку, для которой откроется диалог. При значении NULL будет показано содержимое папки *MQL5/Files*. Эта же папка используется, если в *initDir* указан несуществующий путь.

С помощью параметра *filter* можно ограничить набор расширений файлов, которые будут показаны в диалоговом окне. Файлы других форматов будут скрыты. Значение NULL означает отсутствие ограничений.

Формат строки *filter* следующий:

```
"<описание 1>|<расширение 1>|<описание 2>|<расширение 2>..."
```

В качестве *описания* допустима любая строка. В качестве *расширения* можно написать любой фильтр с подстановочными символами '*' и '?', которые мы рассматривали в разделе Поиск файлов и папок. Символ '|' является разделителем.

Поскольку соседние описание и расширение образуют логически связанную пару, общее количество элементов в строке должно быть четным, а количество разделителей — нечетным.

Каждое сочетание описания и расширения генерирует отдельный вариант выбора в выпадающем списке диалога. Описание показывается пользователю, а расширение используется для фильтрации.

Например, "Text documents (*.txt)|*.txt|All files (*.*)|*.*", при этом первое расширение "Text documents (*.txt)|*.txt" будет выбрано как тип файла по умолчанию.

В параметре *flags* можно указать с помощью оператора '|' битовую маску, задающую режимы работы. Для неё определены следующие константы:

- FSD_WRITE_FILE — режим для записи файлов ("Сохранить как"); при отсутствии данного флага по умолчанию используется режим чтения ("Открыть"); при наличии данного флага всегда разрешен ввод произвольного нового имени, независимо от флага FSD_FILE_MUST_EXIST;
- FSD_SELECT_FOLDER — режим выбора папки (только одной и только существующей); при данном флаге все прочие флаги кроме FSD_COMMON_FOLDER игнорируются или вызывают ошибку; запросить явным образом создание папки нельзя, но в диалоге есть возможность создать папку интерактивно и тут же её выбрать;
- FSD_ALLOW_MULTISELECT — разрешение выбирать несколько файлов в режиме чтения; этот флаг игнорируется, если указан FSD_WRITE_FILE или FSD_SELECT_FOLDER;
- FSD_FILE_MUST_EXIST — выбранные файлы должны существовать; если пользователь попытается указать произвольное имя, диалог выведет предупреждение и останется открытым; данный флаг игнорируется, если указан режим FSD_WRITE_FILE;
- FSD_COMMON_FOLDER — диалог открывается для общей "песочницы" всех клиентских терминалов.

Функция заполнят массив строк *filenames* именами выбранных файлов или папки. Если массив динамический, его размер изменяется под фактическое количество данных, в частности, расширяется или наоборот урезается вплоть до 0, если ничего не было выбрано. Если массив фиксированный, он должен быть достаточного размера, чтобы принять ожидаемые данные. В противном случае возникнет ошибка 4007 (ARRAY_RESIZE_ERROR).

Параметр *defaultName* указывает имя файла/папки по умолчанию, которое будет подставлено в соответствующее поле ввода сразу после открытия диалога. Если параметр равен NULL, поле будет изначально пустым.

> Если параметр *defaultName* задан, то во время невизуального тестирования MQL-программы вызов *FileSelectDialog* вернёт 1, а само значение *defaultName* будет скопировано в массив *filenames*.

Функция возвращает количество выбранных элементов (0, если пользователь ничего не выбрал) или -1 в случае ошибки.

Рассмотрим примеры работы функции в скрипте *FileSelect.mq5*. В функции *OnStart* будем последовательно вызывать *FileSelectDialog* с разными настройками. Пока пользователь выбирает что-то (не нажимает кнопку "Отмена" в диалоге), тест продолжается вплоть до последнего шага (даже если функция выполнилась с кодом ошибки).

```
void OnStart()
{
   string filenames[]; // динамический массив подойдет для любого вызова
   string fixed[1]; // слишком маленький массив, если файлов больше 1
   const string filter = // пример фильтров
      "Text documents (*.txt)|*.txt"
      "|Files with short names|????.*"
      "|All files (*.*)|*.*";
```

Сперва запросим у пользователя один файл из папки "MQL5Book". Можно выбрать существующий файл или ввести новое имя (потому что нет флага 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"
```

В предположении, что в папке присутствуют как минимум 5 файлов из поставки книги, здесь выбран один из них.

Затем сделаем аналогичный запрос в режиме "для записи" (с флагом 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"
```

Здесь пользователь также сможет выбрать как существующий файл, так и ввести новое имя. Проверку на то, что пользователь собирается перезаписать существующий файл, должен сделать программист (диалог предупреждений не выдает).

Теперь проверим выбор нескольких файлов (FSD_ALLOW_MULTISELECT) в динамический массив.

```
   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"
```

Наличие флага FSD_FILE_MUST_EXIST означает, что диалог выведет предупреждение и останется открытым, если попытаться ввести новое имя.

Если попробовать похожим образом выбрать более 1 файла в массив фиксированного размера, мы получим ошибку.

```
   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
```

Наконец, протестируем работу с папками (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"
```

В данном случае в качестве стартового пути указана несуществующая вложенная папка "nonexistent", поэтому диалог откроется в корне "песочницы" — *MQL5/Files*. Там мы выбрали "MQL5book".

Если скомбинировать некорректное сочетание флагов, получим еще одну ошибку.

```
   if(PRTF(FileSelectDialog(NULL, "MQL5book", NULL,
      FSD_SELECT_FOLDER | FSD_WRITE_FILE, filenames, NULL)) == 0) return;
   // -1 / INTERNAL_ERROR(4001)
   ArrayPrint(filenames); // "MQL5Book"
}
```

Из-за ошибки функция не стала модифицировать переданный массив, и в нем остался прежний элемент "MQL5Book".

В данном тесте мы намеренно проверяли результаты только на 0, чтобы продемонстрировать все варианты вне зависимости от наличия ошибок. В реальной программе проверяйте результат функции с учетом ошибок, т.е. с условиями на три исхода: выбор сделан (>0), выбор не сделан (==0) и ошибка (<0).

4.6 Глобальные переменные терминала

В предыдущей главе мы изучили функции MQL5 для работы с файлами. Они предоставляют широкие, гибкие возможности по записи и чтению произвольных данных. Однако иногда MQL-программе требуется более простой способ сохранить и восстановить состояние какого-либо атрибута между запусками.

Например, мы хотим подсчитывать некую статистику — сколько раз была запущена программа, сколько её копий выполняется параллельно на разных графиках и т.д. Накапливать эту информацию внутри самой программы невозможно. Это должно быть некое внешнее долговременное "хранилище". Но создавать ради этого файл было бы накладно, хотя, конечно, возможно.

Кроме того, многие программы проектируются с расчетом на взаимодействие друг с другом, то есть они должны как-то обмениваться информацией. Если речь об интеграции с программой, внешней по отношению к терминалу, или о передаче большого объема данных, то здесь без файлов, действительно трудно обойтись. Однако, когда пересылаемых данных мало, а все программы написаны на MQL5 и выполняются внутри MetaTrader 5, использование файлов кажется избыточным. Ведь в терминале на этот случай реализована более простая технология: глобальные переменные.

Глобальная переменная — это именованная ячейка в общей памяти терминала. Она может создаваться, модифицироваться, удаляться любой MQL-программой, но не будет принадлежать исключительно ей и доступна всем другим MQL-программам. Имя глобальной переменной — это любая уникальная (среди всех переменных) строка длиной не более 63 символов. Эта строка не обязана отвечать требованиям идентификаторов переменных в MQL5, поскольку глобальные переменные терминала не являются переменными в привычном понимании. Программист не определяет их в исходном коде в соответствии с синтаксисом, который мы изучали в главе Переменные, они не являются составной частью MQL-программы, и любое действие с ними выполняется только через вызов одной из специальных функций, которые мы опишем в этой главе.

Глобальные переменные терминала позволяют хранить только значения типа *double*. При необходимости вы можете упаковать/конвертировать значения других типов в *double* или использовать часть имени переменной (например, после определенного префикса) для хранения строк.

Пока терминал работает, глобальные переменные хранятся в оперативной памяти и доступны практически моментально: единственные накладные расходы связанны с вызовом функций. Это выгодно отличает глобальные переменные от использования файлов, в случае которых получение дескриптора — относительно медленный процесс, да и сам дескриптор потребляет некоторые дополнительные ресурсы.

При завершении рабочей сессии терминала глобальные переменные выгружаются в специальный файл (*gvariables.dat*) и затем восстанавливаются из него при следующем запуске терминала.

Конкретная глобальная переменная автоматически уничтожается терминалом, если она не была востребована в течение 4 недель. Такое поведение основывается на отслеживании и сохранении времени последнего использования переменной, причем под использованием понимается установка нового значения или чтение старого (но не проверка на существование или получение времени последнего использования).

> Обратите внимание, что глобальные переменные не привязаны ни к счету, ни к профилю, ни к каким-либо еще характеристикам торговой среды. Поэтому, если в них предполагается хранить нечто, относящееся к окружению (например, некие общие лимиты для конкретного счета), имена переменных следует конструировать с учетом всех факторов, которые влияют на алгоритм и принятие решений. Для различения глобальных переменных нескольких экземпляров одного и того же эксперта может потребоваться добавить в имя рабочий инструмент, таймфрейм или "магическое число" из настроек эксперта.

Кроме MQL-программ глобальные переменные может вручную создавать и пользователь. Список существующих глобальных переменных, а также средства интерактивного управления ими находятся в диалоге, открываемом в терминале по команде *Сервис -> Глобальные переменные* (F3).

С помощью соответствующих кнопок здесь можно *Добавить* и *Удалить* глобальную переменную, а двойной щелчок мыши в колонках *Переменная* или *Значение* позволяет редактировать имя или величину конкретной переменной. Для работы с клавиатуры действуют следующие горячие клавиши: F2 — редактирование имени, F3 — редактирование значения, Ins — добавление новой переменной, Del — удаление выбранной переменной.

Чуть позже мы изучим два основных типа MQL-программ — эксперты и индикаторы. Их отличает способность запускаться в тестере, где функции для глобальных переменных также работают. Однако там глобальные переменные создаются, хранятся и управляются агентом тестирования.

Иными словами, списки глобальных переменных терминала недоступны в тестере, а те переменные, которые создаются тестируемой программой, принадлежат конкретному агенту и время их жизни ограничено одним тестовым проходом. То есть, глобальные переменные агента не видны из других агентов и будут удалены по окончании теста. В частности, если эксперт оптимизируется на нескольких агентах, он может манипулировать глобальными переменными для "общения" с индикаторами, используемыми им в контексте того же агента, поскольку они выполняются там совместно, но на параллельно работающих агентах другими копиями эксперта будут формироваться собственные списки переменных.

> Обмен данными между MQL-программ с помощью глобальных переменных — не единственный и не всегда самый подходящий способ. В частности, эксперты и индикаторы являются интерактивными типами MQL-программ, способными генерировать и принимать события на графиках. В параметрах событий можно передавать информацию различных типов. Кроме того, массивы расчетных данных могут подготавливаться и предоставляться другим MQL-программам в виде буферов индикаторов. MQL-программы, располагающиеся на графиках, могут использовать для передачи и хранения информации интерфейсные графические объекты.

Чисто технически максимальное количество глобальных переменных ограничено лишь ресурсами операционной системы. Однако для большого количества элементов рекомендуется использовать более подходящие средства: файлы или базы данных.

4.6.1 Запись и чтение глобальной переменной

Для записи и чтения глобальных переменных MQL5 API предоставляет 2 функции: *GlobalVariableSet* и *GlobalVariableGet* (в двух вариантах).

datetime GlobalVariableSet(const string name, double value)

Функция устанавливает новое значение *value* глобальной переменной с именем name. Если переменной не существовало до вызова функции, она будет создана. Если переменная уже была, прежнее значение будет заменено на *value*.

При успешном выполнении функция возвращает время модификации переменной (текущее локальное время компьютера). В случае ошибки получим 0.

double GlobalVariableGet(const string name)
bool GlobalVariableGet(const string name, double &value)

Функция возвращает значение глобальной переменной по имени name. Результат вызова первого варианта функции содержит непосредственно значение переменной (в случае успеха) или 0 (в случае ошибки). Поскольку переменная может содержать значение 0 (что совпадает с индикацией ошибки), данный вариант требует анализировать внутренний код ошибки *_LastError*, если получен ноль, чтобы отличить штатное исполнение от нештатного. В частности, если делается попытка прочитать несуществующую переменную, генерируется внутренняя ошибка 4501 (GLOBALVARIABLE_NOT_FOUND).

Данный вариант функции удобно применять в алгоритмах, где получение нуля является подходящим аналогом инициализации по умолчанию для несуществовавшей ранее переменной (см. пример ниже). Если же отсутствие переменной нужно обработать особым образом (в частности, вычислить какое-то другое стартовое значение), следует предварительно проверить переменную на существование с помощью функции *GlobalVariableCheck* и в зависимости от её результата выполнять разные ветки кода. Или же можно воспользоваться вторым вариантом.

Второй вариант функции возвращает *true* или *false* в зависимости от успешности выполнения. В случае успеха значение глобальной переменной терминала помещается в приемную переменную *value*, передаваемую по ссылке вторым параметром. Если переменной нет, получим *false*.

В тестовом скрипте *GlobalsRunCount.mq5* используем глобальную переменную для подсчета количества раз, которое он запускался. В качестве имени переменной выступает имя исходного файла.

```
const string gv = __FILE__;
```

Напомним, встроенный макрос __FILE__ (см. раздел Предопределенные константы) раскрывается компилятором в название компилируемого файла, то есть в данном случае "GlobalsRunCount.mq5".

В функции *OnStart* попытаемся прочитать данную глобальную переменную и сохраним полученный результат в локальную переменную *count*. Если глобальной переменной еще не было, получим 0, что нас устраивает (начинаем подсчет с нуля).

Перед сохранением значения в *count* необходимо делать приведение к (*int*), поскольку функция *GlobalVariableGet* возвращает *double*, и без приведения компилятор выдает предупреждение о потенциальной потере данных (он не в курсе, что мы планируем хранить исключительно целые числа).

```
void OnStart()
{
   int count = (int)PRTF(GlobalVariableGet(gv));
   count++;
   PRTF(GlobalVariableSet(gv, count));
   Print("This script run count: ", count);
}
```

Затем мы увеличиваем счетчик на 1 и записываем его обратно в глобальную переменную с помощью *GlobalVariableSet*. Если запустить скрипт несколько раз, получим примерно следующие записи в журнале (метки времени у вас будут отличаться):

```
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
```

Важно отметить, что при первом запуске мы получили значение 0, и был взведен внутренний признак ошибки 4501. Все последующие вызовы уже выполняются без проблем, так как переменная существует (её можно увидеть в окне "Глобальные переменные" терминала). Желающие могут закрыть терминал, запустить его вновь и выполнить скрипт еще раз: счетчик продолжит увеличиваться с прошлого значения.

4.6.2 Проверка существования и времени последнего действия

Как мы видели в предыдущем разделе, проверить существование глобальной переменной можно, попытавшись прочитать её значение: если это не приведет к появлению кода ошибки в *_LastError*, значит, глобальная переменная существует, причем мы уже получили её величину и можем использовать в алгоритме. Однако, если в каких-то условиях требуется лишь проверить существование, но не считывать глобальную переменную, более удобно применить другую функцию, специально предназначенную для этого: *GlobalVariableCheck*.

Существует и еще один вариант проверки — с помощью функции *GlobalVariableTime*. Как ясно из её названия, она позволяет узнать время последнего использования переменной. Но если переменной не существует, то и время её использования отсутствует, то есть равно 0.

bool GlobalVariableCheck(const string name)

Функция проверяет существование глобальной переменной с указанным именем и возвращает результат: *true* (переменная есть) или *false* (переменной нет).

datetime GlobalVariableTime(const string name)

Функция возвращает время последнего использования глобальной переменной с указанным именем. Фактом использования может быть модификация или чтение значения переменной.

Проверка переменной на существование с помощью *GlobalVariableCheck* или получение её времени посредством *GlobalVariableTime* не меняют времени использования.

В скрипте *GlobalsRunCheck.mq5* мы слегка дополним код из *GlobalsRunCount.mq5*, чтобы в самом начале функции *OnStart* проверять наличие переменной и время её использования.

```
void OnStart()
{
   PRTF(GlobalVariableCheck(gv));
   PRTF(GlobalVariableTime(gv));
   ...
```

Далее идет код без изменений. Но обратите внимание, что переменная *gv*, определенная через __FILE__, на этот раз будет содержать новое имя скрипта "GlobalsRunCheck.mq5" в качестве имени глобальной переменной (т.е. каждый скрипт у нас имеет свой глобальный счетчик).

Все запуски кроме самого первого покажут значение *true* от функции *GlobalVariableCheck* (переменная существует), а время переменной — с предыдущего запуска. Вот пример журнала:

```
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 Получение списка глобальных переменных

Довольно часто MQL-программе требуется просмотреть существующие глобальные переменные и выбрать из них подходящие по некоторому критерию. Например, если программа использует часть имени переменной для хранения текстовой информации, то заранее известен лишь префикс, предназначенный для идентификаций "своей" переменной, а "полезная нагрузка", пристыкованная к префиксу, не дает возможности искать переменную по точному имени.

MQL5 API имеет две функции, которые позволяют организовать перебор глобальных переменных.

int GlobalVariablesTotal()

Функция возвращает общее количество глобальных переменных.

string GlobalVariableName(int index)

Функция возвращает имя глобальной переменной по порядковому номеру в списке глобальных переменных. Параметр *index* с номером запрашиваемой переменной должен находиться в диапазоне от 0 до *GlobalVariablesTotal() - 1*.

В случае ошибки функция вернет NULL, а код ошибки можно получить из служебной переменной *_LastError* или функции *GetLastError*.

Проверим эту пару функций с помощью скрипта *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));
   }
}
```

В первой строке намеренно запрашивается имя переменной с большим номером, которой, скорее всего, не существует, что должно вызвать ошибку. Далее производится запрос реального

количества переменных и цикл по всем из них, с выводом имени и значения. В журнал ниже попали переменные, созданные предыдущими тестовыми скриптами, и одна сторонняя.

```
GlobalVariableName(1000000)= / GLOBALVARIABLE_NOT_FOUND(4501)
GlobalVariablesTotal()=3 / ok
0 GlobalsRunCheck.mq5=3.000000
1 GlobalsRunCount.mq5=4.000000
2 abracadabra=0.000000
```

Порядок, в котором терминал "отдает" переменные по индексу, не определен.

4.6.4 Удаление глобальных переменных

При необходимости MQL-программа может удалить ставшую ненужной глобальную переменную или их группу. Список глобальных переменных потребляет некоторые ресурсы компьютера, а хороший стиль программирования предполагает, что ресурсы следует по возможности освобождать.

bool GlobalVariableDel(const string name)

Функция удаляет глобальную переменную с именем name. При успешном выполнении функция возвращает *true*, иначе — *false*.

int GlobalVariablesDeleteAll(const string prefix = NULL, datetime limit = 0)

Функция удаляет глобальные переменные с указанным префиксом в имени и более старым временем использования, чем значение в параметре *limit*.

Если указан префикс NULL (по умолчанию) или пустая строка, то под критерий удаления попадают все глобальные переменные, соответствующие также и критерию удаления по дате (если он задан).

Если параметр *limit* равен нулю (по умолчанию), то удаляются глобальные переменные с любой датой, с учетом префикса.

Если указаны оба параметра, то удаляются глобальные переменные, соответствующие одновременно и префиксу, и критерию по времени.

Будьте осторожны: вызов *GlobalVariablesDeleteAll* без параметров приведет к удалению всех переменных.

Функция возвращает количество удаленных переменных.

Рассмотрим скрипт *GlobalsDelete.mq5*, эксплуатирующий две новых функции.

```
void OnStart()
{
   PRTF(GlobalVariableDel("#123%"));
   PRTF(GlobalVariablesDeleteAll("#123%"));
   ...
```

В начале делается попытка удалить несуществующие глобальные переменные по точному имени и префиксу. Обе не оказывают эффекта на существующие переменные.

Вызов *GlobalVariablesDeleteAll* с фильтром по времени в прошлом (больше 4 недель назад) также имеет нулевой результат, потому что столь "древние" переменные терминал удаляет сам автоматически (такие переменные не могут существовать).

```
   PRTF(GlobalVariablesDeleteAll(NULL, D'2021.01.01'));
```

Далее для тестирования создается переменная с именем "abracadabra" (если её не было) и тут же удаляется. Эти вызовы должны закончиться успешно.

```
   PRTF(GlobalVariableSet(abracadabra, 0));
   PRTF(GlobalVariableDel(abracadabra));
```

Наконец, удалим переменные, начинающиеся с префикса "GlobalsRun": они должны были быть созданы тестовыми скриптами из двух предыдущих разделов по именам файлов (соответственно, "GlobalsRunCount.mq5" и "GlobalsRunCheck.mq5").

```
   PRTF(GlobalVariablesDeleteAll("GlobalsRun"));
   PRTF(GlobalVariablesTotal());
}
```

Скрипт должен вывести в журнал примерно следующее (некоторые показатели зависят от внешних условий и времени запуска).

```
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
```

В конце мы вывели общее количество оставшихся глобальных переменных (в данном случае получили 0, то есть переменных нет). Оно может у вас отличаться, если глобальные переменные создавались другими MQL-программами или пользователем.

4.6.5 Временные глобальные переменные

В подсистеме глобальных переменных терминала существует возможность делать некоторые переменные временными: они хранятся только в памяти и не записываются на диск при закрытии терминала.

В силу своей специфики временные глобальные переменные используются исключительно для обмена данными между MQL-программами и не подходят для сохранения состояния между запусками MetaTrader 5. Одно из наиболее очевидных применений для временных переменных: различные показатели операционной деятельности (например, счетчики запущенных копий программ), которые должны динамически пересчитываться при каждом старте, а не восстанавливаться с диска.

Глобальную переменную следует объявить временной заранее, до назначения ей какого-либо значения, с помощью функции *GlobalVariableTemp*.

К сожалению, по имени глобальной переменной нельзя узнать, является ли она временной: MQL5 не предоставляет для этого средств.

Временные переменные могут создавать только MQL-программы. Временные переменные отображаются в окне "Глобальные переменные" наравне с обычными (персистентными) глобальными переменными, но пользователь не имеет возможности добавить свою временную переменную из GUI.

bool GlobalVariableTemp(const string name)

Функция создает новую глобальную переменную с указанным именем, которая будет существовать только до конца текущего сеанса работы терминала.

Если переменная с таким именем уже существовала, она не будет преобразована во временную.

Если же переменой до сих пор не было, она получит значение 0. После этого с ней можно работать как обычно, в частности, присваивать другие значения функцией *GlobalVariableSet*.

Пример для данной функции мы покажем вместе с функциями следующего раздела.

4.6.6 Синхронизация программ с помощью глобальных переменных

Поскольку глобальные переменные существуют вне MQL-программ, их удобно применять для организации внешних флагов, управляющих несколькими копиями одной и той же программы или передающих сигналы между разными программами. Наиболее простой пример заключается в ограничении количества запускаемых копий программы. Это может быть необходимо, чтобы предотвратить случайное дублирование эксперта на разных графиках (из-за чего торговые приказы могут задваиваться), или для реализации демонстрационной версии.

На первый взгляд, подобная проверка могла бы быть сделана в исходном коде следующим образом.

```
void OnStart()
{
   const string gv = "AlreadyRunning";
   // если переменная есть, значит один экземпляр уже выполняется
   if(GlobalVariableCheck(gv)) return;
   // создаем переменную, как флаг сигнализирующий наличие работающей копии
   GlobalVariableSet(gv, 0);
   
   while(!IsStopped())
   {
      // рабочий цикл
   }
   // удаляем переменную перед выходом
   GlobalVariableDel(gv);
}
```

Здесь показан простейший вариант на примере скрипта — для других типов MQL-программ общий принцип проверки будет тот же самый, хотя место расположения инструкций может отличаться: вместо бесконечного рабочего цикла в экспертах и индикаторах используются характерные для них обработчики событий, многократно вызываемые терминалом. Эти нюансы мы изучим позднее.

Проблема с представленным кодом заключается в том, что здесь не учтено параллельное выполнение MQL-программ.

MQL-программа, как правило, выполняется в собственном потоке. Для трех из четырех типов MQL-программ, а именно для экспертов, скриптов и сервисов, система совершенно точно выделяет отдельные потоки. В случае индикаторов по одному общему потоку выделяется на все их экземпляры, работающие на одинаковом сочетании рабочего инструмента и таймфрейма. Но индикаторы на разных сочетаниях по-прежнему принадлежат разным потокам.

Практически всегда в терминале выполняется очень много потоков — значительно больше, чем количество ядер процессора. Из-за этого каждый поток периодически ненадолго приостанавливается системой, чтобы дать возможность поработать и другим потокам. Поскольку все такие переключения между потоками происходят очень быстро, мы как пользователи не замечаем этой "внутренней кухни". Однако каждая приостановка может влиять на последовательность, в которой разные потоки получают доступ к разделяемым общим ресурсам. И глобальные переменные как раз являются такими ресурсами.

С точки зрения программы приостановка может случиться между любыми соседними инструкциями. Если, зная это, взглянуть снова на наш пример, нетрудно увидеть место, где логика работы с глобальной переменной может быть нарушена.

Действительно, первая копия (поток) может выполнить проверку и не обнаружить переменную, но быть тут же приостановленной. В результате, она еще не успеет создать переменную своей следующей инструкцией, как контекст выполнения переключится на вторую копию. Та тоже не обнаружит переменную и "решит" продолжить работу, как и первая. Для наглядности одинаковый исходный код двух копий приведен ниже в виде двух столбцов инструкций в порядке их перемежающегося исполнения.

Копия 1

```
void OnStart()
{
   const string gv = "AlreadyRunning";

   if(GlobalVariableCheck(gv)) return;
   // переменной нет

   GlobalVariableSet(gv, 0);
   // "я - первый и единственный"
   while(!IsStopped())

   {
      ;

   }
   GlobalVariableDel(gv);

}
```

Копия 2

```
void OnStart()
{

   const string gv = "AlreadyRunning";

   if(GlobalVariableCheck(gv)) return;
   // переменной все еще нет

   GlobalVariableSet(gv, 0);
   // "нет, я - первый и единственный"
   while(!IsStopped())
   {

      ;
   }

   GlobalVariableDel(gv);
}
```

Разумеется, такая схема переключений между потоками обладает изрядной долей условности. Но здесь важна сама возможность нарушения логики программы, хотя бы и в одной единственной строке. Когда программ (потоков) много, вероятность непредвиденных действий с общими ресурсами возрастает. Этого может оказаться достаточно, чтобы в самый неожиданный момент увести эксперт в убыток или получить искаженные оценки технического анализа.

Самое неприятное в ошибках такого рода в том, что их очень сложно обнаружить. Их не способен обнаружить компилятор, и на стадии выполнения они проявляют себя спорадически. Но если ошибка не дает о себе знать долгое время, это не значит, что её нет.

Для решения подобных проблем необходимо неким образом синхронизировать доступ всех копий программ к разделяемым ресурсам (в данном случае, к глобальным переменным).

В информатике имеется специальное понятие — мьютекс (от английского mutex, mutual exclusion) — объект по обеспечению исключительного доступа к разделяемому ресурсу из параллельно выполняющихся программ. Мьютекс предотвращает потерю или порчу данных из-за асинхронных изменений. Обычно обращение к мьютексу синхронизирует разные программы за счет того, что только одна из них может редактировать защищенные данные, захватив мьютекс в конкретный момент, а остальные вынуждены ждать, пока мьютекс не освободится.

В MQL5 готовых мьютексов в чистом виде не существует. Но для глобальных переменных похожий эффект позволяет получить следующая функция, которую мы рассмотрим.

bool GlobalVariableSetOnCondition(const string name, double value, double precondition)

Функция устанавливает новое значение *value* существующей глобальной переменной *name* при условии, что её текущее значение равно *precondition*.

При успешном выполнении функция возвращает *true*, иначе — *false*, и код ошибки будет доступен в *_LastError*. В частности, если переменной не существует, функция сгенерирует ошибку ERR_GLOBALVARIABLE_NOT_FOUND (4501).

Функция обеспечивает атомарный доступ к глобальной переменной, то есть неразделимым образом проводит два действия: проверяет её текущее значение, и если оно соответствует условию, присваивает новую величину *value*.

Эквивалентный код функции можно представить примерно следующим образом (почему "примерно" — поясним далее):

```
bool GlobalVariableZetOnCondition(const string name, double value, double preconditic
{
   bool result = false;
   { /* включить защиту от прерывания */ }
   if(GlobalVariableCheck(name) && (GlobalVariableGet(name) == precondition))
   {
      GlobalVariableSet(name, value);
      result = true;
   }
   { /* выключить защиту от прерывания */ }
   return result;
}
```

Реализовать такой код, работающий по задумке, невозможно по двум причинам. Во-первых, блоки с включением и выключением защиты от прерывания нечем реализовать на чистом MQL5 (внутри встроенной функции *GlobalVariableSetOnCondition* это обеспечивает само ядро). Во-вторых, вызов функции *GlobalVariableGet* меняет время последнего использования переменной, в то время как функция *GlobalVariableSetOnCondition* не меняет его, если предусловие не было выполнено.

Для демонстрации работы с *GlobalVariableSetOnCondition* мы впервые обратимся к новому типу MQL-программ: сервисам. Подробно мы изучим их в отдельном разделе. Пока же достаточно будет отметить, что их структура максимально похожа на скрипты: и там, и там имеется лишь одна главная функция (точка входа) — знакомая нам *OnStart*. Единственное существенное отличие заключается в том, что скрипт выполняется на графике, а сервис — сам по себе (в фоновом режиме).

Необходимость замены скриптов на сервисы объясняется тем, что прикладной смысл задачи, в которой мы используем *GlobalVariableSetOnCondition*, заключается в подсчете количества запущенных экземпляров программы, с возможностью установки лимита. При этом коллизии с одновременной модификацией разделяемого счетчика могут возникать только в момент запуска множества программ. А в случае скриптов довольно сложно запустить несколько их копий на разных графиках в относительно короткий промежуток времени. Для сервисов же, наоборот, в интерфейсе терминала имеется удобный механизм для пакетного (группового) запуска. Кроме того, все активированные сервисы автоматически стартуют при очередной загрузке терминала.

Предлагаемый механизм подсчета количества копий будет востребованным и для MQL-программ других типов. Поскольку эксперты и индикаторы остаются прикрепленными на графики даже при выключении терминала, при следующем его включении происходит практически одновременное чтение всеми программами их настроек и общих ресурсов. Поэтому, если в какие-то эксперты и индикаторы встроено ограничение на количество копий, критически важно синхронизировать подсчет на основе глобальных переменных.

Сперва рассмотрим сервис, реализующий контроль копий в наивном режиме, без применения *GlobalVariableSetOnCondition*, и убедимся, что проблема сбоев счетчика реальная. Сервисы находятся в выделенном подкаталоге в общем каталоге исходных кодов, поэтому приведем расширенный путь — *MQL5/Services/MQL5Book/p4/GlobalsNoCondition.mq5*.

В начале файла сервиса должна идти директива:

```
#property service
```

В сервисе предусмотрим 2 входных переменных, чтобы задать ограничение на количество разрешенных параллельно выполняющихся копий (limit) и задержку для эмуляции прерывания выполнения из-за массовой нагрузки на диск и CPU компьютера, что часто бывает при запуске терминала. Это облегчит воспроизведение проблемы без необходимости перезагружать терминал много раз в надежде добиться рассинхронизации — ведь мы собираемся отлавливать ошибку, которая может возникать лишь эпизодически, но при этом, уж если случилась, то чревата серьезными последствия.

```
input int limit = 1;         // Limit
input int startPause = 100; // Delay (ms)
```

Эмуляция задержки построена на функции *Sleep*.

```
void Delay()
{
   if(startPause > 0)
   {
      Sleep(startPause);
   }
}
```

Внутри функции *OnStart* первым делом декларируется временная глобальная переменная. Поскольку она предназначена для подсчета работающих копий программы, не имеет смысла делать её постоянной: при каждом запуске терминала нужно вести подсчет заново.

```
void OnStart()
{
   PRTF(GlobalVariableTemp(__FILE__));
   ...
```

На тот случай, что хитрый пользователь сделал одноименную переменную заранее и присвоил ей отрицательное значение, делаем защиту.

```
   int count = (int)GlobalVariableGet(__FILE__);
   if(count < 0)
   {
      Print("Negative count detected. Not allowed.");
      return;
   }
```

Далее начинается фрагмент с основным функционалом. Если счетчик уже больше или равен предельно допустимому количеству, прерываем запуск программы.

```
   if(count >= limit)
   {
      PrintFormat("Can't start more than %d copy(s)", limit);
      return;
   }
```

В противном случае увеличиваем счетчик на 1 и записываем в глобальную переменную. Предварительно эмулируем задержку, чтобы спровоцировать ситуацию, когда между чтением переменной и её записью в нашей программе могла вклиниться другая программа.

```
   Delay();
   PRTF(GlobalVariableSet(__FILE__, count + 1));
```

Если это действительно произойдет, наша копия программы будет делать приращение и присваивать уже устаревшую, некорректную величину. Получится, что в другой копии программы, выполняющейся параллельно с нашей, такое же значение *count* уже было обработано или будет обработано вторично.

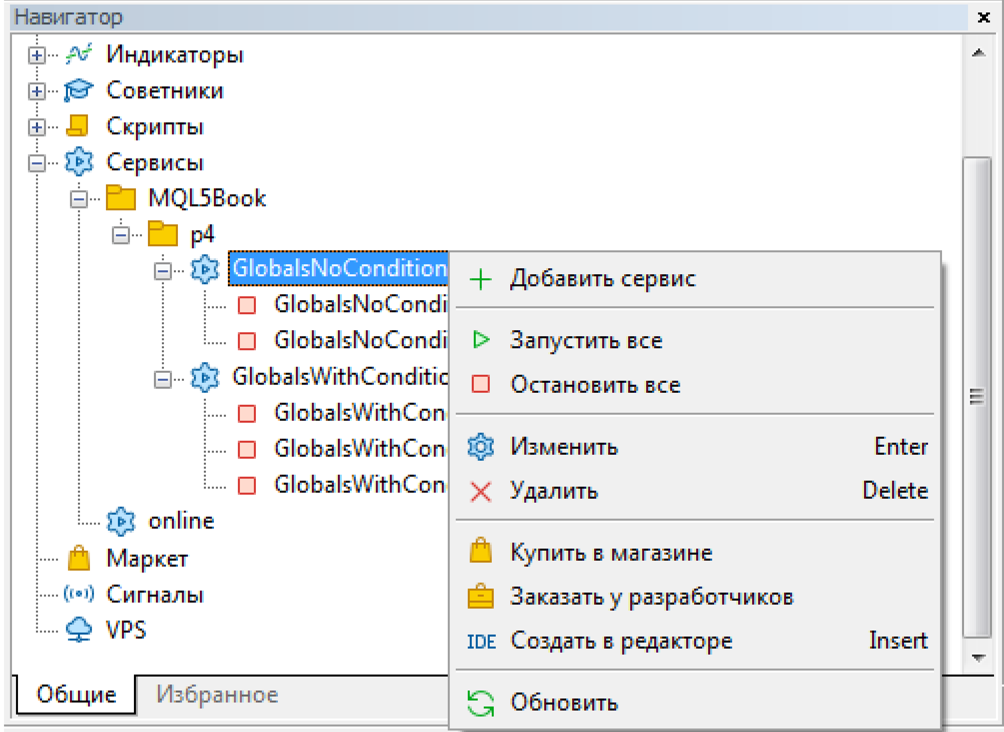
Полезную работу сервиса изображает следующий цикл.

```
     int loop = 0;
     while(!IsStopped())
     {
        PrintFormat("Copy %d is working [%d]...", count, loop++);
        // ...
        Sleep(3000);
     }
```

После того как пользователь остановит сервис (для этого в интерфейсе имеется контекстное меню — о нем чуть ниже), цикл завершится, и нам нужно уменьшить счетчик.

```
     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");
     }
  }
```

Откомпилированные сервисы попадают в соответствующую ветвь "Навигатора".



Сервисы в "Навигаторе" и их контекстное меню

По щелчку правой кнопки мыши откроем контекстное меню и создадим два экземпляра сервиса *GlobalsNoCondition.mq5*, вызвав дважды команду *Добавить сервис*. При этом каждый раз будет открываться диалог с настройками сервиса, где следует оставить параметрам значения по умолчанию.

Важно отметить, что команда *Добавить сервис* сразу же запускает созданный сервис. Но нам это не нужно. Поэтому следует сразу же после запуска каждой копии снова вызвать контекстное меню и выполнить команду *Остановить* (если выделен конкретный экземпляр) или *Остановить все* (если выделена программа, то есть вся группа порожденных экземпляров).

Первый экземпляр сервиса получит по умолчанию название, полностью совпадающее с файлом сервиса ("GlobalsNoCondition"), а во всех последующих будет автоматически добавляться увеличивающийся номер. В частности, второй экземпляр значится как "GlobalsNoCondition 1". Терминал позволяет переименовать экземпляры в произвольный текст с помощью команды *Переименовать*, но мы этого делать не будем.

Теперь все готово для эксперимента. Попробуем запустить два экземпляра одновременно. Для этого выполним команду *Запустить все* для соответствующей ветви *GlobalsNoCondition*.

Напомним, в параметрах было задано ограничение в 1 экземпляр. Однако, судя по журналам, оно не сработало.

```
GlobalsNoCondition    GlobalVariableTemp(GlobalsNoCondition.mq5)=true / ok
GlobalsNoCondition 1  GlobalVariableTemp(GlobalsNoCondition.mq5)=false / GLOBALVARIAB
GlobalsNoCondition    GlobalVariableSet(GlobalsNoCondition.mq5,count+1)=2021.08.31 17
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:
GlobalsNoCondition 1  Count underflow
```

Обе копии "думают", что имеют номер 0 (выводят "Copy 0" из рабочего цикла) и общее их число ошибочно равно 1, потому что именно это значение обе копии сохранили в переменную-счетчик.

Именно из-за этого при остановке сервисов (команда *Остановить все*) мы получили сообщение о некорректном состоянии ("Count underflow"): ведь каждая из копий пытается уменьшить счетчик на 1, и в результате та из них, что выполнилась второй, получила отрицательное значение.

Для решения проблемы необходимо использовать функцию *GlobalVariableSetOnCondition*. На основе исходного кода предыдущего сервиса была подготовлена усовершенствованная версия *GlobalsWithCondition.mq5*. В целом она повторяет логику работы предшественника, но есть и существенные отличия.

Вместо простого вызова *GlobalVariableSet* для увеличения счетчика пришлось написать более сложную конструкцию.

```
const int maxRetries = 5;
int retry = 0;

while(count < limit && retry < maxRetries)
{
   Delay();
   if(PRTF(GlobalVariableSetOnCondition(__FILE__, count + 1, count))) break;
   // условие не выполнено (count устарело), присваивание не состоялось,
   // попробуем еще раз с новым условием, если цикл не превысит лимит
   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;
}
...
```

Поскольку функция *GlobalVariableSetOnCondition* может не записать новое значение счетчика, если старое уже устарело, мы в цикле считываем глобальную переменную еще раз и повторяем попытки её инкрементировать, до тех пор, пока не превышено максимально допустимое значение счетчика. Также условием цикла ограничено количество попыток. Если цикл закончится с нарушением одного из условий, значит обновить счетчик не удалось, и программа не должна дальше выполняться.

Стратегии синхронизации

В принципе, для реализации захвата разделяемого ресурса существует несколько стандартных стратегий.

Первая заключается в "мягкой" проверке, свободен ли ресурс, и его последующем блокировании, только если он в тот момент свободен. Если же он занят, алгоритм планирует следующую попытку через некоторый период, а сам в это время занимается другими задачами (именно поэтому данный подход предпочтителен для программ, в которые "вшито" несколько сфер деятельности/ответственности). Аналогом данной схемы поведения в переложении для функции *GlobalVariableSetOnCondition* является одиночный вызов, без цикла, с выходом из текущего блока в случае неудачи. Изменение переменной откладывается "до лучших времен".

Вторая стратегия более настойчивая, применена в нашем скрипте. Это цикл, который повторяет запрос ресурса заданное количество раз или предопределенное время (допустимый период ожидания ресурса). Если цикл истекает, а положительный результат не достигнут (вызов функции *GlobalVariableSetOnCondition* так и не вернул true), программа также выходит из текущего блока и, вероятно, планирует повторить попытки через некоторое время.

Наконец, третья стратегия, самая жесткая, предполагает запрос ресурса "до победного конца". Её можно представить как бесконечный цикл с вызовом функции. Такой подход имеет смысл использовать в программах, которые ориентированы на одну конкретную задачу, и не

могут продолжать работу без захваченного ресурса. В MQL5 используйте для этого цикл *while(!IsStopped())* и не забывайте внутри вызывать *Sleep*.

Здесь важно отметить потенциальную проблему с "жестким" захватом нескольких ресурсов. Представьте, что MQL-программа модифицирует несколько глобальных переменных (что является, в принципе, обычной ситуацией). Если одна её копия захватит одну переменную, а вторая — другую, и обе будут ждать освобождения, наступит их взаимная блокировка (deadlock).

Исходя из вышеизложенного, совместный доступ к глобальным переменным и другим ресурсам (например, файлам), следует тщательно проектировать и анализировать на предмет блокировок и так называемых "гонок" (race condition), когда параллельное исполнение программ приводит к неопределенному результату (в зависимости от порядка их работы).

После завершения рабочего цикла в новой версии сервиса по аналогичному принципу изменен алгоритм уменьшения счетчика.

```
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);
}
```

Ради интереса создадим для нового сервиса уже три экземпляра. В настройках каждого из них укажем в параметре Limit: 2 экземпляра (чтобы провести тест в измененных условиях). Напомним, что создание каждого экземпляра сразу же его запускает, что нам не нужно, и потому каждую только что созданную копию следует остановить.

Экземпляры получат имена по умолчанию "GlobalsWithCondition", "GlobalsWithCondition 1" и "GlobalsWithCondition 2".

Когда все будет готово, запустим сразу все копии и получим примерно такие записи в журнале.

```
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
                        » 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,
GlobalsWithCondition    Stopped copy 0: count: 2, retries: 0
GlobalsWithCondition 1  Copy 1 (out of 1) is stopping
GlobalsWithCondition 1  GlobalVariableSetOnCondition(GlobalsWithCondition.mq5,last-1,
GlobalsWithCondition 1  Stopped copy 1: count: 1, retries: 0
```

Прежде всего, обратите внимание на случайную, но вместе с тем наглядную демонстрацию описываемого эффекта переключения контекста параллельного выполняющихся программ. Первым экземпляром, который создал временную переменную, был "GlobalsWithCondition" без номера: это видно по результату функции *GlobalVariableTemp*, равному *true*. Однако в журнале эта строчка занимает лишь третью позицию, а две предыдущих содержат результаты вызова той же функции в копиях под именами с номерами 1 и 2, и в них функция *GlobalVariableTemp* вернула *false*. Это означает, что в этих копиях проверка переменной выполнялась позднее, хотя затем их потоки обогнали поток "GlobalsWithCondition" без номера и оказались в журнале раньше.

Но вернемся к нашему основному алгоритму подсчета программ. Экземпляр "GlobalsWithCondition" первым преодолел проверку и запустился в работу под внутренним идентификатором "Copy 0" (из кода сервиса мы не можем узнать, как пользователь назвал экземпляр: такой функции в MQL5 API нет, по крайней мере пока).

В экземплярах под номерами 1 и 2 ("GlobalsWithCondition 1", "GlobalsWithCondition 2") благодаря функции *GlobalVariableSetOnCondition* был обнаружен факт модификации счетчика: при старте он был равен 0, но "GlobalsWithCondition" увеличил его на 1. Оба припозднившихся экземпляра вывели сообщение "Counter is already altered by other instance: 1". Одной из этих копий ("GlobalsWithCondition 1") удалось раньше номера 2 получить новое значение 1 из переменной и увеличить его до 2. Об этом говорит успешный вызов *GlobalVariableSetOnCondition* (он вернул *true*). И далее появилось сообщение о начале работы "Copy 1 is working".

Тот факт, что значение внутреннего счетчика совпало с внешним номером экземпляра, чисто случаен. Вполне могло быть, что "GlobalsWithCondition 2" запустился бы раньше "GlobalsWithCondition 1" (или в какой-то другой последовательности, учитывая, что копий целых три). Тогда внешняя и внутренняя нумерация отличались бы. Вы можете повторять эксперимент с запуском и остановкой всех сервисов много раз, и скорее всего, последовательность, в которой копии увеличивают переменную-счетчик, будет отличаться. Но в любом случае ограничение на общее количество отсечет один лишний экземпляр.

Когда последний экземпляр "GlobalsWithCondition 2" получает право доступа к глобальной переменной, там уже хранится значение 2. Так как это заданный нами предел, программа не запускается.

```
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
```

Далее копии "GlobalsWithCondition" и "GlobalsWithCondition 1" "крутятся" в рабочем цикле, пока сервисы не останавливаются.

Вы можете попробовать остановить лишь один экземпляр. Тогда появится возможность запустить другой, получивший ранее запрет на выполнение из-за превышения квоты.

Разумеется, предложенный вариант защиты от параллельной модификации эффективен только при координации поведения собственных программ, но не для ограничения на единственную копию демо-версии, поскольку пользователь может просто удалить глобальную переменную. Для этой цели глобальные переменные можно использовать по-другому — в привязке к идентификатору графика: MQL-программа работает только до тех пор, пока в созданной ею глобальной переменной находится идентификатор её графика. Другие способы контролировать разделяемые данные (счетчики и прочую информацию) предоставляют ресурсы и база данных.

4.6.7 Сброс глобальных переменных на диск

Для оптимизации быстродействия глобальные переменные постоянно находятся в памяти, пока работает терминал. Однако между сеансами переменные хранятся, как мы знаем, в специальном файле. Это касается всех глобальных переменных, кроме временных. Обычно запись переменных в файл происходит, когда терминал закрывается. Однако при внезапном сбое работы компьютера данные могут потеряться. Поэтому бывает полезно принудительно инициировать запись, чтобы гарантировать сохранность данных во всяких непредвиденных ситуациях. Для этой цели в MQL5 API имеется функция *GlobalVariablesFlush*.

void GlobalVariablesFlush()

Функция принудительно записывает содержимое глобальных переменных на диск. Функция не имеет параметров и ничего не возвращает.

Простейший пример приведен в скрипте *GlobalsFlush.mq5*.

```
void OnStart()
{
   GlobalVariablesFlush();
}
```

С помощью него вы можете сбрасывать переменные на диск в любой момент, по необходимости. В своем любимом файловом менеджере вы можете убедиться, что дата и время у файла *gvariables.dat* меняется сразу после запуска скрипта. Однако учтите, что файл обновится, только если глобальные переменные каким-либо образом редактировались или хотя бы считывались (это изменяет время доступа) после предыдущего сохранения.

Данный скрипт пригодится тем, кто держит терминал включенным длительное время, и в нем выполняются программы, модифицирующие глобальные переменные.

4.7 Функции для работы со временем

Время является фундаментальным фактором большинства процессов и играет важнейшую прикладную роль для трейдинга.

Как мы знаем, основная система координат в торговле построена на двух измерениях: цене и времени. На графиках они отображаются, как правильно, по вертикальной и горизонтальной оси, соответственно. Позднее мы затронем еще одну важную ось, которую можно представить перпендикулярной двум первым и уходящей как бы вглубь графика, — на ней отмечаются торговые объемы. Но пока сконцентрируемся на времени.

Это измерение является общим для всех графиков, использует одни и те же единицы измерения и, как бы это странно ни звучало, характеризуется постоянством (ход времени предсказуем).

Общее число встроенных средств, связанных с расчетом и анализом времени, в терминале очень велико. Поэтому знакомиться с ними мы будем постепенно, по мере продвижения по главам книги, от простого — к сложному.

В данной главе мы изучим функции, которые позволяют контролировать время и приостанавливать активность программы на заданный интервал.

В разделе Дата и время в главе про преобразование данных мы уже видели пару функций, связанных со временем: *TimeToStruct* и *StructToTime*. Они позволяют разделить значение типа *datetime* на составляющие или наоборот, сконструировать *datetime* из отдельных полей: напомним, что они сведены в структуру *MqlDateTime*.

```
struct MqlDateTime
{
   int year;        // год (1970 — 3000)
   int mon;         // месяц (1 — 12)
   int day;         // день (1 — 31)
   int hour;        // часы (0 — 23)
   int min;         // минуты (0 — 59)
   int sec;         // секунды (0 — 59)
   int day_of_week; // день недели, нумерация с 0 (воскресенье) по 6 (субботу)
                    //    согласно перечислению ENUM_DAY_OF_WEEK
   int day_of_year; // порядковый номер дня в году, начиная с 0 (1 января)
};
```

Но откуда MQL-программа может получить само значение *datetime*?

Например, исторические цены и время отражены в котировках, а текущие "живые" данные поступают с тиками. И то, и другое обладает метками времени, которые мы научимся получать в соответствующих разделах: про таймсерии и события терминала. Однако MQL-программа имеет возможность запросить текущее время само по себе (без цен или другой торговой информации) с помощью нескольких функций.

Несколько функций потребовалось потому, что система является распределенной: она состоит из клиентского терминала и сервера брокера, находящихся в произвольных частях мира, которые, вполне вероятно, относятся к разным часовым поясам.

Любой часовой пояс характеризуется временным смещением относительно общемировой точки отсчета времени — нулевого гринвичского меридиана (Greenwich Mean Time, GMT). Как правило, смещение часового пояса составляет целое число часов N (хотя есть и экзотические зоны с получасовым шагом) и потому обозначается как GMT+N или GMT-N, в зависимости от того, восточнее или западнее от меридиана находится пояс. Например, расположенная восточнее от Лондона континентальная Европа использует центральноевропейское время (Central European Time, CET), равное GMT+1, или восточноевропейское время (Eastern European Time, EET), равное GMT+2, а в Америке расположены "отрицательные" зоны, такие как Eastern Standard Time (EST) или GMT-5.

> Следует отметить, что GMT соответствует астрономическому (солнечному) времени, которое обладает едва заметной нелинейностью, поскольку вращение Земли постепенно замедляется. В связи с этим в последние десятилетия фактически произошел переход к более точной системе учета времени (на основе атомных часов), в которой общемировое время называется всемирным координированным временем (Coordinated Universal Time, UTC). Во многих прикладных областях, включая и трейдинг, различие между GMT и UTC несущественно, поэтому обозначения часовых поясов в новом формате UTC±N и старом GMT±N следует считать аналогами. Например, многие брокеры уже указывают в спецификациях времена сессий через UTC, в то время как в MQL5 API исторически используется обозначение GMT.

MQL5 API позволяет узнать текущее время терминала (фактически, локальное время компьютера) и время сервера: их возвращают, соответственно, функции *TimeLocal* и *TimeCurrent*. Кроме того, MQL-программа может получить текущее время GMT (функция *TimeGMT*), основываясь на настройках часового пояса Windows. Таким образом, у трейдера и программиста появляется привязка локального времени к общемировому, а по разнице между локальным и серверным временем можно определить "таймзону" сервера и котировок. Но здесь есть пара нюансов.

Во-первых, во многих странах мира существует практика перехода на "летнее" время из соображений более эффективного использования дневного света (Daylight Savings Time , DST).

Обычно это означает добавление 1 часа к стандартному (зимнему) времени примерно с марта/апреля по октябрь/ноябрь (в северном полушарии, в южном — наоборот). При этом время GMT/UTC всегда остается постоянным, то есть не подвергается поправке DST, и потому потенциально возможны различные варианты схождения/расхождения клиентского и серверного времени:

- в разных странах даты перехода могут отличаться;
- некоторые страны не осуществляют переход на летнее время;

В связи с этим, некоторым MQL-программам требуется отслеживать подобные моменты перестройки часовых поясов, если алгоритмы основаны на привязке к внутридневному времени (например, к выходу новостей), а не к движениям цен или концентрации объемов.

И если перевод времени на компьютере пользователя определить довольно легко, благодаря функции *TimeDaylightSavings*, то для серверного времени готового аналога нет.

Во-вторых, штатный тестер MetaTrader 5, в котором мы можем отлаживать или оценивать MQL-программы таких типов как эксперты и индикаторы, к сожалению, не эмулирует время торгового сервера. Вместо этого все три вышеуказанные функции *TimeLocal*, *TimeGMT*, *TimeCurrent* вернут одно и то же время, то есть часовой пояс всегда виртуально равен GMT.

Абсолютное и относительное время

Учет времени в алгоритмах, как и в жизни, может вестись в абсолютных или относительных координатах. Каждый момент в прошлом, в настоящем и в будущем описывается абсолютным значением, на которое мы можем сослаться, чтобы указать начало отчетного периода или время выхода экономических новостей. Именно это время мы храним в MQL5 с помощью типа *datetime*. Вместе с тем, зачастую требуется заглянуть в будущее или отступить в прошлое на заданное количество единиц времени от текущего момента. При этом нас интересует не абсолютное значение, а временной интервал.

В частности, в алгоритмах существует понятие таймаута — периода времени, за который должно выполниться определенное действие, и если оно не выполнилось по любым причинам, мы его отменяем, перестаем ждать результата (потому что, видимо, что-то пошло не так). Измерять интервал можно в разных единицах: часах, секундах, миллисекундах или даже микросекундах (ведь, компьютеры нынче быстры).

В MQL5 часть функций, связанных со временем, работает с абсолютными значениями (например, *TimeLocal*, *TimeCurrent*), а часть с интервалами (например, *GetTickCount*, *GetMicrosecondCount*).

Однако измерение интервалов или активация программы через заданные промежутки времени могут осуществляться не только с помощью функций из данного раздела, но и встроенных таймеров, работающих по известному принципу будильника. Будучи взведенными, они используют для уведомления MQL-программ специальные события и реализуемые нами функции обработки этих событий — *OnTimer* (они похожи на *OnStart*). Этим аспектом управления временем мы займемся в отдельном разделе, после изучения общей концепции событий в MQL5 (см. раздел Обзор функций обработки событий).

4.7.1 Время локальное и серверное

На платформе MetaTrader 5 всегда существует два типа времени: локальное (клиентское) и серверное (брокера).

Локальное время соответствует времени компьютера, на котором запущен терминал, и увеличивается непрерывно, с равной скоростью, как и в реальном мире.

Серверное время течет иначе. Основу для него задает время на компьютере у брокера, однако к клиенту информация о нем поступает только вместе с очередными изменениями цен, которые упаковываются в специальные структуры, называемые тиками (см. раздел про *MqlTick*), и передаются MQL-программам с помощью событий.

Таким образом, обновленное серверное время становится известно в терминале только в результате изменения цены хотя бы одного финансового инструмента на рынке, то есть из числа тех, что выбраны в окне "Обзор рынка". Последнее известное время сервера отображается в заголовке этого окна. Если тиков нет, серверное время в терминале стоит на месте. Это особенно заметно в выходные и праздничные дни, когда все биржи и площадки Forex закрыты.

В частности, в воскресенье серверное время будет, скорее всего, отображаться как вечер пятницы. Исключение составят лишь те экземпляры MetaTrader 5, в которых доступны непрерывно торгуемые инструменты, такие как криптовалюты. Однако и в этом случае, в периоды низкой волатильности серверное время может заметно отставать от локального.

Все функции данного раздела оперируют временем с точностью до секунды (точность представления времени в типе *datetime*).

Для получения локального и серверного времени MQL5 API предоставляет 3 функции: *TimeLocal*, *TimeCurrent* и *TimeTradeServer*. Все три функции имеют по два варианта прототипа: первый возвращает время как значение типа *datetime*, второй дополнительно принимает по ссылке и заполняет компонентами времени структуру *MqlDateTime*.

datetime TimeLocal()
datetime TimeLocal(MqlDateTime &dt)

Функция возвращает локальное компьютерное время в формате *datetime*.

Важно отметить, что время включает в себя поправку на летнее время, если она активирована. То есть *TimeLocal* равна стандартному времени часового пояса компьютера за вычетом поправки *TimeDaylightSavings*. Условно формулу можно представить так:

```
TimeLocalsummer() = TimeLocalwinter() - TimeDaylightSavings()
```

Здесь *TimeDaylightSavings* равно обычно -3600, то есть переводу часов на 1 час вперед (1 час пропадает). Таким образом, летнее значение *TimeLocal* больше зимнего (при равенстве астрономического времени суток) относительно UTC. Например, если зимой *TimeLocal* равен UTC+2, то летом UTC+3. Универсальное время UTC можно получить с помощью функции *TimeGMT*.

datetime TimeCurrent()
datetime TimeCurrent(MqlDateTime &dt)

Функция возвращает последнее известное время сервера в формате *datetime*. Это время прихода последней котировки из списка всех финансовых инструментов, подключенных в "Обзоре рынка". Единственное исключение: в обработчике события *OnTick* в экспертах данная функция

вернет время обрабатываемого тика (даже если в обзоре рынка уже случились тики с более свежим временем).

Также отметим, что время на горизонтальной оси всех графиков в MetaTrader 5 соответствует времени сервера (на истории). Последний (текущий, самый правый) бар содержит в себе время *TimeCurrent*. Подробности см. в разделе Графики.

datetime TimeTradeServer()
datetime TimeTradeServer(MqlDateTime &dt)

Функция возвращает расчетное текущее время торгового сервера. В отличие от функции *TimeCurrent*, результаты которой могут не меняться в отсутствии новых котировок, *TimeTradeServer* позволяет получить оценку непрерывно увеличивающегося времени сервера. В основу расчета берется последняя известная разница "таймзон" клиента и сервера, которая прибавляется к текущему локальному времени.

В тестере значение *TimeTradeServer* всегда равно *TimeCurrent*.

Пример работы функций приведен в скрипте *TimeCheck.mq5*.

В главной функции организован бесконечный цикл, который выводит в журнал все типы времени каждую секунду, пока пользователь не остановит скрипт.

```
void OnStart()
{
   while(!IsStopped())
   {
      PRTF(TimeLocal());
      PRTF(TimeCurrent());
      PRTF(TimeTradeServer());
      PRTF(TimeTradeServerExact());
      Sleep(1000);
   }
}
```

Помимо стандартных функций здесь применена пользовательская функция *TimeTradeServerExact*.

```
datetime TimeTradeServerExact()
{
   enum LOCATION
   {
      LOCAL,
      SERVER,
   };
   static datetime now[2] = {}, then[2] = {};
   static int shiftInHours = 0;
   static long shiftInSeconds = 0;

   // постоянно засекаем 2 последних метки времени там и тут
   then[LOCAL] = now[LOCAL];
   then[SERVER] = now[SERVER];
   now[LOCAL] = TimeLocal();
   now[SERVER] = TimeCurrent();

   // при первом вызове еще не имеем 2-х меток,
   // необходимых для расчета стабильной разницы
   if(then[LOCAL] == 0 && then[SERVER] == 0) return 0;

   // когда ход времени одинаков на клиенте и на сервере,
   // и сервер не "заморожен" из-за выходных/праздников,
   // обновляем разницу
   if(now[LOCAL] - now[SERVER] == then[LOCAL] - then[SERVER]
   && now[SERVER] != then[SERVER])
   {
      shiftInSeconds = now[LOCAL] - now[SERVER];
      shiftInHours = (int)MathRound(shiftInSeconds / 3600.0);
      // отладочная печать
      PrintFormat("Shift update: hours: %d; seconds: %lld", shiftInHours, shiftInSecc
   }

   // NB: встроенная функция TimeTradeServer вычисляет так:
   //                TimeLocal() - shiftInHours * 3600
   return (datetime)(TimeLocal() - shiftInSeconds);
}
```

Она потребовалась потому, что алгоритм встроенной функции *TimeTradeServer* может устроить не всех. Встроенная функция находит разницу между локальным и серверным временем в часах (то есть разницу "таймзон"), и затем получает серверное время как коррекцию локального времени на эту разницу. В результате, если минуты и секунды тикают на клиенте и сервере не совсем синхронно (что весьма вероятно), стандартная аппроксимация серверного времени покажет минуты и секунды клиента, а не сервера.

В идеале "бортовые" часы всех компьютеров должны быть синхронизированы с глобальным временем, но на практике случаются отклонения, и достаточно небольшого сдвига на одной из сторон, как *TimeTradeServer* перестанет максимально точно повторять время на сервере.

В нашей реализации этой же функции на MQL5 мы не округляем разницу между временем клиента и сервера до часовых "таймзон". Вместо этого в расчете используется точная разница в

секундах. Поэтому *TimeTradeServerExact* возвращает время, в котором минуты и секунды тикают так же, как на сервере.

Вот пример журнала, генерируемого скриптом.

```
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
```

Видно, что часовые пояса клиента и сервера совпадают, но есть рассинхронизация в несколько минут (для наглядности). При первом вызове *TimeTradeServerExact* вернула 0. Далее данные для расчета разницы уже поступят, и мы увидим все 4 "типа" времени, размеренно "шагающие" с секундным интервалом.

```
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 Переход на летнее время (локальное)

Для определения факта перевода локальных часов на летнее время в MQL5 имеется функция *TimeDaylightSavings*. Она берет настройки из вашей операционной системы.

Определить режим летнего времени на сервере так же легко не получится. Для этого потребуется реализовать на MQL5 анализ котировок, событий экономического календаря или времени ролловера/начисления свопов в торговой истории счета. В примере далее мы покажем один из вариантов.

int TimeDaylightSavings()

Функция возвращает поправку в секундах, если осуществлен переход на летнее время. Зимнее время является стандартным для каждого часового пояса, так что поправка в этот период равна нулю. В условном виде формулу получения поправки можно записать следующим образом:

```
TimeDaylightSavings() = TimeLocalwinter() - TimeLocalsummer()
```

Например, если стандартный часовой пояс (*winter*) равен UTC+3 (то есть время пояса на 3 часа больше, чем UTC), то при переходе на летнее время (*summer*) добавляется 1 час и получается UTC+4. При этом *TimeDaylightSavings* вернет -3600.

Пример применения функции приведен в скрипте *TimeSummer.mq5*, где также предлагается один из возможных эмпирических способов выявления соответствующего режима на сервере.

```
void OnStart()
{
   PRTF(TimeLocal());          // локальное время терминала
   PRTF(TimeCurrent());        // последнее известное время сервера
   PRTF(TimeTradeServer());    // расчетное время сервера
   PRTF(TimeGMT());            // время GMT (расчет от локального через сдвиг таймзон
   PRTF(TimeGMTOffset());      // сдвиг таймзоны от GMT в секундах
   PRTF(TimeDaylightSavings());// поправка на летнее время в секундах
   ...
```

Для начала выведем все типы времени и его коррекции, предоставляемые MQL5 (функции *TimeGMT* и *TimeGMTOffset* будут представлены в следующем разделе про Универсальное время, но их суть должна уже быть в целом ясна из предыдущего описания).

Предполагается, что скрипт запускается в торговые дни. Записи в журнале у вас будут соответствовать настройкам вашего компьютера и сервера брокера.

```
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
```

В данном случае часовой пояс клиента отстоит от GMT на 3 часа (UTC+3), поправки на летнее время нет.

Теперь займемся сервером. По значению функции *TimeCurrent* мы можем определить текущее время сервера, но не его стандартный часовой пояс, поскольку в этом времени может участвовать переход на летнее время (а используется ли оно в принципе, и не включено ли в данный момент, — информации в MQL5 нет).

Для определения реальной "таймзоны" сервера и режима летнего времени воспользуемся тем фактом, что перевод времени сервера сказывается на котировках. Как большинство эмпирических способов решения задач, этот может давать не совсем верные результаты в тех или иных обстоятельствах. Если сверка с другими источниками покажет разночтения, следует подобрать иной способ.

Открытие рынка Форекс происходит в воскресенье вечером в 22 часа по всемирному времени (это соответствует началу утренних торгов в азиатско-тихоокеанском регионе), а закрытие — в пятницу в 22 часа (закрытие торгов в Америке). Это значит, что на серверах в зоне UTC+2 (Восточная Европа), первые бары появятся ровно в 0 часов 0 минут в понедельник. По центральноевропейскому времени, которое соответствует UTC+1, начало торговой недели происходит в 23 часа воскресного вечера.

Подсчитав статистику внутрисуточного смещения первого бара H1 после каждого перерыва на выходные дни, мы получим оценку "таймзоны" сервера. Разумеется, для этого лучше использовать наиболее ликвидный инструмент Forex, то есть EURUSD.

Если в данной статистике за годовой период обнаружится два максимума внутрисуточного смещения и они расположены рядом друг с другом, это будет означать, что брокер производит переход на летнее время и обратно.

Обратите внимание, что периоды летнего и зимнего времени не равны. Так, при переходе на летнее время в начале марта и возврате на зимнее в начале ноября получим примерно 8 месяцев летнего времени. Это будет сказываться на соотношении максимумов в статистике.

Имея две "таймзоны", легко определить, какая из них активна в данный момент и, тем самым, выяснить текущее наличие или отсутствие поправки на летнее время.

При переводе часов на летнее время у брокера с UTC+2 таймзона поменяется на UTC+3, из-за чего начало недели сместится с 22:00 на 21:00. Это скажется на структуре баров H1: чисто визуально на графике мы увидим три бара вечером воскресенья вместо двух.

Перевод часов с зимнего (UTC+2) на летнее (UTC+3) время на графике EURUSD H1

Для воплощения задуманного оформлена отдельная функция *ServerTimeZone*. За получение котировок (а точнее, меток времени баров) в ней отвечает вызов встроенной функции *CopyTime* (мы изучим её в разделе про доступ к таймсериям).

```
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)) >
  {
     // здесь получим примерно 6000 баров в массиве
     const int n = ArraySize(array);
     PrintFormat("Got %d H1 bars, ~%d days", n, n / 24);
     // (-V-) цикл по барам H1
     ...
  }
}
```

В качестве параметров в *CopyTime* передается рабочий инструмент, таймфрейм H1, и диапазон дат за последний год. Значение NULL вместо инструмента означает символ текущего графика, на который будет помещен скрипт, поэтому рекомендуется выбрать окно с EURUSD. Константа PERIOD_H1 соответствует H1, как нетрудно догадаться. Функция *TimeCurrent* нам уже знакома: она вернет текущее, самое последнее известное время сервера. А если вычесть из него количество секунд в году, которое положено в переменную *year*, мы получим дату и время ровно год назад. Результаты поступят в массив *array*.

Для подсчета статистики, сколько раз неделя открывалась баром на конкретном часе, зарезервируем массив *hours[24]*. Сам расчет будем выполнять в цикле по полученному массиву *array*, то есть по барам из прошлого в настоящее. На каждой итерации открывающий час просматриваемой недели будем хранить в переменной *current*. Когда цикл закончится, в *current* останется актуальный часовой пояс сервера, поскольку последней обработается текущая неделя.

```
     // (-v-) цикл по барам H1
     int hours[24] = {};
     int current = 0;
     for(int i = 0; i < n; ++i)
     {
        // (-V-) обработка i-го бара H1
        ...
     }

     Print("Week opening hours stats:");
     ArrayPrint(hours);
```

Внутри цикла по дням воспользуемся классом *DateTime* из заголовочного файла *MQL5Book/DateTime.mqh* (см. раздел Дата и время).

```
      // (-v-) обработка i-го бара H1
      // находим день недели бара
      const ENUM_DAY_OF_WEEK weekday = TimeDayOfWeek(array[i]);
      // пропускаем все дни кроме воскресенья и понедельника
      if(weekday > MONDAY) continue;
      // анализируем первый бар H1 очередной торговой недели
      // находим час первого бара после выходных
      current = _TimeHour();
      // подсчитываем статистику часов открытия
      hours[current]++;

      // пропускаем 2 следующих дня
      // (т.к. статистика начала этой недели уже обновлена)
      i += 48;
```

Предложенный алгоритм не является оптимальным, но зато не требует вникания в технические детали организации таймсерий, которые нам еще не известны.

Некоторые недели бывают неформатными (начинаются после праздников). Если такая ситуация случится на последней неделе, переменная *current* будет содержать необычное смещение. Это можно проверить по статистике: для полученного часа окажется очень малое количество зафиксированных "открытий" недели. В тестовом скрипте в таком случае просто выводится сообщение в журнал. На практике следует уточнить стандартное открытие по предыдущим одной-двум неделям.

```
   // (-V-) цикл по барам H1
   ...
   if(hours[current] <= 52 / 4)
   {
      // TODO: уточнить по предыдущим неделям
      Print("Extraordinary week detected");
   }
```

Если брокер не переходит на летнее время, в статистике получится один максимум, в который попадут все или почти все недели. Если брокер практикует смену поясов, в статистике будет два максимума.

```
   // находим самый частый временной сдвиг
   int max = ArrayMaximum(hours);
   // затем проверяем, нет ли еще одного регулярного сдвига
   hours[max] = 0;
   int sub = ArrayMaximum(hours);
```

Нам нужно определить, насколько значителен второй экстремум (т.е. отличен от случайных праздников, которые могли сместить начало недели). Для этого оцениваем статистику относительно четверти года (52 недели / 4). Если этот предел превышен, брокером поддерживается режим летнего времени.

```
      int DST = 0;
      if(hours[sub] > 52 / 4)
      {
         // DST поддерживается в принципе
         if(current == max || current == sub)
         {
            if(current == MathMin(max, sub))
               DST = fabs(max - sub); // DST включен сейчас
         }
      }
```

Если смещение открытия текущей недели (в переменной current) совпадает с одним из двух основных экстремумов, значит, текущая неделя открылась штатно, и по ней можно делать вывод о "таймзоне" (это защитное условие необходимо, потому что выше у нас нет корректировки для нестандартной недели, а вместо этого только выдается предупреждение).

Теперь все готово, чтобы сформировать ответ нашей функции: часовой пояс сервера и признак включенного режима летнего времени.

```
      current += 2 + DST; // +2 для получения смещения от UTC
      current %= 24;
      // таймзоны всегда в диапазоне [UTC-12,UTC+12]
      if(current > 12) current = current - 24;
```

Поскольку мы имеем две характеристики для возврата из функции (*current* и *DST*), и помимо этого можем сообщить вызываемому коду, использует ли брокер летнее время в принципе (даже если сейчас зима), имеет смысл объявить специальную структуру *ServerTime* со всеми необходимыми полями.

```
struct ServerTime
{
   int offsetGMT;      // таймзона в секундах относительно UTC/GMT
   int offsetDST;      // DST коррекция в секундах (включена в offsetGMT)
   bool supportDST;    // DST коррекция обнаружена в принципе в котировках
   string description; // описание результата
};
```

Тогда в функции *ServerTimeZone* мы можем заполнить и вернуть такую структуру в качестве результата работы.

```
      ServerTime st = {};
      st.description = StringFormat("Server time offset: UTC%+d, including DST%+d", cu
      st.offsetGMT = -current * 3600;
      st.offsetDST = -DST * 3600;
      return st;
```

Если по какой-либо причине функция не сможет получить котировки, вернем пустую структуру.

```
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;
}
```

Проверим новую функцию в действии, для чего в *OnStart* добавим следующие инструкции:

```
    ...
    ServerTime st = ServerTimeZone();
    Print(st.description);
    Print("ServerGMTOffset: ", st.offsetGMT);
    Print("ServerTimeDaylightSavings: ", st.offsetDST);
}
```

Посмотрим на возможные результаты.

```
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
```

Согласно собранной статистике баров H1, неделя у данного брокера открывается строго в 00:00 понедельника. Таким образом, реальная "таймзона" равна UTC+2, а поправка на летнее время отсутствует, то есть время сервера должно совпадать с EET (UTC+2). Однако на практике, как мы видели в первой части лога, время на сервере отличается от GMT на 3 часа.

Здесь можно предположить, что нам встретился сервер, круглогодично работающий в летнем времени. В таком случае функция *ServerTimeZone* не сумеет отличить корректировку от дополнительного часа в "таймзоне": в результате, режим DST будет равен нулю, а время GMT, рассчитанное по котировкам сервера, сместится вправо на час от реального. Или наше исходное предположение о том, что котировки начинают поступать в 22:00 воскресенья, не соответствуют режиму работы данного сервера. Такие моменты следует выяснять у службы поддержки брокера.

4.7.3 Универсальное время

MQL5 позволяет узнать глобальное время GMT (UTC), основываясь на локальном времени компьютера и его часовом поясе.

datetime TimeGMT()
datetime TimeGMT(MqlDateTime &dt)

Функция возвращает время GMT в формате *datetime*, отсчитывая его от локального времени компьютера, с учетом перехода на зимнее или летнее время.

Обобщенная формула расчета:

```
TimeGMT() = TimeLocal() + TimeGMTOffset()
```

Таким образом, точность представления всемирного времени зависит от правильности настройки часов локального компьютера. В идеале полученное значение должно совпадать со значением, известным на сервере.

Для торговых стратегий, основанных на внешних экономических новостях, проще всего использовать календари в "таймзоне" GMT: тогда приближающиеся события можно отслеживать по *TimeGMT*. Чтобы сделать привязку события к времени сервера на графике, следует скорректировать событие на разницу между серверным часовым поясом и GMT (*TimeTradeServer() - TimeGMT()*). Но напомним, что в MQL5 имеется свой собственный, встроенный календарь.

int TimeGMTOffset()

Функция возвращает текущую разницу между временем GMT и локальным временем компьютера в секундах, исходя из настройки "таймзоны" в Windows, с учетом действующего перехода на зимнее или летнее время. В большинстве случаев часовой пояс задан целым числом часов относительно GMT, поэтому *TimeGMTOffset* равно часовому поясу, умноженному на -3600 (перевод в секунды). Например, зимой "таймзона" может быть равна UTC+2, что дает смещение -7200, а летом — UTC+3, что дает -10800. Минус нужен, потому что "положительные" "таймзоны" при пересчете их времени в GMT требуют вычитания вышеозначенного количества секунд, а "отрицательные" — прибавления.

Скрипт с использованием *TimeGMT* и *TimeGMTOffset* был приведен в предыдущем разделе.

4.7.4 Приостановка выполнения программы

Как мы видели ранее в примерах, программам иногда требуется повторять некоторые действия периодически — по простому расписанию или в случае неудачных предыдущих попыток. Когда это делается в цикле, рекомендуется регулярно приостанавливать программу, чтобы предотвратить слишком частые запросы и лишнюю нагрузку на CPU, а также чтобы дать время внешним "игрокам" выполнить свою работу (например, если мы ожидаем данные из другой программы, подгрузку истории котировок и т.д.).

Для этой цели в MQL5 имеется функция *Sleep*. В данном разделе приводится её формальное описание, а пример будет приведен в следующем разделе, вместе с функциями измерения временных интервалов.

void Sleep(int milliseconds)

Функция приостанавливает выполнение MQL-программы на заданное количество миллисекунд. После их истечения продолжат выполняться следующие инструкции, расположенные непосредственно после вызова *Sleep*.

Функцию имеет смысл использовать в первую очередь в скриптах и сервисах, потому что данные типы программ не имеют иных способов ожидания.

В экспертах и индикаторах рекомендуется пользоваться таймерами и событием *OnTimer*. В этой схеме MQL-программа возвращает управление терминалу и будет вызвана через заданный интервал.

Более того, функцию *Sleep* нельзя вызывать из индикаторов, поскольку они выполняются в интерфейсных потоках терминала, приостановка которых скажется на отрисовке графиков.

Если пользователь прервет работу MQL-программы из интерфейса терминала в то время, как она ожидает завершения вызова *Sleep*, выход из функции происходит немедленно (в пределах 100мс), то есть пауза заканчивается досрочно. При этом будет взведен флаг остановки *_StopFlag* (доступный также через функцию *IsStopped*), и программа должна максимально быстро и корректно прекратить исполнение.

4.7.5 Счетчики временных интервалов

Чтобы засечь интервал времени с точностью до секунды, достаточно взять разницу между двумя значениями *datetime*, полученными с помощью *TimeLocal*. Однако иногда нам требуется более высокая точность. Для этой цели MQL5 позволяет получать системные счетчики миллисекунд (*GetTickCount*, *GetTickCount64*) или микросекунд (*GetMicrosecondCount*).

uint GetTickCount()
ulong GetTickCount64()

Функции возвращают количество миллисекунд, прошедших с момента загрузки операционной системы. Точность отсчета времени ограничена стандартным системным таймером (~10-15 миллисекунд). Для более точного измерения интервалов используйте функцию *GetMicrosecondCount*.

В случае функции *GetTickCount* тип возвращаемого значения *uint* предопределяет период времени, через который счетчик будет переполняться: примерно 49.7 суток. Иными словами, отсчет снова пойдет с 0, если компьютер не выключался столь длительное время.

В отличие от неё, функция *GetTickCount64* возвращает значения *ulong*, и этот счетчик не переполняется в обозримом будущем (584'942'417 лет).

ulong GetMicrosecondCount()

Функция возвращает количество микросекунд, прошедших со старта MQL-программы.

Примеры использования функций счетчиков и *Sleep* сведены в скрипт *TimeCount.mq5*.

```
void OnStart()
{
   const uint startMs = GetTickCount();
   const ulong startMcs =  GetMicrosecondCount();

   // цикл на 5 секунд
   while(PRTF(GetTickCount()) < startMs + 5000)
   {
      PRTF(GetMicrosecondCount());
      Sleep(1000);
   }

   PRTF(GetTickCount() - startMs);
   PRTF(GetMicrosecondCount() - startMcs);
}
```

Вот как могут выглядеть результаты скрипта в журнале.

```
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 Взаимодействие с пользователем

Связь программы с "внешним миром" всегда двунаправленная, и средства для её организации можно условно разделить на категории для ввода и выводы данных. В классическом варианте пользователь предоставляет программе некие настройки и получает из неё результат. Если программа интегрируется с каким-либо внешним приложением или сервисом, ввод и вывод, как правило, осуществляются по специальным протоколам обмена (через файлы, сеть, разделяемую память и т.д.), минуя пользовательский интерфейс.

Среда исполнения MQL-программ позволяет организовать взаимодействие с пользователем MetaTrader 5 множеством способов.

В этой главе мы познакомимся с самыми простыми из них, позволяющими выводить сообщения в журнал или на график, показывать простое диалоговое окно и выдавать звуковые оповещения.

Напомним, что стандартом для ввода данных в MQL-программу являются входные переменные. Однако их можно задать только при инициализации программы. Изменение свойств программы через диалог настроек означает её "перезапуск" с новыми значениями (о некоторых нюансах, связанных здесь с типом MQL-программы, из-за чего слово *перезапуск* взято в кавычки, мы поговорим позднее).

Более гибкое интерактивное взаимодействие предполагает возможность управлять поведением программы без её остановки. В элементарных случаях для этого подойдет, например, диалоговое окно MessageBox, которое мы рассмотрим ниже, но для большинства практических применений этого недостаточно.

Поэтому в следующих частях книги мы существенно расширим перечень средств для реализации пользовательского интерфейса и научимся создавать интерактивные программы на основе интерфейсных объектов, отображать графическую информацию в индикаторах или ресурсах, отсылать push-уведомления на мобильные устройства пользователя и многому другому.

4.8.1 Вывод сообщений в журнал

Вывод информации в журнал является наиболее общеупотребительным способом сообщить пользователю текущую информацию о работе программы. Это может быть статус штатного завершения, индикация прогресса при длительном расчете или отладочные данные для поиска и воспроизведения ошибок.

К сожалению, ни один программист не застрахован от ошибок в своем коде и потому, как правило, в любой программе разработчики пытаются оставлять так называемый "след из хлебных крошек": фиксацию в логе основных этапов выполнения программы (как минимум, последовательность вызовов функций).

Мы уже знакомы с двумя функциями для вывода в журнал — *Print* и *PrintFormat* — они активно использовались в примерах предыдущих разделов. Нам пришлось в упрощенном режиме "ввести их в обиход" раньше времени, поскольку обойтись без них практически невозможно.

Один вызов функции порождает, как правило, одну запись. Однако если в выводимой строке встречается символ перевода строки ('\n'), он поделит информацию на две части.

Напомним, что все вызовы *Print* и *PrintFormat* трансформируются в записи в журнале на вкладке *Эксперты* окна *Инструменты*. Хотя вкладка называется *Эксперты*, в ней собираются результаты всех инструкций "печати", вне зависимости от типа MQL-программы.

Журналы хранятся в файлах, организованных по принципу "одни сутки — один файл": они имеют имена вида YYYYMMDD.log (Y — год, M — месяц, D — день). Располагаются файлы в папке *<каталог данных>/MQL5/Logs* (не следует путать их с системными журналами терминала в папке *<каталог данных>/Logs*).

> Учтите, что при массированном выводе в журнал (если вызовы функции *Print* генерируют большой объем информации за короткий промежуток времени) терминал отображает в окне лишь некоторые записи. Это делается для оптимизации быстродействия. Кроме того, пользователь в любом случае не способен увидеть все сообщения на лету. Для того чтобы увидеть полную версию журнала, необходимо выполнить команду *Просмотр* контекстного меню. В результате откроется окно с логом.
>
> Также следует помнить, что информация из журнала кэшируется при записи на диск, то есть пишется в файлы большими блоками в отложенном режиме, из-за чего в каждый момент времени файл с журналом, как правило, не содержит самых последних записей (хотя они видны в окне). Чтобы инициировать сброс кэша на диск, можно выполнить команду *Просмотр* или *Открыть* в контекстном меню журнала.

Каждая запись лога предваряется временем с точностью до миллисекунды, а также названием программы (и её графика), породившей или вызвавшей это сообщение.

void Print(argument, ...)

Функция печатает одно или больше значений в журнал экспертов, в одну строку (если в выводимых данных не встретится символ '\n').

Аргументы могут иметь любой встроенный тип и разделяются запятыми. Количество параметров не может превышать 64. Их переменное число обозначено в прототипе многоточием, но MQL5 не позволяет описывать свои собственные функции с подобной характеристикой: переменное количество параметров имеют лишь некоторые встроенные функции API (в частности, *StringFormat*, *Print*, *PrintFormat* и *Comment*).

Для структур и классов следует реализовать встроенный метод печати или выводить их поля по отдельности.

Также функция не способна обрабатывать массивы. Их можно выводить поэлементно или воспользоваться функцией *ArrayPrint*.

Значения типа *double* выводятся функцией с точностью до 16 значащих цифр (совокупно в мантиссе и дробной части). Число может выводиться либо в традиционном, либо в научном формате (с показателем степени), в зависимости от того, какая запись более компактна. Значения типа *float* выводятся с точностью до 7 десятичных разрядов. Для вывода вещественных чисел с другой точностью или для явного указания формата необходимо использовать функцию *PrintFormat*.

Значения типа *bool* выводятся в виде строк "true" или "false".

Даты выводятся с указанием дня и времени с максимальной точностью (до секунды), в формате "YYYY.MM.DD hh:mm:ss". Для вывода даты в другом формате необходимо использовать функцию *TimeToString* (см. раздел Дата и время).

Значения перечислений выводятся как целые числа. Для отображения названий элементов используйте функцию *EnumToString* (см. раздел Перечисления).

Однобайтовые и двухбайтовые символы также выводятся как целые числа. Для отображения символов в виде знаков или букв используйте функции *CharToString* или *ShortToString* (см. раздел Работа с символами и кодовыми страницами).

Значения типа *color* выводятся либо в виде строки с тройкой чисел, обозначающих интенсивность каждой компоненты цвета ("R,G,B"), либо в виде названия цвета, если этот цвет присутствует в наборе цветов.

Более подробно о преобразованиях значений разных типов в строки описано в главе Преобразование данных встроенных типов (в частности, в разделах Числа в строки и обратно, Дата и время, Цвет).

При работе в тестере стратегий в режиме одиночного прохода (тестирования эксперта или индикатора), результаты функции *Print* выводятся в журнал агента тестирования.

При работе в тестере стратегий в режиме оптимизации вывод в журнал подавляется из соображений повышения быстродействия, поэтому функция *Print* не имеет видимого эффекта. Тем не менее, все выражения, указанные в качестве аргументов, вычисляются.

Все аргументы, после преобразования в строковое представление, состыковываются в одну общую строку без каких-либо символов-разделителей. Если это требуется, необходимо явным образом прописать такие символы в списке аргументов. Например,

```
int x;
bool y;
datetime z;
...
Print(x, ", ", y, ", ", z);
```

Здесь в журнал выводятся 3 переменные, разделенные запятыми. Если бы не промежуточные литералы ", ", значения переменных склеились бы в записи журнала.

Множество примеров использования *Print* можно найти, начиная с самых первых разделов книги (например, Первая программа, Присваивание и инициализация, выражения и массивы, и в других).

В качестве нового приема работы с *Print* реализуем простой класс, который позволит выводить последовательность произвольных значений, не указывая между каждыми соседними величинами символ-разделитель. Используем подход с перегрузкой оператора '<<', схожий с тем, что применяется в потоках ввода/вывода C++ (std::cout).

Определение класса поместим в отдельный заголовочный файл *OutputStream.mqh*. В упрощенном виде класс представлен ниже.

```
class OutputStream
{
protected:
   ushort delimiter;
   string line;

   // добавить очередной аргумент, через разделитель (если он есть)
   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); // вывод собранной строки
         line = NULL;
      }
      return &this;
   }
};
```

Его суть заключается в том, чтобы накапливать в строковой переменной *line* строковые представления любых аргументов, переданные с помощью оператора '<<'. Если в конструкторе класса задан символ-разделитель, он будет автоматически вставляться между аргументами. Поскольку перегруженный оператор возвращает указатель на объект, мы сможем нанизывать передачу последовательности аргументов в цепочку:

```
OutputStream out(',');
out << x << y << z << out;
```

В качестве признака окончания сбора данных и для фактического вывода содержимого *line* в журнал используется перегрузка того же оператора для самого объекта.

Реальный класс несколько сложнее. Он, в частности, позволяет задать не только символ-разделитель, но и точность отображения вещественных чисел, а также флаги для выбора полей в значениях даты и времени. Кроме того класс поддерживает печать символов *ushort* в виде знаков (вместо целочисленных кодов), упрощенный вывод массивов (отдельной строкой), цвета в

шестнадцатеричном формате единой величиной (а не тройкой чисел через запятую, так как запятая часто используется как символ-разделитель, и тогда компоненты цвета выглядят в журнале как 3 разных переменных).

Демонстрация использования класса приведена в скрипте *OutputStream.mq5*.

```
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;

   /*
      пример вывода

      3.14159,text,clrBlue,true
      [100,0,-100]
      2021.09.07 17:38,clr7F8081,@
   */
}
```

void PrintFormat(const string format, ...) ≡ void printf(const string format, ...)

Функция выводит в журнал набор аргументов, руководствуясь указанной форматной строкой. Параметр *format* не только предоставляет шаблон выводимой строки с произвольным текстом, который отображается "как есть", но и может содержать управляющие последовательности, описывающие способ форматирования конкретных аргументов.

Общее количество параметров, включая форматную строку, не может превышать 64. Ограничения на типы параметров аналогичны функции *Print*.

Принципы работ *PrintFormat* и спецификации форматов идентичны тем, что были описаны для функции *StringFormat* (см. раздел Универсальный форматированный вывод данных в строку). Единственное отличие заключается в том, что *StringFormat* возвращает сформированную строку в вызывающий код, а *PrintFormat* отправляет в журнал. Можно сказать, что *PrintFormat* имеет следующий условный эквивалент:

```
Print(StringFormat(<список аргументов как есть, включая format>))
```

Помимо полного названия *PrintFormat* можно использовать более краткий алиас *printf*.

Как и функция *Print*, *PrintFormat* имеет особенности при работе в тестере в режиме оптимизации: её вывод в журнал подавляется для повышения быстродействия.

Скрипты с использованием *PrintFormat* уже встречались нам во многих разделах, например: Переход return, Цвет, Динамические массивы, Управление дескрипторами файлов, Получение списка глобальных переменных.

4.8.2 Предупреждающие сигналы (алерты)

В данном разделе под сигналом будет пониматься функция *Alert* для выдачи предупреждений пользователю терминала.

Термин "алерт", к сожалению, имеет множественные значения в MetaTrader 5. Существует 2 контекста, в которых он используется:

- настраиваемые пользователем вручную алерты во вкладке *Алерты* в панели *Инструменты*; с помощью них можно отслеживать срабатывание простых условий по превышению ценой, объемами или временем заданных значений, и выдавать уведомления различными способами;
- программные "алерты", генерируемые из MQL-кода функцией *Alert*; они никак не связаны с предыдущими;

Кроме того, слово "алерт", являющееся транслитерацией иностранного слова, часто заменяется в русском на "сигнал". И в этом случае возможно ошибочное толкование из-за совпадения с торговыми сигналами — сервисом mql5.com, доступном также и в MQL5 через специальный набор API (выходит за рамки данной книги, смотрите документацию на сайте).

void Alert(argument, ...)

Функция выводит сообщение в диалоговое немодальное окно, сопровождая это стандартным звуковым сигналом (в соответствии с выбором в диалоге *Настройки*, на закладке *События*, в терминале). Если окно скрыто, оно будет показано поверх главного окна терминала (его можно затем закрыть, свернуть или отодвинуть, продолжая работу с главным окном). Сообщение также добавляется и в журнал экспертов, с пометкой "Alert".

> В интерфейсе MetaTrader 5 нет команды, чтобы вручную открыть окно с алертами, если оно было перед этим закрыто. Чтобы вновь увидеть список предупреждений (в чистом виде, без необходимости фильтровать журнал), потребуется каким-либо образом сгенерировать новый сигнал.

Передача аргументов, вывод информации и общие принципы работы функции полностью совпадают с тем, что было изложено для функции *Print*.

Демонстрация работы функции *Alert* с изображением экрана была показана во вводном примере с приветствиями, в первой главе, в разделе Вывод данных.

Используйте *Alert* вместо *Print* в тех случаях, когда следует обратить внимание пользователя на выводимую информацию. Однако не следует злоупотреблять ею, поскольку частое появление окна может затруднить работу пользователя, вынудит его игнорировать сообщения или остановить MQL-программу. Предусмотрите в своей программе алгоритм по ограничению частоты возможной генерации сообщений.

4.8.3 Вывод сообщений в окно графика

Как мы видели в предыдущих разделах, MQL5 позволяет выводить сообщения в журнал или в окно алертов. Первый способ предназначен, прежде всего, для технической информации и не может гарантировать, что пользователь заметит сообщение (поскольку окно с журналами может быть скрыто). В то же время, второй метод способен показаться слишком навязчивым, если его применять для отображения часто меняющегося статуса программы. Промежуточный вариант предлагает функция *Comment*.

void Comment(argument, ...)

Функция выводит сообщение, составленное из всех переданных аргументов, в левый верхний угол графика. Сообщение остается там, пока эта или какая-либо другая программа не удалит его или не заменит другим.

Окно может содержать только один комментарий: при каждом вызове *Comment* прежнее содержимое (если оно было) заменяется на новое.

Чтобы очистить комментарий, достаточно вызвать функцию с пустой строкой: *Comment("")*.

Количество параметров не должно превышать 64. Поддерживаются только аргументы встроенных типов. Принципы формирования результирующей строки из переданных значений аналогичны тем, что описаны для функции *Print*.

Общая длина выводимого сообщения ограничена 2045 символами. В случае превышения лимита конец строки будет обрезан.

> Текущее содержимое комментария является одним из строковых свойств графика, которые можно узнать с помощью вызова функции *ChartGetString(NULL, CHART_COMMENT)*. Об этом и других свойствах графиков (не только строковых) мы поговорим в отдельной главе.

Так же как и в функциях *Print*, *PrintFormat*, *Alert*, строковые аргументы могут содержать символ перевода строки ('\n' или '\r\n'), в результате чего сообщение будет разделено на соответствующее количество строк. Для *Comment* это единственная возможность показать многострочное сообщение. Если для получения того же эффекта с помощью функций печати и сигналов можно вызвать их несколько раз, то с *Comment* так сделать нельзя, поскольку каждый вызов заменит старую строку на новую.

Пример работы функции *Comment* приведен на изображении окна со скриптом приветствия из первой главы, в разделе Вывод данных.

Дополнительно разработаем класс и упрощенные функции для вывода многострочных комментариев на основе кольцевого буфера заданного размера. Тестовый скрипт (*OutputComment.mq5*) и заголовочный файл с кодом класса (*Comments.mqh*) прилагаются к книге.

```
class Comments
{
   const int capacity; // максимальное количество строк
   const bool reverse; // порядок отображения (новые наверху, если true)
   string lines[];     // текстовый буфер
   int cursor;         // куда помещаем следующую строку
   int size;           // актуальное количество сохраненных строк

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();
};
```

Основную работу выполняет метод *add*.

```
void Comments::add(const string line)
{
   ...
   // если переданный текст содержит несколько строк,
   // разбиваем его на элементы по символу перевода строк
   string inputs[];
   const int n = StringSplit(line, '\n', inputs);

   // добавляем все новые элементы в кольцевой буфер
   // перезаписывая по курсору самые старые записи
   // курсор увеличивается по модулю емкости (сброс на 0 по переполнению)
   for(int i = 0; i < n; ++i)
   {
      lines[cursor] = inputs[reverse ? n - i - 1 : i];
      cursor = (cursor + 1) % capacity;
      if(size < capacity) size++;
   }
   // объединяем все текстовые записи в прямом или обратном порядке
   // склеивая символами перевода строки
   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";
      }
   }

   // выводим результат
   Comment(result);
}
```

При необходимости комментарий и текстовый буфер можно очистить методом *clear* или вызвав *add(NULL)*.

```
void Comments::clear()
{
   Comment("");
   cursor = 0;
   size = 0;
}
```

При наличии такого класса вы можете определить объект с необходимой ёмкостью буфера и направлением вывода, а затем использовать его методы.

```
Comments c(30/*capacity*/, true/*order*/);

void function()
{
   ...
   c.add("123");
}
```

Но для упрощения генерации комментариев в привычном функциональном стиле, по аналогии с функцией *Comment*, реализована пара вспомогательных функций.

```
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);
}
```

Они отличаются только направлением вывода буфера. *MultiComment* отображает строки в обратном хронологическом порядке, то есть самые свежие наверху, как на доске объявлений. Эта функция рекомендуется для неопределенно долгого эпизодического вывода информации с сохранением истории. *ChronoComment* отображает строки в прямом порядке, то есть новые добавляются вниз. Эта функция рекомендуется для пакетного вывода многострочных сообщений.

Количество строк буфера равно по умолчанию N_LINES (10). Если определить этот макрос с другим значением до включения заголовочного файла, он изменит размер.

Тестовый скрипт содержит цикл, в котором периодически генерируются сообщения.

```
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();
}
```

На каждой десятой итерации комментарий очищается. На каждой третьей — создается сообщение из двух строк (на остальных — из одной). Задержка в 1 секунду позволяет рассмотреть динамику в действии.

Вот пример окна во время работы скрипта (в режиме "новые сообщения наверху").

Многострочные комментарии на графике

Отображение многострочной информации в комментарии обладает довольно ограниченными возможностями. Если необходимо организовать вывод данных по колонкам, подсветку цветом или разными шрифтами, реакцию на нажатия мышью, произвольное расположение на графике — следует воспользоваться графическими объектами.

4.8.4 Диалоговое окно сообщений

MQL5 API включает функцию *MessageBox* для интерактивного запроса пользователя о подтверждении действий или выбора варианта обработки конкретной ситуации.

int MessageBox(const string message, const string caption = NULL, int flags = 0)

Функция открывает немодальное диалоговое окно с заданным сообщением (*message*), заголовком (*caption*) и настройками (*flags*). Окно остается видимым поверх главного окна терминала, пока пользователь не закроет его, нажатием на одну из доступных кнопок (см. далее).

Сообщение также выводится в журнал экспертов с пометкой "Message".

Если параметр *caption* равен NULL, в качестве заголовка используется название MQL-программы.

Параметр *flags* должен содержать комбинацию битовых флагов, объединенных операцией ИЛИ ('|'). Общий набор поддерживаемых флагов делится на 3 группы, определяющих:

- набор кнопок в диалоге;
- изображение значка в диалоге;
- выбор активной кнопки по умолчанию;

В следующей таблице приведены константы и значения флагов для определения кнопок диалога.

Константа	Значение	Описание
MB_OK	0x0000	1 кнопка OK (по умолчанию)
MB_OKCANCEL	0x0001	2 кнопки: OK и Отмена (Cancel)
MB_ABORTRETRYIGNORE	0x0002	3 кнопки: Прервать (Abort), Повтор (Retry), Пропустить (Ignore)
MB_YESNOCANCEL	0x0003	3 кнопки: Да (Yes), Нет (No), Отмена (Cancel)
MB_YESNO	0x0004	2 кнопки: Да (Yes) и Нет (No)
MB_RETRYCANCEL	0x0005	2 кнопки: Повтор (Retry) и Отмена (Cancel)
MB_CANCELTRYCONTINUE	0x0006	3 кнопки: Отмена (Cancel), Повторить (Try Again), Продолжить (Continue)

В следующей таблице перечислены доступные изображения (выводятся слева от сообщения).

Константа	Значение	Описание
MB_ICONSTOP MB_ICONERROR MB_ICONHAND	0x0010	знак "STOP"
MB_ICONQUESTION	0x0020	вопросительный знак
MB_ICONEXCLAMATION MB_ICONWARNING	0x0030	восклицательный знак
MB_ICONINFORMATION MB_ICONASTERISK	0x0040	знак информации

Все значки зависят от версии операционной системы. Приведенные образцы могут отличаться на вашем компьютере.

Для выбора активной кнопки зарезервированы следующие значения.

Константа	Значение	Описание
MB_DEFBUTTON1	0x0000	первая кнопка (по умолчанию), если не выбрана ни одна из других констант
MB_DEFBUTTON2	0x0100	вторая кнопка
MB_DEFBUTTON3	0x0200	третья кнопка
MB_DEFBUTTON4	0x0300	четвертая кнопка

Может возникнуть вопрос, что это за 4-я кнопка, если вышеприведенные константы позволяют задать не более трех. Дело в том, что среди флагов имеется также MB_HELP (0x00004000). Он предписывает показать в диалоге кнопку Справка. Она то и может стать по счету четвертой, если основных кнопок три. Однако нажатие на кнопку Справка не приводит к закрытию диалога, в отличие от остальных кнопок. По стандарту Windows с программой может быть связан файл справки, который должен открываться с необходимой подсказкой по нажатию кнопки Справка. Однако в настоящий момент MQL-программы пока не поддерживают эту технологию.

Функция возвращает одно из предопределенных значений в зависимости от того, каким способом был закрыт диалог (какая кнопка нажата).

Константа	Значение	Описание
IDOK	1	кнопка OK
IDCANCEL	2	кнопка Отмена (Cancel)
IDABORT	3	кнопка Прервать (Abort)
IDRETRY	4	кнопка Повтор (Retry)
IDIGNORE	5	кнопка Пропустить (Ignore)
IDYES	6	кнопка Да (Yes)
IDNO	7	кнопка Нет (No)
IDTRYAGAIN	10	кнопка Повторить (Try Again)
IDCONTINUE	11	кнопка Продолжить (Continue)

Если окно сообщения имеет кнопку Отмена (Cancel), то функция возвращает значение IDCANCEL при нажатии клавиши ESC (помимо кнопки Отмена). Если окно сообщения не имеет кнопки Отмена (Cancel), нажатие ESC не дает никакого эффекта.

Вызов *MessageBox* приостанавливает выполнение текущей MQL-программы до тех пор, пока пользователь не закроет диалог. В связи с этим использование *MessageBox* запрещено в индикаторах, поскольку индикаторы выполняются в интерфейсном потоке терминала, и ожидание ответа пользователя затормозило бы обновление графиков.

Также функцию нельзя использовать в сервисах, потому что они не имеют связи с пользовательским интерфейсом, в то время как MQL-программы остальных типов выполняются в контексте графика.

При работе в тестере стратегий функция *MessageBox* не имеет эффекта и возвращает значение 0.

После получения результата из вызова функции вы можете обработать его желаемым способом, например:

```
int result = MessageBox("Continue?", NULL, MB_YESNOCANCEL);
// используем switch или if по необходимости
switch(result)
{
case IDYES:
  // ...
  break;
case IDNO:
  // ...
  break;
case IDCANCEL:
  // ...
  break;
}
```

Для тестирования функции *MessageBox* написан скрипт *OutputMessage.mq5*, в котором пользователь может с помощью входных переменных выбрать параметры диалога и увидеть его в действии.

Группы настроек кнопок, значков и выделенной по умолчанию кнопки, а также коды возврата описаны в специальных перечислениях: ENUM_MB_BUTTONS, ENUM_MB_ICONS, ENUM_MB_DEFAULT, ENUM_MB_RESULT. Это обеспечивает наглядный ввод данных через выпадающие списки и упрощение их конвертации в строки с помощью *EnumToString*.

Вот, например, как определены два первых перечисления.

```
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)
};
```

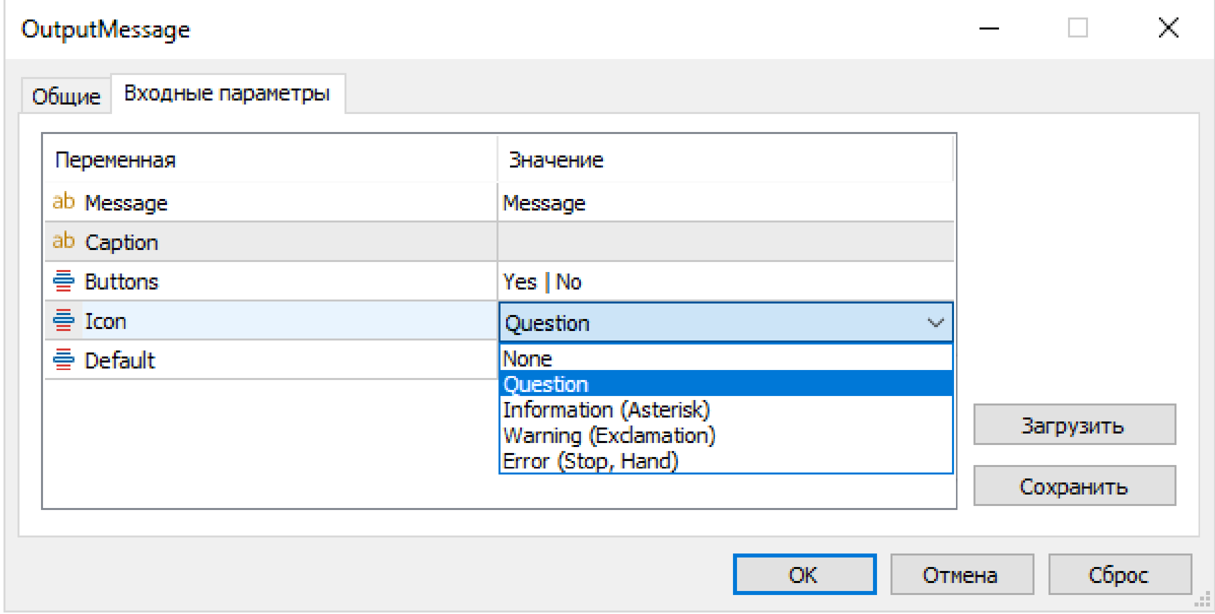
С остальными можно ознакомиться в исходном коде.

Они затем используются в качестве типов входных переменных (причем комментарии элементов обеспечивают более дружественное представление в пользовательском интерфейсе).

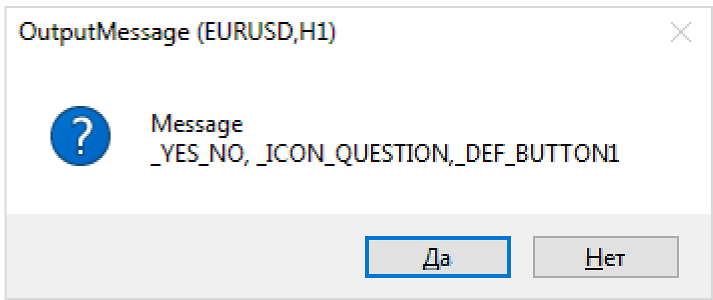
```
input string Message = "Message";
input string Caption = "";
input ENUM_MB_BUTTONS Buttons = _OK;
input ENUM_MB_ICONS Icon = _ICON_NONE;
input ENUM_MB_DEFAULT Default = _DEF_BUTTON1;

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));
}
```

Скрипт показывает в окне указанное сообщение, а также заданные настройки диалога. Результат диалога выводится в журнал.

Копия экрана при выборе параметров и получившийся диалог показаны на следующих изображениях.



Диалог настройки свойств окна



Полученное диалоговое окно с сообщением

4.8.5 Звуковые оповещения

Для работы со звуком MQL5 API предоставляет одну функцию: *PlaySound*.

bool PlaySound(const string soundfile)

Функция воспроизводит указанный звуковой файл в формате *wav*.

Если имя файла указано без пути (например, "Ring.wav"), он должен быть расположен в папке *Sounds* внутри каталога установки терминала. При необходимости внутри папки *Sounds* можно организовать вложенные папки — в таких случаях имя файла в параметре *soundfile* следует предварить относительным путем. Например, "Example/Ring.wav" ссылается на папки и файл *Sounds/Example/Ring.wav* внутри каталога установки терминала.

Кроме того можно использовать звуковые файлы, находящиеся в любой другой подпапке MQL5 в каталоге данных терминала. Такой путь следует указывать с начальной косой чертой (прямой одиночной '/' или удвоенной обратной '\\'), то есть с того символа-разделителя, который вы используете между соседними уровнями папок в файловой системе. Например, если звуковой файл *Demo.wav* лежит в папке *каталог_данных_терминала/MQL5/Files*, то в вызове *PlaySound* мы напишем путь "/Files/Demo.wav".

Вызов функции с параметром NULL останавливает воспроизведение звука. Вызов функции с новым файлом в тот момент, когда еще звучит старый, приведет к прерыванию старого и началу воспроизведения нового.

Кроме файлов, находящихся в файловой системе, в функцию можно передать путь к ресурсам — встраиваемым в MQL-программу блокам данных. В частности, разработчик может создать звуковой ресурс из файла, доступного локально во время компиляции в пределах "песочницы". Все ресурсы находятся внутри ex5-файла, что гарантирует их наличие у пользователя и упрощает распространение программы в виде единого модуля.

Подробный рассказ о всех способах применения ресурсов, включая не только звук, но и изображения, произвольные двоичные и текстовые данные, зависимые программы (индикаторы), представлен в соответствующем разделе в седьмой Части книги.

Функция *PlaySound* возвращает *true*, если файл найден, или *false* в противном случае. Учтите, что даже если файл не является звуковым и не может быть проигран, функция вернет *true*.

Проигрывание звука выполняется асинхронно, параллельно выполнению последующих инструкций программы. Иными словами, функция возвращает управление вызывающему коду сразу после вызова, не дожидаясь, когда аудиоэффект завершится.

В тестере стратегий функция *PlaySound* не выполняется.

Протестировать работу функции позволяет скрипт *OutputSound.mq5*.

```
void OnStart()
{
   PRTF(PlaySound("new.txt"));
   PRTF(PlaySound("abracadabra.wav"));
   const uint start = GetTickCount();
   PRTF(PlaySound("request.wav"));
   PRTF(GetTickCount() - start);
}
```

Программа пытается воспроизвести несколько файлов. Файл "new.txt" существует (создан специально для проверки), файл "abracadabra.wav" отсутствует, а файл "request.wav" входит в стандартную поставку MetaTrader 5. Время последнего вызова функции измеряется с помощью пары обращений к *GetTickCount*.

В результате запуска скрипта получим следующие записи в журнале:

```
PlaySound(new.txt)=true / ok
PlaySound(abracadabra.wav)=false / FILE_NOT_EXIST(5019)
PlaySound(request.wav)=true / ok
GetTickCount()-start=0 / ok
```

Файл "new.txt" был найден, и потому функция вернула *true*, хотя и не породила звука. Вызов для второго, несуществующего файла вернул *false*, а код ошибки в *_LastError* равен 5019 (FILE_NOT_EXIST). Наконец, проигрывание последнего файла (при условии его наличия) должно завершиться успешно во всех смыслах: функция вернет *true*, а терминал воспроизведет аудио. Время обработки вызова практически равно нулю (длительность звука не имеет значения).

4.9 Среда исполнения MQL-программ

Как мы знаем, исходные тексты MQL-программы после компиляции в бинарный исполняемый код в формате *ex5* готовы к работе в терминале или на агентах тестирования. Таким образом, терминал или тестер предоставляют общую среду, внутри которой и "живут" MQL-программы.

Напомним, что встроенный тестер поддерживает только 2 типа MQL-программ: эксперты и индикаторы. Про типы MQL-программ и их особенности мы подробно поговорим в пятой Части книги. А в этой главе остановимся на тех функциях MQL5 API, которые являются общими для всех типов и позволяют анализировать среду исполнения, а также, до некоторой степени, управлять ею.

Большинство свойств среды доступно только на чтение через функции *TerminalInfoInteger*, *TerminalInfoDouble*, *TerminalInfoString*, *MQLInfoInteger*, *MQLInfoString*. Из названий можно понять, что каждая функция возвращает значения определенного типа. Такая организация приводит к тому, что прикладной смысл свойств, объединенных в одной функции, может быть самым разным. Иную группировку может обеспечить реализация собственной объектной прослойки на MQL5 (пример приведем чуть ниже, в разделе об использовании свойств для привязки к программному окружению).

Указанный набор функций имеет явное логическое разделение на общие свойства терминала (с префиксом "Terminal") и свойства отдельной MQL-программы (с префиксом "MQL"). Однако во многих случаях требуется совместно анализировать схожие характеристики и терминала, и программы. Например, разрешения на использование DLL или выполнение торговых операций

выдаются как в целом для терминала, так и конкретной программе. Именно поэтому функции из данного набора имеет смысл рассматривать в комплексе, как единое целое.

На запись доступны лишь некоторые свойства среды, связанные с кодами ошибок, в частности сброс предыдущей ошибки (*ResetLastError*) и установка пользовательской ошибки (*SetUserError*).

Также в этой главы мы рассмотрим функции для программного закрытия терминала (*TerminalClose*, *SetReturnError*) и приостановки программы в отладчике (*DebugBreak*).

4.9.1 Получение общего списка свойств терминала и программы

Имеющиеся встроенные функции получения свойств среды используют обобщенный подход: свойства каждого конкретного типа объединены в отдельную функцию с единственным аргументом, задающим запрашиваемое свойство. Для идентификации свойств определены перечисления: каждый элемент описывает одно свойство.

Как мы увидим далее, данный подход часто применяется в MQL5 API и в других, в том числе прикладных областях. В частности, похожие наборы функций используются для получения свойств торгового счета и финансового инструмента.

Для описания среды достаточно свойств трех простых типов: *int*, *double*, *string*. При этом с помощью значений *int* представлены не только целочисленные свойства, но и логические флаги (в частности, разрешения/запреты, наличие сетевого соединения и т.д.), а также прочие встроенные перечисления (например, типы MQL-программ и типы лицензий).

С учетом условного деления на свойства терминала и свойства конкретной MQL-программы, существуют следующие функции, описывающие среду.

```
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)
```

Данные прототипы задают соответствие типов значений и перечислений. Например, "терминальные" свойства типа *int* сведены в ENUM_TERMINAL_INFO_INTEGER, а его же свойства типа *double* перечислены в ENUM_TERMINAL_INFO_DOUBLE, и т.д. Список доступных перечислений и их элементов можно найти в документации, в разделах, посвященных свойствам терминала и MQL-программы.

В следующих разделах мы рассмотрим все свойства, группируя их по назначению. Здесь же обратимся к задаче получения общего списка всех существующих свойств и их значений. Это часто бывает необходимо для выявления "узких мест" или особенностей работы MQL-программ на конкретных экземплярах терминала. Довольно распространена ситуация, когда на одном компьютере MQL-программа работает, а на другом — не работает совсем или работает с отклонениями.

Перечень свойств постоянно пополняется по мере развития платформы, поэтому желательно делать их запрос не на основе жестко вшитого в исходный код списка, а автоматически.

В разделе Перечисления мы создали шаблонную функцию *EnumToArray* для получения полного списка элементов перечисления (файл *EnumToArray.mqh*). Также в том разделе был представлен скрипт *ConversionEnum.mq5*, использующий указанный заголовочный файл. В скрипте была

реализована вспомогательная функция *process*, которая получала массив с кодами элементов перечисления и выводила их в журнал. Мы возьмем эти наработки в качестве отправной точки для дальнейшего усовершенствования.

Нам необходимо модифицировать функцию *process* таким образом, чтобы не только получить перечень элементов конкретного перечисления, но и запросить соответствующие свойства с помощью одной из встроенных функций свойств.

Дадим новой версии скрипта имя *Environment.mq5*.

Поскольку свойства среды рассредоточены по нескольким разным функциям (в данном случае, пяти), необходимо научиться передавать в новую версию функции *process* указатель на требуемую встроенную функцию (см. раздел Указатели на функции (typedef)). Однако MQL5 не позволяет присвоить указателю на функцию адрес встроенной функции. Это можно сделать только с прикладной функцией, реализованной на MQL5. Поэтому мы создадим функции-обертки. Например:

```
int _MQLInfoInteger(const ENUM_MQL_INFO_INTEGER p)
{
   return MQLInfoInteger(p);
}
// пример описания типа указателя
typedef int (*IntFuncPtr)(const ENUM_MQL_INFO_INTEGER property);
// инициализация переменных-указателей
IntFuncPtr ptr1 = _MQLInfoInteger;  // ok
IntFuncPtr ptr2 = MQLInfoInteger;   // ошибка компиляции
```

Выше показан "двойник" для *MQLInfoInteger* (очевидно, он должен иметь отличное, но — желательно, похожее имя). Остальные функции "упаковываются" аналогичным образом. Всего их получится пять.

Если в старой версии *process* был только один параметр шаблона, задающий перечисление, то в новой нам необходимо также передать тип возвращаемого значения (поскольку MQL5 не "понимает" слов в названии перечислений): хоть в имени ENUM_MQL_INFO_INTEGER и присутствует окончание "INTEGER", компилятор не способен увязать его с типом *int*).

Однако помимо увязки типов возвращаемого значения и перечисления нам требуется каким-то образом передать в функцию *process* указатель на соответствующую функцию-обертку (одну из пяти, определенных нами ранее). Ведь компилятор не умеет сам определять по аргументу, например, типа ENUM_MQL_INFO_INTEGER, что нужно вызвать *MQLInfoInteger*.

Для решения данной проблемы была создана специальная шаблонная структура, соединяющая в себе все три фактора воедино.

```
template<typename E,typename R>
struct Binding
{
public:
   typedef R (*FuncPtr)(const E property);
   const FuncPtr f;
   Binding(FuncPtr p): f(p) { }
};
```

Два параметра шаблона позволяют определить тип указателя функции (*FuncPtr*) с требуемым сочетанием результата и входного параметра. В экземпляре структуры имеется поле *f* для указателя этого, только что определенного типа.

Теперь новая версия функции *process* может быть описана следующим образом.

```
template<typename E,typename R>
void process(Binding<E,R> &b)
{
   E e = (E)0; // отключаем предупреждение об отсутствии инициализации
   int array[];
   // получаем перечень элементов перечисления в массив
   int n = EnumToArray(e, array, 0, USHORT_MAX);
   Print(typename(E), " Count=", n);
   ResetLastError();
   // для каждого элемента выводим название и значение,
   // получаемое через вызов указателя в структуре Binding
   for(int i = 0; i < n; ++i)
   {
      e = (E)array[i];
      R r = b.f(e); // вызываем функцию, потом анализируем _LastError
      const int snapshot = _LastError;
      PrintFormat("% 3d %s=%s", i, EnumToString(e), (string)r +
         (snapshot != 0 ? E2S(snapshot) + " (" + (string)snapshot + ")" : ""));
      ResetLastError();
   }
}
```

Входной параметр является структурой *Binding* и содержит в себе указатель конкретной функции получения свойств (это поле заполнит вызывающий код).

Данная версия алгоритма выводит в журнал порядковый номер, идентификатор свойства и его значение. Еще раз обратим внимание на то, что первое число в каждой записи будет содержать порядковый номер элемента в перечислении, а не значение (значения могут быть назначены элементам с пропусками). При необходимости вы можете добавить вывод переменной *e* "в чистом виде" внутри инструкции *PrintFormat*.

Кроме того вы можете модифицировать process, чтобы она собирала в массив (или другой контейнер, такой как карта (map)) получаемые значения свойств и возвращала их "наружу".

Было бы потенциальной ошибкой обращаться к указателю функции непосредственно в инструкции *PrintFormat* вместе с анализом кода ошибки *_LastError*. Дело в том, что последовательность вычисления аргументов функции (см. раздел Параметры и аргументы) и операндов в выражении (см. раздел Базовые понятия) в данном случае не определена. Поэтому

при вызове указателя в той же строке, где считывается *_LastError*, компилятор может решить выполнить второе раньше первого. В результате мы увидим нерелевантный код ошибки (например, с предыдущего вызова функции).

Но и это еще не всё. Встроенная переменная *_LastError* может изменить свое значение практически в любом месте вычисления выражения, если какая-либо операция выполнится с ошибкой. В частности, функция *EnumToString* может потенциально взвести код ошибки, если в качестве аргумента передано значение, отсутствующее в перечислении. В данном фрагменте мы защищены от этой проблемы, потому что наша функция *EnumToArray* возвращает массив только с проверенными (правильными) элементами перечисления. Однако в общем случае в любой "сложносочиненной" инструкции может оказаться много мест, где *_LastError* будет изменена. В связи с этим желательно зафиксировать код ошибки сразу после интересующего нас действия (здесь это вызов функции по указателю), сохранив его в промежуточную переменную *snapshot*.

Но вернемся к основной задаче. Мы можем, наконец, организовать вызов новой функции *process* для получения различных свойств программного окружения.

```
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));
}
```

Ниже приведен фрагмент сгенерированных записей в журнале.

```
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\Commor
  5 TERMINAL_LANGUAGE=Russian
```

Эти и другие свойства будут описаны в следующих разделах.

Стоит отметить, что некоторые свойства унаследованы с прошлых этапов развития платформы и оставлены только для совместимости. В частности, свойство TERMINAL_X64 в *TerminalInfoInteger* возвращает признак того, является ли терминал 64-разрядным. Сегодня разработка 32-битных версий прекращена, и потому это свойство всегда равно 1 (*true*).

4.9.2 Номер сборки терминала

Поскольку терминал постоянно совершенствуется и в его новых версиях появляются новые возможности, MQL-программе может потребоваться анализ текущей версии, чтобы применить различные варианты алгоритмов. Кроме того, ни одна программа не застрахована от ошибок, включая и сам терминал. Поэтому при возникновении проблем следует предусмотреть вывод

диагностики, включающей текущую версию терминала. Это может помочь в воспроизведении и исправлении ошибок.

Получить номер сборки терминала позволяет свойство TERMINAL_BUILD в ENUM_TERMINAL_INFO_INTEGER.

```
if(TerminalInfoInteger(TERMINAL_BUILD) >= 3000)
{
   ...
}
```

Напомним, что номер сборки компилятора, с помощью которого собирается программа, доступна в исходном коде через макроопределения __MQLBUILD__ или __MQL5BUILD__ (см. раздел Предопределенные константы).

4.9.3 Тип и лицензия программы

Один и тот же исходный код может тем или иным образом входить в состав MQL-программ разных типов. Кроме варианта включения исходных кодов (директивой препроцессора *#include*) в общий продукт на стадии компиляции, следует отметить возможность сборки библиотек — двоичных программных модулей, подключаемых к основной программе на стадии выполнения.

Однако некоторые функции разрешено использовать только в программах определенных типов. Например, функцию *OrderCalcMargin* нельзя использовать в индикаторах. Хотя это ограничение не представляется фундаментально обоснованным, разработчик универсального алгоритма подсчета залоговых средств, который может быть встроен не только в эксперты, но и в индикаторы, должен учитывать этот нюанс и предоставлять для индикаторов альтернативный способ расчета.

Полный список ограничений по типам программ будет приведен в соответствующем разделе каждой главы. Во всех подобных случаях бывает важно знать тип "родительской" программы.

Для определения типа программы существует свойство MQL_PROGRAM_TYPE в ENUM_MQL_INFO_INTEGER. Возможные значения свойства описаны в перечислении ENUM_PROGRAM_TYPE.

Идентификатор	Значение	Описание
PROGRAM_SCRIPT	1	скрипт
PROGRAM_EXPERT	2	эксперт
PROGRAM_INDICATOR	4	индикатор
PROGRAM_SERVICE	5	сервис

Во фрагменте журнала, приведенном в предыдущем разделе, мы видели что свойство PROGRAM_SCRIPT равно 1, потому что наш тест является скриптом. Для получения строкового описания можно применить функцию *EnumToString*.

```
ENUM_PROGRAM_TYPE type = (ENUM_PROGRAM_TYPE)MQLInfoInteger(MQL_PROGRAM_TYPE);
Print(EnumToString(type));
```

Другим свойством MQL-программы, которое удобно анализировать для включения/отключения тех или иных возможностей, является тип лицензии. Как известно, MQL-программы могут распространяться свободно или в рамках онлайн магазина MQL5 Маркет. Более того, программа в магазине может быть куплена или скачана в виде демо-версии. Эти факторы легко проверить и, по желанию, адаптировать алгоритмы под них. Для этих целей существует свойство MQL_LICENSE_TYPE в ENUM_MQL_INFO_INTEGER, использующее в качестве типа перечисление ENUM_LICENSE_TYPE.

Идентификатор	Значение	Описание
LICENSE_FREE	0	бесплатная неограниченная версия
LICENSE_DEMO	1	демо-версия платного продукта из Маркета, работает только в тестере стратегий
LICENSE_FULL	2	купленная лицензионная версия, допускает не менее 5 активаций (может быть увеличено продавцом)
LICENSE_TIME	3	версия с ограничением по времени (пока не реализовано)

Здесь важно отметить, что лицензия относится к тому двоичному ex5-модулю, из которого делается запрос с помощью *MQLInfoInteger(MQL_LICENSE_TYPE)*. Внутри библиотеки данная функцию вернет собственную лицензию библиотеки, а не основной программы, к которой подключена библиотека.

В качестве примера для проверки обеих функций раздела, к книге прилагается простой сервис *EnvType.mq5*. Он не содержит рабочего цикла и потому сразу же завершится после выполнения двух инструкций в *OnStart*.

```
#property service

void OnStart()
{
   Print(EnumToString((ENUM_PROGRAM_TYPE)MQLInfoInteger(MQL_PROGRAM_TYPE)));
   Print(EnumToString((ENUM_LICENSE_TYPE)MQLInfoInteger(MQL_LICENSE_TYPE)));
}
```

Чтобы упростить его запуск, то есть исключить необходимость создавать экземпляр службы и запускать его через контекстное меню Навигатора в терминале, предлагается воспользоваться отладчиком: достаточно в MetaEditor открыть исходный код и выполнить команду *Отладка -> Начать на реальных данных* (F5, или кнопка в инструментальной панели).

Мы должны получить в журнале записи:

```
EnvType (debug)      PROGRAM_SERVICE
EnvType (debug)      LICENSE_FREE
```

Здесь наглядно видно, что тип программы — это сервис, а лицензия — фактически отсутствует (свободное пользование).

4.9.4 Режимы работы терминала и программы

Среда MetaTrader 5 обеспечивает решение различных задач на стыке трейдинга и программирования, что обуславливает необходимость нескольких режимов работы как самого терминала, так и конкретной программы.

С помощью MQL5 API можно отличить штатную деятельность онлайн от тестирования на истории, отладку исходного кода (с целью выявления потенциальных ошибок) от анализа быстродействия (поиска "узких мест" в коде), а также локальную копию терминала от облачной (MetaTrader VPS).

Всё это описывается несколькими флагами, каждый из которых содержит значение логического типа: *true* или *false*.

Идентификатор	Описание
MQL_DEBUG	программа работает в режиме отладки
MQL_PROFILER	программа работает в режиме профилирования кода
MQL_TESTER	программа работает в тестере
MQL_FORWARD	программа выполняется в процессе форвардного тестирования
MQL_OPTIMIZATION	программа выполняется в процессе оптимизации
MQL_VISUAL_MODE	программа выполняется в режиме визуального тестирования
MQL_FRAME_MODE	эксперт выполняется на графике в режиме сбора фреймов результатов оптимизации
TERMINAL_VPS	терминал работает на виртуальном сервере MetaTrader Virtual Hosting (MetaTrader VPS)

Флаги MQL_FORWARD, MQL_OPTIMIZATION, MQL_VISUAL_MODE подразумевают наличие взведенного флага MQL_TESTER.

Некоторые попарные сочетания флагов являются взаимоисключающими, то есть такие флаги не могут быть включены одновременно.

В частности, наличие MQL_FRAME_MODE исключает MQL_TESTER, и наоборот. MQL_OPTIMIZATION исключает MQL_VISUAL_MODE, а MQL_PROFILER исключает MQL_DEBUG.

Все флаги, относящиеся к тестированию (MQL_TESTER, MQL_VISUAL_MODE), мы изучим в разделах, посвященных экспертам и, отчасти, индикаторам. Все, что касается оптимизации экспертов (MQL_OPTIMIZATION, MQL_FORWARD, MQL_FRAME_MODE), будет освещено в отдельном разделе.

Сейчас познакомимся с принципами чтения флагов на примере режимов отладки (MQL_DEBUG) и профилирования (MQL_PROFILER). Заодно вспомним, как данные режимы активируются из редактора MetaEditor (подробности смотрите в документации, а разделах Отладка и Профилирование).

Для этой цели подготовлен скрипт *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));
}
```

Прежде чем запускать программу, следует проверить настройки для отладки/профилирования. Для этого выполните в MetaEditor команду *Сервис -> Настройки* и посмотрите значения полей на закладке *Отладка/Профилирование*. Если включена опция *Использовать указанные настройки*, то именно значения нижележащих полей будут влиять на то, на графике какого финансового инструмента и с каким таймфреймом будет запускаться программа. Если опция отключена, будет использован первый финансовый инструмент в *Обзоре рынка* и таймфрейм H1.

На данном этапе выбор опции не принципиален.

После приготовлений запустим скрипт с помощью команды *Отладка -> Начать на реальных данных* (F5). Поскольку скрипт только выводит запрошенные свойства в журнал (и нам не нужны в нем точки остановки), его исполнение будет моментальным. При необходимости пошаговой отладки мы могли бы поставить точку остановки (F9) на любой инструкции в исходном коде, и выполнение скрипта замерло бы там на любой нужный нам срок, давая возможность изучить содержание всех переменных в MetaEditor, а также построчно продвигаться (F10) по алгоритму.

В журнале MetaTrader 5 (закладка Эксперты) мы должны увидеть следующее:

```
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
```

Флаги всех режимов сброшены, за исключением MQL_DEBUG.

Теперь запустим тот же скрипт из *Навигатора* в MetaTrader 5 (достаточно перетащить его мышью на любой график). Мы получим почти идентичный набор флагов, но MQL_DEBUG на этот раз будет равен 0 (потому что программа выполнялась штатным образом, а не под отладчиком).

> Обратите внимание, что запуск программы с отладкой предваряется её перекомпиляцией в особом режиме, когда в выполняемый файл добавляется служебная информация, позволяющая проводить отладку. Такой бинарный файл больше по размеру и работает медленнее обычного. Поэтому после завершения отладки, перед использованием в реальной торговле, передачей заказчику или загрузкой в Маркет, программу следует перекомпилировать командой *Файл -> Компилировать* (F7).
>
> Метод компиляции не влияет напрямую на свойство MQL_DEBUG. Отладочная версия программы, как мы видим, может быть запущена в терминале без отладчика, и MQL_DEBUG в этом случае будет сброшен. Определить способ компиляции позволяют два встроенных макроса: _DEBUG и _RELEASE (см. раздел Предопределенные константы). Они являются

константами, а не функциями, потому что данное свойство "зашивается" в программу на стадии компиляции и затем не может измениться (в отличие от среды исполнения).

Теперь выполним в MetaEditor команду *Отладка -> Начать профилирование на реальных данных*. Особого смысла профилировать столь простой скрипт, разумеется, нет, но наша задача сейчас — убедиться в том, что в свойствах среды включится соответствующий флаг. И действительно, против MQL_PROFILER теперь стоит единица.

```
MQLInfoInteger(MQL_TESTER)=0 / ok
MQLInfoInteger(MQL_DEBUG)=0 / ok
MQLInfoInteger(MQL_PROFILER)=1 / ok
...
```

Запуск программы с профилированием также предваряется её перекомпиляцией в другом особом режиме, добавляющем в бинарный файл другую служебную информацию, необходимую для замера скорости выполнения инструкций. После анализа отчета профилировщика и исправления "узких мест" следует перекомпилировать программу обычным образом.

В принципе отладка и профилирование могут выполняться как онлайн, так и в тестере (MQL_TESTER) на исторических данных, однако тестер поддерживает лишь эксперты и индикаторы. Поэтому на примере скрипта невозможно увидеть взведенный флаг MQL_TESTER или MQL_VISUAL_MODE.

Как известно, MetaTrader 5 позволяет тестировать торговые программы в быстром режиме (без графика) и в визуальном режиме (на отдельном графике). Именно во втором случае и будет включено свойства MQL_VISUAL_MODE. Его имеет смысл проверять, в частности, для отключения манипуляций с графическими объектами в отсутствии визуализации.

Для отладки на истории в визуальном режиме необходимо предварительно включить опцию *Использовать визуальный режим для отладки на истории* в диалоге настроек MetaEditor. Аналитические программы (индикаторы) тестируются всегда в визуальном режиме.

Имейте в виду, что отладка онлайн небезопасна для торгующих экспертов.

4.9.5 Разрешения

MetaTrader 5 позволяет ограничить выполнение некоторых действий MQL-программами, из соображений безопасности. Некоторые из этих ограничений являются двухуровневыми, то есть устанавливаются раздельно для терминала в целом, и для конкретной программы. Настройки терминала имеют приоритет или выступают значениями по умолчанию для настроек любой MQL-программы. Например, трейдер может запретить всю автоматическую торговлю, установив соответствующий флажок в диалоге настроек MetaTrader 5. При этом частные разрешения на торговлю, "выданные" ранее конкретным роботам в их диалогах, утрачивают силу.

В MQL5 API подобные ограничения (или наоборот, разрешения) доступны на чтение через функции *TerminalInfoInteger* и *MQLInfoInteger*. Поскольку они имеют одинаковый эффект на MQL-программу, та обязана проверять общие и частные запреты в равной степени тщательно (во избежание генерации ошибки при попытке выполнить недопустимое действие). Поэтому в данном разделе опции разных уровней объединены в единый список.

Все разрешения представляют собой логические флаги, то есть хранят значение *true* или *false*.

Идентификатор	Описание
TERMINAL_DLLS_ALLOWED	разрешение на использование DLL
TERMINAL_TRADE_ALLOWED	разрешение на автоматическую торговлю онлайн
TERMINAL_EMAIL_ENABLED	разрешение на отправку писем (SMTP-сервер и логин должны быть указаны в настройках терминала)
TERMINAL_FTP_ENABLED	разрешение на отправку файлов по FTP на указанный сервер (в том числе отчетов для указанного в настройках терминала торгового счета)
TERMINAL_NOTIFICATIONS_ENABLED	разрешение на отправку push-уведомлений на смартфон
MQL_DLLS_ALLOWED	разрешение на использование DLL для данной программы
MQL_TRADE_ALLOWED	разрешение на автоматическую торговлю для данной программы
MQL_SIGNALS_ALLOWED	разрешение на работу с сигналами для данной программы

Разрешение использовать DLL на уровне терминала означает, что при запуске MQL-программы, в которой есть ссылка на какую-либо динамическую библиотеку, в диалоге её свойств будет по умолчанию включен флаг Разрешить импорт DLL на закладке Зависимости. Если флаг в настройках терминала сброшен, то и опция в свойствах MQL-программы будет по умолчанию выключена. Пользователь в любом случае должен разрешить импорт для отдельной программы (есть одно исключение для скриптов, о котором сказано ниже). Если он этого не сделает, программа просто не запустится.

Иными словами, проверить флаги TERMINAL_DLLS_ALLOWED и MQL_DLLS_ALLOWED может либо программа без привязки к DLL, либо программа с привязкой, но для которой MQL_DLLS_ALLOWED однозначно равно true (в силу того, что она уже запустилась). Таким образом, в составе программных комплексов, требующих DLL, вероятно, имеет смысл предусмотреть независимую утилиту, которая отслеживала бы состояние флага и выводила диагностику для пользователя, если он его вдруг отключит. Например, эксперту может требоваться индикатор, использующий DLL. Тогда прежде чем пытаться загрузить индикатор и получить его дескриптор, эксперт может проверить флаг TERMINAL_DLLS_ALLOWED и выдать предупреждение, если он сброшен.

Для скриптов поведение слегка отличается из-за того, что диалог с настройками запускаемого скрипта открывается, только если в исходном коде присутствует директива *#property script_show_inputs*. Если её нет, то диалог появляется, когда в настройках терминала сброшен флаг TERMINAL_DLLS_ALLOWED (и пользователь обязан флаг включить, чтобы скрипт заработал). Когда же общий флаг TERMINAL_DLLS_ALLOWED включен, скрипт запускается без подтверждения пользователя, то есть значение MQL_DLLS_ALLOWED подразумевается равным *true* (согласно TERMINAL_DLLS_ALLOWED).

> При работе в тестере флаги TERMINAL_TRADE_ALLOWED и MQL_TRADE_ALLOWED всегда равны *true*. Однако в индикаторах доступ ко всем торговым функциям запрещен вне зависимости от данных флагов. В тестере не допускается проверка MQL-программ с зависимостью от DLL.

От флагов TERMINAL_EMAIL_ENABLED, TERMINAL_FTP_ENABLED, TERMINAL_NOTIFICATIONS_ENABLED зависит работоспособность функций *SendMail*, *SendFTP*,

SendNotification, которые описаны в разделе Сетевые функции. Флаг MQL_SIGNALS_ALLOWED влияет на доступность группы функций, управляющих подпиской на торговые сигналы mql5.com (не рассматриваются в данной книге). Его состояние соответствует опции *Разрешить изменение настроек сигналов* на закладке *Общие* в свойствах MQL-программы.

Поскольку проверка некоторых свойства требует дополнительных усилий, имеет смысл обернуть флаги в класс, скрывающий в своих методах множественные вызовы разных системных функций. Это тем более необходимо из-за того, что некоторые разрешения не ограничиваются только вышеприведенными опциями. Например, разрешение на торговлю может быть установлено (или снято) не только на уровне терминала или MQL-программы, но и для отдельного финансового инструмента — согласно его спецификации у вашего брокера и расписанию торговых сессий на бирже. В связи с этим мы пока представим заготовку класса Permissions, которая будет содержать лишь знакомые элементы, а затем усовершенствуем её с учетом прикладных API.

Благодаря программной прослойке в виде класса, программист избавлен от необходимости запоминать, какие разрешения определены для *TerminalInfo*-функций, а какие — для *MqlInfo*-функций.

Исходный код находится в файле *EnvPermissions.mq5*.

```
class Permissions
{
public:
   static bool isTradeEnabled(const string symbol = NULL, const datetime session = 0)
   {
      // TODO: будет дополнено прикладными проверками символа и сессий
      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));
   }
};
```

Все методы класса — статические и вызываются в *OnStart*.

```
void OnStart()
{
   Permissions::isTradeEnabled();
   Permissions::isDllsEnabledByDefault();
   Permissions::isDllsEnabled();
   Permissions::isEmailEnabled();
   Permissions::isPushEnabled();
   Permissions::isSignalsEnabled();
}
```

Пример порождаемых записей в журнале показан ниже.

```
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
```

Для самостоятельного изучения в скрипт встроена (но закомментирована) возможность подключить системные DLL для чтения содержимого буфера обмена Windows. Создание и использование библиотек, в частности директива *#import*, будут рассмотрены в седьмой Части книги, в разделе Библиотеки.

Предположим, что глобальная опция импорта DLL в терминале отключена (это рекомендуемая настройка из соображений безопасности). Тогда, если к скрипту подключены DLL, его удастся запустить, только разрешив импорт в его личном диалоге настроек, в результате чего *MQLInfoInteger(MQL_DLLS_ALLOWED)* станет возвращать 1 (*true*). Если дано глобальное разрешение для DLL, то получим *TerminalInfoInteger(TERMINAL_DLLS_ALLOWED)=1*, и MQL_DLLS_ALLOWED унаследует это значение.

4.9.6 Проверка сетевых подключений

Как известно, платформа MetaTrader 5 является распределенной системой, включающей несколько звеньев. Помимо клиентского терминала и сервера брокера, в неё входит сообщество MQL5, Маркет, облачные сервисы и многое другое. По сути распределенной является и клиентская часть, состоящая из терминала и агентов тестирования, которые могут быть развернуты на множестве компьютеров локальной сети. При этом связь между любыми звеньями может, потенциально, нарушаться по тем или иным причинам. И хотя инфраструктура MetaTrader 5 пытается автоматически восстановить свою работоспособность, не всегда это удается сделать быстро.

Поэтому в MQL-программах следует учитывать возможность отсутствия связи. MQL5 API позволяет контролировать наиболее важные связи: с торговым сервером и сообществом MQL5. Следующие свойства доступны в *TerminalInfoInteger*.

Идентификатор	Описание
TERMINAL_CONNECTED	наличие подключения к торговому серверу
TERMINAL_PING_LAST	последнее известное значение скорости связи с торговым сервером в микросекундах
TERMINAL_COMMUNITY_ACCOUNT	наличие авторизационных данных MQL5.community в терминале
TERMINAL_COMMUNITY_CONNECTION	наличие подключения к MQL5.community
TERMINAL_MQID	наличие MetaQuotes ID для отправки push-уведомлений

Все свойства кроме TERMINAL_PING_LAST являются булевыми флагами. TERMINAL_PING_LAST содержит значение типа *int*.

Помимо самого соединения, MQL-программа часто должна убедиться в актуальности имеющихся данных. В частности, взведенный флаг TERMINAL_CONNECTED еще не означает, что интересующие вас котировки синхронизированы с сервером. Для этого нужно дополнительно делать проверки с помощью *SymbolIsSynchronized* или *SeriesInfoInteger(..., SERIES_SYNCHRONIZED)* — об этих функциях мы поговорим в главе о таймсериях.

Кроме того функция *TerminalInfoDouble* поддерживает еще одно интересное свойство: TERMINAL_RETRANSMISSION. Это процент повторно отправляемых сетевых пакетов в TCP/IP протоколе для всех запущенных приложений и служб на данном компьютере. Даже в самой быстрой и правильно настроенной сети иногда происходят потери пакетов и, как следствие, отсутствие подтверждений о доставке пакетов между получателем и отправителем. В таких случаях производится повторная отправка "потерянного" пакета. Сам терминал не считает показатель TERMINAL_RETRANSMISSION, а запрашивает его раз в минуту в операционной системе.

Высокое значение данного показателя может говорить о проблемах внешнего, по отношению к MetaTrader 5, характера (на магистральном канале в Интернет, у вашего провайдера, в локальной сети или на компьютере), что способно ухудшить качество подключения терминала.

Если есть подтвержденное соединение с сообществом (TERMINAL_COMMUNITY_CONNECTION), MQL-программа может запросить текущий баланс пользователя, вызвав *TerminalInfoDouble(TERMINAL_COMMUNITY_BALANCE)*. Это позволяет пользоваться автоматизированной подпиской на платные торговые сигналы (документация API доступна на сайте mql5.com).

Проверим перечисленные свойства с помощью скрипта *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));
}
```

Вот пример лога (значения будут соответствовать вашим настройкам).

```
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 Вычислительные ресурсы: память, диск, процессор

Как и все программы, MQL-программы потребляют компьютерные ресурсы, внося свой вклад в расход памяти, места на диске и загрузку процессора терминалом. Учитывая, что терминал сам по себе весьма требователен к ресурсам (в частности, припомним потенциальную загрузку

котировок и тиков для множества финансовых инструментов, с многолетней историей), иногда требуется анализировать и контролировать ситуацию на предмет близости доступных лимитов.

MQL5 API предоставляет несколько свойств, позволяющих оценить предельно достижимые и израсходованные ресурсы. Свойства сведены в перечисления ENUM_MQL_INFO_INTEGER и ENUM_TERMINAL_INFO_INTEGER.

Идентификатор	Описание
MQL_MEMORY_LIMIT	максимально возможный объём динамической памяти для MQL-программы в KB
MQL_MEMORY_USED	размер использованной памяти MQL-программой в MB
MQL_HANDLES_USED	количество объектов классов
TERMINAL_MEMORY_PHYSICAL	размер физической оперативной памяти в системе в MB
TERMINAL_MEMORY_TOTAL	размер памяти (физической+файл подкачки, то есть виртуальной), доступной процессу терминала (агента) в MB
TERMINAL_MEMORY_AVAILABLE	размер свободной памяти процесса терминала (агента) в MB, часть TOTAL
TERMINAL_MEMORY_USED	размер памяти, использованной терминалом (агентом) в MB, часть TOTAL
TERMINAL_DISK_SPACE	объем свободной памяти на диске, с учетом возможных квот для папки MQL5/Files терминала (агента), в MB
TERMINAL_CPU_CORES	количество ядер процессора в системе
TERMINAL_OPENCL_SUPPORT	версия поддерживаемой OpenCL в виде 0x00010002 = 1.2; "0" означает, что OpenCL не поддерживается

Максимальный объем памяти, доступный MQL-программе, описывается свойством MQL_MEMORY_LIMIT. Это единственное свойство из перечисленных, использующее килобайты (Kb). Все остальные возвращаются в мегабайтах (Mb). Как правило, MQL_MEMORY_LIMIT равно TERMINAL_MEMORY_TOTAL, то есть вся имеющаяся на компьютере память по умолчанию может быть выделена одной MQL-программе. Однако терминал, в особенности его облачная реализация для MetaTrader VPS, и облачные агенты тестирования могут ограничивать размер памяти для отдельной MQL-программы. Тогда MQL_MEMORY_LIMIT будет существенно меньше TERMINAL_MEMORY_TOTAL.

Поскольку Windows обычно создает файл подкачки, равный по размеру физической памяти (ОЗУ), свойство TERMINAL_MEMORY_TOTAL может в 2 раза превышать TERMINAL_MEMORY_PHYSICAL.

Вся доступная виртуальная память TERMINAL_MEMORY_TOTAL делится между использованной (TERMINAL_MEMORY_USED) и еще свободной (TERMINAL_MEMORY_AVAILABLE).

К книге прилагается скрипт *EnvProvision.mq5*, в котором в журнал выводятся все указанные свойства.

```
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)); // выделяем 10 Mb
   PRTF(MQLInfoInteger(MQL_MEMORY_USED));
   PRTF(TerminalInfoInteger(TERMINAL_MEMORY_AVAILABLE));
   PRTF(TerminalInfoInteger(TERMINAL_MEMORY_USED));
}
```

После первичного вывода свойств мы распределяем массив на 10 Mb, и затем снова проверяем память. Пример результата приведен ниже (у вас будут свои значения).

```
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
```

Обратите внимание, что общая виртуальная память (8190) равна удвоенной физической (4095). Доступный для скрипта объем памяти 8388608 Kb практически равен всей памяти 8190 Mb. Свободная (7842) и занятая (348) системная память в сумме также дают 8190.

Если до выделения памяти под массив MQL-программа занимала 1 Mb, то после выделения — уже 11 Mb. При этом объем памяти, занятой терминалом увеличился всего на 5 Mb (с 348 до 353), поскольку некоторые ресурсы были зарезервированы заранее.

4.9.8 Характеристики экрана

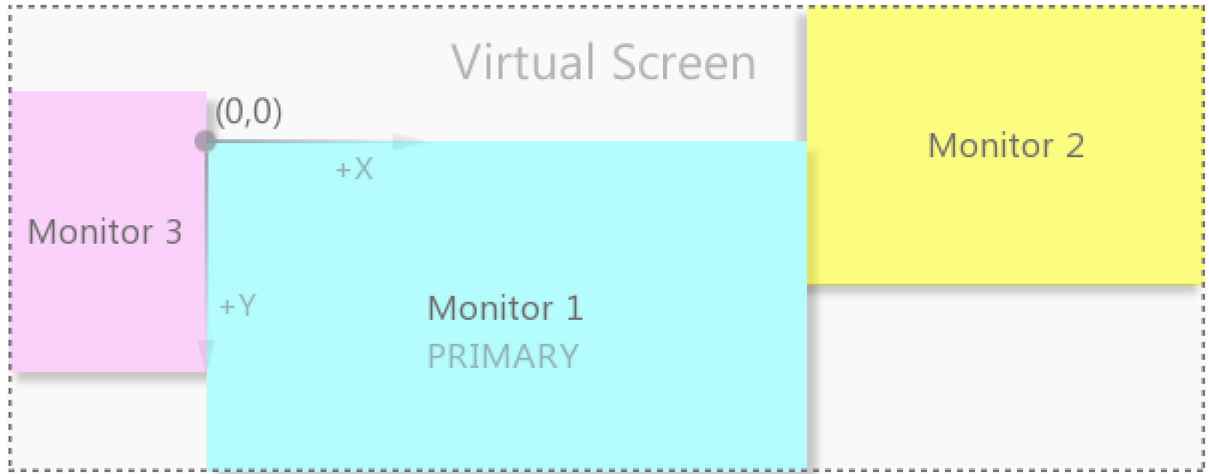
Несколько свойств, предоставляемых функцией *TerminalInfoInteger*, относятся к видео подсистеме компьютера.

Идентификатор	Описание
TERMINAL_SCREEN_DPI	разрешающая способность вывода информации на экран измеряется в количестве отображаемых точек на погонный дюйм (DPI, Dots Per Inch)
TERMINAL_SCREEN_LEFT	левая координата виртуального экрана
TERMINAL_SCREEN_TOP	верхняя координата виртуального экрана
TERMINAL_SCREEN_WIDTH	ширина виртуального экрана
TERMINAL_SCREEN_HEIGHT	высота виртуального экрана
TERMINAL_LEFT	левая координата терминала относительно виртуального экрана
TERMINAL_TOP	верхняя координата терминала относительно виртуального экрана
TERMINAL_RIGHT	правая координата терминала относительно виртуального экрана
TERMINAL_BOTTOM	нижняя координата терминала относительно виртуального экрана

Знание параметра TERMINAL_SCREEN_DPI позволяет задавать размеры графических объектов таким образом, чтобы они выглядели одинаково на мониторах с различной разрешающей способностью. Например, если требуется создать кнопку с видимым размером X сантиметров, то определить для неё количество экранных точек (пикселей) можно следующей функцией:

```
int cm2pixels(const double x)
{
   static const double inch2cm = 2.54; // 1 дюйм равен 2.54 см
   return (int)(x / inch2cm * TerminalInfoInteger(TERMINAL_SCREEN_DPI));
}
```

Виртуальным экраном является прямоугольник, охватывающий все мониторы. Если в системе имеется больше одного монитора и порядок их расположения отличается от строго размещения слева направо, то левая координата виртуального экрана может оказаться отрицательной, а центр (точка отсчета) будет на границе двух мониторов (в верхнем левом углу основного монитора).



Виртуальный экран из нескольких мониторов

Если в системе один монитор, то размер виртуального экрана полностью соответствует ему.

В координатах терминала не учитывается его возможная текущая максимизация (то есть, если главное окно максимизировано, свойства возвращают размер без максимизации, хотя терминал развернут на весь монитор).

В скрипте *EnvScreen.mq5* проверим чтение экранных свойств.

```
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));
}
```

А вот пример полученных записей в журнале.

```
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
```

Помимо общих размеров экрана и окна терминала в MQL-программах довольно часто требуется анализировать текущий размер графика (дочернего окна внутри терминала). Для этих целей имеется специальный набор функций (в частности, *ChartGetInteger*), который мы рассмотрим в разделе Графики.

4.9.9 Строковые свойства терминала и программы

Функции *MQLInfoString* и *TerminalInfoString* позволяют узнать несколько строковых свойств терминала и MQL-программы.

Идентификатор	Описание
MQL_PROGRAM_NAME	имя запущенной MQL-программы
MQL_PROGRAM_PATH	путь для данной запущенной MQL-программы
TERMINAL_LANGUAGE	язык терминала
TERMINAL_COMPANY	имя компании (брокера)
TERMINAL_NAME	имя терминала
TERMINAL_PATH	папка, из которой запущен терминал
TERMINAL_DATA_PATH	папка, в которой хранятся данные терминала
TERMINAL_COMMONDATA_PATH	общая папка всех клиентских терминалов, установленных на компьютере

Имя запущенной программы (MQL_PROGRAM_NAME), как правило, совпадает с именем главного компилируемого модуля (mq5-файла), но может и отличаться. В частности, если ваш исходный код компилируется в библиотеку, которая импортируется в другую MQL-программу (эксперт, индикатор, скрипт или сервис), то свойство MQL_PROGRAM_NAME вернет имя основной программы, а не библиотеки (библиотека не является самостоятельной программой, которую можно запустить).

Принципы организации рабочих папок терминала рассмотрены в разделе Работа с файлами. С помощью перечисленных свойств можно узнать, в частности, куда установлен терминал (TERMINAL_PATH), корневые папки с рабочими данными текущего экземпляра (TERMINAL_DATA_PATH) и всех экземпляров (TERMINAL_COMMONDATA_PATH) терминалов.

Простой скрипт *EnvDescription.mq5* выводит в журнал все эти свойства.

```
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));
}
```

Ниже представлен пример результата.

```
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
```

Интерфейсный язык терминала можно узнать не только в виде строки в свойстве TERMINAL_LANGUAGE, но и как номер кодовой страницы (см. свойство TERMINAL_CODEPAGE в следующем разделе).

4.9.10 Настраиваемые свойства: лимит баров и язык интерфейса

Особняком среди свойств терминала стоят два, которые пользователь может менять интерактивно, по своему усмотрению. Это максимальное количество баров, выводимых на каждый график по умолчанию (оно соответствует значению поля *Макс. баров в окне* в диалоге *Настроек*), а также язык интерфейса (выбирается с помощью команды *Вид* -> *Languages*).

Идентификатор	Описание
TERMINAL_MAXBARS	максимальное количество баров на графике
TERMINAL_CODEPAGE	номер кодовой страницы языка, выбранного в клиентском терминале

Обратите внимание, что значение TERMINAL_MAXBARS задает верхний предел для отображения баров, но на деле их количество может быть меньше, если на каком-либо таймфрейме глубина имеющейся истории котировок недостаточно глубока. С другой стороны, длина истории может и превышать заданный лимит TERMINAL_MAXBARS. Тогда узнать количество потенциально доступных баров можно узнать с помощью функции из группы свойств таймсерий — *SeriesInfoInteger*, со свойством SERIES_BARS_COUNT. Учтите, что значение TERMINAL_MAXBARS напрямую влияет на расход оперативной памяти.

4.9.11 Привязка программы к свойствам среды исполнения

В качестве примера работы со свойствами, описанными в предыдущих разделах, рассмотрим востребованную задачу привязки MQL-программы к аппаратному окружению, чтобы защитить её от копирования. Когда программа распространяется через MQL5 Маркет, привязку обеспечивает сам сервис. Однако если программа разрабатывается в заказном порядке, привязать её можно либо к номеру счета, либо к имени клиента, либо к доступным свойствам терминала (компьютера). Первое не всегда удобно, потому что многие трейдеры имеют несколько реальных счетов (вероятно, у разных брокеров), не говоря уже о демо-счетах с ограниченным сроком существования. Второе может быть вымышленным или слишком расхожим. Поэтому мы реализуем прототип алгоритма для привязки программы к выбранному набору свойств окружения. Более серьезные схемы защиты, вероятно, могли бы использовать DLL и напрямую

читать аппаратные метки устройств из системы Windows, однако не каждый клиент согласится запускать потенциально небезопасные библиотеки.

Наш вариант защиты представлен в скрипте *EnvSignature.mq5*. Его суть заключается в том, чтобы вычислить хеши от заданных свойств среды и создать на их основе уникальную сигнатуру (оттиск).

Хешированием называется специальная обработка произвольной информации, в результате чего создается новый блок данных, обладающий следующими характеристиками (их гарантирует используемый алгоритм):

- совпадение значений хешей для двух исходных наборов данных означает практически со 100% вероятностью, что данные идентичны (вероятность случайного совпадения пренебрежимо мала);
- если исходные данные изменятся, значение их хеша также изменится;
- по значению хеша невозможно математически восстановить исходные данные (они остаются в секрете), если только не выполнить полный перебор возможных исходных значений (с увеличением их исходного размера и при отсутствии знаний о структуре, задача нерешаема в обозримом будущем);
- размер хеша фиксирован (не зависит от объема исходных данных).

Предположим, одно из свойств среды описано строкой: "TERMINAL_LANGUAGE=Russian". Его можно получить простой инструкцией вроде следующей (упрощенно):

```
string language = EnumToString(TERMINAL_LANGUAGE) +
            "=" + TerminalInfoString(TERMINAL_LANGUAGE);
```

Разумеется, фактический язык будет соответствовать настройкам. Имея гипотетическую функцию хеширования *Hash*, мы можем вычислить сигнатуру.

```
string signature = Hash(language);
```

Когда свойств будет больше, мы просто повторим процедуру для всех из них или запросим хеш от сложенных строк (пока это псевдо-код, а не часть реальной программы).

```
string properties[];
// заполняем строки свойств по своему желанию
// ...
string signature;
for(int i = 0; i < ArraySize(properties); ++i)
{
   signature += properties[i];
}
return Hash(signature);
```

Полученную сигнатуру пользователь может сообщить разработчику программы, который её особым образом "подпишет", получив строку валидации, подходящую только для этой сигнатуры. "Подпись" также базируется на хешировании и требует знания некоего секрета (пароля-фразы), известного только разработчику и зашитого в программу (для фазы проверки).

Затем разработчик передаст строку валидации пользователю, и тот сможет запускать программу, указав эту строку в параметрах.

При запуске без строки валидации программа должна генерировать новую сигнатуру для текущего окружения, выводить её в журнал и завершать работу (эту информацию полагается передать разработчику). С неверной строкой валидации программа должна выводить сообщение об ошибке и также завершать работу.

Для самого разработчика можно предусмотреть несколько режимов запуска: с сигнатурой, но без строки валидации (чтобы сгенерировать последнюю), или с сигнатурой и строкой валидации (здесь программа подпишет сигнатуру заново и сравнит с заданной строкой валидации — просто для проверки).

Прикинем, насколько избирательной получится такая защита. Ведь все-таки привязка здесь выполняется не к уникальному идентификатору чего-либо.

В следующей таблице приведена статистика по двум характеристикам: размеру экрана и объему оперативной памяти. Очевидно, что значения со временем будут меняться, но примерное распределение будет оставаться таким же: несколько значений характеристик будет наиболее популярно, а некоторые "новые" продвинутые и "старые", уходящие из оборота, составят уменьшающиеся "хвосты".

Screen	**1920x1080**	**1536x864**	**1440x900**	**1366x768**	**800x600**
RAM	**21%**	**7%**	**5%**	**10%**	**4%**
4Gb 20%	4.20	1.40	1.00	2.0	0.8
8Gb 20%	4.20	1.40	1.00	2.0	0.8
16Gb 15%	3.15	1.05	0.75	1.5	0.6
32Gb 10%	2.10	0.70	0.50	1.0	0.4
64Gb 5%	1.05	0.35	0.25	0.5	0.2

Наибольшую обеспокоенность для нас должны представлять ячейки с наибольшими величинами, потому что они означают совпадение сигнатур (если не ввести в них элемент случайности, о чем пойдет речь ниже). В данном случае наиболее вероятны два сочетания характеристик в верхнем левом углу, каждое по 4.2%. Но это только две характеристики. Если добавить в оцениваемое окружение язык интерфейса, таймзону, количество ядер, рабочий путь к данным (желательно общий, так как он содержит имя пользователя Windows), то количество потенциальных совпадений заметно снизится.

Для хеширования воспользуемся встроенной функцией *CryptEncode* (она будет описана в разделе Криптография), поддерживающей метод хеширования SHA256. Согласно его имени, он производит хеш длиной 256 бит, то есть 32 байта. Если бы нам потребовалось показать его пользователю, то при переводе в текст шестандцатеричного представления мы получили бы строку длиной 64 символа.

Чтобы сделать сигнатуру короче, мы преобразуем её с помощью кодировки Base64 (она также поддерживается функцией *CryptEncode* и парной ей *CryptDecode*), что даст строку длиной 44

символа. В отличие от односторонней операции хеширования, кодирование в Base64 обратимо, то есть из него можно восстановить исходные данные.

Основную работу выполняет класс *EnvSignature*. В нем определена строка *data*, где должны накапливаться некие фрагменты, описывающие среду. Публичный интерфейс состоит из нескольких перегруженных версий функции *append* для добавления строк со свойствами среды. По сути, они состыковывают название запрошенного свойства и его значение, используя некий абстрактный элемент, возвращаемый виртуальным методом pepper, в качестве связующего звена. Производный класс определит его как конкретную строку (но она может быть и пустой).

```
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)));
   }
```

Для добавления в объект произвольной строки имеется универсальный метод *append*, который вызывается в вышеприведенных методах.

```
   bool append(const string s)
   {
      data += s;
      return true;
   }
```

При желании, разработчик может добавить в хешируемые данные так называемую "соль". Это массив со случайно генерируемыми данными, необходимый для усложнения подбора хешей. Иными словами, каждая генерация сигнатуры будет отличаться от предыдущей, несмотря на то, что окружение остается постоянным. Реализация данной возможности оставлена для самостоятельного изучения, как и другие более специфические аспекты защиты (такие как применение симметричного шифрования, динамическое вычисление "секрета").

Так как окружение состоит из известных свойств (их список ограничен константами MQL5 API), и не все они обладают достаточной уникальностью, наша защита, как мы подсчитали, может генерировать одинаковые сигнатуры для разных пользователей, если не использовать "соль". Совпадение сигнатур не позволит идентифицировать источник утечки лицензии, если она случилась.

Поэтому повысить эффективность защиты можно за счет изменения способа представления свойств перед хешированием под каждого заказчика. Разумеется, сам способ не должен раскрываться. В рассматриваемой реализации, это сводится к изменению содержимого метода *pepper* и перекомпиляции продукта. Это может быть накладно, но зато позволяет отказаться от случайной "соли".

При заполненной строке свойств, мы можем сгенерировать сигнатуру. Этим занимается метод *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);
}
```

Суть её работы сводится к тому, чтобы добавить к данным некий "секрет" (последовательность байтов, известную только разработчику и внутри программы) и вычислить для общей строки хеш. "Секрет" получается из виртуального метода *secret*, который также определит производный класс.

Полученный байтовый массив с хешем кодируется в строку с помощью Base64.

Теперь у нас на очереди самая главная функция класса — *check*. Именно она "подписывает" сигнатуру у разработчика и проверяет её у пользователя.

```
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;

   /*
      ПРЕДУПРЕЖДЕНИЕ
      Следующий код должен присутствовать только в утилите разработчика.
      Программа, поставляемая пользователю, должна компилироваться без этого 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;
}
```

При штатной работе (у пользователя) метод считает хеш от полученной сигнатуры, дополненной "секретом", и сравнивает его со значением из валидационной строки (её предварительно нужно раскодировать из Base64 в "сырое" бинарное представление хеша). Если два хеша совпали, проверка пройдена успешно: валидационная строка соответствует набору свойств. Очевидно, что пустая валидационная строка (или строка, введенная наобум) проверку не пройдет.

На компьютере разработчика, в исходном коде утилиты для "подписи", должен быть определен макрос I_AM_DEVELOPER, в результате чего пустая валидационная строка обрабатывается иначе. В этом случае полученный хеш кодируется с помощью Base64, и эта строка передается наружу через тот же параметр *validation*. Таким образом, утилита сможет вывести разработчику готовую валидационную строку для заданной сигнатуры.

Для создания объекта необходим конкретный производный класс, в котором определены строки с "секретом" и "перчиком".

```
// ПРЕДУПРЕЖДЕНИЕ: измените макро на собственный набор случайных байтов
#define PROGRAM_SPECIFIC_SECRET "<PROGRAM-SPECIFIC-SECRET>"
// ПРЕДУПРЕЖДЕНИЕ: выберите свои символы для связки в парах name'='value
#define INSTANCE_SPECIFIC_PEPPER "=" // для демо выбран очевидный одиночный знак
// ПРЕДУПРЕЖДЕНИЕ: следующее макро нужно отключить в реальном продукте,
//                 оно должно быть только в утилите подписи
#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;
   }
};
```

Выберем навскидку несколько свойств для заполнения сигнатуры.

```
void FillEnvironment(EnvSignature &env)
{
   // порядок не важен, можно перемешать
   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);
}
```

Теперь все готово для проверки нашей схемы защиты в функции *OnStart*. Но сначала обратим внимание на входные переменные. Поскольку одна и та же программа будет компилироваться в двух вариантах — для конечного пользователя и для разработчика — существует два набора входных переменных: для ввода регистрационных данных пользователем и для генерации этих данных на основе сигнатуры у разработчика. Входные переменные, предназначенные для

разработчика, были описаны выше с помощью макроса INPUT. А для пользователя доступна только строка валидации.

```
input string Validation = "";
```

Когда строка пуста, программа соберет данные об окружении, сгенерирует новую сигнатуру и выведет её в журнал. На этом работа скрипта заканчивается, так как доступ к полезному коду пока не подтвержден.

```
void OnStart()
{
   MyEnvSignature env;
    string signature;
   if(StringLen(Signature) > 0)
   {
     // ... здесь будет код для "подписи" автором
   }
   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
   {
     // ... здесь нужно проверить строку валидации
   }
   Print("The script is validated and running normally");
   // ... фактический рабочий код здесь
}
```

Если переменная *Validation* заполнена, проверяем её соответствие сигнатуре и завершаем работу в случае неудачи.

```
      if(StringLen(Validation) == 0)
      {
         ...
      }
      else
      {
         validation = Validation; // нужен non-const аргумент
         const bool accessGranted = env.check(Signature, validation);
         if(!accessGranted)
         {
            Print("Wrong validation string, terminating");
            return;
         }
         // успех
      }
      Print("The script is validated and running normally");
      // ... фактический рабочий код здесь
   }
```

Если расхождений нет, алгоритм "проваливается" к рабочему коду программы.

На стороне разработчика (в той версии программы, что собрана с макросом I_AM_DEVELOPER) может быть введена сигнатура, и мы по ней восстанавливаем состояние объекта *MyEnvSignature* и рассчитываем строку валидации.

```
   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
      }
      ...
```

Разработчик может указать не только сигнатуру, но и подпись: это приведет к продолжению исполнения кода, как у пользователя (в целях отладки).

При желании вы можете сымитировать изменение окружения, например, таким образом:

```
      FillEnvironment(env);
      // искусственно делаем изменение в среде (добавляем таймзону)
      // env.append("Dummy" + (string)(TimeGMTOffset() - TimeDaylightSavings()));
      const string update = env.emit();
      if(update != signature)
      {
         Print("Signature and environment mismatch");
         return;
      }
```

Рассмотрим несколько тестовых логов.

При первом запуске скрипта *EnvSignature.mq5* "пользователь" увидит примерно следующий лог (значения будут отличаться из-за различий среды):

```
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=
```

Он "отправляет" сгенерированную сигнатуру "разработчику" (в ходе теста это мы сами, поэтому все роли "пользователя" и "разработчика" взяты в кавычки), который вводит её в утилиту подписи (скомпилированную с макросом I_AM_DEVELOPER), в параметр *Signature*. В результате, программа выдаст строку валидации:

```
Validation:YBpYpQ0tLIpUhBslIw+AsPhtPG48b0qut9igJ+Tk1fQ=
```

"Разработчик" "отправляет" её обратно "пользователю", и тот, введя её в параметр *Validation*, получит активированный скрипт:

```
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
```

Для демонстрации действенности защиты давайте продублируем скрипт в виде сервиса: для этого скопируем файл в папку *MQL5/Services/MQL5Book/p4/* и в исходном коде заменим строку:

```
#property script_show_inputs
```

на

```
#property service
```

Откомпилируем сервис, создадим и запустим его экземпляр, а во входных параметрах укажем полученную ранее строку валидации. В результате, сервис прервет работу (не доходя до инструкций с "полезным" кодом) с сообщением:

```
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
```

Дело в том, что среди свойств окружения у нас использована строка MQL_PROGRAM_TYPE. Поэтому выписанная лицензия на один тип программы не подойдет для программы другого типа, даже если она выполняется на компьютере того же пользователя.

4.9.12 Проверка состояния клавиатуры

Функция *TerminalInfoInteger* позволяет узнать состояние управляющих клавиш, называемых также виртуальными. К их числу относятся, в частности, *Ctrl*, *Alt*, *Shift*, *Enter*, *Ins*, *Del*, *Esc*, стрелки для движения курсора и так далее. Виртуальными они называются потому, что на клавиатурах, как правило, имеется несколько способов сгенерировать одно и то же управляющее действие. Например, *Ctrl*, *Shift*, *Alt* дублируются слева и справа от пробела, а движение курсором может выполняться как выделенными клавишами, так и основными при нажатии *Fn*. Таким образом, различить способы управления на физическом уровне (например, левый *Shift* от правого) с помощью данной функции нельзя.

В API определены константы для следующих клавиш.

Идентификатор	Описание
TERMINAL_KEYSTATE_LEFT	Стрелка влево
TERMINAL_KEYSTATE_UP	Стрелка вверх
TERMINAL_KEYSTATE_RIGHT	Стрелка вправо
TERMINAL_KEYSTATE_DOWN	Стрелка вниз
TERMINAL_KEYSTATE_SHIFT	Shift
TERMINAL_KEYSTATE_CONTROL	Ctrl
TERMINAL_KEYSTATE_MENU	Windows
TERMINAL_KEYSTATE_CAPSLOCK	CapsLock
TERMINAL_KEYSTATE_NUMLOCK	NumLock
TERMINAL_KEYSTATE_SCRLOCK	ScrollLock
TERMINAL_KEYSTATE_ENTER	Enter
TERMINAL_KEYSTATE_INSERT	Insert
TERMINAL_KEYSTATE_DELETE	Delete
TERMINAL_KEYSTATE_HOME	Home
TERMINAL_KEYSTATE_END	End
TERMINAL_KEYSTATE_TAB	Tab
TERMINAL_KEYSTATE_PAGEUP	PageUp
TERMINAL_KEYSTATE_PAGEDOWN	PageDown
TERMINAL_KEYSTATE_ESCAPE	Escape

Функция возвращает однобайтовое целочисленное значение, в котором с помощью пары битов сообщается текущее состояние запрошенной клавиши.

Младший бит позволяет отслеживать нажатия клавиши с момента предыдущего вызова функции. Например, если *TerminalInfoInteger(TERMINAL_KEYSTATE_ESCAPE)* вернула в какой-то момент 0, затем пользователь нажал *Escape*, то при следующем вызове *TerminalInfoInteger(TERMINAL_KEYSTATE_ESCAPE)* вернет 1. Если клавиша будет нажата еще раз, то значение снова станет равно 0. И так далее.

Для клавиш, отвечающих за переключение режимов ввода, таких как *CapsLock*, *NumLock*, *ScrollLock*, положение бита указывает, включен ли или выключен соответствующий режим.

Старший бит байта (0x80) взводится в том случае, если клавиша нажата (и не отпущена) в текущий момент.

Данная функция не может использоваться для отслеживания нажатий буквенно-цифровых и функциональных клавиш. Для этой цели необходимо реализовать в программе обработчик *OnChartEvent* и перехватывать сообщения с кодом CHARTEVENT_KEYDOWN. Обратите внимание, что события генерируются на графике и доступны только для экспертов и индикаторов. Программы других типов (скрипты и сервисы) не поддерживают событийную модель программирования.

В скрипте *EnvKeys.mq5* организован цикл по всем константам TERMINAL_KEYSTATE.

```
void OnStart()
{
   for(ENUM_TERMINAL_INFO_INTEGER i = TERMINAL_KEYSTATE_TAB;
      i <= TERMINAL_KEYSTATE_SCRLOCK; ++i)
   {
      const string e = EnumToString(i);
      // пропускаем значения, не являющиеся элементами перечисления
      if(StringFind(e, "ENUM_TERMINAL_INFO_INTEGER") == 0) continue;
      PrintFormat("%s=%2X", e, (uchar)TerminalInfoInteger(i));
   }
}
```

Вы можете поэкспериментировать с нажатиями клавиш и включением/отключением режимов клавиатуры, и понаблюдать за тем, как меняются значения в журнале.

Например, если капитализация по умолчанию отключена, увидим строку:

```
TERMINAL_KEYSTATE_SCRLOCK= 0
```

Если нажать клавишу ScrollLock и, не отпуская, запустить скрипт еще раз, получим запись:

```
TERMINAL_KEYSTATE_CAPSLOCK=81
```

То есть режим уже включен, а клавиша нажата. Отпустим клавишу, и в следующий раз скрипт выдаст:

```
TERMINAL_KEYSTATE_SCRLOCK= 1
```

Режим остался включенным, но клавиша отпущена.

TerminalInfoInteger не подходит для проверки статуса клавиш (TERMINAL_KEYSTATE_XYZ) в зависимых индикаторах созданных вызовом *iCustom* или *IndicatorCreate* — в них функция всегда возвращает 0, даже если индикатор был добавлен на график с помощью *ChartIndicatorAdd*.

Также функция не работает в те моменты, когда график MQL-программы не является активным (то есть пользователь переключился на другой). MQL5 не предоставляет средств для постоянного контроля за клавиатурой.

4.9.13 Проверка статуса и причины остановки MQL-программы

Во многих примерах книги мы уже встречали функцию *IsStopped*. Её необходимо периодически вызывать в тех случаях, когда MQL-программа производит длительные вычисления. Это позволяет проверить, не инициировал ли пользователь закрытие программы (т.е. не попытался ли он удалить её с графика).

bool IsStopped() ≡ bool _StopFlag

Функция возвращает *true*, если программа была прервана пользователем (например, нажатием кнопки Удалить в диалоге, открываемом по команде Список экспертов контекстного меню).

Программе дается 3 секунды на то, чтобы корректно приостановить расчеты, при необходимости сохранить промежуточные результаты, и завершить свою работу. Если этого не произойдет, программа будет снята с графика принудительно.

Вместо функции *IsStopped* можно проверять значение встроенной переменной *_StopFlag*.

Тестовый скрипт *EnvStop.mq5* эмулирует длительные расчеты в цикле: поиск простых чисел. Условие выхода из цикла *while* записано с использованием функции *IsStopped*. Поэтому, когда пользователь удаляет скрипт, цикл прерывается штатным образом и в журнал выводится статистика найденных простых чисел (скрипт мог бы также сохранить сами числа в файл).

```
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()) // попробуйте заменить на while(true)
   {
      // эмулируем длительные расчеты
      if(isPrime(candidate))
      {
         Comment("Count:", ++count, ", Prime:", candidate);
      }
      ++candidate;
      Sleep(10);
   }
   Comment("");
   Print("Total found:", count);
}
```

Если же заменить условие цикла на *true* (бесконечный цикл), скрипт перестанет реагировать на запрос пользователя остановиться и будет выгружен с графика принудительно. В результаты мы увидим в журнале ошибку "Abnormal termination", а комментарий в левом верхнем углу окна остается неочищенным. Таким образом, все инструкции, которые в данном примере символизируют сохранение данных и очистку занятых ресурсов (а это могло бы быть, например, удаление собственных графических объектов из окна), игнорируются.

После того как программе был послан запрос на остановку (и значение *_StopFlag* равно *true*), из неё можно узнать причину остановки с помощью функции *UninitializeReason*.

К сожалению, данная возможность доступна только для экспертов и индикаторов.

int UninitializeReason() ≡ int _UninitReason

Функция возвращает один из предопределенных кодов, описывающих причины деинициализации.

Константа	Значение	Описание
REASON_PROGRAM	0	была вызвана функция *ExpertRemove*, доступная только в советниках и скриптах
REASON_REMOVE	1	программа удалена с графика
REASON_RECOMPILE	2	программа перекомпилирована
REASON_CHARTCHANGE	3	символ или период графика был изменен
REASON_CHARTCLOSE	4	график закрыт
REASON_PARAMETERS	5	входные параметры программы были изменены
REASON_ACCOUNT	6	активирован другой счет либо произошло переподключение к торговому серверу
REASON_TEMPLATE	7	применен другой шаблон графика
REASON_INITFAILED	8	обработчик события *OnInit* вернул признак ошибки
REASON_CLOSE	9	терминал был закрыт

Вместо функции можно обращаться к встроенной глобальной переменной *_UninitReason*.

Код причины деинициализации передается также в качестве параметра функции-обработчика события *OnDeinit*.

В дальнейшем при изучении Особенности запуска и остановки программ разных типов будут представлены индикатор (*Indicators/MQL5Book/p5/LifeCycle.mq5*) и эксперт (*Experts/MQL5Book/p5/LifeCycle.mq5*), которые выводят в журнал причины деинициализации и позволяют исследовать поведение программ в зависимости от действий пользователя.

4.9.14 Программное закрытие терминала и код возврата

MQL5 API содержит несколько функций, которые позволяют не только читать, но изменять программное окружение. Одна из самых радикальных из них — *TerminalClose*. С помощью неё MQL-программа может закрыть терминал (без подтверждения пользователя!).

bool TerminalClose(int retcode)

Функция имеет один параметр *retcode* — код, возвращаемый процессом terminal64.exe в операционную систему Windows. Подобные коды можно анализировать в командных файлах (*.bat и *.cmd), а также в скриптах оболочки (Windows Script Host (WSH), поддерживающий VBScript и JScript, или Windows PowerShell (WPS), с файлами расширений .ps*) и прочих средствах автоматизации (например, встроенный планировщик Windows, подсистема поддержки Linux под Windows с файлами *.sh, и т.д.).

Функция не производит немедленной остановки работы терминала, а просто посылает терминалу команду на завершение.

Если результат вызова равен *true*, значит команда успешно "принята к рассмотрению", и терминал постарается максимально быстро, но корректно закрыться (с уведомлением и

остановкой прочих выполняющихся MQL-программ). В вызывающем коде, разумеется, также должны быть сделаны все приготовления для немедленного завершения работы (в частности, закрыты все ранее открытые файлы), а после вызова функции управление следует вернуть терминалу.

Другая функция, связанная кодом возврата процесса, называется *SetReturnError*. Она позволяет заранее назначить этот код, не отправляя команду на немедленное закрытие.

void SetReturnError(int retcode)

Функция устанавливает код, который процесс терминала вернёт системе Windows при завершении работы.

Важно отметить, что терминал не обязательно должен быть принудительно закрыт функцией *TerminalClose*. Штатное закрытие терминала пользователем также произойдет с указанным кодом. Также данный код попадет в систему, если терминал закроется из-за непредвиденной критической ошибки.

Если функция *SetReturnError* вызывалась многократно и/или из разных MQL-программ, то терминал вернёт последний установленный код.

Для тестирования обеих функций написан скрипт *EnvClose.mq5*.

```
#property script_show_inputs

input int ReturnCode = 0;
input bool CloseTerminalNow = false;

void OnStart()
{
   if(CloseTerminalNow)
   {
      TerminalClose(ReturnCode);
   }
   else
   {
      SetReturnError(ReturnCode);
   }
}
```

Чтобы проверить его в действии нам также потребуется командный файл *envrun.bat* (с книгой он поставляется в папке *MQL5/Files/MQL5Book/*).

```
terminal64.exe
@echo Exit code: %ERRORLEVEL%
```

По сути он только запускает терминал, а после его завершения выводит полученный код в консоль. Файл следует расположить в папке рядом с терминалом (или актуальный экземпляр MetaTrader 5 из числа нескольких установленных в системе должен быть прописан в системной переменной PATH).

Например, если запустить терминал с помощью bat-файла и выполнить скрипт *EnvClose.mq5*, скажем, с параметрами *ReturnCode=100*, *CloseTerminalNow=true*, увидим в консоли примерно такие строки:

```
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>
```

Напомним, что MetaTrader 5 поддерживает различные опции при запуске из командной строки (подробности см. в разделе документации Запуск торговой платформы). Таким образом, можно организовать, например, пакетное тестирование различных экспертов или настроек, а также последовательное переключение между тысячами контролируемых счетов, что было бы нереально сделать при постоянной параллельной работе такого количества экземпляров на одном компьютере.

4.9.15 Обработка ошибок времени исполнения программы

Любая программа, составленная достаточно корректно, чтобы компилироваться без ошибок, все равно не застрахована от ошибок во время работы. Они могут возникать как по недосмотру разработчика, так и вследствие непредвиденных обстоятельств, возникших в программной среде ("пропал" Интернет, кончилась память). Но не менее вероятна и ситуация, когда ошибка возникает из-за неправильного применения программы: ведь, документацию пользователи читают в последнюю очередь. Во всех этих случаях программа должна иметь возможность проанализировать суть проблемы и адекватно её обработать.

Каждая инструкция MQL5 является потенциальным источником ошибок выполнения, и если таковые случаются, терминал сохраняет в специальную переменную *_LastError* описательный код. Разумеется, анализировать код следует непосредственно после каждой инструкции, поскольку потенциальные ошибки в последующих инструкциях могут перезаписать это значение на свое.

Обратите внимание, что существует ряд критических ошибок, при возникновении которых выполнение программы немедленно прерывается:

- деление на ноль;
- выход индекса за пределы массива;
- использование некорректного указателя объекта;

Полный перечень кодов ошибок, и что они значат, приведен в документации.

В разделе Открытие и закрытие файлов мы уже обращались к проблеме диагностики ошибок в рамках написания полезного макроса PRTF. Там, в частности, был введен вспомогательный заголовочный файл *MQL5/Include/MQL5Book/MqlError.mqh*, в котором перечисление MQL_ERROR позволяет простым способом преобразовать цифровой код ошибки в её имя с помощью *EnumToString*.

```
enum MQL_ERROR
{
   SUCCESS = 0,
   INTERNAL_ERROR = 4001,
   WRONG_INTERNAL_PARAMETER = 4002,
   INVALID_PARAMETER = 4003,
   NOT_ENOUGH_MEMORY = 4004,
   ...
   // начало области для ошибок, определенных программистом (см. след. раздел)
   USER_ERROR_FIRST = 65536,
};
#define E2S(X) EnumToString((MQL_ERROR)(X))
```

Здесь в качестве параметра *X* макроса *E2S* как раз должна выступать переменная *_LastError* или эквивалентная ей функция *GetLastError*.

int GetLastError() ≡ int _LastError

Функция возвращает код последней ошибки, произошедшей в инструкциях MQL-программы. Изначально, в отсутствие ошибок, значение равно 0. Разница между чтением *_LastError* и вызовом функции *GetLastError* — чисто синтаксическая (выбирайте подходящий вариант в соответствие с предпочтительным стилем).

Следует иметь в виду, что штатное безошибочное выполнение инструкций не сбрасывает код ошибки. Обращение к *GetLastError* также этого не делает.

Таким образом, если есть последовательность действий, из которых лишь одно выставит признак ошибки, этот признак будет возвращаться функцией и для последующих (успешных) действий. Например,

```
// _LastError = 0 по умолчанию
действие1; // ok, _LastError не меняется
действие2; // ошибка, _LastError = X
действие3; // ok, _LastError не меняется, то есть по-прежнему равен X
действие4; // другая ошибка, _LastError = Y
действие5; // ok, _LastError не меняется, то есть по-прежнему равен Y
действие6; // ok, _LastError не меняется, то есть по-прежнему равен Y
```

Такое поведение затруднило бы локализацию проблемного места, поэтому для сброса переменной *_LastError* в 0 существует отдельная функция *ResetLastError*.

void ResetLastError()

Функция устанавливает значение встроенной переменной *_LastError* в ноль.

Функцию рекомендуется вызывать перед любым действием, которое может привести к ошибке, и после которого планируется проводить анализ с помощью *GetLastError*.

Хорошим примером использования обеих функций является уже упомянутый макрос PRTF (файл PRTF.mqh). Напомним его код:

```
#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; // зафиксируем _LastError на входе
   const string err = E2S(snapshot) + "(" + (string)snapshot + ")";
   Print(s, "=", retval, " / ", (snapshot == 0 ? "ok" : err));
   ResetLastError(); // очистка признака ошибки для следующих вызовов
   return retval;
}
```

Задача макроса и обернутой в него функции *ResultPrint*, вывести в журнал переданное значение, текущий код ошибки и тут же очистить код ошибки. Таким образом, последовательное применение PRTF на ряде инструкций всегда гарантирует, что выведенная в журнал ошибка (или признак успеха) соответствует именно последней инструкции, с помощью которой было получено значение параметра *retval*.

Сохранение *_LastError* в промежуточной локальной переменной *snapshot* потребовалось по той причине, что *_LastError* может изменить свое значение практически в любом месте вычисления выражения, если какая-либо операция выполнится с ошибкой. В частности, здесь в макросе E2S используется функция *EnumToString*, которая может взвести свой код ошибки, если в качестве аргумента передано значение, отсутствующее в перечислении. Тогда в последующих частях того же выражения при формировании строки будет фигурировать уже не изначальная ошибка, а наведенная.

В любой инструкции может оказаться несколько мест, где *_LastError* вдруг изменится. В связи с этим желательно зафиксировать код ошибки сразу после интересующего нас действия.

4.9.16 Пользовательские ошибки

Разработчик может использовать встроенную переменную *_LastError* в собственных, прикладных целях. Это позволяет сделать функция *SetUserError*.

void SetUserError(ushort user_error)

Функция устанавливает встроенную переменную *_LastError* в значение *ERR_USER_ERROR_FIRST + user_error*, где ERR_USER_ERROR_FIRST равно 65536. Все коды ниже этой величины зарезервированы за системными ошибками.

С помощью данного механизма можно отчасти обойти ограничение MQL5, связанное с тем, что в языке не поддерживаются исключения.

Довольно часто функции используют возвращаемое значение как признак ошибки. Однако бывают алгоритмы, в которых функция должна возвращать величину прикладного типа. Допустим речь о *double*. Если функция имеет область определения от минус до плюс бесконечности, любое значение, выбранное нами для индикации ошибки (например, 0), будет невозможно отличить от реального результата вычисления. В случае *double*, конечно, есть вариант вернуть специально сконструированное значение NaN (Not a Number, см. раздел Проверка вещественных чисел на нормальность). Но что если функция возвращает структуру или объект класса? Одно из решений

— возвращать результат через параметр по ссылке или указателю, но такая форма делает невозможным использование функций в качестве операндов выражений.

В контексте разговора о классах уместно вспомнить об особых функциях — конструкторах. Они обязаны вернуть новый экземпляр объекта. Однако иногда обстоятельства мешают сконструировать объект полностью, и тогда вызывающий код вроде бы и получает объект, но не должен им пользоваться. Хорошо, если в классе можно предусмотреть дополнительный метод, который позволил бы проверить полноценность объекта. Но в качестве единообразного альтернативного подхода (например, охватывающего все классы) можно задействовать *SetUserError*.

В разделе Перегрузка операторов мы познакомились с классом *Matrix*. Дополним его методами расчета определителя и обратной матрицы, а затем используем для демонстрации пользовательских ошибок (см. файл *Matrix.mqh*). Напомним, что для матриц были определены перегруженные операторы, позволяющие составлять их в цепочки операторов, в едином выражении, и потому внедрять в него проверку на потенциальные ошибки было бы неудобно.

> Наш класс *Matrix* является собственной альтернативной реализацией для недавно добавленного в MQL5 встроенного объектного типа *matrix*.

Прежде всего, в конструкторах основного класса *Matrix* сделаем проверку на корректность входных параметров. Если кто-то попытается создать матрицу нулевого размера, установим пользовательскую ошибку ERR_USER_MATRIX_EMPTY (одну из нескольких предусмотренных).

```
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);
      }
   }
```

Вышеупомянутые новые операции определены только для квадратных матриц, поэтому создадим производный класс с соответствующим ограничением на размеры.

```
class MatrixSquare : public Matrix
{
public:
   MatrixSquare(const int n, const int _ = -1) : Matrix(n, n)
   {
      if(_ != -1 && _ != n)
      {
         SetUserError(ERR_USER_MATRIX_NOT_SQUARE);
      }
   }
   ...
```

Второй параметр в конструкторе, по логике вещей, должен отсутствовать (подразумевается равным первому), но он нам потребовался, потому что в классе *Matrix* появился шаблонный метод для транспонирования, в котором все типы T должны поддерживать конструктор с двумя целочисленными параметрами.

```
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;
   }
```

Из-за того, что параметров в конструкторе *MatrixSquare* два, нам также приходится проверять их на обязательное равенство, и если оно нарушено, устанавливаем ошибку ERR_USER_MATRIX_NOT_SQUARE.

Наконец, в ходе вычисления обратной матрицы мы можем обнаружить, что матрица вырождена (определитель равен 0). На этот случай зарезервирована ошибка ERR_USER_MATRIX_SINGULAR.

```
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();
   }
   ...
```

Для наглядного вывода ошибок в журнал добавлен статический метод, возвращающий перечисление ENUM_ERR_USER_MATRIX, которое легко передать в *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;
   }
```

С полным кодом всех методов можно ознакомиться в прилагаемом файле.

Проверку кодов прикладных ошибок проведем в тестовом скрипте *EnvError.mq5*.

Сначала убедимся в работоспособности класса: инвертируем матрицу и проверим, что произведение исходной и инвертированной дает единичную матрицу.

```
void OnStart()
{
   Print("Test matrix inversion (should pass)");
   double a[9] =
   {
      1,  2,  3,
      4,  5,  6,
      7,  8,  0,
   };

   ResetLastError();
   MatrixSquare mA(a);   // присваиваем данные в исходную матрицу
   Print("Input");
   mA.print();
   MatrixSquare mAinv(3);
   mAinv = !mA;          // инвертируем и сохраняем в другой матрице
   Print("Result");
   mAinv.print();

   Print("Check inverted by multiplication");
   MatrixSquare test(3); // умножаем первую на вторую
   test = mA * mAinv;
   test.print();         // получаем единичную матрицу
   Print(EnumToString(Matrix::lastError())); // ok
   ...
```

Этот фрагмент кода генерирует следующие записи в журнале.

```
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
```

Обратите внимание, что в единичной матрице из-за погрешностей вычислений с плавающей точкой некоторые нулевые элементы на самом деле представляют собой очень малые, близкие к нулю величины и потому имеют знаки.

Затем посмотрим, как алгоритм справится с вырожденной матрицей.

```
   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
   ...
```

Результаты представлены ниже.

```
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
```

В данном случае мы просто выводим описание ошибки, но в реальной программе следует предусмотреть выбор варианта продолжения, в зависимости от сути проблемы.

Наконец, смоделируем ситуации для двух оставшихся прикладных ошибок.

```
   Print("Empty matrix creation");
   MatrixSquare m0(0);
   Print(EnumToString(Matrix::lastError()));
   
   Print("'Rectangular' square matrix creation");
   MatrixSquare r12(1, 2);
   Print(EnumToString(Matrix::lastError()));
}
```

Здесь мы описываем пустую матрицу и якобы квадратную матрицу, но с отличающимися размерами.

```
Empty matrix creation
ERR_USER_MATRIX_EMPTY
'Rectangular' square matrix creation
ERR_USER_MATRIX_NOT_SQUARE
```

В этих случаях мы не можем не создать объект, поскольку компилятор делает это автоматически.

Конечно, в данном тесте очевидны нарушения контрактов (спецификаций того, что классы и методы "считают" корректными данными и действиями). Однако на практике аргументы часто получаются из других частей кода, в ходе обработки объемных, "чужих" данных, и обнаружить отклонения от ожиданий — не такая уж тривиальная задача.

Способность программы "переваривать" некорректные данные без фатальных последствий является важнейшим показателем её качества, наравне с выдачей правильных результатов для правильных входных данных.

4.9.17 Управление отладкой

Встроенный в MetaEditor отладчик позволяет устанавливать в исходном коде точки останова — строки, на которых исполнение программы должно приостанавливаться. Иногда эта система дает сбой, то есть пауза не срабатывает, и тогда можно воспользоваться функцией *DebugBreak*, которая прописывает остановку в явном виде.

void DebugBreak()

Вызов функции "замораживает" программу и делает активным окно редактора в режиме отладки, со всеми средствами просмотра переменных, стека вызовов и дальнейшего пошагового исполнения.

Прерывание выполнения программы происходит только в том случае, если программа запущена из редактора в режиме отладки (командами *Отладка -> Начать на реальных данных* или *Начать на исторических данных*). Во всех других режимах — штатный запуск (в терминале) или на профилирование — функция не оказывает никакого эффекта.

4.9.18 Предопределенные переменные

В каждой MQL-программе доступен некоторый общий набор глобальных переменных, предоставляемых терминалом: большинство из них мы уже рассмотрели в предыдущих разделах, а ниже приведем сводную таблицу. Практически все переменные доступны только на чтение. Исключение составляет переменная *_LastError*, которая может быть обнулена функцией *ResetLastError*.

Переменная	Значение
_LastError	Значение последней ошибки, аналог функции *GetLastError*
_StopFlag	Флаг остановки программы, аналог функции *IsStopped*
_UninitReason	Код причины деинициализации программы, аналог функции *UninitializeReason*
_RandomSeed	Текущее внутреннее состояние генератора псевдослучайных целых чисел
_IsX64	Признак 64-битного терминала, аналог *TerminalInfoInteger* для свойства TERMINAL_X64

Кроме того, для MQL-программ, выполняющихся в контексте графика — экспертов, скриптов и индикаторов, — поддерживается ряд предопределенных переменных со свойствами графика (они также не могут быть изменены из программы).

Переменная	Значение
_Symbol	Имя символа текущего графика, аналог функции *Symbol*
_Period	Значение таймфрейма текущего графика, аналог функции *Period*
_Digits	Количество десятичных знаков после запятой в цене символа текущего графика, аналог функции *Digits*
_Point	Размер пункта в ценах текущего символа (в валюте котировки), аналог функции *Point*
_AppliedTo	Тип данных, на которых рассчитывается индикатор (только для индикаторов)

4.9.19 Предопределенные константы языка MQL5

В данном разделе собраны все константы, определенные средой исполнения для любой программы. Часть из них мы уже видели в предыдущих разделах, а часть относится к прикладным аспектам программирования на MQL5, которые представлены в более поздних главах.

Константа	Описание	Значение
CHARTS_MAX	Максимально возможное количество одновременно открытых графиков	100
clrNONE	Отсутствие цвета	-1 (0xFFFFFFFF)
EMPTY_VALUE	Пустое значение в индикаторном буфере	DBL_MAX
INVALID_HANDLE	Некорректный дескриптор	-1
NULL	Ноль любого типа	0
WHOLE_ARRAY	Означает количество элементов, оставшееся до конца массива, то есть, будет обработан весь массив	-1
WRONG_VALUE	Константа может неявно приводиться к типу любого перечисления	-1

Как было покаано в главе про файлы, константа INVALID_HANDLE может использоваться для проверки файловых дескрипторов на корректность.

Константа WHOLE_ARRAY предназначена для функций работы с массивами, которые требуют указания количества элементов в обрабатываемых массивах: если необходимо обработать все значения массива с указанной позиции и до конца, то достаточно указать значение WHOLE_ARRAY.

Константа EMPTY_VALUE обычно присваивается тем элементам в индикаторных буферах, которые не должны отрисовываться на графике. Иными словами, данная константа обозначает пустое значение, используемое по умолчанию. Позднее мы расскажем, как его можно заменить для конкретного индикаторного буфера на другое значение, например, 0.

Константа WRONG_VALUE предназначена для тех случаев, когда требуется обозначить некорректное значение перечисления.

Кроме того существует 2 константы, которые имеют разные значения в зависимости от способа компиляции.

Константа	Описание
IS_DEBUG_MODE	Признак работы mq5-программы в режиме отладки: в режиме отладки не равно нулю, в противном случае 0
IS_PROFILE_MODE	Признак работы mq5-программы в режиме профилирования: в режиме профилирования не равно нулю, в противном случае 0

Константа IS_PROFILE_MODE позволяет изменить работу программы для корректного сбора информации в режиме профилирования. Профилирование позволяет замерить время выполнения отдельных фрагментов программы (функций и отдельных строк).

Значение константы IS_PROFILE_MODE задается компилятором в момент компиляции, и в обычном режиме выставляется равным 0. При запуске программы в режиме профилирования производится специальная компиляция, и в этом случае вместо IS_PROFILE_MODE подставляется значение отличное от 0.

Константа IS_DEBUG_MODE работает по похожему принципу: равна 0 в результате штатной компиляции и больше 0 после компиляции для отладки. Она пригодится в тех случаях, когда необходимо немного изменить работу MQL-программы для проверки: например, вывести дополнительную информацию в журнал или создавать вспомогательные графические объекты на графике.

Напомним, что препроцессор определяет похожие по смыслу константы _DEBUG и _RELEASE (см. Предопределенные константы препроцессора).

Более подробную информацию о режиме работы программы можно узнать во время выполнения с помощью функции *MQLInfoInteger* (см. Режимы работы терминала и программы). В частности, отладочная сборка программы может быть запущена и не под отладчиком.

4.10 Матрицы и векторы

Для решения большого класса математических задач в язык MQL5 встроены специальные объектные типы данных — матрицы и векторы. Эти типы предоставляют методы для написания краткого и понятного кода, который близок к математической записи линейных или дифференциальных уравнений.

В каждом языке программирования существует понятие массива, как набора множества элементов. На основе массивов числовых типов (*int*, *double*) или, может быть, структур, строится большинство алгоритмов, особенно в алготрейдинге. Доступ к элементам массива осуществляется по индексу, что позволяет производить операции внутри циклов. Как мы знаем, массивы могут иметь одно, два или более измерений.

Для относительно простых задач хранения и обработки данных возможностей массивов обычно хватает. Но когда дело доходит до комплексных математических задач, из-за большого количества вложенных циклов работа с массивами становится сложной в плане как программирования, так и

чтения кода. Даже самые простые операции линейной алгебры требуют большого количества кода и хорошего понимания математики. Упростить эту задачу позволяет функциональная парадигма программирования, воплощенная в виде функций-методов матриц и векторов, выполняющих массу рутинных действий "за сценой".

Современные технологии, такие как машинное обучение, нейронные сети и 3D-графика, широко используют решения задач из линейной алгебры, в которой применяются операции над векторами и матрицами. Именно поэтому для быстрой и удобной работы с такими объектами в MQL5 были добавлены новые типы данных.

> Во время написания книги набор функций для работы с матрицами и векторами активно пополнялся, поэтому многие интересные новинки могут оказаться не упомянутыми здесь. Следите за анонсами релизов и разделом статей на сайте *mql5.com*.

В этой главе мы дадим краткое описание, а подробную информацию о матрицах и векторах смотрите в соответствующем разделе справки Методы матриц и векторов.

Также предполагается, что читатель знаком с теорией линейной алгебры. При необходимости, обращайтесь к справочной литературе и пособиям в Сети.

4.10.1 Типы матриц и векторов

Вектор — это одномерный массив вещественного или комплексного типа, а матрица — двумерный массив вещественного или комплексного типа. Таким образом, перечень допустимых числовых типов для элементов этих объектов включает *double* (считается типом по умолчанию), *float* и *complex*.

С точки зрения линейной алгебры (но не компилятора!) простое число также является минимальным вектором, а вектор, в свою очередь, можно рассматривать, как частный случай матрицы.

Вектор, в зависимости от типа элементов, описывается с помощью одного из ключевых слов *vector* (с суффиксом или без):

- *vector* — вектор с элементами типа *double*;
- *vectorf* — вектор с элементами типа *float*;
- *vectorc* — вектор с элементами типа *complex.*

Хотя векторы бывают вертикальными и горизонтальными, в MQL5 подобное разделение не делается. Необходимая ориентация вектора определяется (подразумевается) местом вектора в выражении.

Над векторами определены операции сложения и умножения, а также введено понятие "Норма" (и соответствующий метод *Norm*) для получения длины или модуля вектора.

Матрицу можно представить как массив, где первый индекс означает номер строки, а второй индекс — номер столбца. Однако нумерация строк и столбцов, в отличие от линейной алгебры, начинается с нуля, как и в массивах.

$$A=\begin{pmatrix} a_{11}\ a_{12} \dots a_{1n} \\ a_{21}\ a_{22} \dots a_{2n} \\ \dots\dots\dots\dots \\ a_{m1}\ a_{m2} \dots a_{mn} \end{pmatrix}$$

Две размерности матриц называются также осями (axes) и нумеруются следующим образом: 0 — горизонтальная ось (вдоль строк), 1 — вертикальная ось (вдоль столбцов). Номера осей используются во многих матричных функциях. В частности, когда мы будем говорить о разбиении матрицы на фрагменты, то разбиение по горизонтали означает "разрез" между (или, если угодно, вдоль) строк, а разбиение по вертикали — разрез между столбцов.

В зависимости от типа элементов, матрица описывается с помощью одного из ключевых слова *matrix* (с суффиксом или без):

- *matrix* — матрица с элементами типа *double*;
- *matrixf* — матрица с элементами типа *float*;
- *matrixc* — матрица с элементами типа *complex.*

Для применения в шаблонных функциях можно использовать запись *matrix<double>, matrix<float>, matrix<complex>, vector<double>, vector<float>, vector<complex>* вместо соответствующих типов.

```
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);
}
```

При выводе в журнал матрицы и векторы печатаются как последовательности чисел, разделенных запятыми и заключенных в квадратные скобки.

```
[0,1,2,3]
[[0,1]
 [2,3]]
```

Для матрицы определены следующие алгебраические операции:

- сложение матриц, имеющих одинаковый размер;
- умножение матриц подходящего размера, при этом число столбцов матрицы слева должно соответствовать числу строк матрицы справа;
- умножение матрицы на вектор-столбец и умножение вектора-строки на матрицу по правилу матричного умножения (вектор является в этом смысле частным случаем матрицы);
- умножение матрицы на число.

Кроме того, типы *matrix* и *vector* имеют встроенные методы, которые соответствуют аналогам библиотеки NumPy (популярный пакет для машинного обучения на Python), поэтому вы можете

получить дополнительные подсказки в документации и примерах для этой библиотеки. Полный перечень методов можно найти в соответствующем разделе справки по MQL5.

К сожалению, в MQL5 не предусмотрено приведение матриц и векторов одного типа к другому (например, от *double* к *float*). Кроме того, вектор автоматически не трактуется компилятором как матрица (с одним столбцом или строкой) в тех выражениях, где ожидается матрица. То есть, принципы наследования (характерные для ООП) между матрицами и векторами не существуют, несмотря на кажущуюся родственную связь между этими "структурами".

4.10.2 Создание и инициализация матриц и векторов

Предусмотрено несколько способов объявления и инициализации матриц и векторов. Их можно условно разделить на несколько категорий по своему назначению:

- объявление без указания размера;
- объявление с указанием размера;
- объявление с инициализацией;
- статические методы создания;
- нестатические методы (пере-)конфигурирования и инициализации.

Самый простой метод создания — это объявление без указания размера, то есть без распределения памяти для данных. Для этого достаточно указать тип и имя переменной:

```
matrix         matrix_a;   // матрица типа double
matrix<double> matrix_a1;  // матрица типа double внутри шаблонов функций или классов
matrix<float>  matrix_a2;  // матрица типа float
vector         vector_v;   // вектор типа double
vector<double> vector_v1;  // другая запись создания вектора типа double
vector<float>  vector_v2;  // вектор типа float
```

Далее вы можете изменить размер созданных объектов и заполнить нужными значениями. Также их можно использовать во встроенных методах матриц и векторов для получения результатов вычислений. Все эти методы будут рассмотрены по группам в разделах этой главы.

Можно объявить матрицу или вектор с указанием размера. При этом произойдет распределение памяти, но без какой-либо инициализации. Для этого после имени переменной в круглых скобках задаем размер(ы) (для матрицы — сначала количество строк, затем количество столбцов):

```
matrix         matrix_a(128, 128);      // в качестве параметров можно указывать
matrix<double> matrix_a1(nRows, nCols); //       как константы, так и переменные
matrix<float>  matrix_a2(nRows, 1);     // аналог вектора-столбца
vector         vector_v(256);
vector<double> vector_v1(nSize);
vector<float>  vector_v2(nSize + 16);   // выражение в качестве параметра
```

Третий способ создания объектов — объявление с инициализацией. Размеры матриц и векторов в этом случае определяются инициализирующей последовательностью, указанной в фигурных скобках:

```
matrix         matrix_a = {{0.1, 0.2, 0.3}, {0.4, 0.5, 0.6}};
matrix<double> matrix_a1 = matrix_a;    // должны быть матрицы одного и того же типа
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;   // должны быть векторы одного и того же типа
```

Существуют также статические методы создания матриц и векторов указанного размера с инициализацией определённым способом (под ту или иную каноническую форму). Все они приведены в списке ниже и имеют схожие прототипы (векторы отличаются от матриц только отсутствием второго размера).

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* — создает матрицу с единицами по указанной диагонали и нулями в остальных местах;
- *Tri* — создает матрицу с единицами по указанной диагонали и ниже нее, и нулями в остальных местах;
- *Identity* — создает единичную матрицу указанного размера;
- *Ones* — создает матрицу (или вектор), заполненную единицами;
- *Zeros* — создает матрицу (или вектор), заполненную нулями;
- *Full* — создает матрицу (или вектор), заполненную заданным значением во всех элементах.

При необходимости можно "превратить" любую уже существующую матрицу в единичную, для чего к ней следует применить нестатический метод *Identity* (без параметров).

Продемонстрируем методы в действии:

```
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);
```

Кроме того, существуют нестатические методы для инициализации матрицы/вектора заданными значениями — *Init* и *Fill*.

void matrix<T>::Init(const ulong rows, const ulong cols, func_reference rule = NULL, ...)
void matrix<T>::Fill(const T value)

Важным достоинством метода *Init* (впрочем, как и конструкторов) является возможность указать в параметрах инициализирующую функцию для заполнения элементов матрицы/вектора по заданному закону (см. пример чуть ниже).

Ссылку на такую функцию можно передать после размеров, указав её идентификатор без кавычек в параметре *rule* (это не указатель в смысле *typedef (*pointer)(...)* и не строка с именем).

Инициализирующая функция должна иметь первым параметром ссылку на заполняемый объект, а также может иметь дополнительные параметры: в этом случае значения для них передаются в *Init*

или конструктор после ссылки на саму функцию. Без указания *rule*-ссылки будет просто создана матрица заданных размеров.

Метод *Init* позволяет заодно изменить конфигурацию матрицы.

Покажем все изложенное на небольших примерах.

```
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]]
*/
```

Здесь метод *Init* был использован для изменения размеров уже инициализированной матрицы, что привело к заполнению новых элементов случайными значениями.

А следующая функция заполняет матрицу числами, увеличивающимся в геометрической прогрессии:

```
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;
      }
   }
}
```

Тогда её можно применить для создания матрицы.

```
void OnStart()
{
   matrix M(3, 6, MatrixSetValues);
   Print("M = \n", M);
}
```

Результат выполнения:

```
M =
[[1,2,4,8,16,32]
 [64,128,256,512,1024,2048]
 [4096,8192,16384,32768,65536,131072]]
```

В данном случае значения для параметра инициализирующей функции не были указаны следом за её идентификатором в вызове конструктора, и потому использовалось значение по умолчанию (1). Но мы можем, например, для той же *MatrixSetValues* передать стартовое значение -1, что приведет к заполнению матрицы отрицательным рядом.

```
matrix M(3, 6, MatrixSetValues, -1);
```

4.10.3 Копирование матриц, векторов и массивов

Наиболее простой и привычный способ копирования матриц и векторов — через оператор присваивания '='.

```
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);
```

Данный фрагмент генерирует в журнале такие записи:

```
matrix a
[[2,2]
 [3,3]
 [4,4]]
matrix b
[[4,4]
 [5,5]
 [6,6]]
matrix c
[[4,4]
 [5,5]
 [6,6]]
```

Однако для копирования матриц и векторов предназначена и пара методов: *Copy* и *Assign*. Отличие метода *Assign* от *Copy* в том, что он позволяет копировать не только матрицы, но и массивы.

bool matrix<T>::Copy(const matrix<T> &source)
bool matrix<T>::Assign(const matrix<T> &source)
bool matrix<T>::Assign(const T &array[])

Аналогичные по действию и прототипам методы реализованы и для векторов.

Через *Assign* возможна запись вектора в матрицу: результатом будет однострочная матрица.

bool matrix<T>::Assign(const vector<T> &v)

Также вектору можно назначить матрицу: будет выполнена её развёртка, то есть все строки матрицы выстраиваются в один ряд (эквивалентно вызову метода *Flat*).

bool vector<T>::Assign(const matrix<T> &m)

На момент написания данной главы в MQL5 не существует метода для экспорта матрицы или вектора в массив, хотя есть механизм для "переноса" данных (см. далее метод *Swap*).

Пример ниже показывает, как целочисленный массив *int_arr* копируется в матрицу типа *double*. При этом результирующая матрица автоматически подстраивается под размеры копируемого массива.

```
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);
```

Получим в журнале следующий вывод.

```
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]]
```

Таким образом, метод *Assign* позволяет переходить в коде от массивов к матрицам с автоматическим приведением размера и типа.

Более эффективно (быстро и без копирования) можно переносить данные между матрицами, векторами и массивами с помощью методов *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[])
```

Они работают по такому же принципу, как и функция *ArraySwap*: местами меняются внутренние указатели на буфера с данными внутри двух объектов. В результате, элементы матрицы или вектора исчезают в исходном объекте и появляются в принимающем массиве, или наоборот "перетекают" из массива в матрицу или вектор.

Метод *Swap* позволяет работать с динамическими массивами, в том числе и с многомерными. На константные размеры старших измерений многомерного массива (*array[][N1][N2]...*) налагается условие: произведение этих размеров должно быть кратно размеру матрицы или вектора. Так массив с геометрией [][2][3] перераспределяется блоками по 6 элементов. Следовательно, он совместим по обмену с матрицами и векторами размерами 6, 12, 18 и т.д.

4.10.4 Копирование таймсерий в матрицу или вектор

Метод *matrix<T>::CopyRates* копирует временные ряды с историей котировок прямо в матрицу или вектор. Данный метод работает по аналогии с функциями, которые мы подробно рассмотрим в 5-ой Части, в главе о временных рядах, а именно: *CopyRates* и раздельные *Copy-функции* для каждого поля структуры *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)
```

В параметрах требуется указать символ, таймфрейм, а также диапазон запрашиваемых баров: либо по номеру и количеству, либо по диапазону дат. Данные копируются таким образом, что самый старый по времени элемент помещается в начало матрицы/вектора.

Параметр *rates_mask* предназначен для указания комбинации флагов из перечисления ENUM_COPY_RATES с набором доступных полей. Комбинация флагов позволяет за один запрос получить несколько временных рядов из истории. При этом порядок строк в матрице будет соответствовать порядку значений в перечислении ENUM_COPY_RATES, в частности, строка с данными *High* будет в матрице всегда выше строки с данными *Low*.

При копировании в вектор можно указать только одно значение из перечисления ENUM_COPY_RATES, иначе будет ошибка.

Идентификатор	Значение	Описание
COPY_RATES_OPEN	1	цены *Open*
COPY_RATES_HIGH	2	цены *High*
COPY_RATES_LOW	4	цены *Low*
COPY_RATES_CLOSE	8	цены *Close*
COPY_RATES_TIME	16	времена открытия баров
COPY_RATES_VOLUME_TICK	32	тиковые объемы
COPY_RATES_VOLUME_REAL	64	реальные объемы
COPY_RATES_SPREAD	128	спреды
		Комбинации
COPY_RATES_OHLC	15	*Open, High, Low, Close*
COPY_RATES_OHLCT	31	*Open, High, Low, Close, Time*

Мы покажем пример использований этой функции в разделе о Решении уравнений.

4.10.5 Копирование истории тиков в матрицу или вектор

Аналогично барам вы можете копировать в матрицу или вектор и тики: для этих целей предоставлена пара перегрузок методов *CopyTicks* и *CopyTicksRange*. Они работают по принципу, схожему с функциями *CopyTicks* и *CopyTicksRange*, но получают данные в вызывающий объект. Эти функции будут подробно описаны в 5-ой главе, в разделе о работе с массивами реальных тиков в структурах *MqlTick*. Здесь мы лишь покажем прототипы и упомянем основные моменты.

```
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)
```

Параметр *symbol* задает название финансового инструмента, тики которого запрашиваются. Диапазон тиков можно указать по-разному:

- в *CopyTicks* — как количество тиков (параметр *count*), начиная с некоторого момента (*from_msc*) в формате времени в миллисекундах;
- в *CopyTicksRange* — как диапазон двух моментов времени(от *from_msc* по *to_msc*).

Состав копируемых данных о каждом тике указывается в параметре *flags* как битовая маска значений из перечисления ENUM_COPY_TICKS.

Идентификатор	Значение	Описание
COPY_TICKS_INFO	1	Тики, вызванные изменениями *Bid* и/или *Ask*
COPY_TICKS_TRADE	2	Тики, вызванные изменениями *Last* и *Volume*
COPY_TICKS_ALL	3	Все тики
COPY_TICKS_TIME_MS	1 << 8	Время в миллисекундах
COPY_TICKS_BID	1 << 9	Цена *Bid*
COPY_TICKS_ASK	1 << 10	Цена *Ask*
COPY_TICKS_LAST	1 << 11	Цена *Last*
COPY_TICKS_VOLUME	1 << 12	Объем
COPY_TICKS_FLAGS	1 << 13	Флаги тика

Первые три бита (младший байт) обуславливают набор запрашиваемых тиков, а остальные биты (старший байт) — свойства этих тиков.

Комбинировать флаги старшего байта можно только для матриц, поскольку в вектор помещается только один ряд со значениями конкретного поля из всех тиков. Таким образом, для заполнения вектора следует выбрать лишь один бит старшего байта.

При выборе нескольких свойств тиков в процессе заполнения матрицы порядок строк в ней будет соответствовать порядку элементов в перечислении. Например, цена *Bid* всегда окажется в строке выше (с меньшим индексом), чем строка с ценами *Ask*.

Пример совместной работы с тиками и векторами будет представлен в разделе о машинном обучении.

4.10.6 Вычисление выражений с матрицами и векторами

Над матрицами и векторами можно поэлементно производить математические операции (применять операторы) — сложение, вычитание, умножение и деление. Для этого оба объекта должны быть одного и того же типа и иметь одинаковые размеры. Каждый член матрицы/вектора взаимодействует с соответствующим элементом второй матрицы/вектора.

В качестве второго слагаемого (множителя, вычитаемого, делителя) можно также использовать скаляр соответствующего типа (*double*, *float* или *complex*). В этом случае каждый элемент матрицы или вектора будет обработан с учетом этого скаляра.

```
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;   // произведение Адамара (поэлементное)
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;                     // возможны операции "по месту"
matrix_a /= 2;
```

Операции "по месту" модифицируют исходную матрицу (или вектор), помещая в неё результат, в отличие от обычных бинарных операций, в которых операнды остаются без изменений, а для результата создается новый объект.

Кроме того, матрицы и векторы можно передавать в качестве параметра в большинство математических функций. В этом случае матрица или вектор обрабатываются по элементам. Например:

```
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]]
*/
```

В случае *MathMod* и *MathPow* в качестве второго параметра может быть использован как скаляр, так и матрица или вектор соответствующего размера.

4.10.7 Манипуляции над матрицами и векторами

При работе с матрицами и векторами доступны базовые манипуляции без проведения каких-либо вычислений. В начале списка приведены исключительно матричные методы, а четыре последних также применимы и для векторов.

- *Transpose* — транспонирование матрицы;
- *Col, Row, Diag* — извлечение и установка строк, столбцов и диагоналей по номеру;
- *TriL, TriU* — получение нижней и верхней треугольной матрицы по номеру диагонали;
- *SwapCols, SwapRows* — перестановка местами указанных по номерам строк и столбцов;
- *Flat* — установка и получение элемента матрицы по сквозному индексу;
- *Reshape* — изменение формы матрицы "по месту";
- *Split, Hsplit, Vsplit* — разделение матрицы на несколько подматриц;

- *Resize* — изменение размера матрицы и вектора "по месту";
- *Compare, CompareByDigits* — сравнение двух матриц или двух векторов с заданной точностью вещественных чисел;
- *Sort* — сортировка "по месту" (перестановка элементов) и через получение вектора или матрицы индексов;
- *Clip* — ограничение диапазона значений элементов "по месту".

Обратите внимание, что разделение вектора не предусмотрено.

Ниже представлены прототипы методов для матриц.

```
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)
```

Для векторов — набор методов поменьше.

```
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)
```

Пример транспонирования матрицы:

```
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]]
*/
```

Несколько примеров установки разных диагоналей методом *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]]
*/
```

Изменение конфигурации матрицы методом *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]]
*/
```

Разделение матриц на подматрицы мы применим в примере в разделе о Решении уравнений.

Методы *Col* и *Row* позволяют не только получать столбцы или строки матрицы по их номеру, но и вставлять их "по месту" в ранее определенные матрицы. При этом не меняются ни размеры матрицы, ни значения элементов за пределами вектора-столбца (для случая *Col*) или вектора-строки (для случая *Row*).

Если же любой из этих двух методов применяется к матрице, размеры которой еще не установлены, то будет создана нулевая матрица размером [N * M], где N и M определяются по-разному для *Col* и *Row*, исходя из длины вектора и заданного индекса столбца или строки:

- в случае *Col*, N — это длина вектора-столбца, а M — на 1 больше заданного индекса вставляемого столбца;
- в случае *Row*, N — на 1 больше заданного индекса вставляемой строки, а M — длина вектора-строки;

На момент написания данной главы в MQL5 не предусмотрено методов для полноценной вставки строк и столбцов с раздвиганием последующих элементов, а также исключение заданных строк и столбцов.

4.10.8 Произведения матриц и векторов

Умножение матриц является одной из базовых операций в различных численных методах. Например, она часто применяется при реализации прямого и обратного распространения сигнала в слоях нейронных сетей.

К разряду матричных произведений можно отнести также различного рода свертки. Группа таких функций в MQL5 выглядит следующим образом:

- *MatMul* — матричное произведение двух матриц;
- *Power* — возведение квадратной матрицы в указанную целочисленную степень;
- *Inner* — внутреннее произведение двух матриц;
- *Outer* — внешнее произведение двух матриц или двух векторов;
- *Kron* — произведение Кронекера двух матриц, матрицы и вектора, вектора и матрицы или двух векторов;
- *CorrCoef* — вычисление корреляции Пирсона между строками или столбцами матрицы, или между векторами;
- *Cov* — вычисление ковариационной матрицы строк или столбцов матрицы, или между двумя векторами;
- *Correlate* — вычисление взаимной корреляции (кросс-корреляции) двух векторов;
- *Convolve* — вычисление дискретной линейной свертки двух векторов;
- *Dot* — скалярное произведение двух векторов.

Чтобы дать общее представление, как управлять этими методами, приведем их прототипы (в порядке от матричных, через смешанные матрично-векторные, к векторным).

```
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)
```

Вот простой пример матричного произведения двух матриц с помощью метода *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]]
*/
```

Умножать между собой можно матрицы вида A[M,N] * B[N,K] = C[M,K], то есть количество столбцов в матрице слева должно быть равно количеству строк в матрице справа. Если размеры не согласованы, результатом будет пустая матрица.

При умножении матрицы и вектора допустимы два варианта:

- горизонтальный вектор (строка) умножается на матрицу справа, длина вектора равна количеству строк матрицы;
- матрица умножается на вертикальный вектор (столбец) справа, длина вектора равна количеству столбцов матрицы.

Векторы также можно перемножать между собой. В *MatMul* это всегда эквивалентно скалярному произведению (метод *Dot*) вектора-строки на вектор-столбец, а вариант, когда вектор-столбец умножается на вектор-строку и получается матрица, поддерживается другим методом — *Outer*.

Продемонстрируем *Outer*-произведение вектора *v5* на вектор *v3* и в обратной последовательности. В обоих случаях слева подразумевается вектор-столбец, а справа — вектор-строка.

```
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 Преобразования (разложение) матриц

Преобразования матриц являются наиболее востребованными операциями при работе с данными. При этом многие сложные преобразования не могут быть выполнены в аналитической форме и с абсолютной точностью.

Матричные преобразования (или по-другому, декомпозиции) — это методы, которые разлагают матрицу на составные части, что облегчает вычисление более сложных матричных операций. Методы матричной декомпозиции, также называемые методами матричной факторизации, являются основой в алгоритмах линейной алгебры, таких как решение систем линейных уравнений, вычисление обратной матрицы или определителя.

В частности, в машинном обучении широко используется сингулярное разложение (Singular Values Decomposition, SVD), которое позволяет представить исходную матрицу как произведение трех других матриц. SVD-разложение используется при решении самых разных задач — от приближения методом наименьших квадратов до сжатия и распознавания изображений.

Список доступных методов:

- *Cholesky* — вычисление декомпозиции Холецкого;
- *Eig* — вычисление собственных значений и правых собственных векторов квадратной матрицы;
- *EigVals* — вычисление собственных значений общей матрицы;
- *LU* — LU-факторизация матрицы как произведения нижнетреугольной матрицы и верхнетреугольной матрицы;
- *LUP* — LUP-факторизация с частичным поворотом, которая является LU-факторизацией только с перестановками строк: PA=LU;
- *QR* — QR-факторизация матрицы;

- *SVD* — разложение по сингулярным значениям.

Ниже представлены прототипы методов.

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)

Покажем пример сингулярного разложения методом SVD (см. файл *MatrixSVD.mq5*). Сначала инициализируем исходную матрицу.

```
matrix a = {{0, 1, 2, 3, 4, 5, 6, 7, 8}};
a = a - 4;
a.Reshape(3, 3);
Print("matrix a \n", a);
```

Сделаем SVD-разложение:

```
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);
```

Проверим разложение: должно выполняться равенство *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);
```

Сравним полученную и исходную матрицу на ошибки.

```
ulong errors = (int)a.Compare(matrix_usvt, 1e-9);
Print("errors=", errors);
```

В журнале должно получиться следующее:

```
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
```

Еще один практический случай использования метода *Convolve* включен в пример в разделе Методы машинного обучения.

4.10.10 Получение статистики

Для получения описательной статистики матриц и векторов предназначены перечисленные ниже методы. Все они применимы к вектору или матрице целиком, а также к заданной оси матрицы (по горизонтали или по вертикали). При применении целиком к объекту эти функции возвращают скаляр (единственное число). При применении к матрице по какой-либо из осей возвращается вектор.

Общий вид прототипов:

```
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)
```

Перечень самих методов:

- *ArgMax, ArgMin* — нахождение индексов максимального и минимального значений;
- *Max, Min* — нахождение максимальное и минимальное значения;
- *Ptp* — нахождение диапазона значений;
- *Sum, Prod* — вычисление суммы или произведения элементов;

- *CumSum, CumProd* — вычисление кумулятивной суммы или произведения элементов;
- *Median, Mean, Average* — вычисление медианы, среднего арифметического или взвешенного среднего арифметического;
- *Std, Var* — вычисление стандартного отклонения и дисперсии;
- *Percentile, Quantile* — вычисление перцентилей и квантилей;
- *RegressionMetric* — вычисление одной из предопределенных регрессионных метрик, как ошибки отклонения от линии регрессии на данных матрицы/вектора.

Пример вычисления стандартного отклонения и перцентилей для размаха баров (в пунктах) текущего символа и таймфрейма приведен в файле *MatrixStdPercentile.mq5*.

```
input int BarCount = 1000;
input int BarOffset = 0;

void OnStart()
{
   // получаем котировки текущего чарта
   matrix rates;
   rates.CopyRates(_Symbol, _Period, COPY_RATES_OPEN | COPY_RATES_CLOSE,
      BarOffset, BarCount);
   // рассчитываем приращения цен на барах
   vector delta = MathRound((rates.Row(1) - rates.Row(0)) / _Point);
   // отладочная печать начальных баров
   rates.Resize(rates.Rows(), 10);
   Normalize(rates);
   Print(rates);
   // печать метрик приращений
   PRTF((int)delta.Std());
   PRTF((int)delta.Percentile(90));
   PRTF((int)delta.Percentile(10));
}
```

Фрагмент журнала:

```
(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 Характеристики матриц и векторов

Следующая группа методов позволяет получить основные характеристики матриц:

- *Rows, Cols* — количество строк и столбцов в матрице;
- *Norm* — одна из предопределенных норм матрицы (ENUM_MATRIX_NORM);
- *Cond* — число обусловленности матрицы;
- *Det* — определитель квадратной невырожденной матрицы;
- *SLogDet* — вычисляет знак и логарифм определителя матрицы;
- *Rank* — ранг матрицы;

- *Trace* — сумма элементов по диагоналям матрицы (след);
- *Spectrum* — спектр матрицы как набор ее собственных значений.

Кроме того для векторов определены такие характеристики:

- *Size* — длина вектора;
- *Norm* — одна из предопределенных норм вектора (ENUM_VECTOR_NORM).

Размеры объектов (впрочем, как и индексация элементов в них) используют значения типа *ulong*.

```
ulong matrix<T>::Rows()
ulong matrix<T>::Cols()
ulong vector<T>::Size()
```

Большинство прочих характеристик — это вещественные числа.

```
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()
```

Ранг и спектр представляют собой, соответственно, целое число и вектор.

```
int matrix<T>::Rank()
vector matrix<T>::Spectrum()
```

Пример вычисления ранга матрицы:

```
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());
```

А вот результат работы скрипта:

```
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 Решение уравнений

В методах машинного обучения и задачах оптимизации часто требуется найти решение системы линейных уравнений. MQL5 содержит четыре метода, которые позволяют решать такие уравнения в зависимости от типа матрицы.

- *Solve* — решает линейное матричное уравнение или систему линейных алгебраических уравнений;
- *LstSq* — решает систему линейных алгебраических уравнений приблизительно (для неквадратных или вырожденных матриц);
- *Inv* — вычисляет мультипликативную обратную матрицу по отношению к квадратной невырожденной матрице методом Жордана-Гаусса;
- *PInv* — вычисляет псевдообратную матрицу методом Мура-Пенроузаю

Ниже приведены прототипы методов.

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()

Методы *Solve* и *LstSq* подразумевают решение системы уравнений вида *A * X = B*, где A — матрица, B — переданный через параметр вектор со значениями функции (или "зависимой переменной").

Попробуем применить метод *LstSq* для решения системы уравнений, которая представляет собой модель идеальной торговли корзиной инструментов (в нашем случае будем анализировать

основные валюты Forex). Для этого на заданном количестве "исторических" баров нужно найти такие размеры лотов для каждой валюты, чтобы линия баланса стремилась к постоянно растущей прямой.

Обозначим *i*-ую валютную пару S_i. Её котировка на баре с индексом *k* равна $S_i[k]$. Нумерация баров будет идти из прошлого в будущее, как в матрицах и векторах, заполненных методом *CopyRates*. Таким образом, начало собранных котировок для "обучения" модели соответствует бару, помеченному у нас номером 0, но на шкале времени это будет самый старый исторический бар (из тех, что мы обрабатываем, согласно настройкам алгоритма). Бары, идущие вправо (в будущее) от него имеют номера 1, 2 и так далее, вплоть до общего количества баров, на которых пользователь закажет расчет.

Изменение цены символа между 0-м баром и N-м баром обуславливает прибыль (или убыток) к моменту N-го бара.

Учитывая множество валют, получим, например, для 1-го бара такое уравнение прибыли:

```
(S1[1] - S1[0]) * X1 + (S2[1] - S2[0]) * X2 + ... + (Sm[1] - Sm[0]) * Xm = B
```

Здесь *m* — общее количество символов, X_i — размер лота каждого символа, *B* — плавающая прибыль (условный баланс, если зафиксировать прибыль).

Для простоты сократим запись: перейдем от абсолютных значений к приращениям цен ($A_i[k] = S_i[k] - S_i[0]$). С учетом движения по барам, получим несколько выражений для виртуальной кривой баланса:

```
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]
```

Успешная торговля характеризуется постоянной прибылью на каждом баре, то есть модель для правостороннего вектора *B* — это монотонно возрастающая линия, в идеале прямая.

Реализуем эту модель и подберем для неё коэффициенты *X* на базе котировок. Поскольку мы еще не знаем прикладных API, то не станем кодировать полноценную торговую стратегию. Просто построим виртуальный график баланса с помощью функции *GraphPlot* из стандартного заголовочного файла *Graphic.mqh* (мы его уже использовали для демонстрации математических функций).

Полный исходный код нового примера находится в скрипте *MatrixForexBasket.mq5*.

Во входных параметрах позволим пользователю выбирать общее количество баров для выборки данных (*BarCount*), а также номер бара внутри этой выборки (*BarOffset*), где заканчивается условное прошлое и начинается условное будущее.

На условном прошлом будет строиться модель (решаться вышеприведенная система линейных уравнений), а на условном будущем мы выполним форвард-тест.

```
input int BarCount = 20;  // BarCount (известная "история" и "будущее")
input int BarOffset = 10; // BarOffset (где начинается "будущее")
input ENUM_CURVE_TYPE CurveType = CURVE_LINES;
```

Для заполнения вектора идеальным балансом напишем функцию *ConstantGrow*: она будет использоваться далее при инициализации.

```
void ConstantGrow(vector &v)
{
   for(ulong i = 0; i < v.Size(); ++i)
   {
      v[i] = (double)(i + 1);
   }
}
```

Список торгуемых инструментов (основные Forex-пары) задан "жестко" в начале функции *OnStart* — отредактируйте его под свои требования и торговое окружение.

```
void OnStart()
{
   const string symbols[] =
   {
      "EURUSD", "GBPUSD", "USDJPY", "USDCAD",
      "USDCHF", "AUDUSD", "NZDUSD"
   };
   const int size = ArraySize(symbols);
   ...
```

Создадим матрицу *rates*, куда будут складываться котировки символов, вектор *model* с желаемой кривой баланса и вспомогательный вектор *close* для по-символьного запроса цен закрытия баров (данные из него будут копироваться в столбцы матрицы *rates*).

```
   matrix rates(BarCount, size);
   vector model(BarCount - BarOffset, ConstantGrow);
   vector close;
```

В цикле по символам копируем цены закрытия в вектор *close*, вычисляем приращения цен и записываем их в соответствующий столбец матрицы *rates*.

```
   for(int i = 0; i < size; i++)
   {
      if(close.CopyRates(symbols[i], _Period, COPY_RATES_CLOSE, 0, BarCount))
      {
         // вычисляем приращения (прибыль на всех и на каждом баре одной строкой)
         close -= close[0];
         // подстраиваем прибыль на стоимость пункта
         close *= SymbolInfoDouble(symbols[i], SYMBOL_TRADE_TICK_VALUE) /
            SymbolInfoDouble(symbols[i], SYMBOL_TRADE_TICK_SIZE);
         // размещаем вектор в столбце матрицы
         rates.Col(close, i);
      }
      else
      {
         Print("vector.CopyRates(%d, COPY_RATES_CLOSE) failed. Error ",
            symbols[i], _LastError);
         return;
      }
   }
   ...
```

Особенности расчета стоимости одного пункта цены (в валюте депозита) мы рассмотрим в Главе 5.

Также стоит отметить, что на разных финансовых инструментах бары с одинаковыми индексами могут иметь разные временные метки, например, если в одной из стран был праздничный день и рынок был закрыт (вне Forex инструменты могут в принципе иметь разные расписания торговых сессий). Для решения этой проблемы, строго говоря, требуется более глубокий анализ котировок с учетом времен баров и их синхронизация перед вставкой в матрицу *rates*. Мы здесь это не делаем для простоты, а также поскольку рынок Forex большую часть времени функционирует по единым правилам.

Разделяем матрицу на две части: начальная часть будет использоваться для нахождения решения (это эмулирует оптимизацию на истории), а последующая часть — для форвард-теста (расчета последующих изменений баланса).

```
matrix split[];
if(BarOffset > 0)
{
   // обучение на BarCount - BarOffset барах
   // проверка на BarOffset барах
   ulong parts[] = {BarCount - BarOffset, BarOffset};
   rates.Split(parts, 0, split);
}

// решаем систему линейных уравнений для модели
vector x = (BarOffset > 0) ? split[0].LstSq(model) : rates.LstSq(model);
Print("Solution (lots per symbol): ");
Print(x);
...
```

Теперь, имея решение, построим кривую баланса на всех барах выборки (в начале будет идти идеальная "историческая" часть, а затем начнется "будущая", не использовавшаяся для подстройки модели).

```
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]);
   }
}
...
```

Оценим качество решения по критерию R2.

```
   if(BarOffset > 0)
   {
      // делаем копию баланса
      vector backtest = balance;
      // выделяем для бэктеста только "исторические" бары
      backtest.Resize(BarCount - BarOffset);
      // бары для форвард-теста приходится скопировать вручную
      vector forward(BarOffset);
      for(int i = 0; i < BarOffset; ++i)
      {
         forward[i] = balance[BarCount - BarOffset + i];
      }
      // вычисляем регрессионные метрики независимо для обеих частей
      Print("Backtest R2 = ", backtest.RegressionMetric(REGRESSION_R2));
      Print("Forward R2 = ", forward.RegressionMetric(REGRESSION_R2));
   }
   else
   {
      Print("R2 = ", balance.RegressionMetric(REGRESSION_R2));
   }
   ...
```

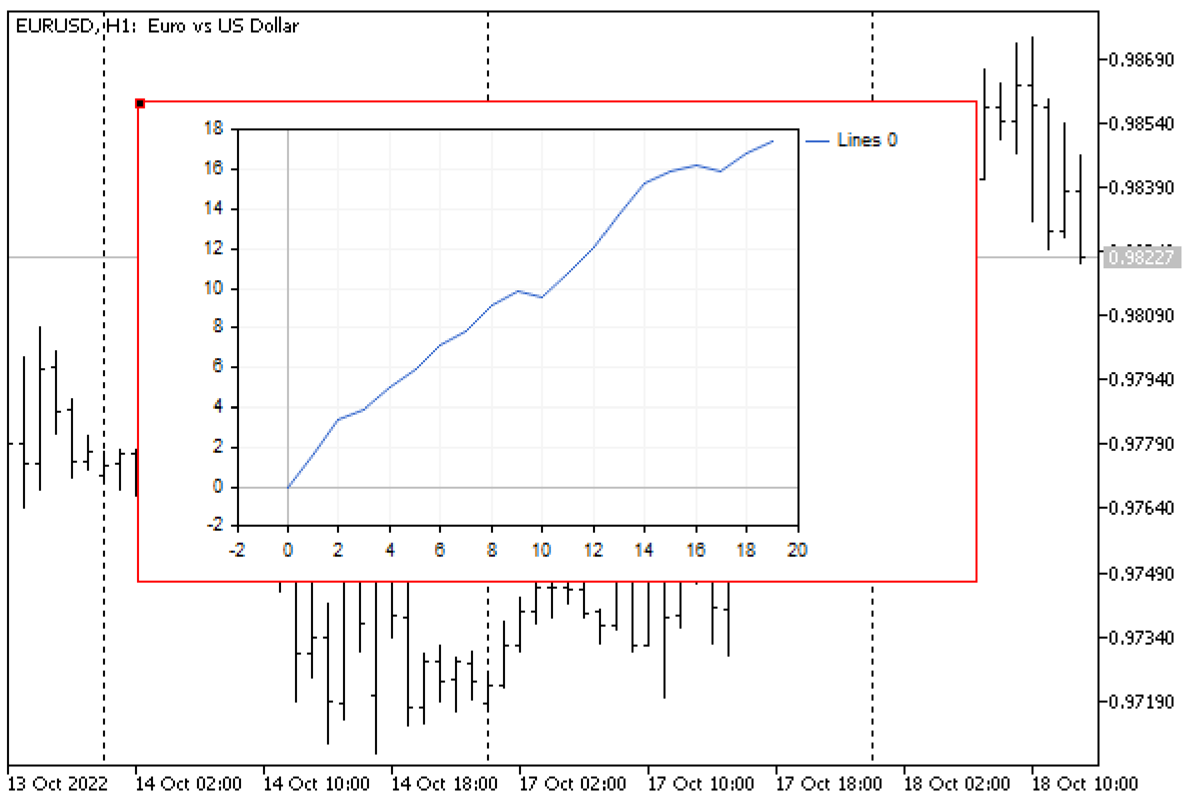
Для отображения кривой баланса на графике необходимо перенести данные из вектора в массив.

```
   double array[];
   balance.Swap(array);

   // распечатаем значения меняющегося баланса с точностью до 2 цифр
   Print("Balance: ");
   ArrayPrint(array, 2);

   // рисуем кривую баланса в объекте-графике ("бэктест" и "форвард")
   GraphPlot(array, CurveType);
}
```

Вот пример журнала, полученного при запуске скрипта на 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
```

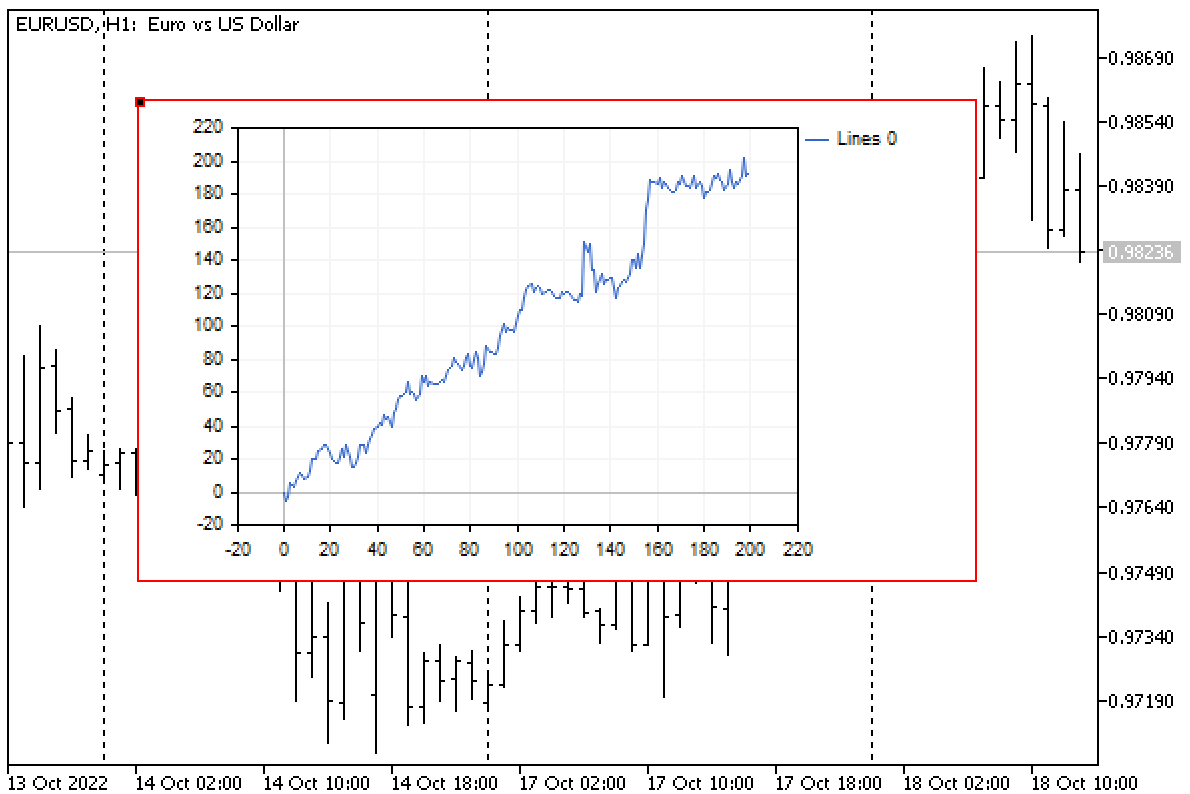
А вот как выглядит кривая виртуального баланса.



Виртуальный баланс торговли корзиной валют лотами согласно решению СЛАУ

Левая половина имеет более ровную форму и более высокий показатель R2, что не удивительно, потому что модель (переменные *X*) подстраивались именно под неё.

Ради интереса увеличим в 10 раз глубину обучения и проверки, то есть зададим в параметрах *BarCount = 200* и *BarOffset = 100*. Получим новую картинку.



Виртуальный баланс торговли корзиной валют лотами согласно решению СЛАУ

Очевидно, что "будущая" половина выглядит более изрезанно, и можно даже сказать, что нам повезло, что она продолжает расти, несмотря на столь простую модель. Как правило, на форвард-тесте кривая виртуального баланса существенно деградирует и начинает идти вниз.

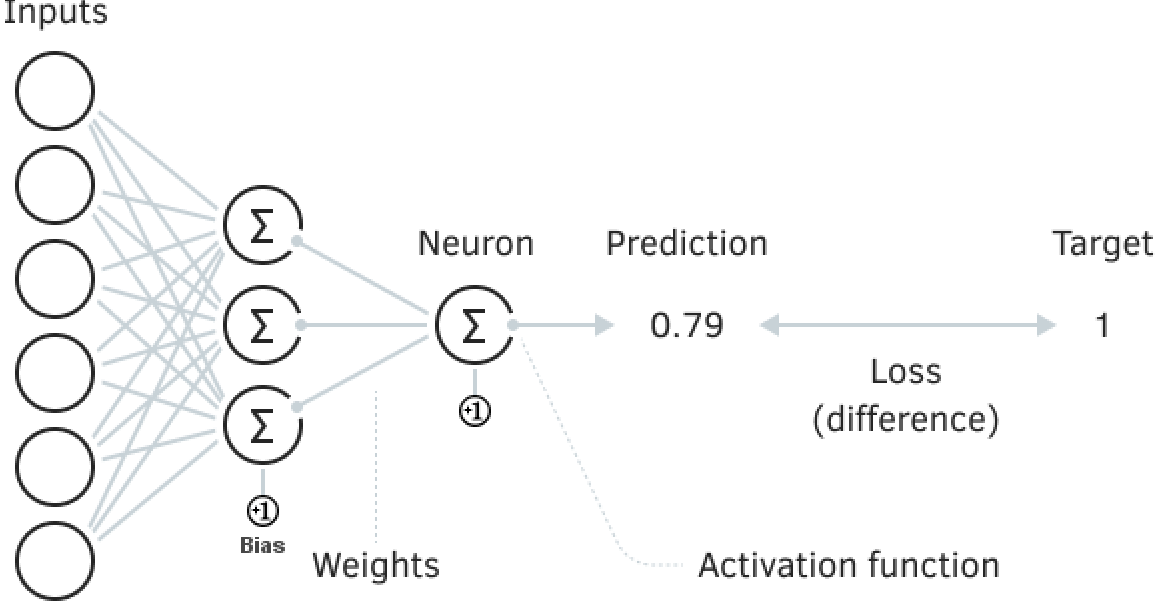
Важно отметить, что для проверки модели мы брали найденные значения *X* из решения системы "как есть", в то время как на практике нам потребуется их нормализовать под минимальные лоты и шаг лотов, что отрицательно скажется на результатах и приблизит их к реальности.

4.10.13 Методы машинного обучения

Среди встроенных методов матриц и векторов предусмотрено несколько, востребованных в задачах машинного обучения, в частности, при реализации нейронных сетей.

Как можно понять из названия, нейронная сеть — это совокупность множества нейронов — примитивных вычислительных ячеек. Примитивными они являются в том смысле, что выполняют довольно простые вычисления: как правило, нейрон обладает набором весовых коэффициентов, которые применяются к неким входным сигналам, после чего взвешенная сумма сигналов подается в функцию активации — нелинейный преобразователь.

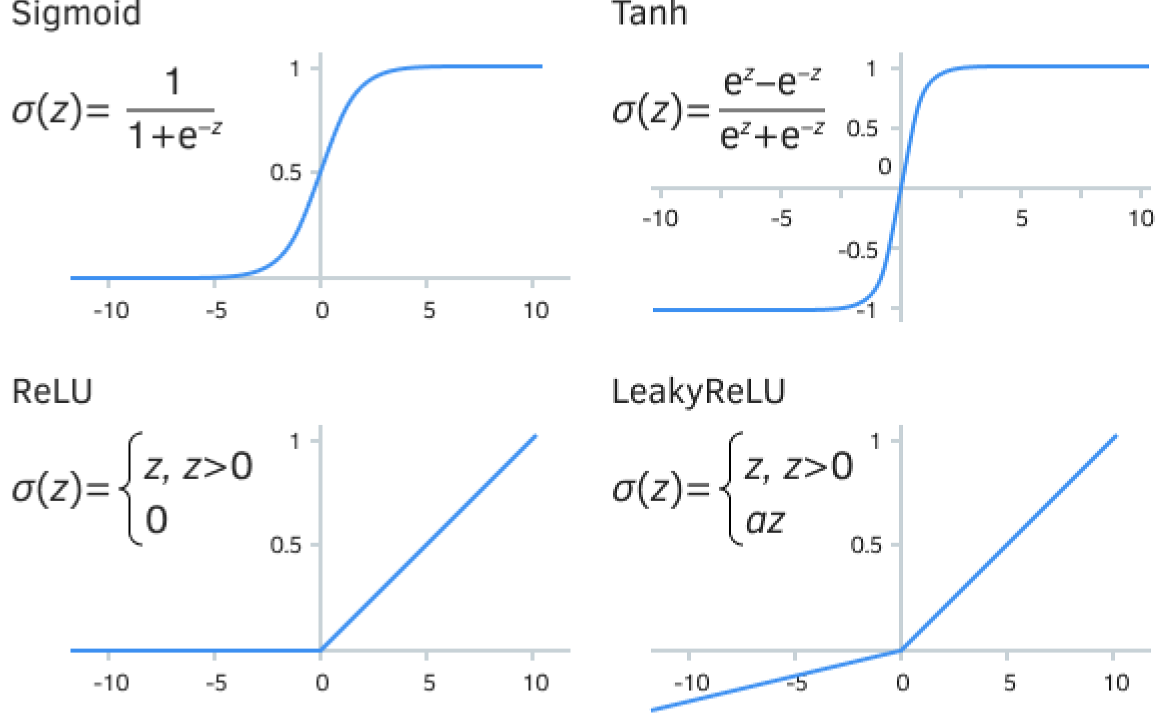
Наличие функции активации позволяет усиливать слабые сигналы и ограничивать слишком сильные, предотвращая переход в насыщение (переполнение вещественных вычислений). Однако самое главное — нелинейность наделяет сеть новыми вычислительными возможностями, существенно усложняет пространство функций (задач), решаемых сетью.



Элементарная нейронная сеть

Сила нейронных сетей проявляется за счет комбинирования большого количество нейронов и установления связей между ними. Обычно нейроны организованы в слои (которые можно сравнить с матрицами или векторами), в том числе и с рекурсивными (рекуррентными) связями, а также могут иметь различные по своему эффекту функции активации. Это позволяет анализировать объемные данные различными алгоритмами, в частности, находя в них скрытые закономерности.

Обратите внимание, что если бы не нелинейность в каждом нейроне, многослойную нейронную сеть можно было бы представить в эквивалентной форме в виде одного слоя, коэффициенты которого получаются матричным произведением всех слоев ($W_{total} = W_1 * W_2 * ... * W_L$, где 1..L — номера слоев). А это получился бы простой линейный сумматор. Таким образом, математически обосновывается важность активационных функций.



Некоторые из наиболее известных активационных функций

Одна из основных классификаций нейронных сетей делит их согласно используемому алгоритму обучения на сети "с учителем" (supervised learning) и "без учителя" (unsupervised learning). Первые требуют, чтобы человек-эксперт предоставил для исходного набора данных желаемые результаты (например, дискретные метки состояния торговой системы или числовые показатели предполагаемых приращений цены). Сети без учителя выявляют кластеры в данных самостоятельно.

В любом случае задача обучения нейронной сети заключается в поиске параметров, минимизирующих ошибку на обучающей и тестовой выборке, для чего используется функция потерь (loss function): она предоставляет качественную или количественную оценку ошибки между целевым и полученным откликом сети.

> Наиболее важными аспектами для успешного применения нейронных сетей являются выбор информативных и взаимно независимых предикторов (анализируемых характеристик), преобразование (нормализация и очистка) данных согласно специфике алгоритма обучения, а также оптимизация архитектуры и размера сети. Само по себе применение алгоритмов машинного обучения не гарантирует успех.

Мы не станем здесь вдаваться в теорию нейронных сетей, их классификацию и типичные решаемые задачи. Эта тема слишком обширна. Желающие могут найти статьи на сайте mql5.com и в других источниках.

В MQL5 предусмотрено три метода для использования в машинном обучении, ставшие частью программного интерфейса матриц и векторов.

- *Activation* — вычисляет значения функции активации;
- *Derivative* — вычисляет значения производной активационной функции;
- *Loss* — вычисляет значение функции потерь.

Производные активационных функций позволяют эффективно обновлять параметры модели на основании её ошибки, изменяющейся в процессе обучения.

Два первых метода записывают результат в переданный вектор/матрицу и возвращают признак успеха (*true* или *false*), а функция потерь возвращает число. Приведем их прототипы (под типом *object*<*T*> обозначены и *matrix*<*T*>, и *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)
```

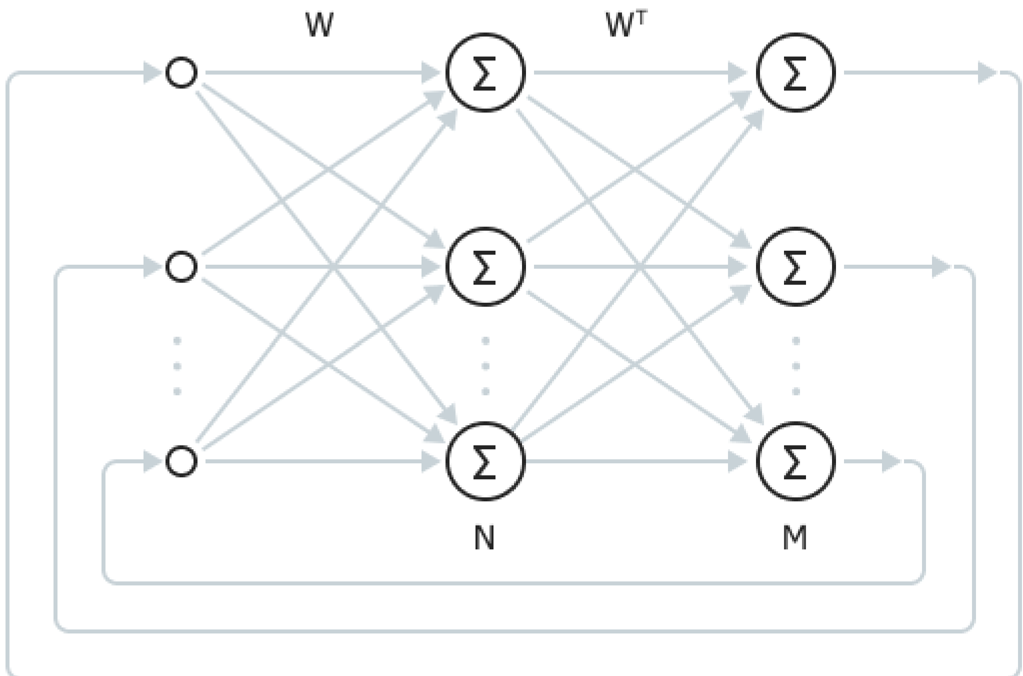
Некоторые функции активации допускают установку параметра с помощью третьего, необязательного аргумента.

Перечень поддерживаемых активационных функций в перечислении ENUM_ACTIVATION_FUNCTION и функций потерь в перечислении ENUM_LOSS_FUNCTION можно найти в справке по MQL5.

В качестве ознакомительного примера рассмотрим задачу анализа потока реальных тиков. Некоторые трейдеры считают тики мусорным шумом, однако есть и те, кто практикует на их основе высокочастотную торговлю. Существует предположение, что высокочастотные алгоритмы, как правило, дают преимущество крупным игроками и основаны исключительно на программной обработке ценовой информации. Исходя из этого выдвинем гипотезу, что в потоке тиков существует эффект краткосрочной памяти, обусловленный текущими активными роботами маркетмейкеров. Тогда можно использовать тот или иной метод машинного обучения, чтобы найти эту зависимость и предсказать несколько будущих тиков.

> Машинное обучение всегда предполагает выдвижение гипотез, синтез модели для них и проверку на практике. Очевидно, что продуктивные гипотезы, а тем более эффективные модели получаются далеко не всегда. Это длительный процесс проб и ошибок, в котором неудачи являются источником для совершенствования и поиска новых путей.

Выберем для нашего случая один из наиболее простых видов нейронных сетей: двунаправленную ассоциативную память (Bidirectional Associative Memory, BAM). Такая сеть имеет всего два слоя: входной и выходной — именно во втором из них формируется некий отклик (ассоциация) в ответ на подачу того или иного сигнала на вход. Размеры слоев могут быть разными. Когда размеры одинаковы, получается конфигурация сети, известная под именем Хопфилда.



Полносвязная двунаправленная ассоциативная память

С помощью такой сети мы будем сопоставлять N недавних предыдущих тиков и M следующих предсказываемых, формируя обучающую выборку из ближайшего прошлого на заданную глубину. Тики будут подаваться в сеть в виде положительных или отрицательных приращений цены, преобразованных к двоичным значениям [+1, -1] (бинарные сигналы являются канонической формой кодирования в сетях BAM и Хопфилда).

Наиболее важным преимуществом BAM является практически мгновенный (по сравнению с большинством других итеративных способов) процесс обучения, который заключается в расчете матрицы весов. Мы приведем формулу чуть ниже.

Но, разумеется, эта простота имеет и обратную "сторону медали": емкость BAM (количество образов, которые она способна запомнить) ограничена размером меньшего слоя, и то — при условии особого распределения +1 и -1 в векторах обучающей выборки.

Таким образом, для нашего случая, сеть "обобщит" все последовательности тиков в обучающей выборке, и далее в процессе штатной работы будет "скатываться" в тот или иной запомненный образ, в зависимости от предъявляемой последовательности новых тиков. Насколько хорошо это будет получаться на практике, зависит от очень большого числа факторов: размера и настроек сети, особенностей текущего потока тиков и так далее.

Поскольку предполагается, что поток тиков обладает лишь кратковременной памятью, сеть желательно переобучать в реальном времени или близко к этому, благо обучение фактически сводится к нескольким матричным операциям.

Итак, чтобы сеть "запомнила" ассоциативные образы (в нашем случае: что было и что будет в потоке тиков) достаточно вычислить следующее уравнение:

$$\mathbf{W} = \Sigma_i(\mathbf{A}_i^T\mathbf{B}_i)$$

где W — весовая матрица сети, а суммирование выполняется по все попарным произведениям входных векторов A_i и соответствующих выходных векторов B_i.

Далее при работе сети мы подаем на первый слой входной образ, применяем к нему матрицу W и тем самым активируем второй слой, где вычисляем активационную функцию для каждого нейрона. Далее с помощью транспонированной матрицы W^T сигнал распространяется обратно на первый слой, где также в нейронах применяются функции активации. Входной образ в этот момент уже не поступает на первый слой, то есть в сети продолжается свободный колебательный процесс. Он продолжается до тех пор, пока изменения сигнала в нейронах сети не стабилизируются (т.е. станут меньше некой заданной величины).

В таком состоянии второй слой сети содержит найденный ассоциированный выходной образ ("предсказание").

Реализуем данный сценарий машинного обучения в скрипте *MatrixMachineLearning.mq5*.

Во входных параметрах можно задать общее количество последних тиков (*TicksToLoad*), запрашиваемых из истории, и какое из них количество отведено для тестирования (*TicksToTest*). Соответственно, модель (веса) будет строиться на основе (*TicksToLoad - TicksToTest*) тиков.

```
input int TicksToLoad = 100;
input int TicksToTest = 50;
input int PredictorSize = 20;
input int ForecastSize = 10;
```

Также во входных переменных выбирается размер входного вектора (количество "известных" тиков *PredictorSize*) и выходного вектора (количество "будущих" тиков *ForecastSize*).

Запрос тиков делается в начале функции *OnStart*. В данном случае мы работаем только с ценами *Ask*, но никто не мешает подключить сюда также *Bid*, *Last* и объем.

```
void OnStart()
{
   vector ticks;
   ticks.CopyTicks(_Symbol, COPY_TICKS_ALL | COPY_TICKS_ASK, 0, TicksToLoad);
   ...
```

Разделим тики на обучающую и тестовую выборки.

```
   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];
   }
   ...
```

Для вычисления приращений цен используем метод *Convolve* с дополнительным вектором {+1, -1}. Отметим, что вектор с приращениями будет на 1 элемент короче исходного.

```
   vector differentiator = {+1, -1};
   vector deltas = ask1.Convolve(differentiator, VECTOR_CONVOLVE_VALID);
   ...
```

Свертка по алгоритму VECTOR_CONVOLVE_VALID означает, что в расчет берутся только полные наложения векторов (т.е. меньший по размеру вектор последовательно сдвигается вдоль большего, не выходя за его границы ни одним элементом). Другие виды сверток допускают наложение векторов только одним элементом или половиной элементов (при этом оставшаяся часть элементов выходит за пределы корреспондирующего вектора, и значения свертки демонстрируют пограничные эффекты).

Для перевода непрерывных значений приращений в единичные импульсы (положительные и отрицательные в зависимости от знака исходного элемента вектора) написана вспомогательная функция *Binary* (здесь не приводится): она возвращает новую копию вектора, в которой каждый элемент равен либо +1, либо -1.

```
   vector inputs = Binary(deltas);
```

На основе полученной входной последовательности с помощью функции *TrainWeights* вычисляется матрица весовых коэффициентов W нейронной сети. Устройство этой функции мы рассмотрим ниже, а пока обратим внимание, что в неё передаются параметры *PredictorSize* и *ForecastSize*, которые позволяют "нарезать" непрерывную последовательность тиков на наборы парных входных и выходных векторов по размерам входного и выходного слоя ВАМ, соответственно.

```
   matrix W = TrainWeights(inputs, PredictorSize, ForecastSize);
   Print("Check training on backtest: ");
   CheckWeights(W, inputs);
   ...
```

Сразу послу "обучения" сети мы проверяем её точность на обучающей выборке — просто, чтобы убедиться, что сеть действительно обучилась. Это действие поручено функции *CheckWeights*.

Однако важнее проверить, как сеть поведет себя на незнакомых тестовых данных. Для этого аналогичным образом "дифференцируем" и "бинаризуем" второй вектор *ask2*, а затем тоже отправляем в *CheckWeights*.

```
   vector test = Binary(ask2.Convolve(differentiator, VECTOR_CONVOLVE_VALID));
   Print("Check training on forwardtest: ");
   CheckWeights(W, test);
   ...
}
```

Настало время познакомиться с функцией *TrainWeights*. В ней мы определяем матрицы A и B для "нарезки" векторов (образов) из переданной входной последовательности — из вектора *data*.

```
template<typename T>
matrix<T> TrainWeights(const vector<T> &data, const uint predictor, const uint respor
   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];
      }
   }
   ...
```

Каждый очередной образ А получается из идущих один за другим тиков в количестве *predictor*, а соответствующий ему "будущий" образ - из следующих *responce* элементов. До тех пор, пока позволяет общий объем данных, это "окно" сдвигается вправо по одному элементу, формируя новые и новые пары образов. Нумерация образов идет по строкам, а тиков в них — по столбцам.

Далее нам следует выделить память под матрицу весов W и заполнить её с помощью матричных методов: последовательно умножаем строки из A и B с помощью *Outer* и матрично суммируем.

```
   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;
}
```

Функция *CheckWeights* выполняет похожие действия для нейронной сети, весовые коэффициенты которой передаются в уже готовом виде в первом аргументе W. Размеры обучающих векторов извлекаются из самой матрицы 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();
   ...
```

Матрицы A и B в данном случае формируются не для расчета W, а выступают "поставщиками" векторов для тестирования. Также нам понадобится и транспонированная копия W для просчета обратных сигналов из второго слоя сети в первый.

Количество итераций, в течение которых в сети допустимы переходные процессы вплоть до сходимости, ограничивается константой *limit*.

```
   const uint limit = 100;

   int positive = 0;
   int negative = 0;
   int average = 0;
```

Переменные *positive*, *negative*, *average* нужны для подсчета статистики успешных и неуспешных предсказаний, чтобы оценить качество обучения.

Далее в цикле по тестовым парам образов выполняется активация сети и снимается её окончательный отклик. Каждый очередной входной вектор записывается в вектор ***a***, а выходной слой ***b*** заполняется нулями. После этого запускаются итерации по передаче сигнала из ***a*** в ***b*** с помощью матрицы W и применения активационной функции AF_TANH, а также обратной передачи сигнала из ***b*** в ***a*** и тоже применения AF_TANH. Процесс продолжается вплоть до достижении *limit* циклов (что маловероятно) или выполнения условия сходимости, при котором векторы состояния нейронов ***a*** и ***b*** уже практически не меняются (здесь используется метод *Compare* и вспомогательные копии векторов ***x*** и ***y*** от предыдущей итерации).

```
   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);
      ...
```

После достижения стабильного состояния, переводим состояния нейронов из непрерывных (вещественных) в бинарные +1 и -1 с помощью функции *Binarize* (она похожа на уже упоминавшуюся функцию *Binary*, но меняет состояние вектора по месту).

Остается посчитать количество совпадений в выходном слое с целевым вектором. Для этого выполняется скалярное умножение векторов. Положительный результат означает, что количество правильно угаданных тиков превышает количество неправильных. Общий счетчик совпадений накапливается в average.

```
      const int match = (int)(b.Dot(B.Row(i)));
      if(match > 0) positive++;
      else if(match < 0) negative++;

      average += match;  // 0 в match означает точность 50/50 (т.е. случайное гадание
   }
```

После завершения цикла по всем тестовым образцам, выводим статистику.

```
   float skew = (float)average / k; // среднее количество совпадений на вектор

   PrintFormat("Count=%d Positive=%d Negative=%d Accuracy=%.2f%%",
      k, positive, negative, ((skew + responce) / 2 / responce) * 100);
}
```

В скрипте также имеется функция *RunWeights*, которая фактически представляет собой рабочий прогон нейронной сети (по её матрице весов W) для онлайн-вектора из последних *predictor* тиков. Функция вернет вектор с предполагаемыми будущими тиками.

```
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;
}
```

В конце *OnStart* мы приостанавливаем выполнение на 1 секунду (чтобы с долей вероятности дождаться новых тиков), запрашиваем последние *PredictorSize + 1* тиков (не забываем +1 для дифференцирования) и делаем для них прогнозирование онлайн.

```
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);
}
```

Запуск скрипта с настройками по умолчанию на EURUSD в вечер пятницы дал такие результаты.

```
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]
```

Символ и время упомянуты не зря, так как обстановка на рынке может существенно влиять на применимость алгоритма и конкретной конфигурации сети. Когда рынок открыт, при каждом запуске скрипта вы будете получать отличные результаты, поскольку поступают все новые и новые тики. Это ожидаемое поведение, согласующееся с гипотезой о формировании короткой памяти.

Как мы видим, точность обучения приемлемая, но на тестовых данных заметно снижается и вполне может стать ниже 50%.

Здесь мы плавно переходим из области программирования в область научных исследований. Встроенный в MQL5 инструментарий машинного обучения позволяет вам реализовать множество других конфигураций нейронных сетей и анализаторов, с иными торговыми стратегиями и принципам подготовки исходных данных.

Часть 5. Создание прикладных программ на MQL5

В этой части мы вплотную займемся изучением тех разделов API, которые связаны с решением прикладных задач алготрейдинга: анализом и обработкой финансовых данных, их визуализацией и разметкой с помощью графических объектов, автоматизацией рутинных действий, интерактивным взаимодействием с пользователем.

Начнем мы с общих принципов создания MQL-программ, их типов, особенностей и модели событий в терминале. Затем коснемся доступа к таймсериям, работы с графиками и графическими объектами. Наконец, разберем принципы создания и применения каждого типа MQL-программы в отдельности.

Активные пользователи MetaTrader 5, несомненно, помнят, что терминал поддерживает пять типов программ:

- технические индикаторы для расчета произвольных показателей в виде временных рядов, с возможностью их визуализации в главном окне графика или в отдельной панели (подокне);
- эксперты (или, иначе, советники), обеспечивающие автоматическую или полуавтоматическую торговлю;
- скрипты для выполнения вспомогательных разовых задач по требованию;
- сервисы для выполнения фоновых задач в непрерывном режиме;
- библиотеки, представляющие собой откомпилированные модули с определенным, обособляемым функционалом, которые подключаются к другим типам MQL-программ во время их загрузки динамически (что кардинально отличает библиотеки от заголовочных файлов, подключаемых на стадии компиляции, статически).

В предыдущих частях книги, по мере освоения основ программирования и общеупотребительных встроенных функций, нам уже пришлось обратиться к реализации скриптов и сервисов в качестве примеров. Эти типы программ были выбраны, как более простые по сравнению с остальными. Теперь мы опишем их целенаправленно и добавим к ним более функциональные и востребованные индикаторы.

С помощью индикаторов и графиков мы изучим некоторые технические приемы, которые будут применимы также и для экспертов. Однако непосредственно разработку экспертов — более сложную по своей сути задачу — мы отложим и вынесем в отдельную, следующую Часть 6, включающую не только автоматическое исполнение приказов и формализацию торговых стратегий, но также их тестирование и оптимизацию на истории.

Что касается индикаторов, MetaTrader 5 поставляется, как известно, с набором встроенных стандартных индикаторов. В данной части мы научимся использовать их программным способом, а также создавать свои собственные индикаторы как с нуля, так и на базе других индикаторов.

Все откомпилированные индикаторы, советники, скрипты и сервисы отображаются в Навигаторе в MetaTrader 5. Библиотеки не являются самостоятельными программами и потому не имеют выделенной ветви в иерархии, хотя, конечно, это было бы удобно с точки зрения единообразного управления всеми бинарными модулями. Как мы увидим далее, те программы, которые зависят от той или иной библиотеки, не могут запуститься без неё. Но сейчас проверить наличие библиотеки можно только в файловом менеджере.

Программирование на MQL5 для трейдеров — исходные коды из книги: Часть 5

Примеры из книги также доступны в публичном проекте \MQL5\Shared Projects\MQL5Book

5.1 Общие принципы выполнения MQL-программ

Все MQL-программы можно условно разделить на несколько групп в зависимости от их возможностей и особенностей.

Большинство программ — эксперты, индикаторы и скрипты — работают в контексте графика. Иными словами, они начинают выполняться только после того, как прикреплены к одному из открытых графиков с помощью команды контекстного меню *Присоединить к графику* в дереве *Навигатора* или путем перетаскивания мышью из *Навигатора* на график.

В противовес этому, сервисы не могут быть размещены на графике, так как они предназначены для выполнения продолжительных, циклических действий в фоновом режиме. Например, в сервисе можно создать пользовательский символ и затем в бесконечном цикле получать для него данные при помощи сетевых функций и непрерывно обновлять. Также сервису логично поручить отслеживание подключения к торговому счету и Интернету, в составе схемы по уведомлению пользователя о проблемах со связью.

Важно отметить, что индикаторы и эксперты сохраняются на графике между сеансами работы с терминалом. Иными словами, если пользователь набросил, например, индикатор на какой-либо график и потом, не удалив его явным образом, закрыл MetaTrader 5, то при следующем запуске терминала индикатор будет восстановлен вместе с графиком, включая и все его настройки.

Кстати говоря, привязка к графику индикаторов и экспертов является основой для шаблонов (см. документацию). Пользователь может собрать на графике набор необходимых ему программ, настроить их и сохранить в особом файле с расширением *tpl*. Это делается с помощью команды контекстного меню *Шаблоны -> Сохранить*. После этого можно применить конкретный шаблон к любому новому графику (команда *Шаблоны -> Загрузить*) и тем самым запустить на нем все привязанные программы. По умолчанию шаблоны хранятся в каталоге *MQL5/Profiles/Templates/*.

Еще одно следствие прикрепления к графику заключается в том, что закрытие графика приводит к выгрузке всех MQL-программ, которые были на нем размещены. Правда, MetaTrader 5 на всякий случай сохраняет особым образом все закрытые графики (по крайней мере, на некоторое время) и потому, если график был закрыт случайно, его можно восстановить вместе со всеми программами (и графическими объектами, если они были) с помощью команды *Файл -> Открыть удаленный*.

Если по какой-либо причине терминал не сможет загрузить файлы графиков, всё состояние MQL-программ (настройки и расположение), будет утеряно. В принципе, это же касается и графических объектов — программы могли добавить их для собственных нужд и ожидать наличия на графике. Делайте резервные копии графиков. Каждый график — это файл с расширением *chr*. Такие файлы хранятся по умолчанию в каталоге *MQL5/Profiles/Charts/Default/*. Это стандартный профиль, создаваемый при установке платформы. Вы можете создать другие профили с помощью команды меню *Файл -> Профили* и затем переключаться между ними (см. документацию).

При необходимости можно остановить работу эксперта и удалить его с графика с помощью команды контекстного меню *Список экспертов* (вызывается по нажатию правой кнопки мыши в окне графика). Она открывает диалог *Эксперты* со списком всех выполняющихся в терминале экспертов. В этом списке следует выбрать ставший ненужным советник и нажать кнопку *Удалить*.

Индикаторы также предполагается удалять явным образом, посредством аналогичной команды контекстного меню — *Список индикаторов*. Она открывает диалог со списком индикаторов текущего графика, где можно выделить конкретный индикатор и нажать кнопку *Удалить*. Кроме того, большинство индикаторов выводит на график различные графические построения (линии, гистограммы), для которых по клику мыши доступно контекстное меню с командами для удаления.

В отличие от индикаторов и экспертов, скрипты не привязываются к графику навсегда. В штатном режиме скрипт удаляется с графика автоматически после выполнения возложенной на него задачи, если это единоразовое, одномоментное действие. Если в скрипте организован цикл для периодических, повторяющихся действий, он, разумеется, продолжит свою работу до прерывания цикла тем или иным образом, но не дольше чем до конца сеанса. Закрытие терминала приводит к откреплению скрипта от графика. После перезагрузки MetaTrader 5 скрипты на графиках не восстанавливаются.

> Обратите внимание, что если переключить график на другой символ или таймфрейм, работающий на нем скрипт будет выгружен. А вот индикаторы и эксперты останутся работать, однако будут заново проинициализированы, причем правила инициализации для них различаются. Эти нюансы будут рассмотрены в разделе Особенности запуска и остановки программ разных типов.

На графике может быть размещен только один эксперт, только один скрипт и произвольное количество индикаторов. Эксперт, скрипт и все индикаторы будут работать параллельно (одновременно).

Что касается сервисов, то их созданные и запущенные экземпляры автоматически восстанавливаются после загрузки терминала. Экземпляр сервиса можно остановить или удалить с помощью контекстного меню в ветви *Сервисы* окна *Навигатора*.

В следующей таблице в обобщенном кратком виде представлены вышеописанные свойства.

Тип программы	Привязка к графику	Количество на графике	Восстановление сеанса
индикатор	требуется	много	с графиком или шаблоном
эксперт	требуется	максимум 1	с графиком или шаблоном
скрипт	требуется	максимум 1	не поддерживается
сервис	не поддерживается	0	с терминалом

Напомним, что все MQL-программы выполняются в клиентском терминале и потому работают только пока терминал открыт. Для постоянного программного контроля за счетом используйте VPS.

5.1.1 Конструирование MQL-программ различных типов

Тип программы является фундаментальным свойством в MQL5. В отличие от C++ или других общецелевых языков программирования, где любую программу можно развивать в произвольных направлениях, например, добавляя графический интерфейс или подкачку данных с сервера по

сети, в MQL5 существует разделение программ на группы по их предназначению. Так, технический анализ с визуализацией временных рядов "поручен" индикаторам, но они не способны торговать. В свою очередь, экспертам доступны функции торгового API, но в них отсутствуют индикаторные буфера (массивы для построения линий).

Поэтому при решении конкретной прикладной задачи разработчик должен, как правило, разложить её на части, функционал каждой из которых укладывается в специализацию отдельного типа. Разумеется, в простых случаях бывает достаточно единственной MQL-программы, но иногда поиск оптимального технического решения неочевиден. Например, для построения "графика" ренко, стоит ли реализовать его как индикатор, как пользовательский символ, генерируемый сервисом, или может быть выполнять расчет "кирпичей" непосредственно в торговом эксперте (если имеется торговая стратегия на ренко)? Все варианты осуществимы.

Тип MQL-программы характеризуется несколькими факторами.

Во-первых, под каждый тип программ выделена собственная папка в рабочем каталоге MQL5. Во введении в первую Часть мы уже упоминали этот факт и перечисляли папки. Так для индикаторов, экспертов, скриптов и сервисов предназначены, соответственно, папки *Indicators*, *Experts*, *Scripts*, и *Services*. Для библиотек в папке MQL5 зарезервирована подпапка *Libraries*. В каждой из них можно организовать дерево вложенных папок произвольной конфигурации.

Напомним, что бинарный файл (готовая программа с расширением *ex5*) — результат компиляции mq5-файла — генерируется в том же каталоге, где лежит исходный mq5-файл. Однако следует упомянуть о существовании в MetaEditor проектов (файлов с расширением *mqproj*) — их мы разберем в главе Проекты. При разработке проекта готовый продукт создается в каталоге рядом с проектом. При создании программы из Мастера MQL5 в редакторе MetaEditor (команда *Файл -> Создать*) исходный файл по умолчанию размещается в папке, соответствующей типу программы. Если вы случайно скопируете программу не в тот каталог, ничего страшного не произойдет: она не превратится, например, из эксперта в индикатор или обратно. Её можно перенести в нужное место непосредственно в редакторе в окне *Навигатор* или во внешнем файловом менеджере. В *Навигаторе* каждый тип программы отображается отличительным значком.

> Размещение внутри каталога MQL5 во вложенной папке, выделенной под конкретный тип программ, не является фактором, определяющим тип конкретной MQL-программы. Тип определяется на основе содержимого исполняемого файла, которое, в свою очередь, формируется компилятором из директив-свойств и инструкций в исходном коде.
>
> Иерархия папок по типам программ введена для удобства. Рекомендуется её придерживаться за исключением случаев, когда речь идет о группе связанных проектов (с программами разных типов), которые логичнее хранить в отдельном каталоге.

Во-вторых, каждый тип программ характеризуется поддержкой ограниченного, специфического набора системных событий, которые активируют программу. Мы дадим Обзор функций обработки событий в отдельном разделе. Для получения событий конкретного типа в программе необходимо описать функцию-обработчик с предопределенным прототипом (название, список параметров, возвращаемое значение).

Например, мы уже видели, что в скриптах и сервисах работа запускается в функции *OnStart*, и поскольку она там единственная, её можно назвать главной "точкой входа", через которую терминал передает управления прикладном коду. В других типах программ ситуация несколько сложнее. В целом отметим, что тип программы характеризуется неким набором обработчиков, часть из которых может быть обязательной, а часть опциональной (но вместе с тем, недопустимой для других типов программ). В частности, в индикаторе требуется функция

OnCalculate (без неё индикатор не скомпилируется, компилятор выдаст ошибку). Однако в эксперте эта функция не используется.

В-третьих, некоторые типы программ требуют особых директив *#property*. В разделе Общие свойства программ мы уже познакомились с директивами, которые могут использоваться во всех типах программ. Однако есть и другие, специализированные директивы. Например, в примерах с сервисами нам встречалась директива *#property service*, которая, собственно, и делает программу сервисом. Без неё даже размещение программы в папке *Services* не позволит запускать её в фоне.

Аналогичным образом определяющую роль в создании библиотек играет директива *#property library*. Все подобные директивы-свойства будут рассмотрены в разделах для соответствующих типов программ.

Сочетание директив и обработчиков событий учитывается в становлении типа MQL-программы следующим образом (обход сверху вниз до первого совпадения):

- Индикатор — наличие обработчика *OnCalculate*;
- Библиотека — наличие *#property library*;
- Скрипт — наличие обработчика *OnStart* и отсутствие *#property service*;
- Сервис — наличие обработчика *OnStart* и *#property service*;
- Эксперт — наличие любого другого обработчика.

Пример того, какой эффект данные свойства оказывают на компилятор, будет приведен в разделе Обзор функций обработки событий.

> Для всех вышеперечисленных пунктов следует учитывать еще один момент. Тип программы определяет главный компилируемый модуль — файл с расширением mq5, в который могут быть подключены с помощью директивы *#include* другие исходные тексты из других каталогов. Все включенные таким образом функции принимаются к рассмотрению наравне с теми, что присутствуют непосредственно в основном mq5-файле.
>
> С другой стороны, директивы *#property* имеют эффект только при размещении в основном, компилируемом mq5-файле. Если директивы встретятся в файлах, включенных в программу с помощью *#include*, они будут проигнорированы.

Главный mq5-файл не обязан буквально содержать функции-обработчики событий. Вполне допустимо поместить часть или целиком весь алгоритм в некоторых заголовочных mqh-файлах, а затем включить их в одну или несколько программ. Например, мы можем реализовать обработчик *OnStart* с набором полезных действий в mqh-файле и с помощью *#include* использовать его внутри двух отдельных программ: скрипта и сервиса.

Попутно отметим, что наличие общих обработчиков событий не является единственным мотивом для выделения общих фрагментов алгоритма в заголовочный файл. Вы можете использовать одну и ту же формулу расчета, например, в индикаторе и в эксперте, оставив их обработчики событий в основных модулях программ.

> Включаемые файлы хоть и принято называть заголовочными и давать им расширение *mqh*, чисто технически это не обязательно. Вполне допустимо (хотя и не рекомендуется) включить в один mq5-файл другой mq5-файл или, например, txt-файл. В них может быть какой-то унаследованный код или, скажем, инициализация неких массивов константами. Для нас здесь важно, что включение другого mq5-файла не делает его главным.
>
> При этом нужно следить, чтобы в программу попали только свойственные её типу функции

обработки событий и среди них не было двойников (как известно, функции идентифицируются по совокупности имени и списку параметров: перегрузка функции разрешена только с различным набором параметров). Обычно это достигают при помощи различных директив препроцессора. Например, определив макрос *#define OnStart OnStartPrevious* перед включением стороннего mq5-файла скрипта в какую-то свою программу, мы фактически превратим описанную в нем функцию *OnStart* в *OnStartPrevious*, и сможем вызывать её как обычную из своих собственных обработчиков событий.

Однако такой подход имеет смысл применять лишь в исключительных случаях, когда исходный код включаемого mq5-файла по тем или иным причинам нельзя подвергнуть модификации, в частности, структурировать его с выделением интересующих алгоритмов в функции или классы в отдельных заголовочных файлах.

По принципу взаимодействия с пользователем MQL-программы можно разделить на интерактивные и утилитарные.

Интерактивные — индикаторы и эксперты — могут обрабатывать события, возникающие в программной среде в ответ на действия пользователя, такие как нажатия кнопок на клавиатуре, перемещение мыши, изменения размера окна, а также многие прочие события, например, связанные с получением котировочных данных или таймером.

Утилитарные — сервисы и скрипты — руководствуются только входными переменными, заданными в момент запуска, и не реагируют на события в системе.

Особняком среди всех типов программ стоят библиотеки. Они всегда выполняются, как часть MQL-программы другого типа (одного из четырех основных), и потому не обладают никакими отличительными характеристиками или поведением. В частности, они не могут напрямую принимать события от терминала и не имеют своих потоков (см. следующий раздел). Одна и та же библиотека может быть подключена ко многим программам, причем происходит это динамически в момент запуска каждой родительской программы. В разделе, посвященном библиотекам, мы узнаем, каким образом описать экспортируемый программный интерфейс библиотеки и импортировать его в родительскую программу.

5.1.2 Потоки

В упрощенном виде программу можно представить как последовательность инструкций, которые разработчик сформировал для компьютера. Основным исполнителем инструкций в компьютере является центральный процессор. Современные компьютеры, как правило, оснащаются процессорами с несколькими ядрами, что равноценно тому, что в них несколько процессоров. Однако количество программ, которые пользователь может захотеть запустить параллельно, практически, не ограничено. Таким образом, число программ всегда во много раз больше доступных ядер/процессоров. В связи с этим каждое ядро на самом деле разделяет свое рабочее время между несколькими разными программами: выделит 1 миллисекунду на выполнение инструкций одной программы, потом 1 миллисекунду — для инструкций другой, потом — третей, и так далее, по кругу. Поскольку переключение происходит очень быстро, пользователь этого не замечает: для него все программы выполняются как бы параллельно и одновременно.

Для того чтобы процессор мог приостановить выполнение инструкций одной программы, а потом возобновить её работу с прежнего места (после того как он незаметно переключался на инструкции других "параллельных" программ), он должен уметь неким образом сохранять и восстанавливать промежуточное состояние каждой программы: текущую инструкцию, переменные, возможно, открытые файлы, сетевые подключения, и так далее. Вся эта

совокупность ресурсов и данных, которые требуются программе для нормальной работы вместе с текущей позицией в последовательности инструкций, называется контекстом выполнения программы. Операционная система, по сути, предназначена для того, чтобы создавать такие контексты под каждую программу по требованию пользователя (или других программ). Каждый такой активный контекст и называется потоком. Многие программы запрашивают для себя много потоков, потому что их функционал подразумевает параллельное ведение нескольких активностей. И MetaTrader 5 — тоже в их числе. Загрузка котировок по множеству символов, построение графиков, реакция на действия пользователя — все это требует потоков. Но плюс ко всему отдельные потоки выделяются и MQL-программам.

Среда исполнения MQL-программ выделяет каждой программе не более одного потока. Эксперты, скрипты и сервисы получают строго по одному потоку, а для индикаторов выделяется один поток на все индикаторы, работающие на одном финансовом инструменте. Причем этот же поток отвечает за отображение графиков соответствующего символа, поэтому занимать его тяжелыми вычислениями не рекомендуется. В противном случае пользовательский интерфейс утратит отзывчивость: действия пользователя будут обрабатываться с задержкой или окно вовсе будет казаться зависшим. Потоки всех прочих типов MQL-программ "не завязаны" на интерфейс, и потому в них можно загружать процессор любой сложной задачей.

Одно из важных свойств потока следует из его определения и назначения: он поддерживает лишь последовательное выполнение указанных инструкций одна за другой. В каждый момент времени в одном потоке исполняется только одна инструкция. Если в программе написан бесконечный цикл, поток "застрянет" на этой инструкции и никогда не доберется до инструкций ниже него. Долгие вычисления также могут создавать эффект бесконечного цикла: они загрузят процессор и не дадут выполниться другим действиям, результаты которых, возможно, ожидает пользователь. Именно поэтому эффективные расчеты в индикаторах важны для гладкой работы графического интерфейса.

Однако и в MQL-программах других типов стоит уделять внимание планированию потоков. В следующих разделах мы познакомимся со специальными функциями обработки событий, которые являются точками входа в MQL-программы. И однопоточная модель означает, что во время обработки одного события программа невосприимчива к другим событиям, которые потенциально могут возникнуть в это же время. Поэтому терминал организует для каждой программы очередь событий. Мы коснемся этого момента более подробно в следующем разделе.

Чтобы ощутить эффекты однопоточности на практике мы чуть позже рассмотрим простой пример в разделе Ограничения и преимущества индикаторов (*IndBarIndex.mq5*). Индикаторы выбраны для этой цели не случайно: они не только делят между собой один поток на каждом символе, но и выводят свои результаты непосредственно на график, что делает потенциальную проблему наиболее наглядной.

5.1.3 Обзор функций обработки событий

Передача управления MQL-программам, то есть, фактически, их выполнение, происходит путем вызова терминалом или агентами тестирования специальных функций, которые MQL-разработчик определяет в своем прикладном коде для обработки предопределенных событий. Такие функции должны иметь заданный прототип, включающий название, список параметров (количество, типы и очередность), а также тип возвращаемого значения.

Название каждой функции соответствует смыслу события, с добавлением префикса *On*. Например, *OnStart* — это главная функция для "старта" скриптов и сервисов: она вызывается терминалом в момент размещения скрипта на графике или запуска экземпляра сервиса.

В рамках данной книги будем называть событие и соответствующий ему обработчик одинаковым именем.

В следующей таблице перечислены все типы событий и то, в каких программах они поддерживаются (— индикатор, — эксперт, — скрипт, — сервис). Подробное описание событий приводится в разделах соответствующих типов программ. Причинами событий инициализации и деинициализации могут быть многие факторы: размещение программы на графике, смена её настроек, смена символа/таймфрейма графика (или шаблона, или профиля), смена счета и другие (см. раздел Особенности запуска и остановки программ разных типов).

Тип программ Событие/Обработчик					Описание
OnStart	-	-	●	●	запуск/выполнение
OnInit	+	+	-	-	инициализация после загрузки (см. подробности в разделе Особенности запуска и остановки программ разных типов)
OnDeinit	+	+	-	-	деинициализация перед остановкой и выгрузкой
OnTick	-	+	-	-	получение новой цены (тика)
OnCalculate	●	-	-	-	запрос на пересчет индикатора из-за получения новой цены или синхронизации старых цен
OnTimer	+	+	-	-	срабатывание таймера с заданной периодичностью
OnTrade	-	+	-	-	завершение торговой операции на сервере
OnTradeTransaction	-	+	-	-	изменение состояния торгового счета (приказов, сделок, позиций)
OnBookEvent	+	+	-	-	изменение стакана заявок
OnChartEvent	+	+	-	-	действия пользователя или MQL-программ на графике
OnTester	-	+	-	-	окончание одиночного прохода в тестере
OnTesterInit	-	+	-	-	инициализация перед оптимизацией
OnTesterDeinit	-	+	-	-	деинициализация после оптимизации
OnTesterPass	-	+	-	-	поступление данных оптимизации с агента тестирования

Обязательные обработчики помечены символом '●', опциональные — символом '+'.

Несмотря на то, что функции-обработчики предназначены, прежде всего, для их вызова средой исполнения, никто не запрещает вызывать их из собственного исходного кода. Например, если в эксперте нужно сделать некий расчет на основании доступных котировок сразу после старта,

причем даже в отсутствие тиков (например, в выходные), можно вызвать *OnTick* перед выходом из *OnInit*. В качестве альтернативы было бы логично выделить расчет в отдельную функцию и вызвать её как из *OnInit*, так и из *OnTick*. Однако следует иметь в виду, что работу функции инициализации желательно выполнять быстро, и если расчет длительный, следует выполнять его по таймеру.

Все MQL-программы (за исключением библиотек) обязаны иметь хотя бы один обработчик события. В противном случае компилятор выдаст ошибку "не найдена функция обработки события" ("event handling function not found").

Наличие некоторых функций-обработчиков определяет тип программы в отсутствие директив #property, устанавливающих другой тип. Например, наличие обработчика *OnCalculate* приводит к генерации индикатора (даже если он размещен в другой папке, например, скриптов или экспертов). Наличие обработчика *OnStart* (если нет *OnCalculate*) подразумевает создание скрипта. При этом, если в индикаторе помимо *OnCalculate* встретится *OnStart*, получим предупреждение компилятора "функция OnStart определена в программе, не являющейся скриптом" ("OnStart function defined in the non-script program").

К книге прилагается пара файлов *AllInOne.mq5* и *AllInOne.mqh*. В заголовочном файле описаны почти пустые заготовки всех основных обработчиков событий: в них ничего нет, кроме вывода названия обработчика в журнал. Синтаксис и особенности применения каждого из обработчиков мы рассмотрим в разделах, посвященных конкретным типам MQL-программ. Смысл данного файла — предоставить поле для экспериментов с компиляцией разных типов программ, в зависимости от наличия тех или иных обработчиков и директив-свойств (*#property*).

Некоторые сочетания могут приводить к ошибкам или предупреждениям.

Если компиляция прошла успешно, то получившийся тип программы автоматически выводится в журнал после её загрузки за счет следующей строки:

```
const string type =
   PRTF(EnumToString((ENUM_PROGRAM_TYPE)MQLInfoInteger(MQL_PROGRAM_TYPE)));
```

Перечисление ENUM_PROGRAM_TYPE и функцию *MQLInfoInteger* мы изучали в разделе Тип и лицензия программы.

Файл *AllInOne.mq5*, в который включен *AllInOne.mqh*, находится изначально в каталоге *MQL5Book/Scripts/p5/*, но его можно скопировать в любую другую папку, в том числе в соседние ветви *Навигатора* (например, в папку советников или индикаторов). Внутри файла в комментариях оставлены варианты для подключения тех или иных конфигураций сборки программы. По умолчанию, если не редактировать файл, получится, как ни странно, советник.

```
//+------------------------------------------------------------------+
//| Раскомментируйте следующую строку для получения сервиса          |
//| NB: также следует активировать #define _OnStart OnStart          |
//+------------------------------------------------------------------+
//#property service

//+------------------------------------------------------------------+
//| Раскомментируйте следующую строку для получения библиотеки       |
//+------------------------------------------------------------------+
//#property library

//+------------------------------------------------------------------+
//| Раскомментируйте следующую строку для скрипта или                |
//| сервиса (должно быть включено свойство #property service)        |
//+------------------------------------------------------------------+
//#define _OnStart OnStart

//+------------------------------------------------------------------+
//| Раскомментируйте одну из двух следующих строк для индикатора     |
//+------------------------------------------------------------------+
//#define _OnCalculate1 OnCalculate
//#define _OnCalculate2 OnCalculate

#include <MQL5Book/AllInOne.mqh>
```

Если прикрепить программу к графику, получим в журнале запись:

```
EnumToString((ENUM_PROGRAM_TYPE)MQLInfoInteger(MQL_PROGRAM_TYPE))=PROGRAM_EXPERT / ok
OnInit
OnChartEvent
OnTick
OnTick
OnTick
...
```

Также, скорее всего, начнет генерироваться поток записей из обработчика *OnTick*, если рынок открыт.

Если продублировать mq5-файл под другим именем и, например, раскомментировать директиву *#property service*, компилятор создаст сервис, но выдаст несколько предупреждений.

```
no OnStart function defined in the script
OnInit function is useless for scripts
OnDeinit function is useless for scripts
```

Первое из них, об отсутствии функции *OnStart*, является на самом деле существенным, потому что при создании экземпляра сервиса в нем не будет вызвана ни одна функция, а только выполнится инициализация глобальных переменных. Правда, за счет этого в журнал (закладка *Эксперты* в терминале) все же будет выведен тип PROGRAM_SERVICE. Но в принципе, в сервисах, также как и в скриптах, предполагается наличие функции *OnStart*.

Два других предупреждения возникают из-за того, что наш заголовочный файл содержит обработчики на все случаи жизни, и компилятор напоминает нам о том, что *OnInit* и *OnDeinit* бесполезны (не будут вызываться терминалом и даже не будут включены в бинарный образ

программы). Разумеется, в реальных программах таких предупреждений быть не должно, то есть все обработчики должны быть задействованы, а всё лишнее — убрано из исходного кода либо физически, либо логически — с помощью директив препроцессора для условной компиляции.

Если создать еще одну копию AllInOne.mq5 и в ней не только активировать директиву *#property service*, но и макрос *#define _OnStart OnStart*, в результате его компиляции получим полностью рабочий сервис. Он при запуске не только выведет название своего типа, но и название сработавшего обработчика *OnStart*.

Макрос потребовался, чтобы иметь возможность по желанию включать/отключать стандартный обработчик *OnStart*. В тексте *AllInOne.mqh* данная функция описана так:

```
void _OnStart() // "лишнее" подчеркивание делает функцию пользовательской
{
   Print(__FUNCTION__);
}
```

Имя, начинающееся с подчеркивания, делает её не стандартным обработчиком, а просто пользовательской функцией с похожим прототипом. Когда мы включаем макрос, компилятор заменяет по ходу компиляции *_OnStart* на *OnStart*, и в результате получается уже стандартный обработчик. Если бы мы явным образом назвали функцию *OnStart*, то согласно приоритетам характеристик, определяющих тип MQL-программы (см. раздел Особенности MQL-программ различных типов), она не позволила бы получить заготовку советника (поскольку *OnStart* идентифицирует программу как скрипт или сервис).

Аналогичная настраиваемая компиляция при помощи макросов *_OnCalculate1* или *_OnCalculate2* потребовалась, чтобы по желанию "прятать" обработчик со стандартным именем *OnCalculate*: в противном случае, при его наличии, у нас всегда получался бы индикатор.

Действительно, если в очередной копии программы активировать макрос *#define _OnCalculate1 OnCalculate*, мы получим пример индикатора (хоть он пустой и ничего не делает). Как мы узнаем далее, для индикаторов существует две разных формы обработчика *OnCalculate*, в связи с чем они представлены под нумерованными именами (*_OnCalculate1* и *_OnCalculate2*). Если запустить индикатор на графике, в журнале можно увидеть имена событий *OnCalculate* (по приходу тиков) и *OnChartEvent* (например, по клику мыши).

При компиляции индикатора компилятор выдаст два предупреждения:

```
no indicator window property is defined, indicator_chart_window is applied
no indicator plot defined for indicator
```

Это происходит потому, что индикаторы, как средства визуализации данных, предполагают наличие в своем коде некоторых специфических настроек, которых здесь нет. На данном этапе поверхностного знакомства с разными типами программ это не важно. Но далее мы научимся описывать в индикаторах их свойства и массивы, определяющие, что и как следует визуализировать на графике. Тогда эти предупреждения исчезнут.

Очередь событий

В момент возникновения нового события оно должно быть доставлено во все MQL-программы, работающие на соответствующем графике. Из-за однопоточной модели исполнения MQL-программ (см. раздел Потоки) может случиться, что очередное событие поступает, когда еще происходит обработка предыдущего. Терминал для подобных случаев поддерживает для каждой интерактивной MQL-программы очередь событий. Все события в ней обрабатываются одно за другим в порядке поступления.

Очереди событий имеют ограниченный размер. Поэтому нерационально написанная программа может за счет медлительных действий спровоцировать переполнение своей очереди. При переполнении новые события отбрасываются без постановки в очередь.

Недостаточно быстрая обработка событий может негативно сказаться на опыте пользователя или качестве данных (представьте, что вы записываете трансляцию изменений стакана цен и пропустите несколько сообщений). Для решения этой проблемы можно искать более эффективные алгоритмы или использовать параллельную работу нескольких взаимосвязанных MQL-программ (например, поручить расчеты индикатору, а в эксперте только считывать готовые данные).

Следует иметь в виду, что терминал складывает в очередь не все события, а избирательно — некоторые типы событий обрабатываются по принципу "не более одного события такого типа в очереди". Например, если в очереди уже есть событие *OnTick* или оно находится в процессе обработки, то новое событие *OnTick* в очередь не ставится. Если в очереди уже находится событие *OnTimer* или событие изменения графика, то новые события таких типов тоже отбрасываются (игнорируются). Речь именно о конкретном экземпляре программы. Другие, менее "занятые" программы, получат это сообщение.

Мы не приводим полный перечень подобных типов событий, потому что данная оптимизация за счет пропуска "накладывающихся" событий может быть изменена разработчиками терминала.

Подход по организации работы программ в ответ на поступающие события называется событийно-ориентированным (event-driven). Еще его можно назвать асинхронным, потому что постановка события в очередь программы и его извлечение (вместе с обработкой) происходят в различные моменты (в идеале, разделенные микроскопическим интервалом, но идеал достижим не всегда). Однако из 4-х типов MQL-программ лишь индикаторы и эксперты в полной мере следуют этому подходу. Скрипты и сервисы имеют, по сути, только главную функцию, которая, будучи вызванной, должна либо быстро выполнить требуемое действие и завершиться, либо запустить бесконечный цикл, в котором поддерживать некую активность (например, чтение данных из сети) вплоть до остановки пользователем. Мы видели примеры таких циклов:

```
while(!IsStopped())
{
   полезный код
   ...
   Sleep(...);
}
```

В подобных циклах важно не забывать использовать *Sleep* с некоторым периодом, чтобы поделиться ресурсом процессора с другими программами. Значение периода выбирается исходя из предполагаемой интенсивности реализуемой активности.

Данный подход можно назвать циклическим или синхронным, или даже, если хотите, режимом "реального времени" (realtime), поскольку период "сна" можно выбирать, чтобы обеспечить постоянную частоту обработки данных, например, так:

```
int rhythm = 100; // 100 мс, 10 раз в секунду
while(!IsStopped())
{
   const int start = (int)GetTickCount();
   полезный код
   ...
   Sleep(rhythm - ((int)GetTickCount() - start));
}
```

Разумеется, "полезный код" должен укладываться по времени в отведенное "окно".

В отличие от этого, при событийном подходе заранее неизвестно, когда в следующий раз сработает фрагмент кода (обработчик). Например, на быстром рынке, во время новостей, тики могут приходить пачками, а по ночам — отсутствовать целыми секундами. В предельном случае, после завершающего тика вечером в пятницу следующее изменение цены по некоторому финансовому инструменту может транслироваться только в понедельник утром, и потому события *OnTick* будут отсутствовать двое суток. Иными словами, в событиях (и моментах активации обработчиков событий) нет регулярности, четкого расписания.

Но при необходимости можно сочетать оба похода. В частности, событие таймера (*OnTimer*) обеспечивает регулярность, а разработчик может периодически генерировать пользовательские события для графика внутри цикла (например, мигать надписью с предупреждением).

5.1.4 Особенности запуска и остановки программ разных типов

В программировании термин инициализация используется во множестве разных контекстов. В MQL5, к сожалению, также возникает некоторая неоднозначность. В разделе Инициализация мы уже применяли это слово для обозначения установки начальных значений переменных. Сейчас мы познакомились с событием инициализации *OnInit* в индикаторах и экспертах. И хотя смысл обоих инициализаций похож (привести программу в рабочее состояние), они на самом деле обозначают разные этапы подготовки MQL-программы к запуску: системный и прикладной.

Жизненный цикл готовой MQL-программы можно представить следующими крупными шагами:

1. Загрузка — чтение программы из файла в память терминала: сюда входят и инструкции, и предопределенные данные (литералы), и ресурсы, и библиотеки. Здесь же вступают в силу директивы *#property*.
2. Выделение памяти под глобальные переменные и установка их начальных значений — это системная инициализация, выполняемая средой исполнения. Напомним, что в разделе Инициализация, изучая старт программы под отладчиком по шагам, мы видели, что в стеке находится запись @global_initializations — это и есть блок кода для данного пункта, который создается неявным образом компилятором. Если в программе используются глобальные объекты классов/структур, на данном этапе будут вызваны их конструкторы.
3. Вызов обработчика события *OnInit* (если он есть): в нем проводится более высокоуровневая, прикладная инициализация — каждая программа выполняет её самостоятельно, по необходимости. Например, это может быть динамическое распределение памяти под массивы объектов, для которых по тем или иным причинам нужно использовать параметрические конструкторы вместо конструкторов по умолчанию. Как нам известно, автоматическое распределение памяти под массивы использует только конструкторы по умолчанию, и потому их невозможно проинициализировать в рамках предыдущего шага (2). Также это может быть открытие файлов, вызов встроенных функций API для включения необходимых режимов графика и т.д.

4. В цикле, пока пользователь не закроет программу или терминал, или не выполнит одно из других действий, требующих переинициализации (см. далее):
   - вызов прочих обработчиков по мере наступления соответствующих событий.
5. Вызов обработчика события *OnDeinit* (если он есть) при обнаружении попытки закрыть программу со стороны пользователя или программно (соответствующая функция ExpertRemove доступна только в экспертах и скриптах).
6. Финализация: освобождение выделенной памяти и прочих ресурсов, которые программист не посчитал нужным освободить в *OnDeinit*. Если в программе используется ООП, здесь вызываются деструкторы глобальных и статических объектов.
7. Выгрузка программы.

Скрипты и сервисы априори не имеют обработчиков *OnInit* и *OnDeinit*, и потому для них шаги 3 и 5 отсутствуют, а шаг 4 вырождается в единственный вызов *OnStart*.

Системная инициализация (под пунктом 2) неотделима от загрузки, то есть всегда следует после неё. А финализация всегда предшествует выгрузке. Однако индикаторы и эксперты по-разному проходят этапы загрузки и выгрузки в различных ситуациях. Поэтому вызовы *OnInit* и *OnDeinit* (пункты 3 и 5) являются теми опорными точками, в которых можно обеспечить согласованную прикладную инициализацию и деинициализацию экспертов и индикаторов.

Загрузка индикаторов и экспертов происходит по следующим причинам:

Причина		
Прикрепление к графику пользователем	+	+
Запуск терминала (если программа была прикреплена к графику перед предыдущим закрытием терминала)	+	+
Загрузка шаблона (если в шаблоне указана прикрепленная к графику программа)	+	+
Смена профиля (если программа прикреплена к одному из графиков профиля)	+	+
После удачной перекомпиляции, если программа была прикреплена к графику	+	+
Смена активного счета	+	+
Смена символа или периода графика, к которому прикреплен индикатор	+	-
Изменение входных параметров индикатора	+	-
Подключение к счету (авторизация), даже если номер счета не менялся	-	+

В более сжатом виде можно сформулировать следующее правило: эксперты не проходят полный жизненный цикл, то есть не перезагружаются при смене символа/таймфрейма графика, а также при изменении входных параметров.

Поэтому аналогичную несимметричность можно наблюдать и при выгрузке программ. Причинами выгрузки для индикаторов и экспертов являются:

Причина		
Удаление с графика	+	+
Закрытие терминала (когда программа прикреплена к графику)	+	+
Загрузка шаблона на график, к которому прикреплена программа	+	+
Закрытие графика, к которому прикреплена программа	+	+
Смена профиля, если программа прикреплена к одному из графиков сменяемого профиля	+	+
Смена счета, к которому подключен терминал	+	+
Смена символа и/или периода графика, к которому прикреплен индикатор	+	-
Изменение входных параметров индикатора	+	-
Прикрепление другого или того же эксперта к графику, на котором уже работает текущий эксперт	-	+
Вызов функции ExpertRemove	-	+

Причину деинициализации можно узнать из программы с помощью функции *UninitializeReason* или флага *_UninitReason* (см. раздел Проверка статуса и причины остановки MQL-программы).

Обратим еще раз внимание: при смене символа или таймфрейма графика, а также при смене входных параметров эксперт остается в памяти, то есть пункты 6-7 (финализация и выгрузка) и пункты 1-2 (загрузка и первичное распределение памяти) не выполняются, поэтому значения глобальных и статических переменных не сбрасываются. При этом последовательно вызываются обработчики *OnDeinit* и *OnInit* на старом и на новом символе/таймфрейме соответственно (или при старых и новых настройках).

Следствием того, что глобальные переменные не очищаются в экспертах, является то, что код деинициализации *_UninitReason* остается в неизменном виде для анализа в обработчике *OnInit*. Новый код будет записан в переменную только в случае следующего события, непосредственно перед вызовом *OnDeinit*.

Все события, поступившие для эксперта до завершения функции *OnInit*, пропускаются.

При первом запуске MQL-программы диалог настроек выводится между шагами 1 и 2. При смене входных параметров диалог настройки по-разному вклинивается в общий цикл в зависимости от типа программы: для индикаторов он по-прежнему появляется перед шагом 2, а для экспертов — перед шагом 3.

К книге прилагается шаблон индикатора и эксперта с одним и тем же названием *LifeCycle.mq5*. В нем реализован вывод в журнал этапов глобальной инициализации/финализации и в обработчиках *OnInit/OnDeinit*. Разместите программы на графике и посмотрите, какие события происходят в ответ на различные действия пользователя: загрузку/выгрузку, смену параметров, переключение символа/таймфрейма.

Загрузка скрипта происходит только при набрасывании его на график. Если скрипт выполняется в цикле, его перекомпиляция не приводит к перезапуску.

Загрузка и выгрузка сервиса происходит по командам контекстного меню в интерфейсе терминала. При перекомпиляции уже выполняющегося сервиса он перезапускается. Напомним, что активные экземпляры сервисов автоматически загружаются при старте терминала и выгружаются при закрытии.

В следующих двух разделах мы рассмотрим особенности запуска разных MQL-программ на уровне обработчиков соответствующих событий.

5.1.5 Опорные события индикаторов и советников: OnInit и OnDeinit

В интерактивных MQL-программах — индикаторах и экспертах — среда генерирует два события для подготовки к запуску (*OnInit*) и к остановке (*OnDeinit*). В скриптах и сервисах таких событий нет, потому что они не принимают асинхронных событий: после передачи управления их единственному обработчику события *OnStart* и вплоть до завершения работы, контекст выполнения потока скрипта/сервиса находится в коде MQL-программы. В отличие от этого, для индикаторов и экспертов нормальный ход работы предполагает, что среда будет многократно вызывать их специфические функции обработки событий (мы рассмотрим их в разделах, посвященных индикаторам и советникам), и каждый раз, предприняв необходимые действия, программы будут возвращать управление терминалу для холостого ожидания новых событий.

int OnInit()

Функция *OnInit* является обработчиком одноименного события, которое генерируется после загрузки эксперта или индикатора. Функцию можно определять только по необходимости.

Функция должна вернуть одно из значений перечисления ENUM_INIT_RETCODE.

Идентификатор	Описание
INIT_SUCCEEDED	Инициализация прошла успешно, выполнение программы можно продолжать; соответствует значению 0
INIT_FAILED	Неудачная инициализация, продолжать выполнение нельзя из-за неустранимых ошибок (например, не удалось создать файл или вспомогательный индикатор); значение 1
INIT_PARAMETERS_INCORRECT	Некорректный набор входных параметров, выполнение программы невозможно
INIT_AGENT_NOT_SUITABLE	Специфический код для работы в тестере: по каким-то причинам данный агент не подходит для проведения тестирования (например, недостаточно оперативной памяти, нет поддержки OpenCL и т.д.)

Если *OnInit* вернет любой ненулевой код возврата, это означает неудачную инициализацию и следом генерируется событие *Deinit* с кодом причины деинициализации REASON_INITFAILED (см. далее).

Функция *OnInit* может быть описана с типом результата *void*: в этом случае инициализация всегда считается успешной.

> В обработчике *OnInit* важно проверять наличие всей необходимой информации окружения, и если она недоступна, откладывать подготовительные действия на следующие события поступления тиков или таймера. Дело в том, что при запуске терминала событие *OnInit* часто срабатывает до того, как установится соединение с сервером, и потому многие свойства финансовых инструментов и торгового счета еще неизвестны. В частности, стоимость одного пункта конкретного символа может возвращаться равной нулю.

void OnDeinit(const int reason)

Функция *OnDeinit* (если она определена) вызывается при деинициализации эксперта или индикатора. Функция необязательна.

Параметр *reason* содержит код причины деинициализации. Возможные значения приведены в следующей таблице.

Константа	Значение	Описание
REASON_PROGRAM	0	Эксперт прекратил свою работу, вызвав функцию *ExpertRemove*
REASON_REMOVE	1	Программа удалена с графика
REASON_RECOMPILE	2	Программа перекомпилирована
REASON_CHARTCHANGE	3	Изменен символ или период графика
REASON_CHARTCLOSE	4	График закрыт
REASON_PARAMETERS	5	Изменены входные параметры
REASON_ACCOUNT	6	Активирован другой счет либо произошло переподключение к торговому серверу вследствие изменения настроек счета
REASON_TEMPLATE	7	Применен другой шаблон графика
REASON_INITFAILED	8	Обработчик *OnInit* вернул ненулевое значение
REASON_CLOSE	9	Терминал был закрыт

Этот же код можно получить в любом месте программы с помощью функции *UninitializeReason*, если в MQL-программе взведен флаг остановки *_StopFlag*.

В файле *AllInOne.mqh* описан класс *Finalizer*, позволяющий "перехватить" код деинициализации в деструкторе через вызов *UninitializeReason*. То же самое значение мы должны получить и в обработчике *OnDeinit*.

```
class Finalizer
{
   static const Finalizer f;
public:
   ~Finalizer()
   {
      PRTF(EnumToString((ENUM_DEINIT_REASON)UninitializeReason()));
   }
};

static const Finalizer Finalizer::f;
```

Для удобства перевода кодов в строковое представление (названия причин) посредством *EnumToString*, в файле *Uninit.mqh* описано перечисление ENUM_DEINIT_REASON с константами из вышеприведенной таблицы. В журнале будут выводиться записи вида:

```
OnDeinit DEINIT_REASON_REMOVE
EnumToString((ENUM_DEINIT_REASON)UninitializeReason())=DEINIT_REASON_REMOVE / ok
```

При смене символа или таймфрейма графика, на котором расположен индикатор, он выгружается и загружается вновь. При этом последовательность срабатывания события *OnDeinit* в старой копии и *OnInit* в новой копии не определена. Это происходит из-за особенностей асинхронной обработки событий терминалом. Иными словами, может случиться не совсем логичная ситуация, что новая копия загрузится и начнет инициализацию до того, как будет полностью выгружена старая. Если индикатор выполняет какую-то настройку графика в *OnInit* (например, создает графический объект), то без принятия особых мер выгружаемая копия может тут же "почистить" график (удалить объект, посчитав его своим). В конкретном случае графических объектов есть частное решение: объектам можно давать имена, включающие префиксы символа и таймфрейма (а также контрольной суммы значений входных переменных), но в общем случае оно не подойдет. Для универсального решения проблемы следует реализовать какой-либо механизм синхронизации, например, на глобальных переменных или ресурсах.

При тестировании индикаторов в тестере разработчики MetaTrader 5 решили не генерировать событие *OnDeinit*. Их идея состоит в том, что индикатор может создавать некоторые графические объекты, которые он обычно удаляет в обработчике *OnDeinit*, однако пользователю желательно их видеть после завершения теста. На самом деле автор MQL-программы может при желании и сам обеспечить аналогичное поведение, оставляя объекты при положительной проверке режима *MQLInfoInteger(MQL_TESTER)*. Это тем более странно, поскольку обработчик *OnDeinit* вызывается после теста эксперта, а эксперт может точно также удалять объекты в *OnDeinit*. Сейчас же исключительно для индикаторов получается, что обеспечить штатное поведение самого обработчика *OnDeinit* в тестере нельзя. Более того, не выполняется и прочая финализация, например, не вызываются деструкторы глобальных объектов.

Таким образом, если вам требуется выполнить после тестового прогона подсчет статистики, сохранение файла или иное действие, изначально предусмотренное для *OnDeinit* индикатора, придется отказаться от теста самого индикатора и перенести алгоритмы в эксперт.

5.1.6 Главная функция скриптов и сервисов: OnStart

Утилитарные программы — скрипты и сервисы — выполняются в терминале посредством вызова их единственной функции обработки событий *OnStart*.

void OnStart()

Функция не имеет параметров и не возвращает значения. Она лишь служит "точкой входа" в прикладную программу со стороны терминала.

Скрипты предназначены, как правило, для разовых действий, выполняемых на графике (позднее мы изучим все возможности, которые предоставляет API графиков). Например, скрипт можно использовать для установки сетки ордеров или, наоборот, для закрытия всех прибыльных открытых позиций, для автоматического нанесения разметки графическими объектами или временного скрытия всех объектов.

Допускается использовать в скриптах постоянные действия, обернутые в "бесконечный" цикл, в котором, как уже говорилось ранее, следует всегда проверять признак остановки (*_StopFlag*) и периодически освобождать процессор (*Sleep*). Здесь следует помнить, что при выключении и включении терминала скрипт прийдется запускать снова.

Поэтому для такой постоянной активности, если она не связана напрямую с графиком, лучше использовать сервис. Стандартным приемом в реализации сервиса является как раз "бесконечный" цикл.

В предыдущих частях книги практически все примеры были реализованы в виде скриптов. В качестве примера сервиса можно напомнить программу *GlobalsWithCondition.mq5* из раздела Синхронизация программ с помощью глобальных переменных. В следующем разделе об остановке советников и скриптов с помощью функции *ExpertRemove* будет приведен еще один.

5.1.7 Программное удаление советников и скриптов: ExpertRemove

При необходимости разработчик может организовать остановку и выгрузку MQL-программ двух типов: экспертов и скриптов. Это делается с помощью функции *ExpertRemove*.

void ExpertRemove()

Функция не имеет параметров и не возвращает значения. Она отправляет в среду исполнения MQL-программ запрос на удаление текущей программы. Фактически это приводит к взведению флага *_StopFlag* и прекращению приема (и обработки) всех последующих событий. После этого программе дается 3 секунды на то, чтобы правильно закончить свою работу: освободить ресурсы, прервать циклы в алгоритмах и т.д. Если программа этого не сделает, она будет выгружена принудительно, с потерей промежуточных данных.

В индикаторах и сервисах данная функция не работает (программа продолжает выполняться).

Каждое обращение к функции сигнализируется в журнале записью "Вызвана функция ExpertRemove()" ("ExpertRemove() function called").

Функция в первую очередь востребована в экспертах, которые невозможно прервать иным способом. В случае скриптов, как правило, более простым способом является прерывание цикла (если он есть) с помощью оператора *break*. Но если циклы вложены, или алгоритм использует множество вызовов функций одна из другой, проще на разных уровнях учитывать флаг остановки в условиях для продолжения расчетов, а в случае ошибочной ситуации взводить этот флаг с помощью *ExpertRemove*. Если не использовать данный встроенный флаг, в любом случае пришлось бы вводить аналогичную по назначению глобальную переменную.

В скрипте *ScriptRemove.mq5* приведен пример использования *ExpertRemove*.

Потенциальную проблему в работе алгоритма, которая приводит к необходимости выгрузки скрипта, эмулирует класс *ProblemSource*. В его конструкторе случайным образом вызывается *ExpertRemove*.

```
class ProblemSource
{
public:
   ProblemSource()
   {
      // имитируем проблему во время создания объекта, например,
      // с захватом неких ресурсов, таких как файл и т.д.
      if(rand() > 20000)
      {
         ExpertRemove(); // установит _StopFlag в true
      }
   }
};
```

Далее объекты этого класса создаются на глобальном уровне и внутри вспомогательной функции.

```
ProblemSource global; // объект может породить ошибку

void SubFunction()
{
   ProblemSource local; // объект может породить ошибку
   // имитируем некую работу (надо проверить целостность объекта!)
   Sleep(1000);
}
```

Теперь используем *SubFunction* в работе *OnStart* внутри цикла с условием на *IsStopped*.

```
void OnStart()
{
   int count = 0;
   // цикл пока не остановит пользователь или сама программа
   while(!IsStopped())
   {
      SubFunction();
      Print(++count);
   }
}
```

Вот пример вывода в журнал (каждый запуск будет отличаться из-за случайности):

```
1
2
3
ExpertRemove() function called
4
```

Обратите внимание, что если ошибка произойдет при создании глобального объекта, цикл не выполнится ни разу.

Поскольку эксперты могут запускаться в тестере, там функция *ExpertRemove* также имеет эффект, причем он зависит от места вызова функции. Если это делается внутри обработчика *OnInit*, произойдет отмена тестирования, то есть одного прогона тестера на текущем наборе параметров эксперта. Такое завершение рассматривается как ошибка инициализации. При вызове *ExpertRemove* в любом другом месте алгоритма тестирование советника прервется досрочно, но будет обработано штатным образом, с вызовом *OnDeinit* и *OnTester*. В этом случае будет получена накопленная торговая статистика и значение критерия оптимизации, с учетом того, что эмулируемое серверное время *TimeCurrent* не достигнет конечной даты в настройках тестера.

5.2 Скрипты и сервисы

В данной главе мы обобщим и изложим полную техническую информацию о скриптах и сервисах, с которыми мы уже начали знакомиться в предыдущих частях книги.

Скрипты и сервисы имеют одинаковые принципы организации и исполнения программного кода. Как мы знаем, их главная функция *OnStart* является и единственной. Скрипты и сервисы не могут обрабатывать другие события.

Вместе с тем, есть пара существенных отличий. Скрипты выполняются в контексте графика и имеют доступ к его свойствам напрямую — через встроенные переменные, такие как *_Symbol*, *_Period*, *_Point* и другие. Мы изучим их в разделе Свойства графика. Сервисы же работают сами по себе, не привязанные ни к каким окнам, хотя и имеют возможность анализировать с помощью специальных функций все графики (эти же *Chart-функции* можно использовать в остальных типах программ: скриптах, индикаторах и экспертах).

С другой стороны, созданные экземпляры сервиса автоматически восстанавливаются терминалом в следующие сеансы работы. Иными словами, сервис, будучи однажды запущенным, всегда остается в деле, пока пользователь его не остановит. В отличие от этого, скрипт удаляется при выключении терминала или закрытии графика.

> Обратите внимание, что сервис выполняется в терминале, как и все остальные типы MQL-программ, а потому закрытие терминала останавливает и сервис. Работа активного сервиса возобновится при следующем запуске терминала. Бесперебойную работу MQL-программ может обеспечить только постоянно загруженный терминал, например, на VPS-сервере.

В скриптах и сервисах можно установить Общие свойства программ с помощью директив *#property*. В дополнение к ним существуют свойства, характерные именно для скриптов и сервисов — о них будет сказано в двух следующих разделах.

Скрипты, выполняющиеся в данный момент на графиках, перечисляются в том же списке, который показывает запущенные эксперты — в диалоге *Эксперты*, открываемом с помощью команды *Список экспертов* контекстного меню графика. Оттуда их можно принудительно удалить с графика.

Управление сервисами возможно только из окна *Навигатора*.

5.2.1 Скрипты

Скриптом является MQL-программа с единственным обработчиком *OnStart*, при условии отсутствия директивы *#property service* (иначе получится сервис, см. следующий раздел).

По умолчанию скрипт сразу начинает выполняться при его размещении на графике. Разработчик имеет возможность запросить у пользователя подтверждение на запуск, добавив в начало файла директиву *#property script_show_confirm*. Терминал в этом случае покажет сообщение с вопросом "Вы действительно хотите присоединить 'программу' к графику 'символ,таймфрейм'?" и кнопками *Да* и *Нет*.

Скрипты, также как и другие программы, могут иметь входные переменные. Однако для скриптов диалог ввода параметров не показывается по умолчанию, даже если в скрипте определены *input*-ы. Чтобы гарантировать открытие диалога свойств перед запуском скрипта, следует применить другую директиву *#property script_show_inputs*. Она имеет приоритет перед *script_show_confirm*, то есть вывод диалога отключает запрос на подтверждение (поскольку диалог сам выступает в аналогичной роли). Директива вызывает диалог даже при отсутствии входных переменных: его имеет смысл открывать, чтобы показать пользователю описание и версию продукта (они выводятся на закладке *Общие*, первой по счету).

В следующей таблице показаны варианты сочетаний директив *#property* и их эффект на программу.

Директива Эффект	script_show_confirm	script_show_inputs
Немедленный запуск	нет	нет
Запрос подтверждения	да	нет
Открытие диалога свойств	неважно	да

Простой пример скрипта с заготовками директив находится в файле *ScriptNoComment.mq5*. Назначение скрипта следующее. Иногда MQL-программы оставляют после себя ненужные комментарии в левом верхнем углу графика. Комментарии хранятся в chr-файлах вместе с графиком, поэтому даже после перезагрузки терминала они восстанавливаются. Данный скрипт позволяет очистить комментарий или установить его в произвольное значение. Если *Назначить горячую клавишу* скрипту с помощью одноименной команды контекстного меню *Навигатора*, можно будет чистить комментарий текущего графика одним нажатием.

Изначально директивы *script_show_confirm* и *script_show_inputs* отключены за счет превращения в строчные комментарии. Вы можете поэкспериментировать с разным сочетанием директив, раскомментируя их по одной или одновременно.

```
//#property script_show_confirm
//#property script_show_inputs

input string Text = "";

void OnStart()
{
   Comment(""); // чистим комментарий
}
```

5.2.2 Сервисы

Сервисом является MQL-программа с единственным обработчиком *OnStart* и директивой *#property service*.

Напомним, что после успешной компиляции сервиса требуется создать и настроить его экземпляр (один или несколько) с помощью команды *Добавить сервис* в контекстном меню *Навигатора* терминала.

В качестве примера сервиса решим небольшую прикладную проблему, которая часто возникает у разработчиков MQL-программ. Многие из них практикуют привязку своих программ к номеру счета пользователя. Речь здесь не обязательно идет о платном продукте, а может касаться распространения среди друзей и знакомых для сбора статистики или удачных настроек. При этом пользователь может помимо рабочего реального счета регистрировать демо-счета. Время существования таких счетов, как правило, ограничено, и потому обновлять ради них привязку каждую пару недель довольно неудобно. Для этого надо править исходный код, компилировать и отсылать программу заново.

Вместо этого мы можем разработать сервис, который будет регистрировать в глобальных переменных (или файлах) номера счетов, к которым произошло успешное подключение из данного терминала.

Технология привязки основывается на попарном шифровании (или, как вариант, хешировании) номеров счетов: прежнего счета логина и нового счета логина. Предыдущий счет должен быть мастер-счетом (на который "выписана" условная привязка), чтобы общая подпись пары расширяла права пользования продуктом на новый счет. В качестве ключа используется секрет, известный только внутри программ (предполагается, что все они поставляются в закрытом, откомпилированном виде). Результатом операции будет строка в формате *Base64*. В реализации применены функции MQL5 API, часть которых еще предстоит изучить, в частности, получение номера счета через *AccountInfoInteger* и шифрование функцией *CryptEncode*. Проверка связи с сервером выполняется известной нам функцией *TerminalInfoInteger* (см. раздел Проверка сетевых подключений).

Сервис не обязан знать, какие счета являются мастер-счетами, а какие дополнительными, — ему достаточно особым образом "подписывать" пары любых последовательно залогиненных счетов. А вот уже конкретная прикладная программа должна дополнить процесс проверки своей "лицензии": помимо сравнения текущего счета с мастер-счетом следует повторить алгоритм сервиса: составить пару [мастер-счет;текущий счет], подсчитать для неё зашифрованную "подпись" и проверить, есть ли она среди глобальных переменных.

Украсть такую лицензию путем переноса на другой компьютер получится, только если подключаться к тому же счету в режиме торговли (не инвестора). Недобросовестный пользователь, конечно, может создавать демо-счета для других людей. Поэтому желательно усовершенствовать защиту. В текущей реализации глобальная переменная просто делается временной, то есть удаляется вместе с окончанием сеанса терминала, но это не предотвращает её возможное копирование.

В качестве дополнительных мер можно, например, шифровать в "подписи" время её создания и предусмотреть истечение прав каждые сутки (или с другой периодичностью). Другой вариант — генерация случайного числа при запуске сервиса и добавление его в подписываемую информацию наравне с номерами счетов. Это число известно только внутри сервиса, но он может транслировать его заинтересованным MQL-программам на графики с помощью функции *EventChartCustom*. Таким образом, "подпись" продолжит действовать в данном экземпляре терминала вплоть до завершения сеанса. В каждом сеансе будет генерироваться и рассылаться новое случайное число, поэтому оно не подойдет для других терминалов. Наконец, самым простым и удобным вариантом будет, вероятно, добавление в "подпись" времени старта системы: *(TimeLocal() - GetTickCount() / 1000)* или производного от него.

Из различных типов MQL-программ только некоторые продолжают выполняться между переключениями счетов и позволяют реализовать данную схему защиты. Поскольку требуется единообразным образом защитить MQL-программы любых типов, включая индикаторы и эксперты (которые перезагружаются при смене счета), имеет смысл поручить эту задачу сервису. Тогда сервис, постоянно работающий с момента загрузки терминала и до его закрытия, будет контролировать логины и генерировать уполномочивающие "подписи".

Исходный код сервиса приведен в файле *MQL5/Services/MQL5Book/p5/ServiceAccount.mq5*. Во входных параметрах задается мастер-счет и префикс глобальных переменных, в которых будут сохраняться "подписи". В реальных программах списки мастер-счетов должны быть зашиты в исходном коде, а вместо глобальных переменных лучше использовать файлы в папке *Common*, чтобы охватить также и тестер.

```
#property service

input long MasterAccount = 123456789;
input string Prefix = "!A_";
```

Главная функция сервиса выполняет свою работу следующим образом: в бесконечном цикле с паузами в 1 секунду отслеживаем смены счетов и сохраняем последний номер, создаем для пары "подпись" и записываем её в глобальную переменную. Созданием подписи занимается функция *Cipher*.

```
void OnStart()
{
   static long account = 0; // предыдущий счет логина
   
   for(; !IsStopped(); )
   {
      // требуем связи, успешного логина и полного доступа (не инвесторского)
      const bool c = TerminalInfoInteger(TERMINAL_CONNECTED)
                  && AccountInfoInteger(ACCOUNT_TRADE_ALLOWED);
      const long a = c ? AccountInfoInteger(ACCOUNT_LOGIN) : 0;
      
      if(account != a) // счет сменился
      {
         if(a != 0) // текущий счет
         {
            if(account != 0) // предыдущий счет
            {
               // перенос авторизации с одного на другой
               const string signature = Cipher(account, a);
               PrintFormat("Account %I64d registered by %I64d: %s",
                  a, account, signature);
               // сохраним запись о связи счетов
               if(StringLen(signature) > 0)
               {
                  GlobalVariableTemp(Prefix + signature);
                  GlobalVariableSet(Prefix + signature, account);
               }
            }
            else // первый счет авторизован, ждем второго
            {
               PrintFormat("New account %I64d detected", a);
            }
            // запомним последний активный счет
            account = a;
         }
      }
      Sleep(1000);
   }
}
```

Функция *Cipher* использует специальное объединение *ByteOverlay2*, чтобы представить пару номеров счетов (типа *long*) в виде байтового массива, который передается для шифрования в *CryptEncode* (здесь выбран метод шифрования CRYPT_DES, но его можно заменить на CRYPT_AES128, CRYPT_AES256 или просто хеширование CRYPT_HASH_SHA256 (с секретом в качестве соли), если восстановление информации из подписи не требуется).

```
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: замените секрет на свое кодовое слово
   // TODO: для методов CRYPT_AES128/CRYPT_AES256 нужны массивы 16/32 байта
   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;
}
```

Далее любая программа в терминале может проверить, нет ли в глобальных переменных "лицензии" для текущего счета. Это делается с помощью функций *CheckAccounts* и *IsCurrentAccountAuthorizedByMaster*. Они приведены в сервисе просто для демонстрации.

Функция *CheckAccounts* выполняет проверку по всем мастер-счетам, зашитым в код, не совпадает ли один из них с текущим.

```
bool CheckAccounts()
{
   const long accounts[] = {MasterAccount}; // TODO: заполнить массив константами
   for(int i = 0; i < ArraySize(accounts); ++i)
   {
      if(IsCurrentAccountAuthorizedByMaster(accounts[i])) return true;
   }
   return false;
}
```

IsCurrentAccountAuthorizedByMaster принимает в качестве параметра номер одного мастер-счета, воссоздает для него "подпись" в паре с текущим счетом и анализирует совпадения.

```
bool IsCurrentAccountAuthorizedByMaster(const long data)
{
   const long a = AccountInfoInteger(ACCOUNT_LOGIN);
   if(a == data) return true; // прямое совпадение
   const string s = Cipher(data, a); // пересчитываем "подпись"
   if(a != 0 && GlobalVariableGet(Prefix + s) == a)
   {
      Print("Sub-License is active: ", s);
      return true;
   }
   return false;
}
```

Предположим, что программам разрешено выполняться на счете 123456789, и он в данный момент активен. При запуске сервис среагирует записью в журнале:

```
New account 123456789 detected
```

Если затем сменить номер счета, например, на 5555555, получим следующую сигнатуру:

```
Account 5555555 registered by 123456789: jdVKxUswBiNlZzDAnV3yxw==
```

Если остановить и снова запустить сервис, увидим проверку счета 5555555 в действии (вызов функции *CheckAccounts* встроен для демонстрации в начало *OnStart*).

```
Sub-License is active: jdVKxUswBiNlZzDAnV3yxw==
Account 123456789 registered by 5555555: ZWcwwJ1d8seN1UrFSzAGIw==
```

Лицензия сработала для нового счета. Если переключиться обратно, сгенерируется "пропуск" с текущего счета на предыдущий (это следствие того, что сервис не знает, какие счета являются основными, а какие временными, и такая "подпись", скорее всего, не потребуется в программах).

Для опосредованной авторизации нового счета потребуется снова войти в мастер-счет и только затем переключиться на новый: это создаст другую глобальную переменную с зашифрованной парой [мастер-счет; новый счет].

Данная версия сервиса не проверяет, чтобы мастер-счет был реальным, а зависимый — демонстрационным. Каждое из этих ограничений можно добавить.

5.2.3 Ограничения для скриптов и сервисов

В скриптах и сервисах запрещены все функции, входящие в группу по работе с индикаторами (эти функции будут описаны в соответствующей главе):

- *SetIndexBuffer*
- *IndicatorSetDouble*
- *IndicatorSetInteger*
- *IndicatorSetString*
- *PlotIndexSetDouble*
- *PlotIndexSetInteger*
- *PlotIndexSetString*
- *PlotIndexGetInteger*

Кроме того, в скриптах и сервисах не имеет смысла использовать обработчик событий таймера *OnTimer* (впрочем, как и любые другие обработчики) и функции таймера:

- *EventSetMillisecondTimer*
- *EventSetTimer*
- *EventKillTimer*

Поскольку скрипты и сервисы не поддерживаются тестером, в них также нельзя использовать *Tester*-функции, они вызовут ошибки ERR_FUNCTION_NOT_ALLOWED (4014).

5.3 Временны́е ряды (таймсерии)

Временны́е ряды — это массивы с данными, в которых индексы элементов соответствуют упорядоченным временным отсчетам. В силу прикладной специфики терминала, практически вся информация, необходимая трейдеру, предоставляется в виде временных рядов или таймсерий. К ним, в частности, относятся массивы котировок, тиков, показания технических индикаторов и другие. С этими данными работает и подавляющее большинство MQL-программ, а потому для них выделена группа функций в MQL5 API, которые мы рассмотрим в данном разделе.

Особенностью доступа к массивам в MQL5 является то, что разработчик может устанавливать по желанию одно из двух направлений индексации:

- Обычное (прямое) — нумерация элементов идет от начала массива к концу (от старых отсчетов к новым);
- Обратное (таймсерия) — нумерация идет от конца массива к началу (от новых отсчетов к старым).

Мы уже освещали данный вопрос в разделе Направление индексации массивов как в таймсерии.

Смена режима индексации выполняется с помощью функции *ArraySetAsSeries* и не влияет на физическое размещение массива в памяти. Изменяется только способ обращения к элементам по номеру: если в обычном мы получаем i-ый элемент как *array[i]*, то в режиме таймсерии эквивалентная формула равна *array[N - i - 1]*, где N — размер массива ("эквивалентная" — потому что прикладному разработчику не нужно везде делать такой пересчет — его автоматически делает терминал, если для массива задан режима индексации таймсерии). Это иллюстрирует следующая таблица (для символьного массива из 10 элементов).

Элементы массива	A	B	C	D	E	F	G	H	I	J
Обычный индекс	0	1	2	3	4	5	6	7	8	9
Индекс как в таймсерии	9	8	7	6	5	4	3	2	1	0

Напомним, что индексация массивов всегда начинается с нуля.

Когда речь идет о массивах котировок и прочих постоянно обновляемых данных, новые элементы дописываются физически в конец массива. Однако с точки зрения торговли следует учитывать самые свежие данные и принимать их за точку отсчета при анализе истории. Именно поэтому удобно постоянно иметь под индексом 0 текущий (последний) бар, а предыдущие отсчитывать от него в прошлое. Таким образом, мы и получаем индексацию таймсерии.

По умолчанию массивы имеют направление индексации слева направо. Если представить, что такой массив отображен на стандартном графике MetaTrader 5, то чисто визуально, элемент с

индексом 0 будет находиться на крайней левой позиции, а последний — на крайней правой. В таймсериях с обратной индексацией 0-й элемент соответствует крайней правой позиции, а последний элемент — крайней левой. Так как таймсерии хранят историю ценовых данных по финансовым инструментам в привязке ко времени, более свежие данные в них всегда находятся правее старых.

Элемент с индексом ноль в массиве-таймсерии содержит информацию о самой последней котировке по инструменту. Нулевой бар, как правило, незавершен, поскольку продолжает формироваться.

Еще одной характеристикой котировочной таймсерии является её период, то есть промежуток времени между соседними отсчетами. Этот период также называется "таймфреймом" и может быть переформулирован более точно. Таймфрейм — промежуток времени, в течение которого формируется один бар котировок, причем его начало и конец выровнены в абсолютном времени с тем же шагом. Например, в таймфрейме "1 час" (H1) бары начинаются строго в 0 минут каждого часа суток. Начало каждого такого периода включено в текущий бар, а конец принадлежит уже следующему бару.

В разделе Символы и таймфреймы приведен полный перечень стандартных таймфреймов.

В рамках метафоры временных рядов и таймсерий работают, как правило, и буфера технических индикаторов, но мы изучим их особенности позднее.

При необходимости в любой MQL-программе можно запросить значения таймсерий по любому символу и таймфрейму, а также значения индикаторов, рассчитанных на любом символе и таймфрейме. Для получения таких данных служат *Copy*-функции, среди которых есть несколько, читающих массивы цен разных типов по отдельности (например, *Open*, *High*, *Low*, *Close*) или массивы структур *MqlRates*, содержащих все характеристики каждого бара.

Бары и тики

Помимо баров с котировками MetaTrader 5 предоставляет пользователям и MQL-программам возможность анализировать тики — элементарные изменения цен, на основе которых и строятся бары. Каждый тик содержит время с точностью до миллисекунды, несколько видов цен (*Bid*, *Ask*, *Last*) и флагов, описывающих суть изменений, а также торговый объем сделки. Чуть позже мы изучим соответствующую структуру *MqlTick* в разделе Работа с массивами реальных тиков.

В зависимости от типа торгового инструмента бары могут строиться по ценам *Bid* или *Last*. В частности, цены *Last* доступны для биржевых инструментов, по которым также транслируется стакан цен ("глубина рынка"). Для небиржевых инструментов, таких как Forex или CFD, используется цена *Bid*.

Те периоды, в течение которых не было изменений цен, не порождают баров — такова особенность представления цен в MetaTrader 5. Например, если таймфрейм равен 1 дню (D1), то пара баров для выходных дней, как правило, отсутствует, и после пятницы сразу идет понедельник.

Котировочный бар появляется, если в соответствующий ему промежуток времени случился хотя бы один тик. При этом время открытия бара всегда выровнено строго по границе периода, даже если первый тик пришел позже (как обычно и происходит). Например, первый бар M1 за сутки может быть сформирован в 00:05, если 4 минуты после полуночи не было тиков, а затем изменение цен случилось в 00:05:15 (то есть на 15-й секунде пятой минуты).

Таким образом, тик включается в тот или иной бар на основе следующего соотношения меток времени: $T_{open} <= T_{tick} < T_{open} + P$, где T_{open} — время открытия бара, T_{tick} — время тика, $T_{open} + P$ — время открытия следующего потенциального бара через период P ("потенциальным" бар назван, потому что его наличие зависит от других тиков).

5.3.1 Символы и таймфреймы

Временные ряды с котировками идентифицируются двумя параметрами: именем символа (финансового инструмента) и таймфреймом (периодом).

Пользователь видит перечень символов в окне *Обзора рынка* и может редактировать его на основе общего списка, предоставляемого брокером (диалог *Символы*). Для MQL-программ существует набор функций, с помощью которых можно делать то же самое: "пролистать" все символы, узнать их свойства и добавить или удалить в/из *Обзор рынка*. Этим функциям будет посвящена отдельная глава.

Однако для запроса таймсерий достаточно лишь знать имя символа — это строка, содержащая обозначение существующего финансового инструмента. Его, например, может задавать пользователь во входной переменной. Кроме того, символ текущего графика можно узнать из встроенной переменной *_Symbol* (или функции *Symbol*), но для нашего удобства все функции таймсерий поддерживают соглашение, что значение NULL также соответствует символу текущего графика.

Теперь обратимся к таймфреймам. В системе определен 21 стандартный таймфрейм: каждый задается элементом в специальном перечислении ENUM_TIMEFRAMES.

Идентификатор	Значение (Hex)	Описание
PERIOD_CURRENT	0	Текущий период графика
PERIOD_M1	1 (0x1)	1 минута
PERIOD_M2	2 (0x2)	2 минуты
PERIOD_M3	3 (0x3)	3 минуты
PERIOD_M4	4 (0x4)	4 минуты
PERIOD_M5	5 (0x5)	5 минут
PERIOD_M6	6 (0x6)	6 минут
PERIOD_M10	10 (0xA)	10 минут
PERIOD_M12	12 (0xC)	12 минут
PERIOD_M15	15 (0xF)	15 минут
PERIOD_M20	20 (0x14)	20 минут
PERIOD_M30	30 (0x1E)	30 минут
PERIOD_H1	16385 (0x4001)	1 час
PERIOD_H2	16386 (0x4002)	2 часа

Идентификатор	Значение (Hex)	Описание
PERIOD_H3	16387 (0x4003)	3 часа
PERIOD_H4	16388 (0x4004)	4 часа
PERIOD_H6	16390 (0x4006)	6 часов
PERIOD_H8	16392 (0x4008)	8 часов
PERIOD_H12	16396 (0x400C)	12 часов
PERIOD_D1	16408 (0x4018)	1 день
PERIOD_W1	32769 (0x8001)	1 неделя
PERIOD_MN1	49153 (0xC001)	1 месяц

Как мы видели в разделе о Предопределенных переменных, период текущего графика программа может узнать из встроенной переменной *_Period* (или функции *Period*). Из столбца значений легко заметить, что передача нуля во встроенные функции, принимающие таймфрейм, будет означать период текущего графика.

Значение для минутных таймфреймов совпадают с количеством минут в них (например, 30 обозначает M30). Для часовых таймфреймов взведен бит 0x4000, а в нижнем байте содержится количество часов (например, 0x4003 для H3). Дневной период D1 кодируется как 24 часа — 0x4018 (0x18 равно 24). Наконец, недельный и месячный таймфреймы имеют свои собственные отличительные биты 0x8000 и 0xC000, соответственно, как индикаторы единиц измерения, а количество (в младшем байте) в обоих случаях равно 1.

Для удобной конвертации элементов перечисления в строки и обратно к книге прилагается заголовочный файл *Periods.mqh* (мы его уже использовали в примере работы с файлами и будем использовать в будущих примерах). Одна из входящих в него функций *StringToPeriod* использует в своем алгоритме вышеописанные особенности внутреннего битового представления элементов перечисления.

```
#define PERIOD_PREFIX_LENGTH 7 // StringLen("PERIOD_")

// получаем сокращенное имя периода без префикса "PERIOD_"
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);
}

// получаем значение периода по полному (PERIOD_H4) или краткому (H4) имени
ENUM_TIMEFRAMES StringToPeriod(string name)
{
   if(StringLen(name) < 2) return 0;
   // преобразуем полное имя "PERIOD_TN" к краткому "TN", если надо
   if(StringLen(name) > PERIOD_PREFIX_LENGTH)
   {
     name = StringSubstr(name, PERIOD_PREFIX_LENGTH);
   }
   // преобразуем цифровое окончание "N" в число, пропускаем "T"
   const int count = (int)StringToInteger(StringSubstr(name, 1));
   // сбрасываем возможную ошибку WRONG_STRING_PARAMETER(5040)
   // например, если на входе строка "MN1", то N1 - не число для StringToInteger
   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;
}
```

> Обратите внимание, что переменные *_Symbol* и *_Period* содержат актуальные данные только в MQL-программах, выполняющихся на графиках: скриптах, экспертах, индикаторах. В сервисах эти переменные пусты, и потому для доступа к таймсериям следует явным образом задавать имя символа и период, либо получать их каким-то образом извне.

Определяющим свойством таймфрейма является его длительность (продолжительность бара). MQL5 позволяет получить количество секунд, формирующих один бар конкретного таймфрейма, с помощью функции *PeriodSeconds*.

```
int PeriodSeconds(ENUM_TIMEFRAMES period = PERIOD_CURRENT)
```

Параметр *period* задает интересующий период как элемент из перечисления ENUM_TIMEFRAMES. Если параметр не указан, то возвращается количество секунд текущего периода графика, на котором запущена программа.

Примеры использования функции мы рассмотрим в индикаторе *IndDeltaVolume.mq5* в разделе Ожидание данных и управление видимостью, а также в индикаторе *UseM1MA.mq5* в разделе Использование встроенных индикаторов.

Для генерации таймфреймов нестандартной длительности, не входящей в указанный список, MQL5 API предоставляет пользовательские символы, однако они не позволяют вести торговлю, как на стандартных графиках, без модификации экспертов.

Кроме того важно отметить, что в MetaTrader 5 продолжительность баров внутри конкретной таймсерии или на графике всегда одинакова. Поэтому для построения графиков, в которых бары формируются не по времени, а по мере накопления иных параметров, в частности, объемов (эквиобъемные графики) или движения цены в одном направлении фиксированными шагами (ренко), также предполагается разрабатывать собственные решения на базе индикаторов (например, с типом отрисовки DRAW_CANDLES или DRAW_BARS) или опять-таки пользовательских символов.

5.3.2 Технические особенности организации и хранения таймсерий

Прежде чем приступать к практическим вопросам использования функций MQL5 API, предназначенных для работы с временными рядами, следует рассмотреть технические основы получения котировочных данных с сервера и их хранения в MetaTrader 5.

Прежде чем ценовые данные будут доступны в терминале для отображения на графиках и передачи в MQL-программы, они скачиваются с сервера и особым образом подготавливаются. Механизм обращения к серверу за данными не зависит от того, каким образом был инициирован запрос — пользователем при навигации по графику или программным способом на языке MQL5.

Данные поступают с сервера в сжатом формате: это экономно упакованные блоки минутных баров, которые, однако, не являются привычными барами M1.

Полученные с сервера данные автоматически распаковываются и сохраняются в специальном промежуточном формате HCC. Данные по каждому символу пишутся в отдельную папку *{каталог_терминала}/bases/{имя_сервера}/history/{имя_символа}*. Например, данные по символу EURUSD с торгового сервера MetaQuotes-Demo могут находиться в папке *C:/Program Files/MetaTrader 5/bases/MetaQuotes-Demo/history/EURUSD/*.

Данные записываются в файлы с расширением *.hcc: каждый файл хранит данные минутных баров за год. Например, файл 2021.hcc в папке EURUSD содержит минутные бары по символу EURUSD за 2021 год. Эти файлы используются для подготовки ценовых данных по всем таймфреймам и не предназначены для прямого доступа.

Служебные файлы в формате HCC исполняют роль источника данных для построения ценовых данных по конкретным таймфреймам. Они создаются только по запросу графика или MQL-программы и сохраняются для дальнейшего использования в файлах с расширением *.hc.

Для каждого таймфрейма данные подготавливаются независимо от других таймфреймов. Правила формирования и доступности данных одинаковы для всех таймфреймов, включая M1. То есть, несмотря на то, что единицей хранения данных в формате HCC является минутный бар, их

наличие не означает наличие и доступность в том же объеме данных таймфрейма M1 в формате НС.

Для экономии ресурсов данные по таймфрейму загружаются и хранятся в оперативной памяти только по необходимости: при длительном отсутствии обращений к данным происходит их выгрузка из оперативной памяти (но они остаются в файле). Это может сказаться на увеличении времени исполнения последующего запроса таймсерии, если она давно не использовалась. Все востребованные таймсерии, в частности, те, для которых открыты графики, доступны практически мгновенно в отсутствие перегрузок по ресурсам компьютера.

Получение новых данных с сервера вызывает автоматическое обновление используемых ценовых данных в формате НС по всем таймфреймам и перерасчет всех зависящих от них индикаторов.

При обращении MQL-программы к данным по конкретному символу и таймфрейму есть вероятность, что требуемая таймсерия еще не сформирована или не синхронизирована с торговым сервером (например, на нем появились обновленные цены). В этом случае следует в том или ином виде реализовать ожидание готовности данных.

Для скриптов единственным решением является использование циклов, так как у них нет другого варианта в виду отсутствия обработки событий. Для индикаторов подобные алгоритмы, как и любые другие циклы ожидания, категорически не рекомендуются, так как приводят к остановке расчета всех индикаторов и другой обработки ценовых данных по данному символу.

Для экспертов и индикаторов лучше использовать событийную модель обработки. Если при обработке события *OnTick* или *OnCalculate* не удалось получить все необходимые данные требуемой таймсерии, то следует выйти из обработчика события, ожидая их появления при следующих вызовах обработчика.

Максимальное количество баров

Стоит отметить, что максимальное количество баров, которое будет рассчитано для каждой запрошенной пары символ/таймфрейм, не превышает значение параметра *Макс. баров в окне* в диалоге *Настроек* терминала. Таким образом, данный параметр накладывает ограничение не только на графики любых таймфреймов, но и на все MQL-программы.

Это ограничение предназначено в первую очередь для экономии ресурсов. Устанавливая большие значения данного параметра, следует помнить, что при наличии достаточно глубокой истории ценовых данных для младших таймфреймов расход памяти на хранение таймсерий и буферов индикаторов может составить сотни мегабайт и занять всю оперативную память.

Изменение лимита баров вступает в силу только после перезапуска клиентского терминала. Оно влияет на объем запрашиваемых у сервера данных для построения требуемого количества баров рабочих таймфреймов.

Ограничение, задаваемое параметром, не является жестким и может быть превышено в определенных случаях. Например, если в начале сеанса история котировок по конкретному таймфрейму достаточна, чтобы выбрать весь лимит, то по мере формирования новых баров их количество может стать больше текущего значения параметра. Актуальное количество доступных баров возвращают функции *Bars/iBars*.

5.3.3 Получение характеристик массивов котировок

Перед чтением массивов таймсерий бывает полезно убедиться в их наличии и соответствии характеристик ожиданиям. Для получения основных свойств, таких как глубина доступной истории в терминале и на сервере, количество построенных баров по конкретному сочетанию символ/период и отсутствии разночтений в котировках между терминалом и сервером предназначена функция SeriesInfoInteger.

Функция имеет 2 формы: первая непосредственно возвращает запрошенное значение (типа *long*), а вторая использует для этого четвертый параметр *result*, передаваемый по ссылке. При этом вторая форма возвращает признак успеха (*true*) или ошибки (*false*). В любом случае код ошибки можно узнать с помощью функции *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)

Функция позволяет узнать одно из свойств временного ряда для указанного символа и таймфрейма, или в целом для всей истории по символу. Запрашиваемое свойство идентифицируется третьим аргументом типа ENUM_SERIES_INFO_INTEGER. Это перечисление включает все доступные свойства:

Идентификатор	Описание	Тип свойства
SERIES_BARS_COUNT	Количество баров по символу/периоду, см. *Bars*	long
SERIES_FIRSTDATE	Самая первая дата по символу/периоду	datetime
SERIES_LASTBAR_DATE	Время открытия последнего бара по символу/периоду	datetime
SERIES_SYNCHRONIZED	Признак синхронизированности данных по символу/периоду на терминале и на сервере	bool
SERIES_SERVER_FIRSTDATE	Самая первая дата в истории по символу на сервере независимо от периода	datetime
SERIES_TERMINAL_FIRSTDATE	Самая первая дата в истории по символу в клиентском терминале независимо от периода	datetime

В зависимости от сути свойства полученную величину следует приводить к значению конкретного типа (см. столбец *Тип свойства*).

Все свойства возвращаются по состоянию на текущий момент.

В скрипте *SeriesInfo.mq5* приводится пример запроса всех свойств.

```
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));
}
```

Вот пример результата, полученного на EURUSD, H1, на сервере 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 / c
SeriesInfoInteger(ABRACADABRA,0,SERIES_BARS_COUNT)=0 / MARKET_UNKNOWN_SYMBOL(4301)
```

5.3.4 Количество доступных баров (Bars/iBars)

Более короткий способ узнать общее количество баров в таймсерии по символу/периоду предоставляют функции *Bars* и *iBars* (какую именно использовать — разницы нет, *iBars* добавлена для совместимости с MQL4).

int Bars(const string symbol, ENUM_TIMEFRAMES timeframe)
int iBars(const string symbol, ENUM_TIMEFRAMES timeframe)

Функции возвращают количество доступных для MQL-программы баров по заданным символу и периоду. На это значение оказывает влияние параметр *Макс. баров в окне* в *Настройках* терминала (см. врезку в разделе Технические особенности организации и хранения таймсерий). Например, если в терминал скачана история, которая при конкретном таймфрейме составляет 20000 баров, но в настройках установлен лимит 10000 баров, то именно вторая величина будет определяющей. Сразу после запуска терминала функции вернут количество 10000 баров, но по мере формирования новых баров оно будет увеличиваться (если позволяет свободная память). Из MQL5 данный лимит можно узнать посредством вызова *TerminalInfoInteger(TERMINAL_MAXBARS)*.

Кроме того у функции *Bars* имеется второй вариант, позволяющий узнать количество баров в диапазоне между двумя датами.

int Bars(const string symbol, ENUM_TIMEFRAMES timeframe, datetime start, datetime stop)

При таком запросе учитываются только те бары, время открытия которых попадает в диапазон от *start* до *stop* (включительно). Причем не важно, в какой последовательности указаны *start* и *stop*: функция в любом случае просканирует фрагмент котировок от меньшего времени до большего.

Если данные для таймсерии с указанными параметрами еще не сформированы или не синхронизированы с торговым сервером при вызове функций *Bars/iBars*, они возвращают нулевое значение. При этом признак ошибки *_LastError* будет также равен 0 (нет ошибки, поскольку данные просто пока не скачаны или не готовы). Получив 0, проверяйте

синхронизацию конкретного таймфрейма с помощью *SeriesInfoInteger(..., SERIES_SYNCHRONIZED)* или синхронизацию символа — для этого есть отдельная функция *SymbolIsSynchronized*.

Примеры работы с функциями будут показаны в скрипте *SeriesBars.mq5*, в следующем разделе, вместе со связанной функцией *iBarShift*.

5.3.5 Поиск индекса бара по времени (iBarShift)

Функция *iBarShift* позволяет получить номер бара для заданного времени. В данном случае нумерация баров всегда подразумевается как в таймсериях, то есть индекс 0 соответствует самому правому, свежему бару, и значения увеличиваются по мере продвижения справа налево (в прошлое).

int iBarShift(const string symbol, ENUM_TIMEFRAMES timeframe, datetime time, bool exact = false)

Функция возвращает индекс бара в таймсерии для указанной пары параметров *symbol/timeframe*, в который попадает значение параметра *time*. Напомним, что каждый бар характеризуется временем открытия и длительностью, общей для всех баров ряда, то есть периодом. Например, на часовом таймфрейме бар, помеченный временем открытия 13:00, длится с 13:00:00 по 13:59:59 (включая всю последнюю минуту и секунду).

Если для указанного времени бар отсутствует (например, время приходится на неторговые часы или дни), то функция ведет себя по-разному в зависимости от параметра exact: при *exact = true* функция вернет -1, а при *exact = false* — индекс ближайшего бара, у которого время открытия меньше указанного. В случае, когда такого бара нет, то есть история раньше указанного времени отсутствует, функция вернет -1. Но здесь есть нюанс.

> Внимание! Возврат из функции *iBarShift* конкретного номера бара, то есть значения, отличного от -1, не означает, что при последующей попытке доступа к таймсериям по этому индексу удастся получить цены или другие характеристики этого бара. В частности, это может произойти, если запрошенный бар имеет индекс, превышающий лимит баров в окне терминала (*TerminalInfoInteger(TERMINAL_MAXBARS)*). Такое может произойти по мере формирования новых баров: тогда более старые бары с номерами, превышающими лимит, "сдвигаются" влево за границу видимых баров в окне и чисто номинально остаются на некоторое время в памяти. Обязанность проверять такие ситуации оставлена на плечах прикладных разработчиков.

Проверим работу функций *Bars/iBars* (см. предыдущий раздел), *iBarShift* с помощью скрипта *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()));
}
```

Здесь нам встречается еще незнакомая функция *ChartTimeOnDropped* (мы опишем её позднее): она возвращает время конкретного бара (в активном графике), на который был мышью перемещен из *Навигатора* и сброшен скрипт. Для начала перетащим скрипт на область графика, где есть котировки.

В журнале создадутся записи следующего вида (числа будут другими, в соответствии с вашими настройками, действиями и текущим временем):

```
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
```

В данном случае скрипт был отбуксирован на бар со временем 2021.10.01 09:00 (использовался часовой таймфрейм). Согласно *iBarShift* это время соответствовало бару под номером 125.

Количество баров от бара под мышью до последнего (текущего времени) составило 126. Это сочетается с номером бара 125, поскольку нумерация начинается с 0.

Общее количество баров на графике, полученное разными способами (*iBars*, *Bars* без диапазона дат и *Bars* с полным диапазоном от 0 до текущего момента *TimeCurrent*), равно 10004. В настройках терминала стояло ограничение 10000, но за время сеанса успели сформироваться дополнительные 4 часовых бара.

Номер бара, в который попадает текущее время *iBarShift(..., TimeCurrent())*, всегда равно 0 для существующего символа и таймфрейма, при условии *exact = false*. Если *exact = true*, то мы можем иногда получить -1, поскольку серверное время увеличивается по приходу тиков всех инструментов рынка, а текущий символ может временно не торговаться. Тогда серверное время может уйти вперед более чем на размер одного бара, и для *TimeCurrent* не найдется нового бара для точного попадания в него.

Если перетащить и сбросить скрипт в пустой области справа от текущего, последнего бара (то есть в будущее), получим примерно такую картину:

```
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
```

Функция *iBarShift* в режиме поиска любого предыдущего бара (*exact* = *false*) возвращает 0, поскольку текущий бар ближе всего к будущему. Однако точный поиск (*exact* = *true*) дает результат -1. Также функции *Bars* с подсчетом количества баров в диапазоне от текущего времени до "целевого" будущего теперь возвращают 0 (баров там пока нет).

Особую важность функция *iBarShift* имеет для написания мультивалютных MQL-программ. Довольно часто расписания торгов разными финансовыми инструментами не совпадают, поэтому для конкретного времени бар может существовать на одном символе, но отсутствовать на другом. С помощью функции *iBarShift* в режиме поиска ближайшего (предыдущего) бара вы всегда можете получить индексы баров с ценами, которые были актуальными для разных символов на один и тот же момент. Как правило, даже для символов Forex индексы исторических баров для одного и того же времени могут отличаться.

Например, следующие инструкции выведут в журнал разные количества баров и их номера на одном и том же диапазоне дат для трех символов: EURUSD, XAUUSD, USDRUB на часовом таймфрейме (сервер 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 Обзор Copy-функций для получения массивов котировок

MQL5 API содержит несколько функций для чтения котировочных таймсерий в массивы. Их имена приведены в следующей таблице.

Функция	Действие
CopyRates	Получение истории котировок в массив структур *MqlRates*
CopyTime	Получение истории времен открытия баров в массив типа *datetime*
CopyOpen	Получение истории цен открытия баров в массив типа *double*
CopyHigh	Получение истории максимальных цен баров в массив типа *double*
CopyLow	Получение истории минимальных цен баров в массив типа *double*
CopyClose	Получение истории цен закрытия баров в массив типа *double*
CopyTickVolume	Получение истории тиковых объемов в массив типа *long*
CopyRealVolume	Получение истории биржевых объемов в массив типа *long*
CopySpread	Получение истории спредов в массив типа *int*

Все функции принимают в качестве двух первых параметров имя требуемого символа и периода, что можно условно представить следующим псевдокодом:

```
int Copy***(const string symbol, ENUM_TIMEFRAMES timeframe, ...)
```

Также все функции имеют по три варианта прототипа, отличающихся способом задания запрашиваемого диапазона:

- начальный индекс бара и количество баров — *Copy***(..., int offset, int count, ...)*;
- время начала диапазона и количество баров — *Copy***(..., datetime start, int count, ...)*;
- время начала и окончания диапазона — *Copy***(..., datetime start, datetime stop, ...).*

При этом в нотации параметров подразумевается, что запрашиваемые данные имеют направление индексации, как в таймсерии, то есть в позиции *offset* с индексом 0 хранятся данные текущего незавершенного бара, и увеличение индексов соответствует продвижению вглубь ценовой истории. Из-за этого, в частности, для второго варианта указанное количество баров *count* будет отсчитываться в прошлое от начала диапазона *offset*, то есть в сторону уменьшения времени.

Дополнительную гибкость предоставляет третий вариант: в нем не важно, в каком порядке задать стартовую и финишную даты (*start/stop*), — функции в любом случае вернут данные в диапазоне от меньшей даты до большей. Подходящие бары отбираются по такому принципу, чтобы их время открытия находилось между временными отсчетами *start/stop* или было равно одному из них, т.е. диапазон [*start;stop*] рассматривается с включением границ.

Какой именно вариант функции выбрать, определяет разработчик, руководствуясь тем, что для него важнее — получить гарантированное количество элементов (например, для алгоритмов машинного обучения) или покрытие конкретного интервала дат (например, с заранее определенным единообразным поведением рынка).

Напомним, что точность представления времени в типе *datetime* составляет 1 секунду. Значения *start/stop* не обязаны быть округленными до размера периода. Например, диапазон от 14:59 до 16:01 позволит выделить на таймфрейме H1 два бара для 15:00 и 16:00. Вырожденный диапазон с равными и округленными метками, например, 15:00 в котировках H1, соответствует одному бару.

Вы можете запрашивать бары на дневном таймфрейме даже при наличии в параметрах start/stop ненулевых значений часов/минут/секунд (несмотря на то, что на таймфрейме D1 метки баров имеют время 00:00). При этом в результат попадут только те бары D1, что имеют время открытия после минимального из *start/stop* и до максимального из *start/stop* (равенство с метками дневных баров в данном случае невозможно, поскольку в искомом времени присутствуют часы/минуты/секунды). Например, между отсечками D'2021.09.01 12:00' и D'2021.09.03 07:00' находятся два времени открытия баров D1 — D'2021.09.02' и D'2021.09.03'. Именно эти бары и попадут в результат. Бар D'2021.09.01' имеет время открытия 00:00 раньше, чем начало диапазона, и потому отбрасывается. Бар D'2021.09.03' включен в результат, несмотря на то, что в диапазон попало только 7 утренних часов из этого дня. С другой стороны, запрос нескольких часов внутри дня, например, между D'2021.09.01 12:00' и D'2021.09.01 15:00' не охватит ни одного дневного бара (время открытия бара D'2021.09.01' не попадает в этот диапазон), и потому приемный массив будет пуст.

Единственное отличие всех функций из таблицы заключается в типе массива, принимающего данные, — он передается последним параметром по ссылке. Например, функция *CopyRates* помещает запрошенные данные в массив структур *MqlRates*, а функция *CopyTime* — время открытия баров в массив типа *datetime*, и так далее.

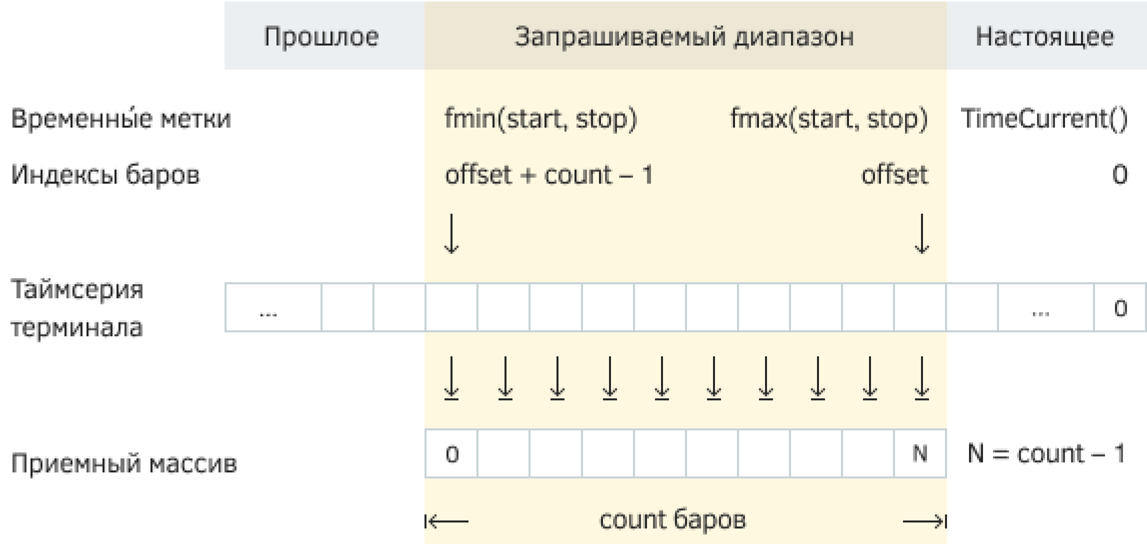
Таким образом, общие прототипы функций можно представить так:

```
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[])
```

Здесь *type* соответствует любому из типов *MqlRates*, *datetime*, *double*, *long* или *int*, в зависимости от конкретной функции.

Функции возвращают количество скопированных элементов в массив либо -1 в случае ошибки. В частности, мы получим результат -1, если в запрашиваемом интервале нет никаких данных на сервере, или интервал находится за пределами максимального количества баров на графике (*TerminalInfoInteger(TERMINAL_MAXBARS)*).

Важно отметить, что в приемном массиве полученные данные всегда физически размещаются в хронологическом порядке, от прошлого к будущему. Таким образом, если для приемного массива используется стандартная индексация (то есть к нему не применялась функция *ArraySetAsSeries*), то элемент с 0-м индексом будет самым старым, а последний элемент — самым новым. Если же для массива была выполнена инструкция *ArraySetAsSeries(result, true)*, то нумерация будет вестись в обратном порядке, как в таймсерии: 0-й элемент окажется самым новым в диапазоне, а последний — самым старым. Это иллюстрирует следующий рисунок.



Таймсерии терминала и приемный массив

В случае успеха в массив-получатель будет скопировано указанное количество элементов из собственной (внутренней) таймсерии терминала. При запросе данных по диапазону дат (*start/stop*), количество элементов в результирующем массиве определится опосредованно, исходя из содержимого истории в этом диапазоне. Поэтому для копирования заранее неизвестного количества значений рекомендуется использовать динамические массивы: функции копирования самостоятельно распределяют необходимый размер массивов-приемников (размер может быть как увеличен, так и уменьшен).

Если необходимо копировать известное количество элементов или делать это часто, например, при каждом вызове *OnTick* в экспертах или *OnCalculate* в индикаторах, то лучше использовать статически распределенные массивы. Дело в том, что операции распределения памяти под динамические массивы требуют дополнительного времени и могут сказаться на быстродействии, в особенности при тестировании и оптимизации.

Доступ к таймсериям осуществляется по-разному для MQL-программ разных типов, если запрашиваемые данные еще не готовы. Так в пользовательских индикаторах *Copy*-функции сразу же возвращают ошибку, поскольку индикаторы выполняются в общем интерфейсном потоке терминала и не могут ожидать получения данных (предполагается, что индикаторы запросят данные при следующих вызовах их обработчиков событий, и таймсерии к тому моменту уже будут закачаны и построены). Кроме того, в главе, посвященной индикаторам, мы узнаем, что для доступа к котировкам "родного" графика, на котором индикатор и размещен, ему не нужно использовать *Copy*-функции, потому что все временные ряды автоматически передаются через параметры-массивы обработчика *OnCalculate*.

При доступе из экспертов и скриптов производится несколько попыток получения данных с небольшой паузой (с ожиданием внутри функции), дающей время для загрузки и расчета недостающих таймсерий. Функция вернет то количество данных, которые будут готовы к моменту истечения этого таймаута, но загрузка истории будет продолжаться, и при следующем аналогичном запросе функция вернет уже больше данных.

В любом случае следует быть готовым, что *Copy*-функция вернет вместо данных ошибку (причин достаточно: нарушение связи, отсутствие запрошенных данных, загруженность процессора, если параллельно запрашивается много новых таймсерий): проанализируйте в коде причину

проблемы (_LastError) и повторите попытку позже, откорректируйте параметры или сообщите пользователю.

Наличие символа в *Обзоре рынка* не является необходимым условием для запроса таймсерий с помощью *Copy*-функкий, однако для символов, включенных в данное окно, запросы, как правило, выполняются быстрее, поскольку некоторые данные уже скачаны с сервера и, вероятно, посчитаны для востребованных периодов. О том, как добавлять символы в *Обзор рынка* программным способом, мы узнаем в разделе Редактирование списка Обзора рынка.

Для пояснения принципов работы функций на практике рассмотрим скрипт *SeriesCopy.mq5*. Он содержит несколько вызовов функции *CopyTime*, что позволяет наглядно увидеть, как соотносятся временные метки и номера баров.

В скрипте определен динамический массив *times* для приема данных. Все запросы делаются для символа "EURUSD" и таймфрейма H1.

```
void OnStart()
{
   datetime times[];
```

Для начала делается запрос 10 баров, начиная с 5 сентября 2021, в прошлое. Поскольку этот день воскресенье, предыдущие бары были в пятницу 3-го числа (см. лог ниже).

```
   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
   */
```

Вывод массива производится по умолчанию в хронологическом порядке (несмотря на то, что параметры функции задаются в обратной системе координат: как в таймсерии). Поменяем порядок индексации в приемном массиве и выведем его еще раз.

```
   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
   */
```

Для следующих экспериментов восстановим обычный порядок.

```
   PRTF(ArraySetAsSeries(times, false)); // true / ok
```

Теперь запросим неопределенное количество баров между двумя временными отсчетами (количество неизвестно, потому что в диапазоне могут оказаться праздники, например). Сделаем это двумя способами: в первом случае укажем диапазон от будущего в прошлое, а во втором — из прошлого в будущее. Оба результата совпадут.

```
   //                                      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
   */
```

Распечатка массивов позволяет убедиться в их идентичности. Вернемся к режиму индексации как в таймсерии и обсудим еще один нюанс.

```
   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
```

Хотя две временных метки отстоят друг от друга на 24 часа, что предполагает получение в массиве 25 элементов (вспоминаем, что начало и конец обрабатываются включительно), результат содержит только 4 бара. Дело в том, что 5-е сентября приходится на воскресенье, и потому из всего диапазона торговля велась только в утренние часы 6-го числа.

Также обратите внимание, что приемный массив был автоматически уменьшен в размере с 10 до 4 элементов.

Наконец, запросим 10 баров, начиная с 100-го бара (полученные результаты будут зависеть от вашего текущего времени и доступной истории).

```
   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
   */
}
```

Из-за индексации как в таймсерии, массив выведен в обратном хронологическом порядке.

5.3.7 Получение котировок в виде массива структур MqlRates

Для запроса массива котировок, включающего все характеристики баров, предназначена функция *CopyRates*, имеющая несколько перегрузок.

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[])

Функция получает в массив *rates* исторические данные для указанных параметров: символа, таймфрейма и диапазона времени, заданного либо номерами баров, либо значениями *start/stop* типа *datetime*.

Функция возвращает количество скопированных элементов массива либо -1 в случае ошибки, код которой можно узнать из *_LastError*. В частности, ошибка возникнет, если задан несуществующий символ, интервал не содержит данных на сервере или выходит за ограничение по количеству баров на графике (*TerminalInfoInteger(TERMINAL_MAXBARS)*).

Принципы работы с данной функцией являются общими для всех *Copy*-функций и были изложены в разделе Обзор Copy-функций для получения массивов котировок.

Структура встроенного типа *MqlRates* описана следующим образом:

```
struct MqlRates
{
   datetime time;         // время открытия бара
   double   open;         // цена открытия
   double   high;         // максимальная цена за бар
   double   low;          // минимальная цена за бар
   double   close;        // цена закрытия
   long     tick_volume;  // тиковый объем за бар
   int      spread;       // минимальный спред за бар в пунктах
   long     real_volume;  // биржевой объем за бар
};
```

Попробуем применить функцию для подсчета среднего размера баров в скрипте *SeriesStats.mq5*. Во входных переменных предоставим возможность выбора рабочего символа, таймфрейма, количества анализируемых баров и начального смещения в прошлое (0 означает анализ с текущего бара).

```
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; // подсчет размаха и движения цен в барах

   PrintFormat("Requesting %d bars on %s %s",
      BarCount, StringLen(WorkSymbol) > 0 ? WorkSymbol : _Symbol,
      EnumToString(TimeFrame == PERIOD_CURRENT ? _Period : TimeFrame));

   // запрашиваем всю информацию о BarCount барах в массив MqlRates
   const int n = PRTF(CopyRates(WorkSymbol, TimeFrame, BarOffset, BarCount, rates));

   // в цикле подсчитываем среднее для размаха и движения
   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));
}
```

Набросив скрипт на график EURUSD,H1, мы можем получить примерно следующий результат.

```
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
```

Поскольку в терминале стояло ограничение на 20000 баров, запрос 100000 баров смог вернуть только 20018 (лимит и новые сформированные бары после начала сеанса). В самый первый элемент массива (с индексом 0) попал бар со временем 2018.07.19 15:00, а в последний — 2021.10.11 17:00.

Согласно статистике средний размах бара за это время составил 128 пунктов, и движение между открытием и закрытием — 62 пункта.

При запросе информации с помощью начальной и конечной даты (*start/stop*) имейте в виду, что обе границы трактуются включительно. Поэтому для задания интервала, соответствующего какому-либо бару старшего таймфрейма, следует отнимать от правой границы 1 секунду. Мы применим этот прием в примере *SeriesSpread.mq5* в разделе Чтение цены, объема, спреда и времени по индексу бара.

5.3.8 Раздельный запрос массивов цен, объемов, спредов, времени

Вместо запроса всех характеристик кодировок в виде массива *MqlRates*, можно читать только данные конкретного поля (цены, объема, спреда или времени) в отдельный массив. Для этого определено несколько функций, действующих по общим принципам, рассмотренным в разделе Обзор Copy-функций для получения массивов котировок.

Следующая схема объединяет описания всех прототипов.

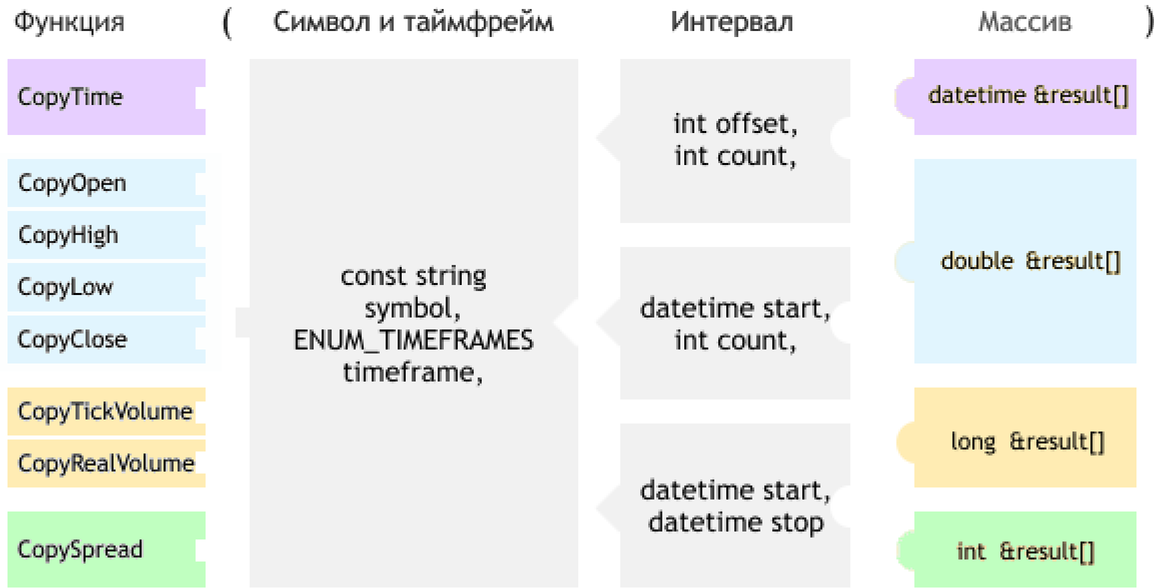


Схема прототипов Copy-функций

В скрипте *SeriesRates.mq5* воспользуемся функциями копирования цен OHLC, чтобы сравнить их с результатом вызова *CopyRates*.

```
void OnStart()
{
   const int N = 10;
   MqlRates rates[];

   // запрашиваем и отображаем всю информацию о N барах из массива MqlRates
   PRTF(CopyRates("EURUSD", PERIOD_D1, D'2021.10.01', N, rates));
   ArrayPrint(rates);

   // теперь запрашиваем цены OHLC раздельно
   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));

   // сравниваем цены полученные разными методами
   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])
      {
         // здесь мы не должны оказаться
         Print("Data mismatch at ", i);
         return;
      }
   }

   Print("Copied OHLC arrays match MqlRates array"); // успех: соответствие есть
}
```

Запустив скрипт, получим в журнале следующие записи.

```
CopyRates(EURUSD,PERIOD_D1,D'2021.10.01',N,rates)=10 / ok
                  [time]  [open]  [high]   [low] [close] [tick_volume] [spread] [real_
[0] 2021.09.20 00:00:00 1.17272 1.17363 1.17004 1.17257         58444        0
[1] 2021.09.21 00:00:00 1.17248 1.17486 1.17149 1.17252         58514        0
[2] 2021.09.22 00:00:00 1.17240 1.17555 1.16843 1.16866         72571        0
[3] 2021.09.23 00:00:00 1.16860 1.17501 1.16835 1.17381         68536        0
[4] 2021.09.24 00:00:00 1.17379 1.17476 1.17007 1.17206         51401        0
[5] 2021.09.27 00:00:00 1.17255 1.17255 1.16848 1.16952         57807        0
[6] 2021.09.28 00:00:00 1.16940 1.17032 1.16682 1.16826         64793        0
[7] 2021.09.29 00:00:00 1.16825 1.16901 1.15894 1.15969         68964        0
[8] 2021.09.30 00:00:00 1.15963 1.16097 1.15626 1.15769         68517        0
[9] 2021.10.01 00:00:00 1.15740 1.16075 1.15630 1.15927         66777        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
Copied OHLC arrays match MqlRates array
```

Напомним, что тиковый объем в поле *tick_volume* представляет собой простой счетчик тиков за период. Биржевой объем в поле *real_volume* равен нулю у небиржевых инструментов (как и у EURUSD, в данном случае).

Другой пример использования функции *CopyTime* был представлен в скрипте *SeriesCopy.mq5* в разделе Обзор Copy-функций для получения массивов котировок.

5.3.9 Чтение цены, объема, спреда и времени по индексу бара

Иногда требуется узнать информацию не о последовательности баров, а лишь об одном. В принципе, для этого можно пользоваться рассмотренными ранее *Copy*-функциями, задав в них количество (параметр *count*) равным 1, но это не очень удобно. Более простой вариант предлагают следующие функции, возвращающие одно значение определенного типа для бара по его номеру в таймсерии.

Все функции имеют похожий прототип, но разные названия и тип возвращаемого значения. Исторически так сложилось, что названия начинаются с префикса *i*, то есть имеют вид *iValue* (данные функции относятся к большой группе встроенных технических индикаторов: ведь характеристики котировок являются первоисточником для теханализа, и почти все индикаторы являются их производными, отсюда и возникла буква *i*).

type iValue(const string symbol, ENUM_TIMEFRAMES timeframe, int offset)

Здесь *type* соответствует одному из типов *datetime*, *double*, *long* или *int*, в зависимости от конкретной функции. Символ и таймфрейм индентифицируют запрашиваемый временной ряд. Индекс требуемого бара *offset* передается в нотации таймсерий, то есть 0 означает самый свежий, правый бар (как правило, еще не завершенный), а увеличение номера приводит к продвижению вглубь истории. Как и в случае *Copy*-функций, можно использовать NULL и 0 для задания символа и периода, равными свойствам текущего графика.

Так как *i*-функции эквиваленты вызову *Copy*-функций, к ним применимы все особенности запроса таймсерий из разных типов программ, описанные в разделе Обзор Copy-функций для получения массивов котировок.

Функция	Описание
iTime	время открытия бара
iOpen	цена открытия бара
iHigh	максимальная цена бара
iLow	минимальная цена бара
iClose	цена закрытия бара
iTickVolume	тиковый объем бара (аналог iVolume)
iVolume	тиковый объем бара (аналог iTickVolume)
iRealVolume	реальный торговый объем бара
iSpread	минимальный спред бара (в пунктах)

Функции возвращают запрошенную величину или 0 в случае ошибки (к сожалению, в некоторых случаях 0 может быть реальным значением). Для получения дополнительной информации об ошибке необходимо вызвать функцию *GetLastError*.

Функции не кэшируют результаты и при каждом вызове возвращают актуальные данные из таймсерии по указанным символу/периоду. Это означает, что при отсутствии готовых данных (на первом вызове или после потери синхронизации) функции может понадобиться некоторое время для подготовки результата.

В качестве примера решим задачу с получением более или менее реалистичной оценки размера спреда для каждого бара. Дело в том, что в котировках хранится минимальное значение спреда, что может вызывать неоправданно завышенные ожидания при конструировании торговых стратегий. Для получения абсолютно точных значений среднего, медианного или максимального спреда за бар потребовалось бы анализировать реальные тики, но мы пока не научились с ними работать. А кроме того, это был бы весьма затратный по ресурсам процесс. Более рациональным является подход по анализу спредов на младшем таймфрейме M1: для баров более старших таймфреймов достаточно искать максимальный спред во внутренних барах M1. Конечено, строго говоря, он будет не максимальным, а максимальным из минимальных, но учитывая быстротечность минутных отсчетов, можно надеяться засечь характерные расширения спредов хотя бы на некоторых барах M1, а этого уже достаточно, чтобы получить приемлемое соотношение точности и скорости анализа.

Один из вариантов алгоритма реализован в скрипте *SeriesSpread.mq5*. Во входных переменных можно задать символ, таймфрейм и количество баров для анализа. По умолчанию обрабатывается символ текущего графика и его период (должен быть больше M1).

```
input string WorkSymbol = NULL; // Symbol (leave empty for current)
input ENUM_TIMEFRAMES TimeFrame = PERIOD_CURRENT;
input int BarCount = 100;
```

Поскольку для каждого бара важна только информации о его времени и спреде, была описана специальная структура с двумя полями. В принципе, можно было использовать стандартную структуру *MqlRates* и складывать "максимальные" спреды в какое-нибудь неиспользуемое поле

(например, *real_volume* для символов Forex), но тогда данные для большинства полей копировались бы и занимали память вхолостую.

```
struct SpreadPerBar
{
   datetime time;
   int spread;
};
```

Используя тип новой структуры, подготовим массив *peaks* для подсчета данных указанного количества баров.

```
void OnStart()
{
   SpreadPerBar peaks[];
   ArrayResize(peaks, BarCount);
   ZeroMemory(peaks);
   ...
```

Далее в цикле по барам выполняется основная часть алгоритма. Для каждого бара с помощью функции *iTime* мы определяем две временных метки, задающих границы бара. Фактически это время открытия i-го бара и соседнего (i+1)-го бара. Учитывая принципы индексации, можно сказать, что (i+1)-й бар является предыдущим (более старым, см. переменную *prev*), а i-й — следующим (более новым, см. переменную *next*). Время открытия бара принадлежит только одному бару, то есть метка *prev* содержится в (i+1)-м баре, а метка *next* — в i-м. Таким образом, при обработке каждого бара его правая граница должна исключаться из интервала [*prev;next*).

Нас интересуют спреды на одноминутном таймфрейме, и потому мы будем использовать функцию *CopySpread* для PERIOD_M1. При этом полуоткрытый интервал достигается путем указания в параметрах *start/stop* точного значения *prev* и уменьшенного на 1 секунду значения *next*. Информация о спредах копируется в динамический массив *spreads* (память под него распределяет сама функция).

```
   for(int i = 0; i < BarCount; ++i)
   {
      int spreads[]; // приемный массив для спредов M1 внутри i-го бара
      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;
      }
   }
```

Далее остается найти максимальное значение в этом массиве и сохранить его в соответствующей структуре *SpreadPerBar* вместе со временем бара. Обратите внимание, что нулевой незавершенный бар в анализе не участвует (Вы можете дополнить алгоритм, если это необходимо).

После завершения цикла выводим массив структур в журнал.

```
   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);
```

Запустив скрипт на графике EURUSD,H1, получим статистику спредов внутри часовых баров (приведено с сокращениями):

```
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
```

Налицо увеличение спредов в ночные часы: так, в окрестности полуночи котировки содержат спреды 7-15 пунктов, а в наших измерениях они равны 15-65. Однако и в другие периоды обнаруживаются ненулевые значения, хотя в метриках часовых баров стоят, как правило, нули.

5.3.10 Поиск максимального и минимального значения в таймсерии

Среди группы функций для работы с временными рядами котировок присутствуют две, предоставляющих простейшую агрегатную обработку: поиск максимального и минимального значений ряда на заданном интервале — соответственно *iHighest* и *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)

Функции возвращают индекс наибольшего/наименьшего значения для конкретного типа таймсерии, которая задается парой параметров *symbol/timeframe*, а также элементом перечисления ENUM_SERIESMODE (оно описывает уже знакомые нам поля котировок).

Идентификатор	Описание
MODE_OPEN	цена открытия
MODE_LOW	минимальная цена
MODE_HIGH	максимальная цена
MODE_CLOSE	цена закрытия
MODE_VOLUME	тиковый объем
MODE_REAL_VOLUME	реальный объем
MODE_SPREAD	спред

Параметр *offset* задает индекс, с которого начинается поиск. Напомним, что нумерация ведется как в таймсерии, то есть увеличение *offset* приводит к смещению в прошлое, а 0-й индекс означает текущий бар (это значение по умолчанию). Количество анализируемых баров указывается в параметре *count* (по умолчанию, весь массив WHOLE_ARRAY).

В случае ошибки функции возвращают -1, а код ошибки можно узнать с помощью *GetLastError*.

Для демонстрации работы одной из этих функций (*iHighest*) модифицируем пример из предыдущего раздела по оценке реальных размеров спредов по барам и сравним результаты — они, разумеется, должны совпасть. Новая версия скрипта прилагается в файле *SeriesSpreadHighest.mq5*.

Изменения коснулись структуры *SpreadPerBar* и рабочего цикла внутри *OnStart*.

В структуру были добавлены поля, которые позволяют понять принцип работы новой функции. По сути алгоритма они не обязательны.

```
struct SpreadPerBar
{
   datetime time;
   int spread;
   int max; // сквозной индекс M1-бара со спредом, значение которого максимально
            // среди всех M1-баров внутри текущего бара старшего таймфрейма
   int num; // количество M1-баров в текущем баре старшего таймфрейма
   int pos; // начальный индекс M1-бара внутри текущего бара старшего таймфрейма
};
```

Основные преобразования затронули *OnStart*, но они локализованы внутри цикла (все остальные фрагменты кода остались без изменений).

```
   for(int i = 0; i < BarCount; ++i)
   {
      const datetime next = iTime(WorkSymbol, TimeFrame, i);
      const datetime prev = iTime(WorkSymbol, TimeFrame, i + 1);
      ...
```

Границы текущего бара *prev* и *next* определяются как раньше. Однако вместо копирования элементов таймсерии между этими метками в собственный массив *spreads* и последующего вызова *ArrayMaximum* для него, мы определяем индексы и количество M1-баров, формирующих текущий бар старшего таймфрейма. Делается это следующим образом.

Функция *iBarShift* позволяет узнать смещение (переменная *p*) в истории M1, где находится правая граница бара с временем *next - 1*. Функция *Bars* вычисляет количество баров M1 (переменная *n*), попадающих между метками *prev* и *next - 1*. Эти два значения становятся параметрами в вызове функции *iHighest*, чтобы найти максимальное значение типа MODE_SPREAD, среди *n* баров M1, начиная с индекса *p*. Если максимум найден без проблем (*m > -1*), нам остается взять соответствующее значение с помощью *iSpread* и поместить в структуру.

```
      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;
      }
   }
```

При выводе массива с результатами в журнал мы теперь дополнительно увидим индексы баров M1, где "начинается" бар старшего таймфрейма и где в нем нашелся максимальный спред. Слово "начинается" взято в кавычки, потому что по мере поступления новых баров M1 эти индексы будут увеличиваться, и виртуальное "начало" каждого будет сдвигаться, хотя времена открытия исторических баров, разумеется, остаются постоянными.

```
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
...
```

Например, на момент запуска скрипта бар с меткой 2021.10.12 14:00 начинался с 67-го бара M1 (т.е. был открыт 67 минут назад), а M1-бар с максимальным спредом внутри этого H1-бара нашелся под индексом 89. Очевидно, что этот индекс должен быть меньше, чем номер M1-бара, на котором начинался предыдущий H1-бар: 2021.10.12 13:00 — он отметился 127 минут назад. В этом H1-баре, в свою очередь, был найден максимальный спред по индексу 181. И это меньше индекса 187 у еще более старого бара 2021.10.12 12:00.

Индексы в колонках *pos* и *max* постоянно возрастают, потому что мы обходим бары в порядке от настоящего в прошлое. В колонке *num* почти всегда будет выводиться 60, поскольку большинство баров H1 состоит из 60 баров M1. Но так бывает не всегда. Например, ниже показаны неполные часовые бары, состоящие из меньшего числа минут: это могут быть как последствия более раннего закрытия рынка из-за праздничного расписания, так и реальные пропуски в торговой активности (отсутствие ликвидности).

```
...
[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 Работа с массивами реальных тиков в структурах MqlTick

MetaTrader 5 обеспечивает возможность работать не только с историей котировок (баров), но и историей реальных тиков. Из пользовательского интерфейса все исторические данные доступны в диалоге *Символы*. В нем имеются 3 закладки: *Спецификация*, *Бары* и *Тики*. Когда в древовидном списке символов на первой закладке выделен конкретный элемент, при переходе на закладки *Бары* и *Тики* можно запросить котировки в виде баров или тиков, соответственно.

Из MQL-программ история реальных тиков также доступна — с помощью функций *CopyTicks* и *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)

Обе функции запрашивают тики для указанного инструмента *symbol* в передаваемый по ссылке массив *ticks*. Структура *MqlTick* содержит всю информацию об одном тике и описана в MQL5 следующим образом:

```
struct MqlTick
{
   datetime time;          // время данного обновления цен
   double   bid;           // текущая цена Bid
   double   ask;           // текущая цена Ask
   double   last;          // цена последней сделки (Last)
   ulong    volume;        // объем для цены Last
   long     time_msc;      // время данного обновления цен в миллисекундах
   uint     flags;         // флаги (какие поля структуры изменились)
   double   volume_real; // объем для цены Last с повышенной точностью
};
```

Поле *flags* предназначено для хранения битовой маски признаков, какие именно поля в структуре тика содержат измененные значения.

Константа	Значение	Описание
TICK_FLAG_BID	2	Изменена цена Bid
TICK_FLAG_ASK	4	Изменена цена Ask
TICK_FLAG_LAST	8	Изменена цена Last
TICK_FLAG_VOLUME	16	Изменен объем
TICK_FLAG_BUY	32	Тик возник в результате сделки на покупку
TICK_FLAG_SELL	64	Тик возник в результате сделки на продажу

Это потребовалось потому, что у каждого тика всегда заполняются все поля, независимо от того, изменились ли данные по сравнению с предыдущим тиком. Это позволяет всегда иметь актуальное состояние цен на любой момент времени без поиска предыдущих значений по тиковой истории. Например, с тиком могла измениться только цена Bid, но в структуре помимо

новой цены будут указаны и остальные параметры: предыдущая цена *Ask*, *Last*, объем и так далее.

Вместе с тем, следует иметь в виду, что в зависимости от типа инструмента некоторые поля в тиках могут быть всегда нулевыми (и для них никогда не взводятся соответствующие биты маски). В частности, для инструментов Forex, как правило, остаются пустыми поля *last*, *volume*, *volume_real*.

Приемный массив *ticks* может быть фиксированного размера или динамический. В фиксированный массив функции скопируют не больше тиков, чем размер массива, невзирая на реальное количество тиков в запрошенном интервале времени (указанном параметрами *from/to* в функции *CopyTicksRange*) или в параметре *count* функции *CopyTicks*. В массиве *ticks* наиболее старые тики размещаются под первыми номерами, а наиболее новые — под последними.

В параметрах обеих функций временные отсчеты задаются в виде миллисекунд, прошедших с 01.01.1970 00:00:00. Так в функции *CopyTicks* диапазон запрашиваемых тиков задается начальным отсчетом *from* и количеством тиков *count*, а в функции *CopyTicksRange* — начальным и конечным отсчетами *from* и *to* (обе границы — включительно).

Иными словами, *CopyTicksRange* предназначена для получения тиков в конкретном интервале, причем их количество заранее не известно. А *CopyTicks* гарантирует получение не более *count* тиков, но не позволяет заранее определить, какой временной интервал эти тики покроют.

Хронологический порядок значений *from* и *to* в *CopyTicksRange* неважен: функция в любом случае отдаст тики, начиная от минимального из двух значений и заканчивая максимальным.

Функция *CopyTicks* расценивает параметр *from* как левую границу с минимальным временем и отсчитывает от неё *count* тиков в будущее. Однако существует важное исключение: значение *from = 0* (по умолчанию) трактуется как текущий момент времени, и от него тики отсчитываются в прошлое. Это дает возможность всегда получить заданное количество последних тиков. Когда параметр *count = 0* (по умолчанию), функция копирует не более 2000 тиков.

Обе функции возвращают количество скопированных тиков либо -1 в случае ошибки. В частности *GetLastError* может возвращать следующие коды ошибок:

- ERR_HISTORY_TIMEOUT — время ожидания синхронизации тиков вышло, функция отдала всё что было;
- ERR_HISTORY_SMALL_BUFFER — статический буфер слишком маленький, отдано столько, сколько поместилось в массив;
- ERR_NOT_ENOUGH_MEMORY — не удалось выделить нужный объем памяти для получения истории тиков из указанного диапазона в динамический массив.

Параметр *flags* определяет тип запрашиваемых тиков.

Константа	Значение	Описание
COPY_TICKS_INFO	1	Тики, вызванные изменениями *Bid* и/или *Ask* (TICK_FLAG_BID, TICK_FLAG_ASK)
COPY_TICKS_TRADE	2	Тики с изменениями *Last* и *Volume* (TICK_FLAG_LAST, TICK_FLAG_VOLUME, TICK_FLAG_BUY, TICK_FLAG_SELL)
COPY_TICKS_ALL	3	Все тики

При любом типе запроса в оставшиеся поля структуры *MqlTick*, не соответствующие флагам, подставляются предыдущие актуальные значения. Например, если запрашивались только информационные тики (COPY_TICKS_INFO), в них все равно будут заполнены остальные поля — так, если изменилась только цена *Bid*, в поля *ask* и *volume* будут записаны последние известные значения. Чтобы узнать, что именно новое содержит тик, необходимо анализировать его поле *flags* (там будет либо значение TICK_FLAG_BID, либо TICK_FLAG_ASK, либо их комбинация). Если тик имеет нулевые значения цен *Bid* и *Ask*, и при этом флаги показывают, что данные цены изменились (*flags == TICK_FLAG_BID | TICK_FLAG_ASK*), то это говорит об опустошении стакана заявок.

Аналогичным образом, если запрашивались торговые тики (COPY_TICKS_TRADE), в их поля *bid* и *ask* будут записаны последние известные значения цен. В данном случае поле *flags* может иметь комбинацию значений TICK_FLAG_LAST, TICK_FLAG_VOLUME, TICK_FLAG_BUY, TICK_FLAG_SELL.

При запросе COPY_TICKS_ALL отдаются все тики.

Вызов любой из функций *CopyTicks/CopyTicksRange* проверяет синхронизацию базы тиков, хранящихся на жёстком диске по заданному символу. Если тиков в локальной базе не хватает, то недостающие тики автоматически будут загружены с торгового сервера. При этом будут синхронизированы тики с учетом старейшей даты из параметров запроса и вплоть до текущего момента. После этого все приходящие по данному символу тики будут поступать в тиковую базу и поддерживать её в актуальном синхронизированном состоянии.

> Тиковые данные имеют значительно больший размер, чем минутные котировки. При первом запросе истории тиков или запуске тестирования по реальным тикам их скачивание может занять продолжительное время. История тиковых данных хранится в файлах внутреннего TKC-формата в каталоге *{каталог_терминала}/bases/{имя_сервера}/ticks/{имя_символа}*. Каждый файл содержит информацию за один месяц.

В индикаторах функции возвращают результат немедленно, то есть копируют имеющиеся тики по символу и запускают фоновый процесс синхронизации базы тиков, если данных не хватило. Все индикаторы на одном символе работают в одном общем потоке, поэтому они не имеют права ждать завершения синхронизации. После окончания синхронизации при последующем вызове функции вернут все запрашиваемые тики.

В экспертах и скриптах функции могут дожидаться результата до 45 секунд: в отличие от индикатора каждый эксперт и скрипт работает в собственном потоке, и поэтому может дожидаться окончания синхронизации в пределах таймаута. Если за это время тики так и не будут синхронизированы в необходимом объёме, то будут возвращены только имеющиеся в наличии тики, а синхронизация продолжится в фоновом режиме.

Напомним, что тики, поступающие в режиме реального времени, транслируются на графики в виде событий: индикаторы получают уведомления о новых тиках в обработчике *OnCalculate*, а эксперты — в обработчике *OnTick*. При этом следует иметь в виду, что система не гарантирует доставку всех событий: если за то время, пока программа обрабатывает текущее событие *OnCalculate/OnTick*, в терминал поступили новые тики, события о них для этой "занятой" программы не формируются и не добавляются в её очередь (см. раздел Обзор функций обработки событий). Более того, в терминал может одновременно прийти несколько тиков, но для каждой MQL-программы будет сгенерировано лишь одно событие — об актуальном состоянии рынка. В связи с этим весьма полезной является функция *CopyTicks*, которая позволяет запрашивать все тики, пришедшие с момента предыдущей обработки события. Вот как выглядит этот алгоритм в псевдокоде:

```
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;
         ... // обработка всех пропущенных тиков
      }
   }
   else
   {
      MqlTick tick;
      SymbolInfoTick(_Symbol, tick);
      prev = tick.time_msc;
      ... // обработка первого тика
   }
}
```

Функция *SymbolInfoTick*, использованная здесь, заполняет передаваемую по ссылке одиночную структуру *MqlTick* информацией о последнем тике. Мы изучим её в отдельном разделе.

Обратите внимание, что при вызове *CopyTicks* к старой метке времени *prev* добавляется одна миллисекунда. Это гарантирует, что прежний тик не будет обработан повторно. Однако, если в течение одной миллисекунды, соответствующей *prev*, на самом деле было несколько тиков, данный алгоритм пропустит их. Если требуется охватить абсолютно все тики, нужно запоминать количество имеющихся тиков с временем *prev* одновременно с обновлением переменной *prev*. Затем при следующем вызове *CopyTicks* следует запрашивать тики с момента *prev* и пропускать (игнорировать в массиве) сохраненное количество "старых" тиков.

Вместе с тем отметим, что вышеприведенный алгоритм требуется далеко не каждой MQL-программе. Большинство из них не проводит анализ каждого тика, а текущее состояние цен, соответствующее последнему известному тику, достаточно оперативно транслируется на графики в модели событий и доступно через свойства символов и графиков.

Для демонстрации функций рассмотрим два примера — по одному на каждую функцию. Для обоих примеров был разработан общий заголовочный файл *TickEnum.mqh*, в котором

вышеприведенные константы для флагов запрашиваемых тиков и флагов состояния тиков сведены в два перечисления.

```
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,
   ...
};
```

Применение перечислений делает более строгой проверку типов в исходном коде, а также упрощает отображение смысла значений в виде строк с помощью *EnumToString*. Кроме того, в перечисление TICK_FLAGS добавлены наиболее ходовые комбинации флагов для оптимизации визуализации или фильтрации тиков. К сожалению, дать элементам перечислений те же имена, что и встроенные константы, нельзя — возникает конфликт имен.

Первый скрипт *SeriesTicksStats.mq5* использует функцию *CopyTicks*, для того чтобы подсчитать количество тиков с различными взведенными флагами на заданную глубину истории.

Во входных параметрах можно задать рабочий символ (по умолчанию, символ графика), количество анализируемых тиков и режим запроса из COPY_TICKS.

```
input string WorkSymbol = NULL; // Symbol (leave empty for current)
input int TickCount = 10000;
input COPY_TICKS TickType = ALL_TICKS;
```

Статистика встречаемости каждого флага (а точнее, каждого бита в битовой маске) в свойствах тиков собирается в структуре *TickFlagStats*.

```
struct TickFlagStats
{
   TICK_FLAGS flag; // маска с взведенным битом (одним или несколькими)
   int count;       // количество тиков с таким битом в поле flags
   string legend;   // описание бита
};
```

В функции *OnStart* описан массив структур *TickFlagStats* размером в 8 элементов: 6 из них (с 1 по 6 включительно) используются для соответствующих TICK_FLAG-битов, а два остальных для комбинаций битов (см. далее). С помощью простого цикла в массиве заполняются элементы для отдельных стандартных битов/флагов, а после цикла — две комбинированных маски (в 0-м элементе будут подсчитываться тики с одновременным изменением *Bid* и *Ask*, а в 7-м элементе — тики с одновременными сделками *Buy* и *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;  // BID И ASK комбинация
   stats[7].flag = TF_BUY_SELL; // BUY И SELL комбинация
   stats[0].legend = "TF_BID_ASK (COMBO)";
   stats[7].legend = "TF_BUY_SELL (COMBO)";
   ...
```

Всю основную работу мы поручим вспомогательной функции *CalcTickStats*, передав в неё входные параметры и подготовленный массив для сбора статистики. После этого останется вывести подсчитанные числа в журнал.

```
   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);
}
```

Наибольший интерес, конечно, представляет сама функция *CalcTickStats*.

```
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);

      // цикл по тикам
      for(int j = 0; j < nt; ++j)
      {
         // цикл по флагам TICK_FLAGs (2 4 8 16 32 64) и комбинациям
         for(int k = 0; k < nf; ++k)
         {
            if((ticks[j].flags & stats[k].flag) == stats[k].flag)
            {
               stats[k].count++;
            }
         }
      }
   }
   return nt;
}
```

Её назначение запросить с помощью *CopyTicks* тики указанного символа (*symbol*), конкретного типа (*type*), начиная с даты *start*, в количестве *count* штук. Параметр *start*, будучи типа *datetime*, должен пересчитываться в миллисекунды при передаче в *CopyTicks*. Напомним, что если *start = 0* (что имеет место в нашем случае, в функции *OnStart*), система вернет последние тики, то есть отсчет ведется от текущего времени. Поэтому при каждом вызове скрипта статистика, скорее всего, будет обновляться за счет прихода новых тиков. Исключение могут составить лишь запросы на выходных или малоликвидных инструментах.

Если *CopyTicks* выполнилась без ошибок, наш код выводит в журнал временной диапазон, покрытый полученными тиками.

Наконец, в цикле мы проходим по всем тикам и подсчитываем количество побитовых совпадений во флагах тиков и масках элементов в подготовленном заранее массиве статистических структур *TickFlagStats*.

Желательно запускать скрипт на инструментах, где имеется информация о реальных объемах и сделках, чтобы протестировать все режимы из перечисления COPY_TICKS (напомним, они соответствуют константам для параметра *flags* в *CopyTicks*: COPY_TICKS_INFO, COPY_TICKS_TRADE и COPY_TICKS_ALL).

Вот пример записей в журнале при запросе статистики для 100000 тиков всех типов (*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)"
```

А вот что получится, если запросить только информационные тики (*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)"
```

Здесь можно проверить точность расчетов: сумма чисел для "TF_BID" и "TF_ASK" за вычетом совпадений "TF_BID_ASK (COMBO)" дает ровно 100000 (общее количество тиков). Тики с объемами и ценой *Last* в результат не попали ни разу, как и ожидалось.

Теперь запустим скрипт еще раз — исключительно для торговых тиков (*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)"
```

Все тики имели флаги "TF_LAST" и "TF_VOLUME", а смешивание направлений сделок случилось 7308 раз. Опять же сумма показателей "TF_BUY" и "TF_SELL" за вычетом их комбинации совпадает с общим количеством тиков.

Второй скрипт *SeriesTicksDeltaVolume.mq5* использует функцию *CopyTicksRange* для расчета дельт объемов на каждом баре. Как известно, котировки MetaTrader 5 содержат лишь обезличенные объемы, в которых покупки и продажи объединены в одной величине для каждого бара. Однако

наличие истории реальных тиков позволяет посчитать отдельно суммы объемов покупок и продаж, а также их разницу. Данные характеристики являются дополнительными важными факторами для принятия торговых решений.

Во входных параметрах задаются похожие настройки, как и в первом скрипте, в частности, название символа для анализа и режим запроса тиков. Правда, в данном случае дополнительно потребуется указать таймфрейм, потому что дельты объемов должны считаться побарово. По умолчанию будет использоваться текущий таймфрей графика. Для указания количества обсчитываемых баров предназначен параметр *BarCount*.

```
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;
```

Статистика по каждому бару хранится в структуре *DeltaVolumePerBar*.

```
struct DeltaVolumePerBar
{
   datetime time; // время бара
   ulong buy;     // чистый объем покупок
   ulong sell;    // чистый объем продаж
   long delta;    // разница объемов
};
```

В функции *OnStart* описан массив таких структур, его размер выделяется под указанное количество баров.

```
void OnStart()
{
   DeltaVolumePerBar deltas[];
   ArrayResize(deltas, BarCount);
   ZeroMemory(deltas);
   ...
```

А вот и основной алгоритм.

```
   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)
      {
         ...
      }
   }
```

В цикле по барам получаем временные рамки для каждого бара: *prev* и *next* (0-й неполный бар не обрабатывается). Вызываем *CopyTicksRange* для этого интервала, не забывая перевести *datetime*

в миллисекунды и вычесть 1 миллисекунду из правой границы, так как это время принадлежит уже следующему бару. В отсутствие ошибок обрабатываем в цикле массив полученных тиков.

```
         deltas[i].time = prev; // запомним время бара
         for(int j = 0; j < n; ++j)
         {
            // когда реальные объемы могут быть доступны, берем их из тиков
            if(TickType == TRADE_TICKS)
            {
               // раздельно аккумулируем объемы по сделкам покупки и продажи
               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;
               }
            }
            // когда реальных объемов нет, оцениваем их по движению цен вверх/вниз
            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);
```

Если в настройках скрипта запрашивался анализ по торговым тикам (TRADE_TICKS), проверяем наличие флагов TICK_FLAG_BUY и TICK_FLAG_SELL, и если хотя бы один из них взведен, учитываем объем из поля *volume* в соответствующей переменной структуры *DeltaVolumePerBar*. Данный режим подходит только для биржевых инструментов. Для инструментов Forex объемы и флаги направлений сделок не заполняются, а потому следует изобрести другой подход.

Если в настройках указаны информационные тики (INFO_TICKS), которые есть для всех инструментов, алгоритм строится на следующих эмпирических правилах. Как известно, покупки толкают цену вверх, а продажи — вниз. Поэтому можно предположить, что если средняя цена *Ask+Bid* двинулась в новом тике вверх относительно предыдущего, то на нем произошла покупка, а если цена двинулась вниз — продажа. Объем можно примерно оценить как количество пройденных пунктов (*_Point*).

Результаты подсчетов выводятся просто как массив структур с собранной статистикой.

```
   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);
}
```

Ниже показана пара фрагментов журналов для режимов TRADE_TICKS и 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
...
```

Значения, конечно, существенно разные, но дело не в абсолютных величинах: в отсутствие биржевых объемов даже такая эмуляция расщепления и динамика дельт позволяет взглянуть на поведение рынка под другим углом.

```
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
...
```

Когда мы научимся создавать индикаторы, то сможем встроить этот алгоритм в один из них (см. пример *IndDeltaVolume.mq5* в разделе Ожидание данных и управление видимостью), чтобы наглядно отобразить дельты непосредственно на графике.

5.4 Создание пользовательских индикаторов

Индикаторы — один из наиболее востребованных типов MQL-программ. Они являются простым и вместе с тем мощным средством технического анализа. Основной механизм их использования заключается в обработке исходных котировочных данных с помощью формул для создания производных таймсерий, которые позволяют оценить и визуализировать специфические характеристики рыночных процессов. В принципе, любая таймсерия, в том числе и та, что получена в результате расчетов индикатора, может быть подана в другой индикатор, и так далее. Формулы многих известных индикаторов (например, MACD) фактически состоят из обращений к нескольким взаимосвязанным индикаторам.

Пользователи терминала, несомненно, знакомы со многими встроенными индикаторами, а также знают, что с помощью языка MQL5 перечень доступных индикаторов может быть расширен. С точки зрения пользователя встроенные и заказные индикаторы, реализованные на MQL5, работают совершенно одинаково.

Как правило, индикаторы отображают результаты своей работы в виде линий, гистограмм и прочих графических построений в окне графика с котировками. Каждая такая диаграмма визуализируется на основе рассчитанных таймсерий, которые хранятся внутри индикаторов в специальных массивах, называемых индикаторными буферами: они доступны для просмотра в *Окне данных* терминала наравне с OHLC-ценами. Однако индикаторы могут предоставлять дополнительный функционал помимо буферов или не иметь их вовсе. Например, индикаторы часто применяются для решения задач, где требуется создавать графические объекты, управлять самим графиком и его свойствами, взаимодействовать с пользователем (см. *OnChartEvent*).

В данной главе мы изучим базовые принципы создания индикаторов на MQL5. Такие индикаторы принято называть "пользовательскими", потому что пользователь может сам написать их с нуля или скомпилировать из готовых исходных кодов. В следующей главе мы обратимся к вопросам программного управления пользовательскими и встроенными индикаторами, что позволит конструировать более сложные индикаторы и подготовит почву для получения на основе индикаторов торговых сигналов и фильтров для экспертов.

Чуть позже мы освоим технологию по внедрению индикаторов внутрь исполняемых MQL-программ в виде ресурсов.

5.4.1 Основные характеристики индикаторов

Индикатор реализует некий расчетный алгоритм, применяемый по барам к заданной исходной таймсерии или нескольким таймсериям. Все такие временные ряды являются *собственными* массивами (см. функцию *ArrayIsSeries*) терминала: он сам распределяет под них память и добавляет новые элементы по мере формирования новых баров. Среди таких массивов основополагающую роль играют, разумеется, массивы с котировками символов на разных таймфреймах: их терминал также заполняет сам. Однако запущенные индикаторы способны существенно расширить набор таймсерий, доступных для анализа.

Дело в том, что результаты своей работы индикатор обычно сохраняет в динамические массивы, которые с помощью специальной функции (*SetIndexBuffer*) регистрируются как индикаторные буфера и становятся также собственными массивами терминала. Помимо выделения под них памяти терминал обеспечивает к этим массивам публичный доступ как к новым таймсериям, на которых могут рассчитываться уже другие индикаторы.

Точкой входа в расчетную часть индикатора является функция *OnCalculate* — обработчик одноименного события. В разделе Обзор функций обработки событий мы уже упоминали данную функцию: одного её наличия в исходном коде достаточно, чтобы MQL-программа стала восприниматься терминалом как индикатор. Функция *OnCalculate* будет подробно описана в следующем разделе. В частности, главной особенностью *OnCalculate* является наличие двух разных форм. Программисту требуется определиться с выбором в пользу одного или другого варианта в самом начале проектирования индикатора, потому что это определяет назначение и возможные сценарии использования.

Функция *OnCalculate* является не единственной отличительной чертой индикатора. Кроме неё исключительно для индикаторов предназначена группа специальных директив препроцессора *#property* — мы рассмотрим их поэтапно в нескольких соответствующих разделах этой Главы. Напомним, что ранее мы встречались с некоторыми Общими свойствами программ, и такие директивы, разумеется, тоже применимы к индикаторам.

Как известно пользователям MetaTrader 5, в каждый индикатор заложен способ отображения его графических построений (таймсерий): либо в главном окне, в котором выводятся котировки, либо в отдельной панели (подокне). Такая панель создается в нижней части окна при набрасывании на график конкретного индикатора (или группы индикаторов), если он спроектирован для работы в подокне. Например, стандартный индикатор Moving Average (MA) рисуется вместе с котировками, а Williams Percent Range (WPR) — в отдельном подокне.

С точки зрения разработчика это означает, что следует изначально определить, будет ли индикатор предназначен для показа на главном окне или в подокне, потому что совместить эти два режима нельзя. Более того, эта характеристика, так же как и количество индикаторных буферов могут задаваться лишь однажды с помощью директив *#property* (см. разделы Два типа

индикаторов и Настройка количества буферов и графических построений), и изменить их затем с помощью вызовов функций MQL5 API не удастся — таких функций просто не предусмотрено. В отличие от этих неизменяемых атрибутов, большинство прочих свойств индикаторов может динамически подстраиваться специальными функциями. Таким образом, по мере изучения технических аспектов программирования индикаторов, мы сможем установить соответствия между свойствами *#property* и функциями MQL5.

Также в индикаторах обычно реализуются обработчики *OnInit* и *OnDeinit* (см. раздел Опорные события индикаторов и советников). *OnInit* особенно важен для назначения массивов, которые будут выполнять роль индикаторных буферов, то есть аккумулировать результаты промежуточных и окончательных расчетов, видимых пользователю и доступных другим программам, таким как советники.

Индикатор относится к числу интерактивных MQL-программ, которые способны, при необходимости, работать с событиями таймера (*OnTimer*) и изменениями графика (*OnChartEvent*), производимыми пользователем или другими программами. Эти технические возможности являются опциональными для индикаторов и основываются на очереди событий графика, а потому будут рассмотрены отдельно, в главе, посвященной графикам.

5.4.2 Главное событие индикаторов: OnCalculate

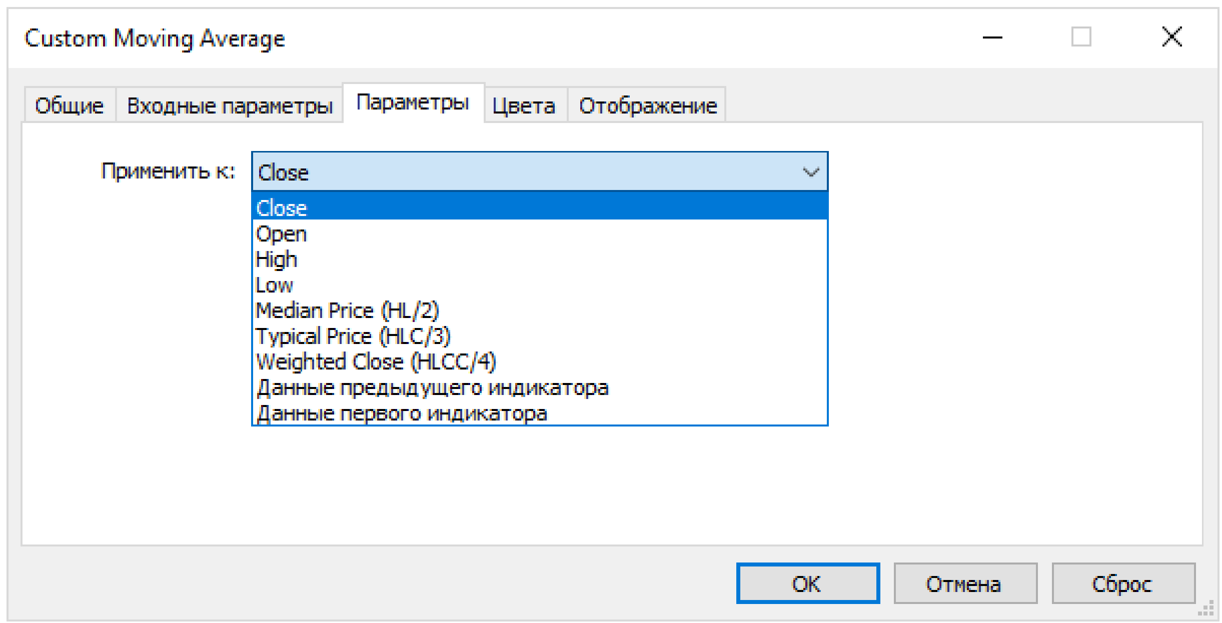
Функция *OnCalculate* является основной точкой входа в MQL5 код индикатора: она вызывается при наступлении одноименного события, которое генерируется при изменений ценовых данных. Например, оно может случиться при поступлении нового тика по символу или изменении старых цен (заполнение пропуска в истории или её докачка с сервера).

Существует два варианта функции, отличающиеся тем, какие данные выступают исходным материалом для расчетов:

- полная — предоставляет в параметрах набор стандартных ценовых таймсерий (цены OHLC, объемы, спреды);
- сокращенная — для одной произвольной таймсерии (не обязательно стандартной);

Конкретный индикатор должен использовать лишь один из двух вариантов, совместить их в одном индикаторе нельзя.

В случае использования сокращенной формы *OnCalculate* во время размещения индикатора на графике в его диалоге свойств становится доступна дополнительная закладка, где с помощью выпадающего списка *Применить к* следует выбрать исходную таймсерию, на основе которой будет рассчитываться индикатор. По умолчанию, если таймсерия не выбрана, расчет производится по значениям цены *Close*.



Выбор исходной таймсерии для индикатора с сокращенной формой OnCalculate

В списке всегда предлагаются стандартные типы цен, но при наличии на графике других индикаторов данная настройка позволяет выбрать один из них в качестве поставщика исходных данных для добавляемого индикатора, тем самым выстраивая цепочку обработки из индикаторов. Мы попробуем построить один индикатор от другого в разделе Пропуск отрисовки на начальных барах. При использовании полной формы данная возможность отсутствует.

> Запрещено применять индикаторы к нескольким встроенным индикаторам: Fractals, Gator, Ichimoku и Parabolic SAR.

Краткая форма *OnCalculate* имеет следующий прототип.

```
int OnCalculate(const int rates_total, const int prev_calculated, const int begin,
  const double &data[])
```

Массив *data* содержит исходные данные для расчета. Это может быть одна из ценовых таймсерий либо рассчитанный буфер другого индикатора. Параметр *rates_total* указывает размер массива *data*. В принципе, вызовы *ArraySize(data)* или *iBars(NULL, 0)* должны дать то же самое значение, что и *rates_total*.

Параметр *prev_calculated* предназначен для эффективного пересчета индикатора на небольшом количестве новых баров (обычно на одном, последнем), вместо полного расчета на всех барах. Значение *prev_calculated* равно результату функции *OnCalculate*, возвращенному среде исполнения из предыдущего вызова функции. Например, если при поступлении очередного тика индикатор посчитал заложенную в него формулу для всех баров, он должен вернуть из *OnCalculate* значение $rates_total_A$ (здесь индекс A означает начальный момент времени). Тогда на следующем тике, при получении события *OnCalculate* терминал установит *prev_calculated* в прежнее значение $rates_total_A$. Однако количество баров за это время может уже измениться и новое значение *rates_total* увеличится — обозначим его $rates_total_B$. Таким образом, пересчету будут подлежать только бары от *prev_calculated* (он же $rates_total_A$) до $rates_total_B$.

Однако наиболее частной является ситуация, когда новые тики укладываются в текущий нулевой бар, то есть *rates_total* не меняется, и потому в большинстве вызовов *OnCalculate* выполняется равенство *prev_calculated == rates_total*. Нужно ли что-то пересчитывать в таком

случае? Это зависит от сути вычислений. Например, если индикатор считается по ценам открытия баров, которые, очевидно, не меняются, то пересчитывать что-либо не имеет смысла. Однако если индикатор использует в расчетах цену закрытия (фактически, цену последнего известного тика) или любую другую сводную цену, зависящую от *Close*, то последний бар следует всегда пересчитывать.

При первом вызове функции *OnCalculate* значение *prev_calculated* равно 0.

Если с момента последнего вызова функции *OnCalculate* ценовые данные были изменены (например, подкачана более глубокая история или в ней заполнены пропуски), то значение параметра *prev_calculated* будет также установлено в 0 самим терминалом. Таким образом, индикатору будет дан сигнал на полный пересчет по всей доступной истории.

Если функция *OnCalculate* возвращает нулевое значение, индикатор не отрисовывается, а названия и значения его буферов в *Окне данных* будут скрыты.

> Обратите внимание, что возврат полного количества баров *rates_total* является единственным стандартным способом сообщить терминалу и другим MQL-программа, которые будут использовать индикатор, о готовности его данных. Даже если индикатор разработан так, чтобы вести расчет и показывать лишь ограниченное количество данных, он должен вернуть *rates_total*.

Направление индексации массива *data* может быть выбрано с помощью вызова *ArraySetAsSeries* (по умолчанию оно равно *false*, в чем можно убедиться, вызвав *ArrayGetAsSeries*). В то же время, если применить к массиву функцию *ArrayIsSeries*, она вернет значение *true*. Это означает, что данный массив является внутренним массивом, управляемым терминалом. Индикатор не может его как-либо менять, а только читать, тем более что в описании параметра присутствует модификатор *const*.

Параметр *begin* сообщает количество начальных значений массива *data*, которые следует исключить из расчета. Параметр устанавливается системой, когда наш индикатор настроен пользователем таким образом, что получает *data* из другого индикатора (см. изображение выше). Например, если выбранный индикатор-источник данных выполняет расчет скользящего среднего периода *N*, то на первых *N - 1* барах исходных данных по определению не существует, поскольку там невозможно провести усреднение по *N* барам. Если в этом индикаторе-источнике его разработчик установил специальное свойство, оно корректно передастся нам в параметре *begin*. Мы скоро проверим этот аспект на практике (см. раздел Пропуск отрисовки на начальных барах).

Попробуем создать пустой индикатор с сокращенной формой *OnCalculate*. Он пока ничего не будет уметь, но послужит заготовкой для дальнейших экспериментов. Исходный файл *IndStub.mq5* можно найти в папке *MQL5/Indicators/MQL5Book/p5/*. Чтобы убедиться в работе индикатора добавим в *OnCalculate* вывод значений *prev_calculated* и *rates_total* в журнал, а также подсчет количества вызовов функции.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   static int count = 0;
   ++count;
   // сравним количество баров на предыдущем вызове и текущем
   if(prev_calculated != rates_total)
   {
      // сигнализируем только при различии
      PrintFormat("calculated=%d rates=%d; %d ticks",
         prev_calculated, rates_total, count);
   }
   return rates_total; // вернем количество обработанных баров
}
```

Условие на неравенство *prev_calculated* и *rates_total* гарантирует, что сообщение появится только при первом размещении индикатора на графике, а также по мере появления новых баров. Все тики, приходящие в процессе формирования текущего бара, не будут менять количество баров, а потому *prev_calculated* и *rates_total* будут равны. Однако мы подсчитаем в переменной count общее количество тиков.

Остальные параметры пока остались не у дел, но мы постепенно задействуем все возможности.

Компиляция данного исходного кода проходит успешно, но генерирует два предупреждения.

```
no indicator window property is defined, indicator_chart_window is applied
no indicator plot defined for indicator
```

Они указывают на отсутствие некоторых директив *#property*, которые хоть и не являются обязательными, но задают базовые свойства индикатора. В частности, первое предупреждение гласит, что для индикатора не выбран способ привязки — к основному окну или подокну, и потому по умолчанию будет использоваться основное окно графика. Второе предупреждение связано с тем, что мы не задали количество диаграмм для отображения. Как уже было сказано, некоторые индикаторы специально проектируются без буферов, поскольку предназначены для выполнения иных действий, но в нашем случае это напоминание о том, чтобы добавить визуальную часть позднее.

Мы займемся ликвидацией предупреждений через пару абзацев, а пока запустим индикатор на графике EURUSD,M1. Таймфрейм M1 выбран не случайно: так мы достаточно быстро сможем увидеть формирование новых баров и появление сообщений в журнале.

```
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
```

Таким образом, мы видим, что обработчик *OnCalculate* вызывается, как ожидалось, и в нем можно выполнять расчеты, как на каждом тике, так и по барам. Напомним, что убрать индикатор с графика можно, вызвав диалог *Список индикаторов* из контекстного меню графика и нажав там кнопку *Удалить*, предварительно выделив требуемый индикатор в списке.

Сейчас вернемся ненадолго к той особенности функции *OnCalculate*, что она имеет два потенциальных прототипа. Мы попробовали в деле сокращенный вариант, но могли бы реализовать точно такую же заготовку и для полной формы.

Полная форма предназначена для расчета на основе стандартных ценовых таймсерий и имеет следующий прототип.

```
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[])
```

Параметры *rates_total* и *prev_calculated* имеют тот же смысл, что и в простой форме *OnCalculate*: *rates_total* задает размер переданных таймсерий (все массивы имеют одинаковую длину, так как это общее количество баров на графике), а *prev_calculated* содержит количество баров, обработанных на предыдущем вызове (то есть, значение, которое функция *OnCalculate* вернула ранее терминалу с помощью инструкции return).

Массивы *open*, *high*, *low* и *close* содержат, соответственно, цены открытия, максимальную, минимальную и закрытия для баров текущего графика — таймсерии рабочей пары символа и таймфрейма. Массив *time* содержит время открытия каждого бара, а *tick_volume* и *volume* — торговые объемы за бар: тиковый и биржевой.

В предыдущей главе мы изучили Временные ряды со стандартными типами цен и объемов, предоставляемые терминалом для MQL-программ через набор функций. Так вот для удобства расчета индикатора эти временные ряды передаются в обработчик *OnCalculate* напрямую по ссылке в виде массивов. Это избавляет от необходимости вызывать эти функции и копировать (дублировать) котировки во внутренние массивы. Разумеется, такой прием подходит только для тех индикаторов, которые рассчитываются на одном сочетании рабочего символа и таймфрейма, совпадающих с текущим графиком. Однако MQL5 позволяет создавать мультивалютные, мультитаймфреймовые индикаторы, а также индикаторы для "чужих" по отношению к графику символов и таймфреймов. Во всех таких случаях уже на обойтись без функций доступа к таймсериям. Чуть позже мы посмотрим, как это делается.

Если проверить для всех переданных массивов атрибут принадлежности терминалу с помощью *ArrayIsSeries*, эта функция вернет *true*. Все массивы доступны только на чтение. Это подчеркивает и модификатор *const* в описании параметров.

Выбор между полной и сокращенной формами делается исходя из потребностей расчетного алгоритма в данных. Например, для сглаживания массива с помощью алгоритма скользящей средней требуется только один входной массив, и потому индикатор может быть построен по любому типу цены, на выбор пользователя. Однако известные индикаторы *ParabolicSAR* или *ZigZag* требуют цен *High* и *Low*, а потому должны использовать полную версию *OnCalculate*. В следующих разделах мы увидим примеры индикаторов как для простой версии *OnCalculate*, так и для полной.

5.4.3 Два типа индикаторов: для главного и отдельного окна

Как известно, индикаторы в MetaTrader 5 способны отображать свои линии в двух местах: в главном окне поверх котировок или в отдельном окне, создаваемой под графиком котировок. Эти два режима являются взаимоисключающими: каждый индикатор проектируется либо для главного окна, либо для отдельного, но не может совместить оба способа.

Для тех случаев, когда от программы требуется визуализация данных и там, и там, существует несколько альтернативных решений. Например, проект можно реализовать в виде двух взаимодействующих индикаторов (техническая сторона взаимодействия остается открытой: это могут быть ресурсы, файлы, СУБД, или разделяемая память, доступная через DLL). Другой подход предполагает использовать индикаторные буфера в одном из окон, например, в нижней панели, а визуализацию на главном графике выполнять с помощью графических объектов.

Напомним, что как в основном окне, так и в нижней панели пользователи могут наносить несколько индикаторов. Если индикатор предназначен для работы в отдельной панели, то его перетаскивание мышью из Навигатора в основное окно приведет к автоматическому созданию новой панели под этот индикатор. Однако если в окне уже существует нижняя панель с другим индикатором, то новый можно перетащить туда же, совместив тем самым два или более индикатора. При этом возможны различные режимы масштабирования индикаторов в одном окне. По умолчанию, построения каждого индикатора масштабируются автоматически и независимо друг от друга на всю высоту панели, но это можно изменить (см. пример *SubScaler.mq5* в разделе про события клавиатуры на графиках).

Выбор окна отображения индикатора производится с помощью одной из двух директив компиляции.

```
#property indicator_chart_window     // выводим индикатор в окно графика
#property indicator_separate_window  // выводим индикатор в отдельное окно
```

Разработчик индикатора должен вставить одну из них в начало исходного кода. Если ни одной из директив нет, по умолчанию предполагается вывод в основное окно, но компилятор выдает предупреждение. Мы видели это в предыдущем разделе. Далее в примерах мы будем обязательно указывать *#property indicator_chart_window* или *#property indicator_separate_window*.

Второе предупреждение при компиляции *IndStub.mq5* касалось отсутствующей настройки буферов и диаграмм. Займемся им в следующем разделе.

> Действие выпадающего списка *Применить к* в настройках индикатора зависит от того, для какого окна он разработан.
>
> Индикатор для отдельного окна можно *Применить к* индикатору в подокне, но не к индикатору в главном окне.
>
> Однако индикатор для главного окна можно *Применить к* любому индикатору, как к тому, что в главном окне, так и в подокне.

5.4.4 Настройка количества буферов и графических построений

Чтобы индикатор мог вывести на график результаты своих расчетов, он должен определить один или несколько массивов и объявить их индикаторными буферами. Количество буферов устанавливается с помощью директивы:

```
#property indicator_buffers N
```

Здесь N — целое число от 1 до 512. Эта директива задает количество буферов, которые будут доступны в коде для расчета индикатора.

В качестве N должна быть указана целочисленная константа (литерал) или эквивалентное ей макроопределение. Поскольку это директива препроцессора, никакие переменные (пусть и с модификатором const) на стадии предобработки исходного кода еще не существуют.

Однако буферов недостаточно для визуализации расчетных данных. В MQL5 система визуализации является двухуровневой. Первый уровень как раз формируют индикаторные буфера — динамические массивы, хранящие данные для отображения. Второй уровень предназначен для управления тем, как эти данные будут отображаться. Строится он на основе новых сущностей — так называемых графических построений (или диаграмм), в англоязычной терминологии — "plot". Дело в том, что различные способы отображения данных могут потребовать разного количества индикаторных буферов. Например, скользящая средняя имеет ровно одно значение на бар, и потому для такой линейной диаграммы достаточно одного индикаторного буфера. Однако для показа диаграммы в виде японских свечей требуется 4 значения на бар (цены OHLC). Таким образом, одному такому графическому построению требуется 4 индикаторных буфера.

Количество диаграмм (P) также должно быть определено в исходном коде с помощью специальной директивы.

```
#property indicator_plots   P
```

В простейшем случае количество буферов и диаграмм совпадает. Но мы скоро разберем примеры, когда буферов потребуется больше, чем графических построений. Кроме ситуаций, в которых графическое построение конкретного вида требует предопределенного количества буферов, нам иногда придется столкнуться с необходимостью выделять один или несколько массивов под промежуточные вычисления. Такие массивы не участвуют напрямую в визуализации, но содержат данные для построения визуализируемых буферов. Конечно, можно использовать для таких целей простые динамические массивы, не объявляя их буферами. Но тогда нам пришлось бы самостоятельно контролировать и изменять их размеры. Гораздо удобнее сделать их буферами и тем самым поручить терминалу выделение памяти.

> Количество буферов и графических построений можно задавать только с помощью директив препроцессора, динамическое изменение этих свойств с помощью функций MQL5 невозможно.

После того как количество буферов и диаграмм будет определено, следует описать в исходном коде сами массивы, которые станут индикаторными буферами.

Начнем разрабатывать новый пример индикатора *IndReplica1.mq5*, чтобы продемонстрировать необходимые конструкции в исходном коде. Суть индикатора будет проста: в его единственном буфере отобразим значения полученного массива-параметра *data*. Как мы уже говорили ранее, выбор конкретной таймсерии для передачи в массив *data* осуществляет пользователь в момент наложения индикатора на график, а по умолчанию там будет предложена таймсерия с ценами закрытия баров.

Добавим директивы с описанием одного буфера и одной диаграммы.

```
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots 1
```

Директивы не выделяют сам буфер, а лишь устанавливают свойства индикатора и подготавливают систему исполнения к тому, что программа далее определит и настроит указанное количество массивов. Поэтому теперь самое время разобраться с регистрацией массива в качестве буфера.

5.4.5 Назначение массива в качестве буфера: SetIndexBuffer

Роль индикаторных буферов могут выполнять любые динамические массивы типа *double* со временем существования от запуска программы и до её остановки. Самый распространенный способ определения такого массива — на глобальном уровне. Но в некоторых случаях удобнее делать массивы членами классов, после чего создавать глобальные объекты с массивами. Мы рассмотрим примеры такого подхода, когда реализуем мультивалютный индикатор (см. пример *IndUnityPercent.mq5* в разделе Мультивалютные и мультитаймфреймовые индикаторы) и индикатор дельты объемов (см. *IndDeltaVolume.mq5* в разделе Ожидание данных и управление видимостью).

Итак, опишем на глобальном уровне динамический массив *buffer* (без указания размера).

```
double buffer[];
```

Чтобы зарегистрировать его как буфер, в терминале существует специальная функция *SetIndexBuffer*. Как правило, она вызывается в обработчике *OnInit*, как и многие другие функции для настройки индикатора, которые мы рассмотрим позднее.

```
bool SetIndexBuffer(int index, double buffer[],
  ENUM_INDEXBUFFER_TYPE mode = INDICATOR_DATA)
```

Функция связывает указанный по индексу (*index*) индикаторный буфер с динамическим массивом *buffer*. Значение *index* должно лежать в пределах от 0 до N - 1, где N — количество буферов, определенное директивой *#property indicator_buffers*.

Сразу после привязки массив еще не готов для работы с данными и даже не меняет свой размер, поэтому инициализацию и все расчеты следует выполнять уже в функции *OnCalculate*. Изменять размер динамического массива после его назначения в качестве индикаторного буфера нельзя. Для индикаторных буферов все операции по изменению размера производит сам терминал.

Направление индексации после связывания массива с индикаторным буфером по умолчанию устанавливается как в обычных массивах, но при необходимости его можно изменить с помощью функцию *ArraySetAsSeries*.

Функция *SetIndexBuffer* возвращает признак успешного выполнения (*true*) или ошибки (*false*).

Опциональный параметр *mode* указывает системе, каким образом будет использоваться буфер. Возможные значения собраны в перечислении ENUM_INDEXBUFFER_TYPE.

Идентификатор	Описание
INDICATOR_DATA	данные для отрисовки
INDICATOR_COLOR_INDEX	цвета отрисовки
INDICATOR_CALCULATIONS	внутренние результаты промежуточных вычислений

По умолчанию индикаторный буфер предназначен для отрисовки данных (INDICATOR_DATA). Это значение привносит еще один эффект помимо отображения массива на графике: значение каждого буфера для бара под курсором мыши показываются в *Окне данных*. Правда, с помощью некоторых настроек индикатора это поведение можно изменить (см. свойство PLOT_SHOW_DATA в разделе Настройка графических построений). Большинство примеров в данной Главе относится к режиму INDICATOR_DATA.

Если для расчета индикатора требуется хранить промежуточные результаты для каждого бара, для них можно выделить вспомогательный неотображаемый буфер (INDICATOR_CALCULATIONS). Это более удобно, чем использование для тех же целей обычного массива, поскольку тогда программист должен самостоятельно управлять его размером. В данной Главе будет представлено два примера с INDICATOR_CALCULATIONS: *IndTripleEMA.mq5* (см. раздел Пропуск отрисовки на начальных барах) и *IndSubChartSimple.mq5* (см. раздел Мультивалютные и мультитаймфреймовые индикаторы).

Некоторые построения позволяют задавать для каждого бара цвет отображения. Для хранения информации о цвете используются цветовые буфера (INDICATOR_COLOR_INDEX). Цвет представлен целочисленным типом *color*, но все индикаторные буфера должны иметь тип *double*, и в них в данном случае хранится номер цвета из особой палитры, заданной разработчиком (см. раздел Поэлементное раскрашивание диаграмм и пример индикатора *IndColorWPR.mq5* в нем).

Значения цветовых и вспомогательных буферов не отображаются в *Окне данных*, а также их нельзя получить с помощью функции *CopyBuffer*, которую мы изучим позднее в Главе про Использование встроенных и пользовательских индикаторов из MQL5.

Индикаторный буфер не инициализируется никакими значениями. Если какие-то его элементы не рассчитываются по тем или иным причинам (например, в настройках индикатора есть ограничение на максимальное количество баров или само графическое построение подразумевает редкие значащие элементы, между которыми должны быть пропуски, как между вершинами ZigZag-а), то их следует явным образом заполнить специальным "пустым" значением. "Пустое" значение не отображается на графике и не выводится в Окне данных. По умолчанию для него существует константа EMPTY_VALUE (DBL_MAX), но при необходимости его можно заменить на любое другое, например, на "обычный" 0. Это делается с помощью функции *PlotIndexSetDouble*.

С учетом новых знаний о функции *SetIndexBuffer* дополним наш очередной пример *IndReplica1.mq5*, который мы начали в предыдущем разделе. В частности, нам потребуется обработчик *OnInit*.

```
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots 1

#include <MQL5Book/PRTF.mqh>

double buffer[]; // глобальный динамический массив

int OnInit()
{
   // регистрируем массив в качестве индикаторного буфера
   PRTF(SetIndexBuffer(0, buffer)); // true / ok
   // здесь сделан намеренно второй некорректный вызов, чтобы показать ошибку
   PRTF(SetIndexBuffer(1, buffer)); // false / BUFFERS_WRONG_INDEX(4602)
   // проверяем размер: по-прежнему 0
   PRTF(ArraySize(buffer)); // 0
   return INIT_SUCCEEDED;
}
```

Количество буферов определено директивой равным 1, поэтому назначение массива для единственного буфера использует индекс 0 (первый параметр *SetIndexBuffer*). Второй вызов

функции является ошибочным и сделан только для демонстрации проблемы: поскольку индекс 1 подразумевает наличие двух объявленных буферов, он генерирует ошибку BUFFERS_WRONG_INDEX (4602).

В самом начале функции *OnCalculate* выведем размер массива еще раз. В этом месте он уже будет распределен под количество баров.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   // после старта проверяем, что платформа автоматически управляет размером массива
   if(prev_calculated == 0)
   {
      PRTF(ArraySize(buffer)); // 10189 - актуальное количество баров
   }
   ...
```

Теперь обратимся к вопросу, что наш индикатор будет считать. Как уже было сказано, мы пока не будем закладывать в него сложные формулы, а просто попытаемся скопировать переданную таймсерию из параметра *data* в буфер. Это отражено и в названии индикатора.

```
   ...
   // на каждом новом баре или множестве баров (включая первый расчет)
   if(prev_calculated != rates_total)
   {
      // заполняем все новые бары
      ArrayCopy(buffer, data, prev_calculated, prev_calculated);
   }
   else // тики на текущем баре
   {
      // обновляем последний бар
      buffer[rates_total - 1] = data[rates_total - 1];
   }

   // сообщаем количество обработанных баров самим себе в будущем
   return rates_total;
}
```

Теперь индикатор компилируется без предупреждений. Мы можем запустить на графике, и при настройках по умолчанию он должен дублировать в буфере значения цены закрытия баров. Это происходит благодаря краткой форме *OnCalculate*, мы рассматривали этот аспект в разделе Главное событие индикаторов: OnCalculate.

Однако мы обнаружим странную вещь: в *Окне данных* наш буфер действительно выводится и содержит правильные значения, однако на графике линии нет. Это следствие того, что за отображение отвечают графические построения, а не буфера. В текущей версии индикатора мы настроили только буфер. В следующем разделе мы создадим новую версию *IndReplica2.mq5* и дополним необходимыми инструкциями.

Вместе с тем описанный эффект может быть полезен для создания "скрытных" индикаторов, которые не выводят свои линии на график, но доступны для программного чтения из других

MQL-программ. При желании разработчик сможет скрыть даже упоминание индикаторных буферов из *Окна данных* (см. PLOT_SHOW_DATA в следующем разделе).

О том, как управлять индикаторами из MQL5-кода, будет рассказано в следующей главе.

5.4.6 Настройка графических построений: PlotIndexSetInteger

За настройки графических построений в MQL5 API отвечает группа функций: *PlotIndexSetInteger*, *PlotIndexSetDouble*, *PlotIndexSetString*. Целочисленные свойства можно также читать через *PlotIndexGetInteger*. Нас в первую очередь как раз интересуют целочисленные свойства.

Функция *PlotIndexSetInteger* имеет две формы. О том, чем они отличаются — чуть ниже.

```
bool PlotIndexSetInteger(int index, ENUM_PLOT_PROPERTY_INTEGER property, int value)
bool PlotIndexSetInteger(int index, ENUM_PLOT_PROPERTY_INTEGER property, int modifier,
  int value)
```

Функция задает значение свойства графического построения под номером index. Значение *index* должно находиться в пределах от 0 до P - 1, где P — количество графических построений, заданное директивой *#property indicator_plots*. Само свойство идентифицируется параметром *property*: допустимые величины следует брать из перечисления ENUM_PLOT_PROPERTY_INTEGER (см. далее). Значение свойства передается в параметре *value*.

Вторая форма функции предназначена для свойств, которые распространяются на несколько компонентов (принадлежащих, тем не менее, одному свойству). В частности, для диаграмм некоторых типов может быть назначен не один цвет, а целая палитра: тогда параметр *modifier* позволяет изменить любой цвет в этом наборе.

В случае успешного выполнения функция возвращает *true*, в противном случае *false*.

В следующей таблице перечислены доступные свойства ENUM_PLOT_PROPERTY_INTEGER.

Идентификатор	Описание	Тип свойства
PLOT_ARROW	Код стрелки из шрифта Wingdings для диаграмм DRAW_ARROW	uchar
PLOT_ARROW_SHIFT	Смещение стрелок по вертикали для диаграмм DRAW_ARROW	int
PLOT_DRAW_BEGIN	Индекс первого бара (слева направо), с которого начинаются данные	int
PLOT_DRAW_TYPE	Тип графического построения (диаграммы)	ENUM_DRAW_TYPE
PLOT_SHOW_DATA	Признак отображения значений построения в *Окне данных* (*true* — видно, *false* — не видно)	bool
PLOT_SHIFT	Сдвиг графического построения индикатора по оси времени в барах (положительный — вправо, отрицательный — влево)	int
PLOT_LINE_STYLE	Стиль отрисовки линий	ENUM_LINE_STYLE
PLOT_LINE_WIDTH	Толщина линий в пикселях (1 - 5)	int
PLOT_COLOR_INDEXES	Количество цветов (1 - 64)	int
PLOT_LINE_COLOR	Цвет отрисовки	color (модификатор — номер цвета)

Постепенно мы изучим все свойства, но сейчас сконцентрируемся на трех основных: PLOT_DRAW_TYPE, PLOT_LINE_STYLE и PLOT_LINE_COLOR.

Индикаторы в MetaTrader 5 поддерживают несколько предопределенных типов диаграмм, называемых графическими построениями. Они определяют одновременно и способ визуального представления и требуемую структуру буферов с исходными данными для отображения.

Всего таких базовых построений 10, и на уровне MQL5 они описываются идентификаторами в перечислении ENUM_DRAW_TYPE. Именно свойству PLOT_DRAW_TYPE следует присвоить одно из значений ENUM_DRAW_TYPE.

Тип визуализации *примеры*	**Описание**	**Количество буферов**
DRAW_NONE *IndDeltaVolume.mq5*	на графике ничего не отображается, но значения соответствующего буфера доступны в Окне данных	1
DRAW_LINE *IndLabelHighLowClose.mq5*, *IndWPR.mq5*, *IndUnityPercent.mq5*	кривая линия по значениям буфера ("пустые" элементы образуют пропуск в линии)	1
DRAW_SECTION	прямые отрезки, образующие ломаную между "непустыми" элементами буфера (при отсутствии пропусков аналогично DRAW_LINE)	1
DRAW_ARROW *IndReplica3.mq5*, *IndFractals.mq5*	символы (метки)	1
DRAW_HISTOGRAM *IndDeltaVolume.mq5*	гистограмма от нулевой линии до значений буфера	1
DRAW_HISTOGRAM2 *IndLabelHighLowClose.mq5*	гистограмма между значениями парных элементов двух индикаторных буферов	2
DRAW_ZIGZAG *IndFractalsZigZag.mq5*	прямые отрезки, образующие ломаную между последовательно встречающимися "непустыми" элементами двух буферов (похож на DRAW_SECTION, но в отличие от него допускает вертикальные отрезки на одном баре)	2
DRAW_FILLING	цветная заливка канала между двумя линиями по парным значениям в двух буферах	2
DRAW_BARS *IndSubChartSimple.mq5*	отображение в виде баров: четыре цены на бар указываются в четверке смежных буферов, в порядке OHLC	4
DRAW_CANDLES *IndSubChartSimple.mq5*	отображение в виде японских свечей: четыре цены на бар указываются в четверке смежных буферов, в порядке OHLC	4

В данной таблице приведены не все элементы ENUM_DRAW_TYPE. Существуют аналоги таких же построений с поддержкой расцвечивания отдельных элементов (баров). Мы представим их в отдельном разделе Поэлементное раскрашивание диаграмм. В документации по MQL5 приведены примеры для всех типов, а в рамках этой книги есть некоторые исключения: наличие демонстрационных индикаторов указано рядом с названиями типов.

Во всех случаях, включая DRAW_NONE, данные из буфера доступны в других программах через функцию *CopyBuffer*.

Дополнительная особенность типа DRAW_NONE заключается в том, что значения такого буфера не участвуют в автоматическом масштабировании графика, которое по умолчанию включено для индикаторов, выводимых в подокно.

Стиль линий в диаграмме определяется свойством PLOT_LINE_STYLE, для которого также имеется перечисление с допустимыми значениями ENUM_LINE_STYLE.

Идентификатор	Описание
STYLE_SOLID	сплошная линия
STYLE_DASH	прерывистая линия
STYLE_DOT	пунктирная линия
STYLE_DASHDOT	штрих-пунктирная линия
STYLE_DASHDOTDOT	штрих - две точки

Наконец, цвет линии задается свойством PLOT_LINE_COLOR. В простейшем случае данное свойство содержит один единственный цвет для всей диаграммы. Для некоторых типов диаграмм, в частности, DRAW_CANDLES, можно указать несколько цветов с помощью параметра-модификатора. Об этом мы поговорим позднее (см. пример *IndSubChartSimple.mq5* в разделе Мультивалютные и мультитаймфреймовые индикаторы).

Вышеназванных трех свойств достаточно для демонстрации отображения индикатора *IndReplica2.mq5*. Добавим два входных параметра *DrawType* и *LineStyle* типов ENUM_DRAW_TYPE и ENUM_LINE_STYLE соответственно, а затем вызовем функцию *PlotIndexSetInteger* несколько раз в *OnInit* для установки свойств отрисовки индикатора.

```
#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()
{
   // регистрируем массив в качестве индикаторного буфера
   SetIndexBuffer(0, buffer);

   // настраиваем свойства диаграммы под номером 0
   PlotIndexSetInteger(0, PLOT_DRAW_TYPE, DrawType);
   PlotIndexSetInteger(0, PLOT_LINE_STYLE, LineStyle);
   PlotIndexSetInteger(0, PLOT_LINE_COLOR, clrBlue);

   return INIT_SUCCEEDED;
}
```

Для свойства PLOT_LINE_COLOR мы не стали заводить входную переменную, поскольку это и некоторые другие свойства напрямую доступны из диалога свойств любого индикатора — на закладке *Цвета*. По умолчанию, то есть сразу после запуска индикатора цвет линии будет синим, но его, а также толщину линии и её стиль можно поменять в диалоге (на указанной закладке). Наш параметр *LineStyle* отчасти дублирует соответствующую ячейку *Стиль* в таблице

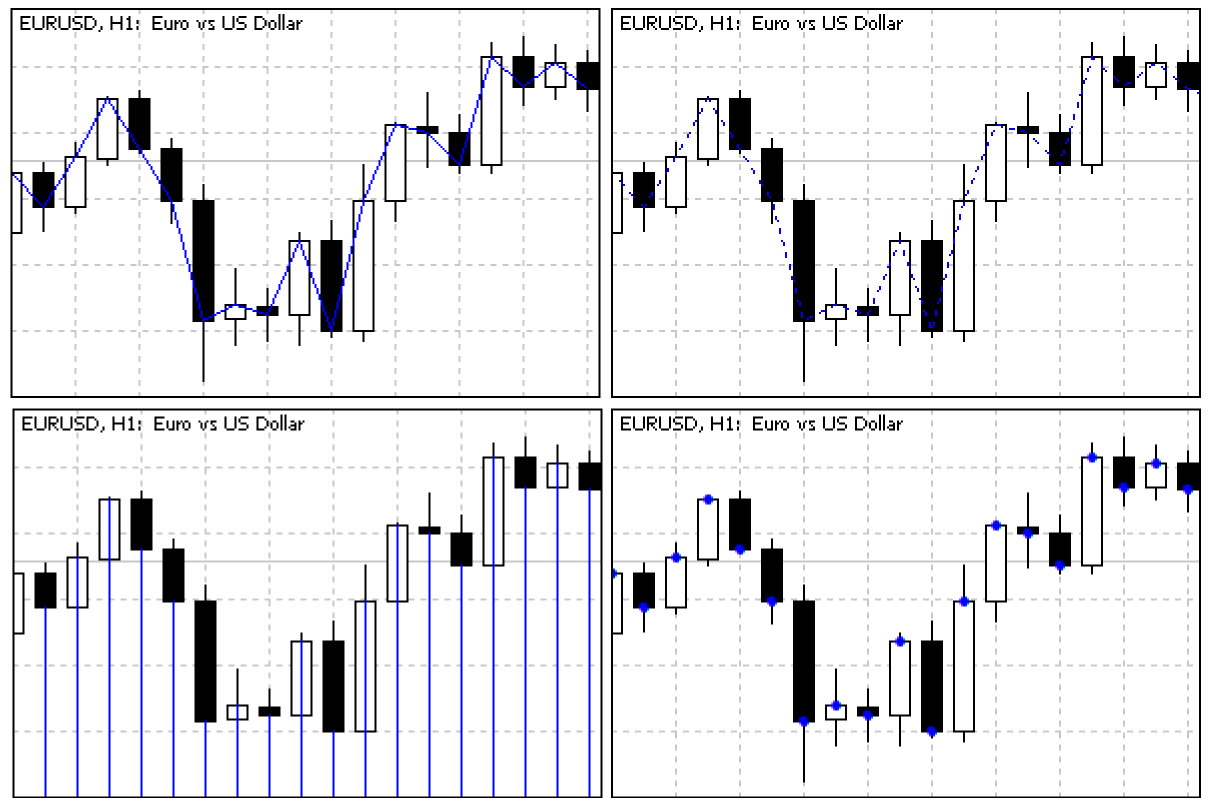
Цвета, но и дает некоторое преимущество. Дело в том, что стандартные "контролы" диалога не позволяют выбрать стиль, когда ширина линии больше 1. А с помощью входной переменной *LineStyle* мы можем получить, например, штрих-пунктирную линию при заданной ширине 3 пикселя.

Заполнение буфера данными в *OnCalculate* остается без изменений по сравнению с *IndReplica1.mq5*.

Откомпилировав и набросив индикатор на график, получим ожидаемую картину: линия синего цвета по ценам закрытия на графике и соответствующие цены закрытия баров в *Окне данных*.

Меняя входной параметр *DrawType*, мы можем наблюдать, как меняется способ отображения данных из буфера. При этом следует выбирать только типы, для которых требуется единственный буфер. Любой другой тип графического построения (DRAW_HISTOGRAM2, DRAW_ZIGZAG, DRAW_FILLING, DRAW_BARS, DRAW_CANDLES) просто не сможет работать на одном буфере и ничего не покажет. Также не имеет смысла выбирать типы построений с цветным раскрашиванием (начинаются со слова "Color"), так как для них требуется дополнительный буфер с номерами цветов на каждом баре (как уже было сказано, мы познакомимся с этой возможностью в разделе Поэлементное раскрашивание диаграмм).

Ниже показаны варианты отображения DRAW_LINE, DRAW_SECTION, DRAW_HISTOGRAM, DRAW_ARROW.



Типы диаграмм на одном буфере

Если бы не выбранные специально разные стили — STYLE_SOLID для DRAW_LINE и STYLE_DOT для DRAW_SECTION, эти типы отрисовки нельзя было бы отличить, потому что в нашем буфере все элементы имеют "непустые" значения. Напомним, что по умолчанию под "пустым" значением подразумевается специальная константа EMPTY_VALUE, которую мы не использовали. Секции (отрезки) в DRAW_SECTION проводятся, минуя "пустые" элементы, и это становится

заметно только при наличии таковых. Про установку "пустых" элементов мы поговорим в разделе Визуализация пропусков данных.

Гистограмма от нулевой линии DRAW_HISTOGRAM обычно применяется в индикаторах с собственным окном, здесь же она приведена для демонстрации. Мы создадим индикатор в подокне с данным типом отрисовки в разделе Ожидание данных и управление видимостью (см. пример *IndDeltaVolume.mq5*).

Для типа DRAW_ARROW система по умолчанию использует символ заполненного кружка (код 159), однако его можно заменить на другой с помощью вызова *PlotIndexSetInteger(индекс, PLOT_ARROW, код)*.

Коды и внешний вид символов шрифта Wingdings можно посмотреть в справке по MQL5.

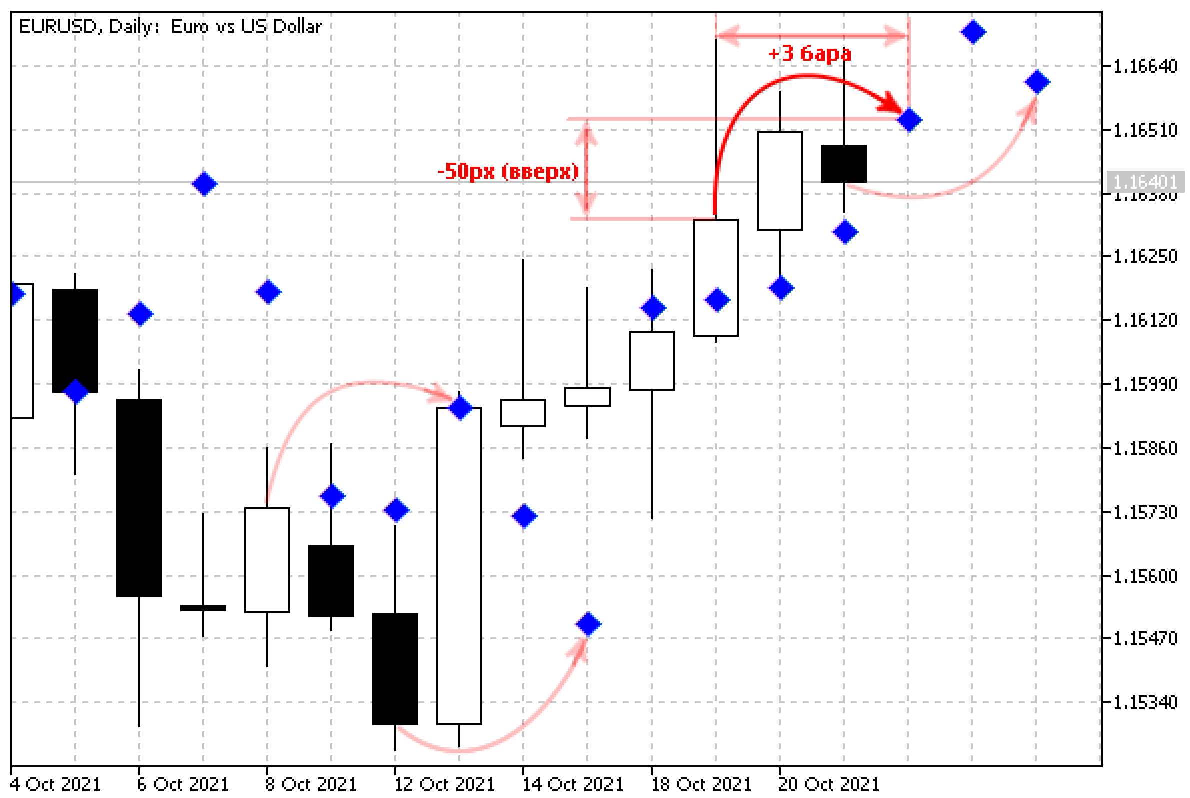
В еще одной модификации индикатора *IndReplica3.mq5* добавим входные параметры для выбора символа "стрелки" (*ArrowCode*), а также для сдвига этих меток на графике по вертикали (*ArrowPadding*) и горизонтали (*TimeShift*).

```
input uchar ArrowCode = 159;
input int ArrowPadding = 0;
input int TimeShift = 0;
```

Вертикальный сдвиг по шкале цен задается в пикселях (положительные значения означают сдвиг вниз, отрицательные — вверх). Горизонтальный по шкале времени — задается в барах (положительные значения — это сдвиг вправо, в будущее, а отрицательные — влево, в прошлое). Новые входные переменные передаются в вызовы *PlotIndexSetInteger* в *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);
   ...
}
```

На следующем скриншоте показан пример выполнения *IndReplica3.mq5* на графике с настройками 117 (ромб), -50 (50 точек вверх), 3 (3 бара вправо/вперед).



Точечная диаграмма со сдвигами меток по вертикали и горизонтали

Наш индикатор по умолчанию строится по типу цены *Close* (хотя пользователь и может поменять это в диалоге свойств, в выпадающем списке *Применить к*). При необходимости можно назначить другую начальную настройку с помощью директивы:

```
#property indicator_applied_price PRICE_TYPE
```

Здесь вместо PRICE_TYPE следует указать любую константу из перечисления ENUM_APPLIED_PRICE. В него же входит и PRICE_CLOSE, соответствующая умолчанию. Например, следующая директива, добавленная в исходный код приведет к тому, что индикатор будет по умолчанию строиться на "типичной" цене.

```
#property indicator_applied_price PRICE_TYPICAL
```

Еще раз отметим, что эта настройки задает только умолчание. Узнать актуальный тип цены, на котором строится индикатор, позволяет встроенная переменная *_AppliedTo*. Если же индикатор строится по дескриптору другого индикатора, то можно будет узнать лишь этот факт, но не название конкретного индикатора-поставщика данных.

Чтобы узнать в исходном коде текущее состояние свойств из перечисления ENUM_PLOT_PROPERTY_INTEGER используйте функцию *PlotIndexGetInteger*.

```
int PlotIndexGetInteger(int index, ENUM_PLOT_PROPERTY_INTEGER property)
int PlotIndexGetInteger(int index, ENUM_PLOT_PROPERTY_INTEGER property, int modifier)
```

Функция обычно применяется в паре с функцией *PlotIndexSetInteger* для копирования свойств рисования из одной линии в другую или для чтения свойств из кодов универсальных mqh-файлов, включаемых в исходный код различных индикаторов.

К сожалению, аналогичных функций *PlotIndexGetDouble* и *PlotIndexGetString* не предусмотрено.

5.4.7 Правила сопоставления буферов и диаграмм

При регистрации диаграмм с помощью *PlotIndexSetInteger(i, PLOT_DRAW_TYPE, type)* каждый вызов последовательно ставит в соответствие i-ой диаграмме некоторое количество буферов согласно их требуемому количеству для типа отрисовки *type* (см. таблицу ENUM_DRAW_TYPE в предыдущем разделе). Тем самым это количество буферов изымается из рассмотрения при привязке буферов к следующим диаграммам (при следующих вызовах *PlotIndexSetInteger*).

Например, если первым графическим построением (под индексом 0) идет DRAW_CANDLES, для которого необходимо 4 индикаторных буфера, то именно такое количество и будет с ним ассоциировано. Таким образом, буфера с индексами от 0 по 3 включительно получат привязку, и следующим свободным для привязки буфером станет буфер под индексом 4.

Если следом регистрируется простая линейная диаграмма DRAW_LINE (её индекс в последовательности диаграмм равен 1), она займет только 1 буфер — как раз под индексом 4.

Если далее настраивается диаграмма DRAW_ZIGZAG (следующий индекс среди диаграмм равен 2), то поскольку она использует два буфера, то к ней отойдут буфера с индексами 5 и 6.

Разумеется, количество буферов должно быть достаточным для всех регистрируемых построений. Вышеприведенный пример иллюстрируется следующей таблицей. В ней всего 7 буферов и 3 графических построения (диаграммы).

<table>
<tr><td>Индекс буфера в SetIndexBuffer</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr>
<tr><td>Индекс диаграммы в PlotIndexSetInteger</td><td colspan="4">0</td><td>1</td><td colspan="2">2</td></tr>
<tr><td>Тип отрисовки</td><td colspan="4">DRAW_CANDLES</td><td>DRAW _LINE</td><td colspan="2">DRAW_ZIGZAG</td></tr>
</table>

Индексация буферов и диаграмм независима в том смысле, что индекс буфера не обязан совпадать с индексом диаграммы. Вместе с тем по мере увеличения индексов диаграмм происходит увеличение индексов привязываемых к ним буферов, причем расхождение в индексации может становиться все больше и больше при наличии типов отрисовок, которые "забирают" под себя более одного буфера.

Хотя принято вызывать функции *SetIndexBuffer* до *PlotIndexSetInteger*, это не обязательно. Важным является лишь правильное соответствие индексов буферов и индексов диаграмм. В случае использования директив (см. следующий раздел), которые являются альтернативой вызовам *PlotIndexSetInteger*, директивы исполняются в любом случае раньше обработчика *OnInit*.

Для демонстрации различия в индексациях буферов и диаграмм рассмотрим простой пример *IndHighLowClose.mq5*. В нем станем рисовать размах каждой свечи между *High* и *Low* в виде гистограммы типа DRAW_HISTOGRAM2, а цену *Close* подчеркнем простой линией DRAW_LINE. Для доступа к таймсериям цен разных типов нам также потребуется сменить форму *OnCalculate* с упрощенной на полную.

Поскольку гистограмма требует 2 буфера, то вместе с еще одним буфером под линию *Close* следует описать три буфера.

```
#property indicator_chart_window
#property indicator_buffers 3
#property indicator_plots 2

double highs[];
double lows[];
double closes[];
```

Регистрируем их в *OnInit* в порядке очередности.

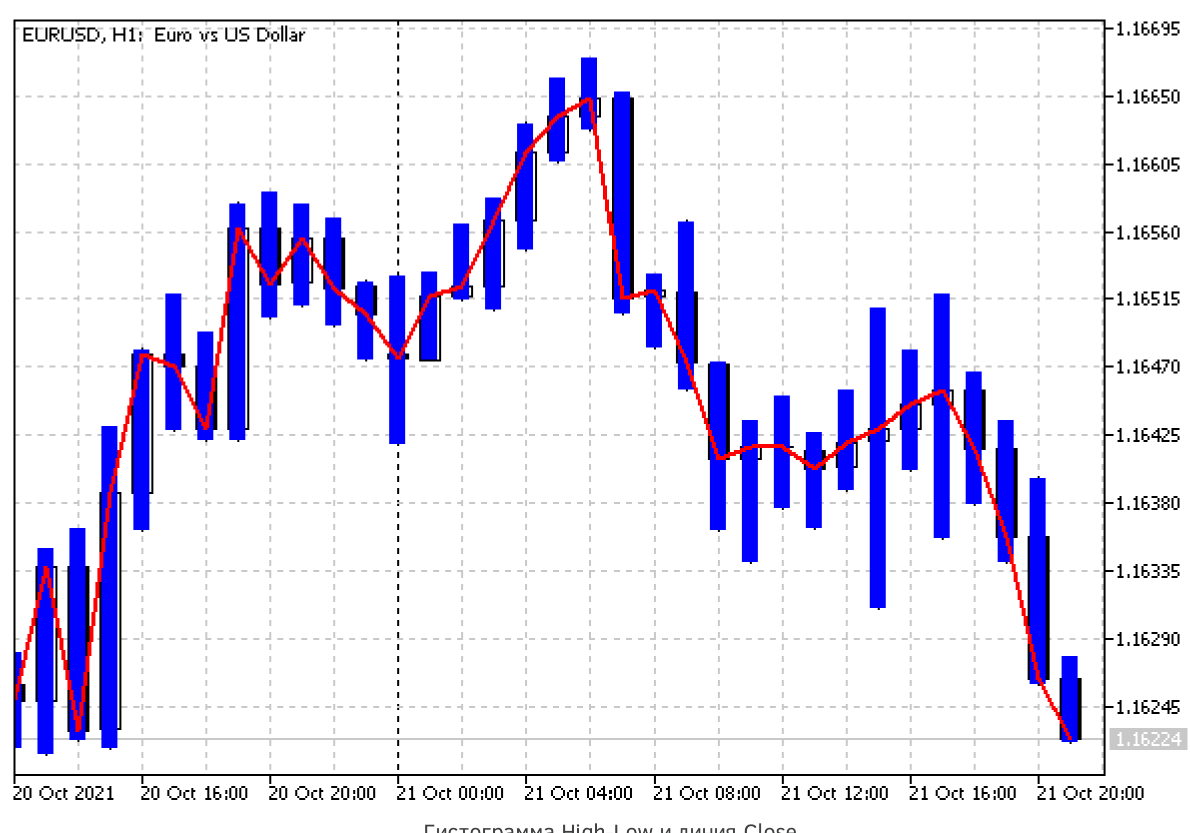
```
int OnInit()
{
   // массивы для буферов под 3 типа цены
   SetIndexBuffer(0, highs);
   SetIndexBuffer(1, lows);
   SetIndexBuffer(2, closes);

   // отрисовка гистограммы между High и Low свечи под индексом 0
   PlotIndexSetInteger(0, PLOT_DRAW_TYPE, DRAW_HISTOGRAM2);
   PlotIndexSetInteger(0, PLOT_LINE_WIDTH, 5);
   PlotIndexSetInteger(0, PLOT_LINE_COLOR, clrBlue);

   // отрисовка линии Close под индексом 1
   PlotIndexSetInteger(1, PLOT_DRAW_TYPE, DRAW_LINE);
   PlotIndexSetInteger(1, PLOT_LINE_WIDTH, 2);
   PlotIndexSetInteger(1, PLOT_LINE_COLOR, clrRed);

   return INIT_SUCCEEDED;
}
```

Попутно ширина гистограммы установлена равной 5-ти пикселям, а ширины линии — 2-м. Стили явным образом не назначаются, и равны по умолчанию STYLE_SOLID.

Теперь собственно функция *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[])
{
   // на каждом новом баре или множесте баров (включая первый расчет)
   if(prev_calculated != rates_total)
   {
      // заполняем все новые бары
      ArrayCopy(highs, high, prev_calculated, prev_calculated);
      ArrayCopy(lows, low, prev_calculated, prev_calculated);
      ArrayCopy(closes, close, prev_calculated, prev_calculated);
   }
   else // тики на текущем баре
   {
      // обновляем последний бар
      highs[rates_total - 1] = high[rates_total - 1];
      lows[rates_total - 1] = low[rates_total - 1];
      closes[rates_total - 1] = close[rates_total - 1];
   }
   // сообщаем количество обработанных баров для следующего вызова
   return rates_total;
}
```

Результат работы этого индикатора приведен на следующем изображении:



Гистограмма High-Low и линия Close

Обратите внимание на один важный момент. Диаграммы наносятся на график в порядке, соответстующем их индексам, в результате чего одни располагаются визуально выше других (перекрывают их). В данном случае сперва отрисовывается гистограмма с индексом 0, а затем поверх неё — линия с индексом 1. Иногда имеет смысл менять порядок регистрации диаграмм, чтобы обеспечить лучшую видимость более мелких графических построений, которые могут быть перекрыты более крупными (широкими).

Установка таких приоритетов по мнимой оси Z, уходящий вглубь экрана (перпендикулярно экрану) называется Z-порядком. Мы еще раз столкнемся с этой техникой при изучении графических объектов.

Напомним также, что по умолчанию индикаторы выводятся поверх графика цен, но это поведение можно изменить в настройках: диалог *Свойства графика*, закладка *Общие*, опция *График сверху*. Аналогичная опция есть и в программном интерфейсе (*ChartSetInteger(CHART_FOREGROUND)*, см. раздел Режимы отображения графика).

5.4.8 Применение директив для настройки графических построений

До сих пор мы настраивали графические построения с помощью вызовов функции *PlotIndexSetInteger*. MQL5 позволяет сделать то же самое с помощью директив препроцессора *#property*. Основная разница между этими двумя способами заключается в том, что директивы обрабатываются на стадии компиляции и описанные с помощью них свойства считываются из исполняемого файла в процессе загрузки, еще до того как выполнится обработчик *OnInit* (если он есть). То есть, директивы предоставляют некоторые значения по умолчанию, которые, в принципе, можно и не менять, если в этом нет необходимости.

С другой стороны, вызов функции *PlotIndexSetInteger* позволяет менять свойства на лету, в процессе выполнения программы. Динамическое изменение свойств с помощью функций

позволяет создавать более гибкие сценарии использования индикатора. Соответствие директив вызовам функции *PlotIndexSetInteger* приведено в следующей таблице.

Директивы	**Функция**	**Описание**
indicator_colorN	PlotIndexSetInteger(N-1, PLOT_LINE_COLOR, color)	Цвет линии для графического построения
indicator_styleN	PlotIndexSetInteger(N-1, PLOT_LINE_STYLE, type)	Стиль отрисовки из перечисления ENUM_LINE_STYLE
indicator_typeN	PlotIndexSetInteger(N-1, PLOT_DRAW_TYPE, type)	Тип отрисовки из перечисления ENUM_DRAW_TYPE
indicator_widthN	PlotIndexSetInteger(N-1, PLOT_LINE_WIDTH, width)	Толщина линии в пикселях (1 - 5)

Обратите внимание, что нумерация графических построений в директивах ведется с 1, в то время как в функциях — с 0. Например, директива *#property indicator_type1 DRAW_ZIGZAG* эквивалентна вызову *PlotIndexSetInteger(0, PLOT_DRAW_TYPE, DRAW_ZIGZAG)*.

Также стоит отметить, что с помощью функции можно задать гораздо больше свойств, чем через директивы: в перечислении ENUM_PLOT_PROPERTY_INTEGER — 10 элементов.

> Свойства, описанные директивами, доступны (видимы и могут редактироваться пользователем) в диалоге настройки параметров индикатора даже при первом его размещении на графике. В частности, сюда относятся толщина, цвет и стиль линий (закладка *Цвета*), количество и размещение уровней (закладка *Уровни*). Те же свойства, заданные функциями (и если они не имеют значений по умолчанию в директивах), появляются в диалоге только во второй и последующие разы.

Адаптируем индикатор *IndHighLowClose.mq5* под использование директив. Новая версия находится в файле *IndPropHighLowClose.mq5*. За счет директив обработчик *OnInit* упрощается, *OnCalculate* не меняется.

```
#property indicator_chart_window
#property indicator_buffers 3
#property indicator_plots 2

// настройки отрисовки гистограммы High-Low (индекс 0 меняем на 1 в директиве)
#property indicator_type1   DRAW_HISTOGRAM2
#property indicator_style1  STYLE_SOLID // по умолчанию, можно опустить
#property indicator_color1  clrBlue
#property indicator_width1  5

// настройки отрисовки линии Close (индекс 1 меняем на 2 в директиве)
#property indicator_type2   DRAW_LINE
#property indicator_style2  STYLE_SOLID // по умолчанию, можно опустить
#property indicator_color2  clrRed
#property indicator_width2  2

double highs[];
double lows[];
double closes[];

int OnInit()
{
   // массивы для буферов под 3 типа цены
   SetIndexBuffer(0, highs);
   SetIndexBuffer(1, lows);
   SetIndexBuffer(2, closes);

   return INIT_SUCCEEDED;
}
```

Внешне новый индикатор неотличим от старого.

5.4.9 Установка названий для графических построений

В предыдущих примерах данной главы индикаторные буфера обозначались в Окне данных названием самого индикатора. Это малоинформативно. MQL5 API предоставляет возможность задать для каждого буфера собственное название. Сделать это можно двумя уже известными нам способами: с помощью директивы *#property* и вызова специальной функции *PlotIndexSetString*.

bool PlotIndexSetString(int index, ENUM_PLOT_PROPERTY_STRING property, string value)

Прототип функции аналогичен *PlotIndexSetInteger* за исключением того, что тип свойств (параметр *value*) — *string*. Функция поддерживает только одно свойство PLOT_LABEL (это константа перечисления ENUM_PLOT_PROPERTY_STRING). Индекс настраиваемой диаграммы в параметре *index* должен быть от 0 до N-1, где N — общее количество диаграмм, указанных в *#property indicator_plots N*.

При использовании директивы к индексу диаграммы следует сделать поправку на 1. Напомним, что нумерация графических построений в директивах начинается с единицы, в то время как в параметрах функций — с нуля.

Директива	Функция	Описание
#property indicator_labelN	PlotIndexSetString(N-1, PLOT_LABEL, string)	Задает текстовую метку для отображения в *Окне данных* и во всплывающих подсказках

Для графических серий, требующих несколько индикаторных буферов (например, DRAW_CANDLES, DRAW_FILLING и других), имена меток задаются через разделитель ';'.

Метки также показываются во всплывающей подсказке при наведении курсора на диаграмму.

В примере *IndLabelHighLowClose.mq5* добавим две директивы по сравнению с *IndPropHighLowClose.mq5*.

```
#property indicator_label1  "High;Low"
#property indicator_label2  "Close"
```

Теперь смысл значений при отображении индикатора в *Окне данных* более понятен.

5.4.10 Визуализация пропусков данных (пустых элементов)

Во многих случаях показания индикаторов должны выводиться лишь на некоторых барах, оставляя остальные бары нетронутыми (визуально, без лишних линий или меток). Например, многие сигнальные индикаторы выводят стрелки вверх или вниз на тех барах, где появляется рекомендация к покупке или продаже. Но сигналы встречаются нечасто.

Для установки "пустого" значения, которое не выводится ни на графике, ни в *Окне данных*, применяется функция *PlotIndexSetDouble*.

bool PlotIndexSetDouble(int index, ENUM_PLOT_PROPERTY_DOUBLE property, double value)

Функция предназначена для установки у диаграммы под номером *index* свойств типа *double*. Набор таких свойств сведен в перечисление ENUM_PLOT_PROPERTY_DOUBLE, но на текущий момент в нем только один элемент: PLOT_EMPTY_VALUE. Он и задает "пустое" значение. Само значение передается в последнем параметре *value*.

В качестве подходящего примера индикатора с "редкими" значениями мы рассмотрим довольно известный детектор фракталов. Суть его работы в том, чтобы пометить на графике высокие цены (*High*), которые выше N соседних баров, и низкие цена (*Low*), которые ниже N соседних баров, в обе стороны. Файл индикатора называется *IndFractals.mq5*.

В индикаторе будет два буфера и два графических построения типа DRAW_ARROW.

```
#property indicator_chart_window
#property indicator_buffers 2
#property indicator_plots   2

// настройки отрисовки
#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"

// индикаторные буфера
double UpBuffer[];
double DownBuffer[];
```

Входная переменная *FractalOrder* позволит задавать количество соседних баров, по которым определяется верхний или нижний экстремум.

```
input int FractalOrder = 3;
```

Для символов-стрелок предусмотрим отступ в 10 пикселей от экстремумов для лучший видимости.

```
const int ArrowShift = 10;
```

В функции *OnInit* выполним традиционные действия по объявлению массивов буферами и их привязке к визуализируемым диаграммам.

```
int OnInit()
{
   // привязка буферов
   SetIndexBuffer(0, UpBuffer, INDICATOR_DATA);
   SetIndexBuffer(1, DownBuffer, INDICATOR_DATA);

   // коды символов-стрелок вверх и вниз
   PlotIndexSetInteger(0, PLOT_ARROW, 217);
   PlotIndexSetInteger(1, PLOT_ARROW, 218);

   // отступ для стрелок
   PlotIndexSetInteger(0, PLOT_ARROW_SHIFT, -ArrowShift);
   PlotIndexSetInteger(1, PLOT_ARROW_SHIFT, +ArrowShift);

   // установка "пустого" значения (можно опустить, т.к. EMPTY_VALUE по умолчанию)
   PlotIndexSetDouble(0, PLOT_EMPTY_VALUE, EMPTY_VALUE);
   PlotIndexSetDouble(1, PLOT_EMPTY_VALUE, EMPTY_VALUE);

   return FractalOrder > 0 ? INIT_SUCCEEDED : INIT_PARAMETERS_INCORRECT;
}
```

Обратите внимание, что по умолчанию "пустым" значением является специальная константа EMPTY_VALUE, поэтому вышеприведенные вызовы *PlotIndexSetDouble* необязательны.

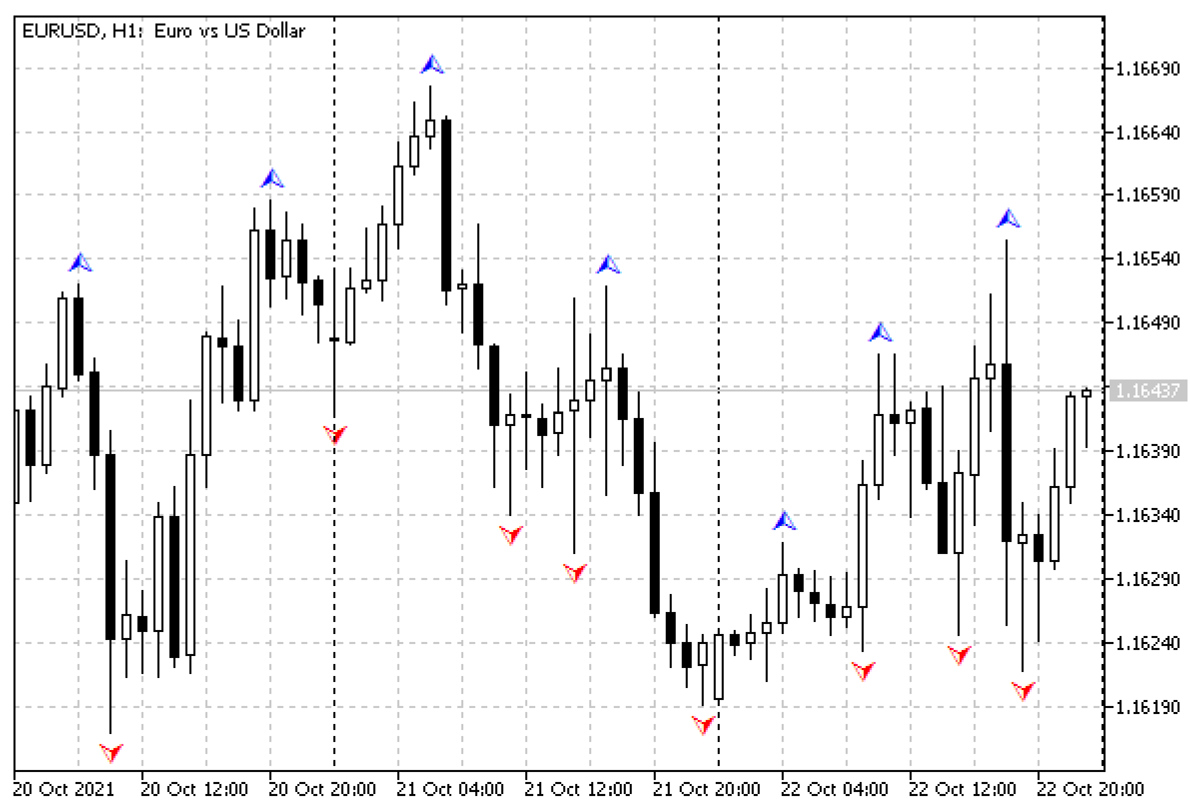
В обработчике *OnCalculate* в момент первого вызова инициализируем оба массива значением EMPTY_VALUE, и далее присваиваем его новым элементам по мере формирования баров. Заполнение необходимо, потому что распределенная под буфера память может содержать произвольные данные (мусор).

```
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)
   {
      // на старте заполняем массивы целиком
      ArrayInitialize(UpBuffer, EMPTY_VALUE);
      ArrayInitialize(DownBuffer, EMPTY_VALUE);
   }
   else
   {
      // на новых барах также чистим элементы
      for(int i = prev_calculated; i < rates_total; ++i)
      {
         UpBuffer[i] = EMPTY_VALUE;
         DownBuffer[i] = EMPTY_VALUE;
      }
   }
   ...
```

В основном цикле по барам сравниваем цены *high* и *low* с этими же типами цен на соседних барах и оставляем метки там, где обнаруживается экстремум среди *FractalOrder* баров в каждую из сторон.

```
   // просматриваем все или новые бары, имеющие в окружении FractalOrder баров
   for(int i = fmax(prev_calculated - FractalOrder - 1, FractalOrder);
       i < rates_total - FractalOrder; ++i)
   {
      // проверяем, что верхняя цена выше соседних баров
      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;
         }
      }

      // проверяем, что нижняя цена ниже соседних баров
      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;
}
```

Посмотрим, как этот индикатор выглядит на графике.



Индикатор фракталов

Теперь заменим тип отрисовки с DRAW_ARROW на DRAW_ZIGZAG и сравним эффект "пустых" значений для обоих вариантов. Очевидно, что в результате должен получиться зигзаг на фракталах. Модифицированная версия индикатора прилагается в файле *IndFractalsZigZag.mq5*.

Одно из главных изменений касается количества диаграмм: она теперь одна, так как DRAW_ZIGZAG "потребляет" оба буфера.

```
#property indicator_chart_window
#property indicator_buffers 2
#property indicator_plots   1

// настройки отрисовки
#property indicator_type1   DRAW_ZIGZAG
#property indicator_color1  clrMediumOrchid
#property indicator_width1  2
#property indicator_label1  "ZigZag Up;ZigZag Down"
...
```

Из *OnInit* уходят все вызовы функций, связанных с настройкой стрелок.

```
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;
}
```

В остальном исходный код без изменений.

На следующем изображении приведен график, на который зигзаг наложен в дополнение к фракталам: тем самым можно наглядно сравнить их результаты. Оба индикатора работают совершенно независимо, но из-за одинакового алгоритма найденные экстремумы совпадают.

Индикатор Зиг-Заг по фракталам

Важно отметить, что в случае, если подряд встречаются экстремумы одного типа, зигзаг использует первый из них. Это следствие того, что в качестве экстремумов используются фракталы. В стандартном зигзаге такого, разумеется, быть не может. При необходимости желающие могут усовершенствовать алгоритм, предварительно проредив последовательности фракталов.

Также следует отметить, что для отрисовки DRAW_ZIGZAG (так же, как и DRAW_SECTION), видимые отрезки соединяют "непустые" элементы и потому, строго говоря, какой-то фрагмент отрезка рисуется на каждом баре, включая и те, что имеют значение EMPTY_VALUE (или другое назначенное вместо него). Однако в *Окне данных* четко видно, что "пустые" элементы действительно пусты: для них значения не выводятся.

5.4.11 Индикаторы с собственным подокном: размер и уровни

До сих пор мы ограничивались индикаторами, работающими в основном окне графика, то есть они имели директиву *#property indicator_chart_window*. Настало время изучить индикаторы, размещаемые в собственном подконе, под графиком котировок. Напомним, что их следует описывать с директивой *#property indicator_separate_window*.

Все, что мы изучили ранее, применимо и к индикатором в подокне, включая описание и привязку буферов, настройку типов и стилей рисования, применения как полной, так и сокращенной формы *OnCalculate*, на выбор. Однако они имеют и некоторые особенности, дополнительные настройки.

Поскольку подокно имеет собственную шкалу значений, MQL5 позволяет задать для него максимальную и минимальную величины (пользователи могут задавать подобные ограничения в диалоге настройки индикатора, на закладке *Шкала*). Программно это делается с помощью функции *IndicatorSetDouble* со следующим прототипом.

```
bool IndicatorSetDouble(ENUM_CUSTOMIND_PROPERTY_DOUBLE property, double value)
bool IndicatorSetDouble(ENUM_CUSTOMIND_PROPERTY_DOUBLE property, int modifier,
  double value)
```

Функция задает для индикатора значение свойства типа *double*. Две формы потребовались потому, что некоторые свойства могут существовать во множественном числе, в частности горизонтальные уровни (о них речь пойдет чуть ниже). Доступные свойства собраны в перечислении ENUM_CUSTOMIND_PROPERTY_DOUBLE.

Идентификатор	Описание
INDICATOR_MINIMUM	Минимум по вертикальной оси
INDICATOR_MAXIMUM	Максимум по вертикальной оси
INDICATOR_LEVELVALUE	Значение горизонтального уровня (номер задается в параметре modifier)

Фиксированный диапазон шкалы применяется во многих осцилляторных индикаторах, например, WPR. Далее мы покажем с ним пример, охватывающий все функции (свойства) из данного раздела.

В случае успешного выполнения функция возвращает *true*, в противном случае *false*.

Кроме управления шкалой индикаторы в подокне могут иметь, как мы уже поняли, горизонтальные уровни. Чтобы задать их количество и атрибуты применяется другая функция *IndicatorSetInteger*. Пользователь может выполнить аналогичные действия в диалоге настройки индикатора на закладке *Уровни*.

```
bool IndicatorSetInteger(ENUM_CUSTOMIND_PROPERTY_INTEGER property, int value)
bool IndicatorSetInteger(ENUM_CUSTOMIND_PROPERTY_INTEGER property, int modifier,
  int value)
```

Функция также имеет две формы и позволяет задать для индикатора значение свойства типа *int* или эквивалентного ему (например, *color* или перечисление). Доступные свойства собраны в перечислении ENUM_CUSTOMIND_PROPERTY_INTEGER. Помимо свойств, связанных с уровнями,

в нем присутствует свойство INDICATOR_DIGITS, являющее общим для индикаторов любых типов: его мы рассмотрим в следующем разделе.

Идентификатор	Описание
INDICATOR_DIGITS	Точность отображения значений индикатора (знаки после десятичной точки)
INDICATOR_HEIGHT	Фиксированная высота собственного окна индикатора в пикселях (команда препроцессора *#property indicator_height*)
INDICATOR_LEVELS	Количество горизонтальных уровней в окне индикатора
INDICATOR_LEVELCOLOR	Цвет линии уровня (имеет тип *color*, параметр *modifier* задает номер уровня)
INDICATOR_LEVELSTYLE	Стиль линии уровня (имеет тип ENUM_LINE_STYLE, параметр *modifier* задает номер уровня)
INDICATOR_LEVELWIDTH	Толщина линии уровня (1-5) (параметр *modifier* задает номер уровня)

Уровни могут иметь текстовые метки. Для их назначения следует использовать функцию *IndicatorSetString*.

```
bool IndicatorSetString(ENUM_CUSTOMIND_PROPERTY_STRING property, string value)
bool IndicatorSetString(ENUM_CUSTOMIND_PROPERTY_STRING property, int modifier,
  string value)
```

Перечисление ENUM_CUSTOMIND_PROPERTY_STRING содержит список строковых свойств индикаторов. Здесь особо следует отметить другое свойство, не относящееся к уровням, — INDICATOR_SHORTNAME: оно также является общим для всех индикаторов и будет рассмотрено в следующем разделе.

Идентификатор	Описание
INDICATOR_SHORTNAME	Публичный заголовок индикатора
INDICATOR_LEVELTEXT	Описание уровня (номер указывается в modifier)

Все упомянутые функции для числовых типов *int* и *double* дублируются специальными директивами (ниже приведена сводная таблица).

Директивы для свойств уровней	Функции-аналоги	Тип свойства	Описание
indicator_levelN	IndicatorSetDouble(INDICATOR_LEVELVALUE, N-1, value)	double	Значение для горизонтального уровня номер N на вертикальной оси
indicator_levelcolor	IndicatorSetInteger(INDICATOR_LEVELCOLOR, N-1, color)	color	Цвет горизонтальных уровней (разный цвет по номеру можно задать только с помощью функции)
indicator_levelwidth	IndicatorSetInteger(INDICATOR_LEVELWIDTH, N-1, width)	int	Толщина линий горизонтальных уровней в пикселях (разную толщину по номеру можно задать только с помощью функции)
indicator_levelstyle	IndicatorSetInteger(INDICATOR_LEVELSTYLE, N-1, style)	ENUM _LINE _STYLE	Стили линий горизонтальных уровней (разные стили по номеру можно задать только с помощью функции)
indicator_minimum	IndicatorSetDouble(INDICATOR_MINIMUM, minimum)	double	Фиксированное минимальное значение, нижнее ограничение шкалы по вертикальной оси
indicator_maximum	IndicatorSetDouble(INDICATOR_MAXIMUM, maximum)	double	Фиксированное максимальное значение, верхнее ограничение шкалы по вертикальной оси

Обратите внимание, что нумерация экземпляров свойств (модификаторов) при использовании директив *#property* начинается с 1 (единицы), в то время как функции используют нумерацию с 0 (нуля).

Внимательный читатель заметит, что для некоторых свойств не существует директив. К их числу относятся INDICATOR_LEVELTEXT, INDICATOR_SHORTNAME, INDICATOR_DIGITS. Предполагается, что данные свойства должны заполняться динамически из MQL-кода, в зависимости от входных переменных и графика, на котором размещен индикатор. INDICATOR_LEVELS задается опосредованно, за счет указания нескольких директив для уровней.

Наконец, отличительной чертой индикаторов в подокне является то, что программа может "заморозить" вертикальный размер своего окна.

Директива для размера подокна	Функция-аналог	Описание
indicator_height	IndicatorSetInteger(INDICATOR_HEIGHT, height)	Фиксированная высота подокна индикатора в пикселях (пользователь не сможет изменить высоту)

Фиксированная высота подокна обычно применяется только для управляющих панелей с элементами управления (кнопками, флагами, полями ввода), реализованными с помощью графических объектов.

Для функций установки свойств, к сожалению, не предусмотрено обратных (*IndicatorGetInteger*, *IndicatorGetDouble*, *IndicatorGetString*). Среди прочего это не позволяет, например, узнать количество и значения горизонтальных уровней, если их сменил пользователь.

В качестве примера работы с фиксированной шкалой и уровнями рассмотрим индикатор *IndWPR.mq5*. В нем воспользуемся стандартным алгоритмом WPR: на заданном количестве прошедших баров (период WPR) найдем максимумы H и минимумы L цены (то есть её размах). Затем вычислим отношение разницы текущей цены C и минимума L, C - L (или разницы -(H - C), со знаком минус) ко всему размаху, и приведем всё в диапазон от 0 до -100. Вот каноническая формула расчета WPR:

```
R% = (-(H - C) / (H - L)) * 100
```

В начале исходного кода добавим несколько директив. Помимо свойства расположения индикатора в собственном окне установим шкалу значений от 0 до -100.

```
#property indicator_separate_window
#property indicator_maximum    0.0
#property indicator_minimum    -100.0
```

Для хранения значений и отображения достаточного одного буфера и одной линейной диаграммы.

```
#property indicator_buffers    1
#property indicator_plots      1
#property indicator_type1      DRAW_LINE
#property indicator_color1     clrDodgerBlue
```

В индикаторе WPR принято выделять два уровня: -20 и -80, как границы областей перекупленности и перепроданности, соответственно. Создадим для них пару горизонтальных линий.

```
#property indicator_level1     -20.0
#property indicator_level2     -80.0
#property indicator_levelstyle STYLE_DOT
#property indicator_levelcolor clrSilver
#property indicator_levelwidth 1
```

Единственная входная переменная позволяет задать период расчета WPR.

```
input int WPRPeriod = 14; // Period
```

Массив для буфера описывается на глобальном уровне и регистрируется в *OnInit*.

```
double WPRBuffer[];

void OnInit()
{
   // проверка на корректность ввода
   if(WPRPeriod < 1)
   {
      Alert(StringFormat("Incorrect Period value (%d). Should be 1 or larger",
         WPRPeriod));
   }

   // привязка массива как буфера
   SetIndexBuffer(0, WPRBuffer);
}
```

Обработчик *OnInit* описан с типом *void*, что неявным образом подразумевает успешную инициализацию. Вместе с тем, если задан период меньше 1, это не позволит делать расчет, в связи с чем выдается предупреждение пользователю.

Для упрощения заголовка функции *OnCalculate* для индикаторов был подготовлен заголовочный файл *IndCommon.mqh* с двумя макросами, описывающими стандартные списки параметров обеих форм обработчика события.

```
#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[]
```

Теперь мы можем применять в этом и других индикаторах сжатое определение *OnCalculate* (при условии, что нас устраивают предложенные названия параметров в макросах).

```
#include <MQL5Book/IndCommon.mqh>

int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   if(rates_total < WPRPeriod || WPRPeriod < 1) return 0;
   ...
   return rates_total;
}
```

В начале *OnCalculate* делается проверка на возможность расчета при текущих величинах *WPRPeriod* и *rates_total*. Если данных недостаточно или период слишком маленький, возвращаем 0, из-за чего окно индикатора останется пустым.

Далее заполняем первые несколько баров, для которых невозможно посчитать WPR заданного периода, пустым значением.

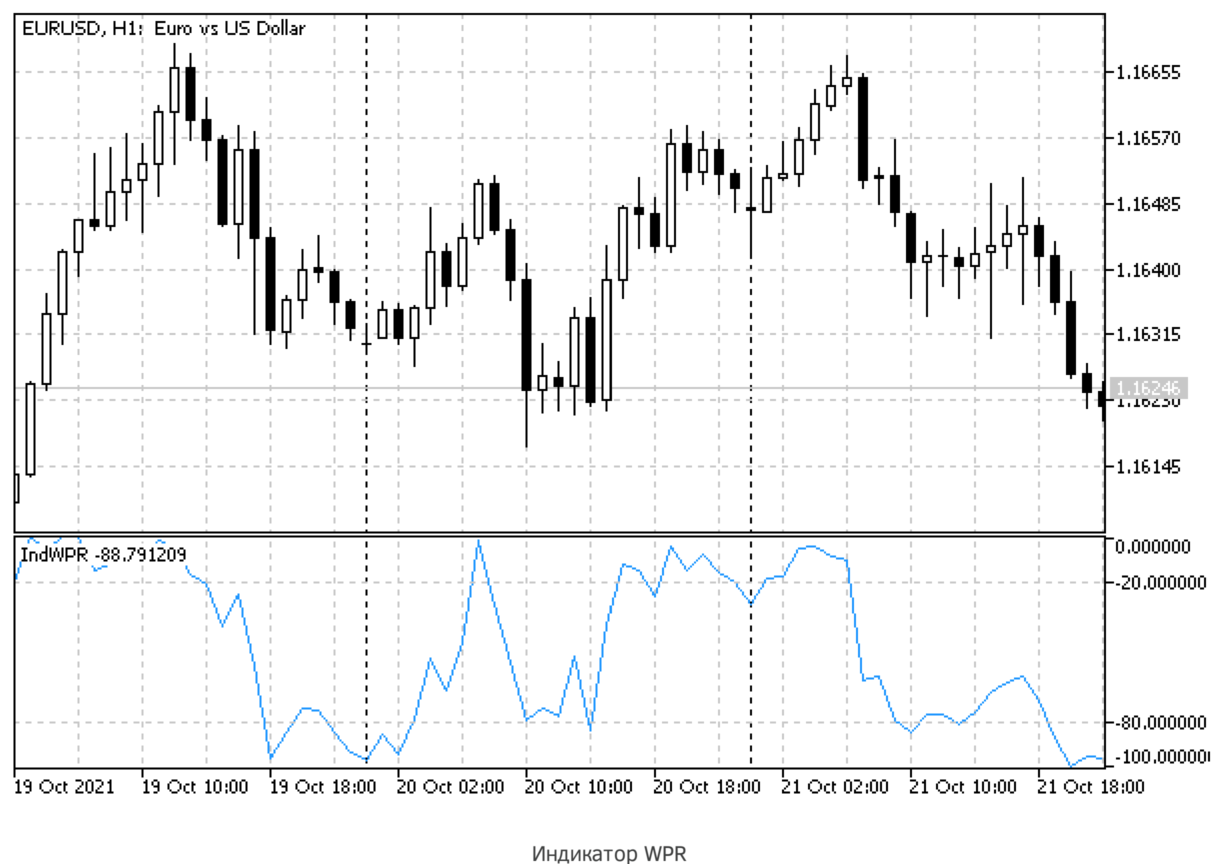
```
int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   ...
   if(prev_calculated == 0)
   {
      ArrayFill(WPRBuffer, 0, WPRPeriod - 1, EMPTY_VALUE);
   }
   ...
}
```

Наконец, запускаем рабочий цикл с расчетом по формуле WPR и складываем результаты в буфер. Обратите внимание, что последний бар обновляется на каждом тике: это достигается за счет того, что цикл начинается с *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;
}
```

Для поиска индексов максимальных *high* и минимальных *low* применяются функции *ArrayMaximum* и *ArrayMinimum*.

Посмотрим, как индикатор выглядит в собственном окне.



Индикатор WPR

В следующих разделах продолжим улучшать данный индикатор, постепенно задействуя другие общеупотребительные свойства.

5.4.12 Общие свойства индикаторов: заголовок и точность значений

Для всех индикаторов поддерживается пара важных свойств, не связанных с расчетами, но улучшающих опыт пользователя. Их корректная установка в обработчике *OnInit* стала частью стандарта разработки индикаторов.

Целочисленное свойство INDICATOR_DIGITS задается с помощью рассмотренной ранее функции *IndicatorSetInteger* и влияет на точность представления вещественных чисел на графике и в *Окне данных*. По умолчанию терминал выводит 6 разрядов после десятичной точки. Если показания индикатора связаны с ценой текущего инструмента, то имеет смысл установить данное свойство равным точности представления цен: *IndicatorSetInteger(INDICATOR_DIGITS, _Digits)*.

В случае WPR значения представляют собой аналог процентов, и потому имеет смысл ограничить выводимые значения двумя знаками после запятой.

```
IndicatorSetInteger(INDICATOR_DIGITS, 2);
```

Вторым общеупотребительным свойством является строковое INDICATOR_SHORTNAME — для него используется функция *IndicatorSetString*. Это заголовок индикатора, выводимый во всплывающих подсказках, а также в левом верхнем углу подокна, если индикатор имеет собственное окно. Когда он не задан явным образом, используется название файла индикатора. В частности, на скриншоте в предыдущем разделе мы видим заголовок "IndWPR".

В заголовке индикатора принято отображать главные входные переменные и режимы работы (если их несколько).

Например, для WPR, как правило, включают в заголовок выбранный пользователем период.

Кроме того, заголовок позволяет сократить название. Это важно, потому что длина названия ограничена 63-мя символами.

Для обновленной версии WPR будем использовать такую настройку:

```
IndicatorSetString(INDICATOR_SHORTNAME, "%R" + "(" + (string)WPRPeriod + ")");
```

Результаты этих усовершенствований проверим уже в следующем разделе, после того как раскрасим зоны перекупленности и перепроданности в разные цвета (см. пример *IndColorWPR.mq5*).

5.4.13 Поэлементное раскрашивание диаграмм

Кроме стандартных типов отрисовки, перечисленных ранее в ENUM_DRAW_TYPE, платформа также предоставляет их варианты с возможностью индивидуальной раскраски значений на каждом бара. Для этих целей используется дополнительный индикаторный буфер, в котором хранятся номера цветов. Номера ссылаются на элементы в специальном массиве, содержащем определенный программистом набор цветов. Максимальное количество цветов — 64.

В следующей таблице приведены элементы ENUM_DRAW_TYPE с поддержкой цвета и количество буферов, требуемых для их отрисовки, включая 1 буфер с индексами цветов.

Тип визуализации	Описание	Количество буферов
DRAW_COLOR_LINE	Разноцветная линия	1+1
DRAW_COLOR_SECTION	Разноцветные отрезки	1+1
DRAW_COLOR_ARROW	Разноцветные стрелки	1+1
DRAW_COLOR_HISTOGRAM	Разноцветная гистограмма от нулевой линии	1+1
DRAW_COLOR_HISTOGRAM2	Разноцветная гистограмма между попарными значениями двух индикаторных буферов	2+1
DRAW_COLOR_ZIGZAG	Разноцветный ZigZag	2+1
DRAW_COLOR_BARS	Разноцветные бары	4+1
DRAW_COLOR_CANDLES	Разноцветные свечи	4+1

В ходе привязки буферов к диаграммам следует учитывать, что дополнительный "цветовой" буфер должен быть указан в первом параметре *SetIndexBuffer* под номером, идущим непосредственно после буферов с данными. Например, для раскрашиваемой линии, где используется 1 буфер с данными и "цветовой" буфер, данные идут под номером 0, а их цвета под номером 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);
   ...
}
```

Начальный набор цветов в палитре для диаграммы N можно задать директивой *#property indicator_colorN*. В ней через запятую указываются необходимые цвета как именованные константы или литералы цвета. Например, следующая запись в индикаторе выберет 6 стандартных цветов для раскраски 0-ой диаграммы (в директивах нумерация начинается с 1):

```
#property indicator_color1   clrRed,clrBlue,clrGreen,clrYellow,clrMagenta,clrCyan
```

Далее в программе следует указывать не сам цвет, которым будет отображаться графическое построение, а только его индекс. Нумерация в палитре ведется как в обычном массиве — начиная с 0. Так, если для i-го бара нужно задать зеленый цвет, то достаточно установить в цветовом буфере индекс зеленого цвета из палитры, то есть 2 в данном случае.

```
   ColorLineColors[i] = 2; // ссылка на элемент с цветом clrGreen
```

Набор цветов для раскрашивания не является раз и навсегда заданным, его можно менять динамически с помощью функции *PlotIndexSetInteger(index, PLOT_LINE_COLOR, color)*.

Например, чтобы в вышеприведенной палитре заменить цвет *clrGreen* на *clrGray* следует выполнить вызов:

```
   PlotIndexSetInteger(0, PLOT_LINE_COLOR, clrGray);
```

Применим цветовую раскраску в нашем индикаторе WPR. Новый файл — *IndColorWPR.mq5*. Изменения касаются следующих сфер.

Количество буферов увеличено на 1. Вместо одного цвета указано три.

```
#property indicator_buffers    2
#property indicator_plots      1
#property indicator_type1      DRAW_COLOR_LINE
#property indicator_color1     clrDodgerBlue,clrGreen,clrRed
```

Добавлен новый массив под буфер цветов и его регистрация в *OnInit*.

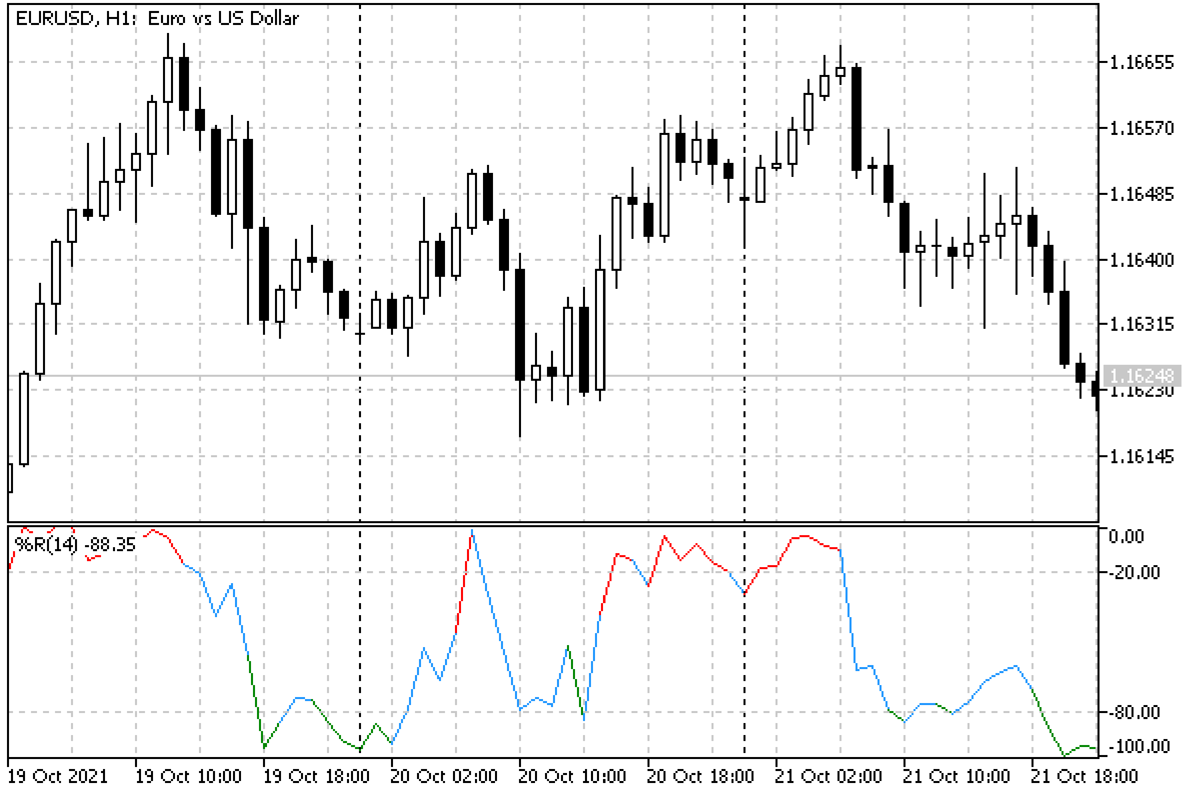
```
double WPRColors[];

void OnInit()
{
   ...
   SetIndexBuffer(1, WPRColors, INDICATOR_COLOR_INDEX);
   ...
```

Если не задать тип буфера INDICATOR_COLOR_INDEX (т.е. с вызовом *SetIndexBuffer(1, WPRColors)* он трактовался бы по умолчанию как INDICATOR_DATA), он станет видимым в *Окне данных*.

В функции *OnCalculate* внутри рабочего цикла добавим раскраску на основе анализа значения i-го бара. По умолчанию берется цвет с 0-м индексом, то есть прежний *clrDodgerBlue*. Если показания индикатора уходят в верхнюю зону, они подсвечиваются цветом номер 2 (*clrRed*), а если в нижнюю зону — цветом номер 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;
}
```

Вот как это выглядит на экране.



Индикатор WPR с цветными зонами перекупленности и перепроданности

Обратите внимание, что фрагмент линии раскрашивается в альтернативный цвет, если его конечная точка (бар) вышла в верхнюю или нижнюю зону. При этом предыдущий отсчет может находиться внутри центральной зоны, из-за чего может складываться впечатление, что подсветка ошибочна. Однако это корректное поведение, соответствующее текущей реализации и особенности применения цвета платформой.

Цвет отрезка линейной диаграммы DRAW_COLOR_LINE между двумя соседними барами определяется цветом правого (более свежего) бара.

Если требуется подсвечивать цветом только фрагменты, у которых оба смежных бара находятся в одной зоне, модифицируйте код на следующий:

```
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;
```

Также напомним, что мы добавили в исходный код установку заголовка и точности представления величин (2 знака). Сравнение нового изображения с прежним позволит заметить эти визуальные отличия. В частности, заголовок теперь имеет вид "%R(14)", а шакала значений по вертикали — намного компактнее.

Последний аспект, который мы изменим в индикаторе *IndColorWPR.mq5* — это пропуск отрисовки на начальных барах.

5.4.14 Пропуск отрисовки на начальных барах

Во многих случаях по условиям алгоритма расчет значений индикатора невозможно начать с первого (самого левого доступного) бара, поскольку требуется обеспечить минимальное заданное количество предыдущих баров в истории. Например, многие виды сглаживания подразумевают, что берется массив цен за предыдущие N баров, и на основании этих значений рассчитывается значение индикатора для текущего бара.

В таких случаях либо нет возможности рассчитать значения индикатора на самых первых барах, либо эти значения не предназначены для отображения на графике и являются только вспомогательными для расчета последующих значений.

Чтобы отказаться от визуализации индикатора на первых N-1 барах истории, следует установить свойству PLOT_DRAW_BEGIN значение N для соответствующего графического построения index: *PlotIndexSetInteger(index, PLOT_DRAW_BEGIN, N)*. По умолчанию данное свойство равно 0, то есть данные выводятся с самого начала.

В принципе, мы можем и сами подавить отображение линии на нужных барах, установив их в "пустое" значение (EMPTY_VALUE по умолчанию). Однако вызов функции *PlotIndexSetInteger* со свойством PLOT_DRAW_BEGIN делает и кое-что другое. Как разработчики своего индикатора, мы тем самым сообщаем внешним программам количество несущественных первых значений в нашем индикаторном буфере. В частности, другие индикаторы, которые потенциально могут быть построены на основе таймсерии нашего индикатора, получат значение свойства PLOT_DRAW_BEGIN в параметре *begin* своего обработчика *OnCalculate*. Таким образом, у них появится возможность пропустить "неполноценные" бары.

В примере индикатора *IndColorWPR.mq5* добавим подобную настройку в функции *OnInit*.

```
input int WPRPeriod = 14; // Period

void OnInit()
{
   ...
   PlotIndexSetInteger(0, PLOT_DRAW_BEGIN, WPRPeriod - 1);
   ...
```

```
}
```

Теперь в функции *OnCalculate* можно было бы убрать принудительную очистку первых баров, так как они всегда будут скрыты.

```
   if(prev_calculated == 0)
   {
      ArrayFill(WPRBuffer, 0, WPRPeriod - 1, EMPTY_VALUE);
   }
```

Но это правильно сработает, только когда пользователь вручную выбрал наш индикатор в качестве источника таймсерии для другого индикатора. Если же какой-то программист решит использовать наш индикатор в своих разработках, то там механизм получения данных другой (мы поговорим о нем в следующей главе), и он не позволит узнать свойство PLOT_DRAW_BEGIN. Поэтому явную инициализацию буфера лучше оставить.

Чтобы продемонстрировать, как данное свойство может эксплуатироваться в другом индикаторе, рассчитываемом на базе нашего, подготовим еще один индикатор. Пусть это будет известный алгоритм тройного экспоненциального сглаживания (Triple Exponential Moving Average), "упакованный" в индикатор *IndTripleEMA.mq5*. Когда он будет готов, его будет легко применить как к ценовым таймсериям, так и к произвольным индикаторам, таким как предыдущий индикатор *IndColorWPR.mq5*.

Попутно мы познакомимся с технической возможностью описывать вспомогательные буфера для расчетов (INDICATOR_CALCULATIONS).

Напомним формулу тройного EMA в виде нескольких вычислительных шагов.

Простое экспоненциальное сглаживание периода P для исходной таймсерии T выражается следующим образом:

```
K = 2.0 / (P + 1)
A[i] = T[i] * K + A[i - 1] * (1 - K)
```

где K — весовой коэффициент учета элементов исходного ряда, рассчитываемый через заданный период P; (1 - K) — коэффициент инерционности, применяемый к элементам сглаженного ряда A. Для получения i-го элемента ряда A суммируем K-ую часть i-го элемента исходного ряда T[i] и (1 - K)-ую часть предыдущего элемента A[i - 1].

Если обозначить сглаживание по указанным формулам как оператор E, то тройное EMA включает, как и следует из названия, троекратное применение E, после чего 3 получившихся сглаженных ряда особым образом комбинируются.

```
EMA1 = E(A, P), по всем i
EMA2 = E(EMA1, P), по всем i
EMA3 = E(EMA2, P), по всем i
TEMA = 3 * EMA1 - 3 * EMA2 + EMA3, по всем i
```

Тройное EMA обеспечивает меньшее отставание от исходного ряда по сравнению с обычным EMA того же периода. Вместе с тем, она характеризуется большей отзывчивостью, которая может спровоцировать неровности на результирующей линии и давать ложные сигналы.

> Сглаживание по EMA позволяет получить приблизительную оценку среднего, уже начиная со второго элемента ряда, причем для этого не требуется менять алгоритм. Это выгодно отличает EMA от других методов сглаживания, которые требуют наличия P предыдущих элементов или менять алгоритм для начальных отсчетов, когда доступно менее P элементов.

> Некоторые разработчики предпочитают объявлять первые P-1 элементов сглаженного ряда недействительными даже при использовании EMA. Однако следует заметить, что влияние прошлых элементов ряда в формуле EMA не ограничено P элементами, и оно становится пренебрежимо малым лишь при стремлении количества элементов к бесконечности (в других известных алгоритмах MA влияние оказывают точно P предыдущих элементов).
>
> В рамках данной книги, для исследования влияния пропуска начальных данных, мы не станем подавлять вывод начальных значений EMA.

Для расчета трех уровней EMA нам потребуются вспомогательные буферы и еще один для финального ряда: он отобразится линейной диаграммой.

```
#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);
   ...
}
```

Входная переменная *InpPeriodEMA* позволяет задать период сглаживания. Вторая переменная *InpHandleBegin* представляет собой переключатель режимов, с помощью которого мы сможем исследовать, как индикатор реагирует на учет или игнорирование параметра *begin* в обработчике *OnCalculate*. Доступные режимы сведены в перечисление BEGIN_POLICY и означают, по порядку:

- строгое выполнение отступа согласно *begin*;
- собственный анализ исходных данных на корректность, без учета *begin*;
- полное отсутствие какой-либо обработки, то есть игнорирование *begin* и прямолинейный расчет по всем данным.

```
enum BEGIN_POLICY
{
   STRICT, // strict
   CUSTOM, // custom
   NONE,   // no
};

input int InpPeriodEMA = 14;                  // EMA period:
input BEGIN_POLICY InpHandleBegin = STRICT;   // Handle 'begin' parameter:
```

Второй режим CUSTOM основан на предварительном сравнении каждого исходного элемента со значением EMPTY_VALUE и замене его на подходящее для алгоритма значение. Это будет корректно работать только с теми индикаторами, которые честно инициализируют неиспользованное начало буферов, не оставляя там мусор. Наш индикатор *IndColorWPR* заполняет буфер как требуется, и потому с ним можно ожидать практически идентичных результатов для режимов STRICT и CUSTOM.

Исходя из *InpPeriodEMA*, подготавливается константа K для расчета EMA.

```
const double K = 2.0 / (InpPeriodEMA + 1);
```

Сама функция EMA достаточно проста (здесь опущен защитный фрагмент для варианта CUSTOM с проверками на 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);
   }
}
```

А вот и полный расчет тройного сглаживания в *OnCalculate*.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
   const int _begin = InpHandleBegin == STRICT ? begin : 0;
   // свежий старт или обновление истории
   if(prev_calculated == 0)
   {
      Print("begin=", begin, " ", EnumToString(InpHandleBegin));

      // мы можем менять настройки диаграмм динамически
      PlotIndexSetInteger(0, PLOT_DRAW_BEGIN, _begin);

      // подготовка массивов
      ArrayInitialize(Ema, EMPTY_VALUE);
      ArrayInitialize(EmaOfEma, EMPTY_VALUE);
      ArrayInitialize(EmaOfEmaOfEma, EMPTY_VALUE);
      ArrayInitialize(TemaBuffer, EMPTY_VALUE);
      Ema[_begin] = EmaOfEma[_begin] = EmaOfEmaOfEma[_begin] = price[_begin];
   }

   // главный цикл с учетом старта от _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) // защита от пустых элементов в начале
      {
         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;
}
```

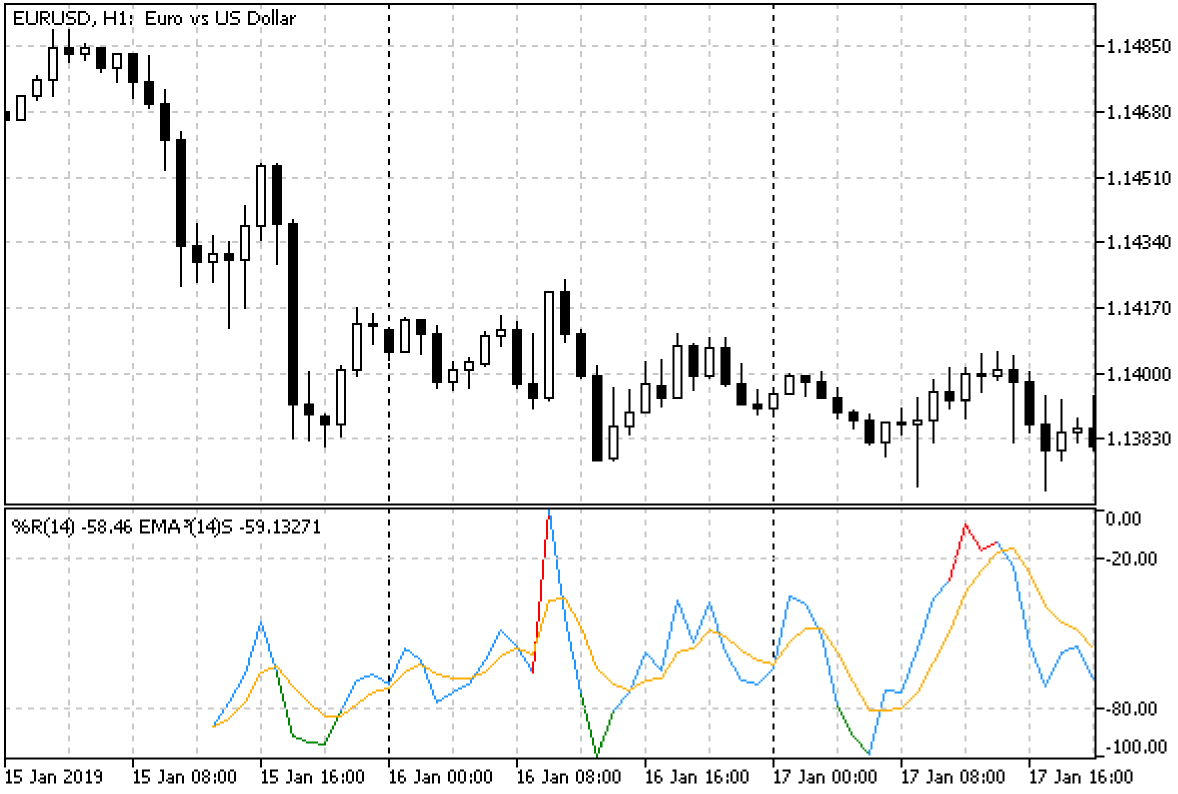
В момент первого запуска или обновления истории в журнал будет записано полученное значение параметра *begin*, а также выбранный пользователем режим его обработки.

После успешной компиляции всё готово для экспериментов.

Первым делом набросим на график индикатор *IndColorWPR* (по умолчанию его период равен 14, что означает, согласно исходным кодам, установку свойства PLOT_DRAW_BEGIN в значение на 1 меньше, т.к. индексация начинается с 0, и 13-й бар будет первым, для которого появится значение). Затем перетащим индикатор *IndTripleEMA* в подокно, в котором выводится WPR. В

открывшемся диалоге настройки свойств следует на закладке *Параметры* выбрать в выпадающем списке *Применить к* вариант *Данные предыдущего индикатора*. На закладке *Входные параметры* оставим значения по умолчанию.

На следующем изображении показано начало графика. В логе появится запись: *begin=13 STRICT*.



Индикатор тройного EMA, примененный к WPR с учетом начала данных

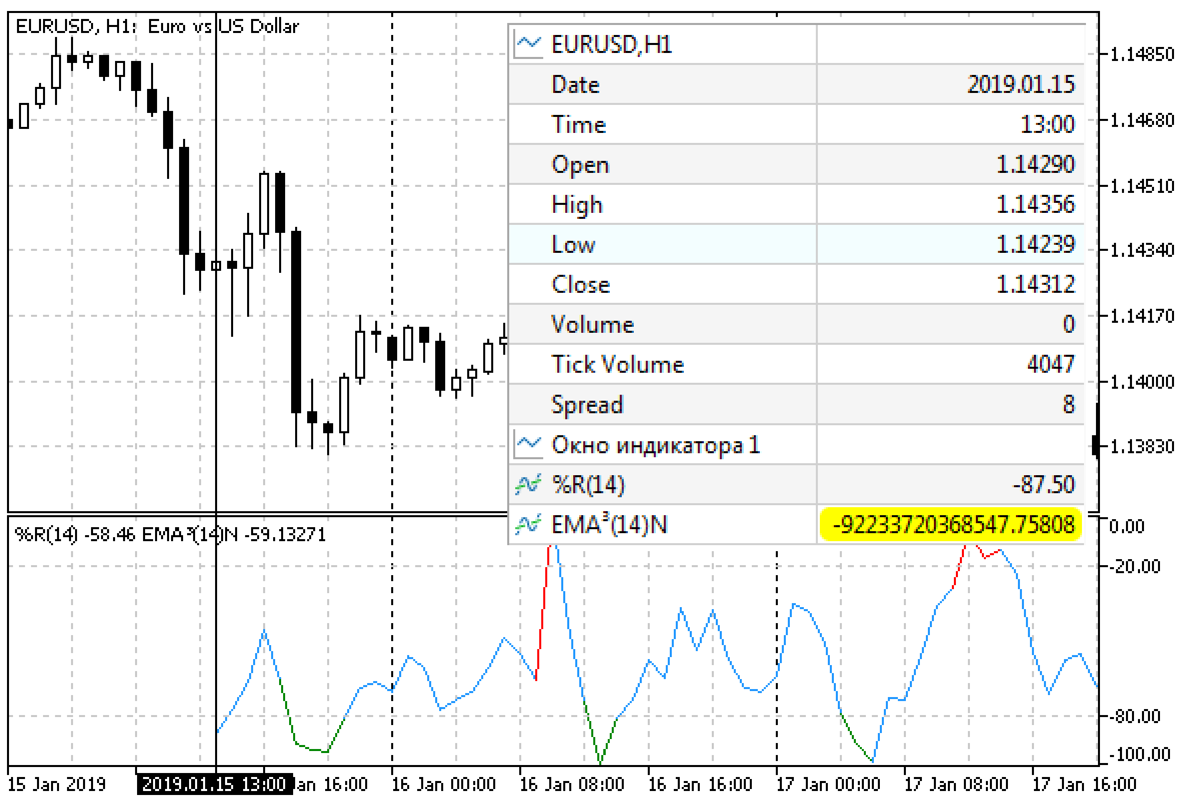
Обратите внимание, что усредненная линия начинается с отступом от начала, синхронно с WPR.

> Внимание! Количество доступных баров для расчета индикаторов *rates_total* (оно же *iBars(_Symbol, _Period)*) может превышать максимально разрешенное количество баров на графике из настроек терминала, если имеется более длинная локальная история котировок. В этом случае пустые элементы в начале линии WPR (или любого другого индикатора с пропуском первых элементов, такого как MA) станут невидны — будут скрыты за левой границей графика. Для воспроизведения ситуации с отсутствием линий на начальных барах потребуется либо увеличить количество баров на графике, либо закрыть терминал и удалить локальную историю для конкретного символа.

Теперь переключим в настройках индикатора *IndTripleEMA* режим на CUSTOM (результат в журнале — *begin=0 CUSTOM*). Ощутимых изменений в показаниях индикатора быть не должно.

Наконец активируем режим NONE. В журнал будет выведено: *begin=0 NONE*.

Здесь ситуация на графике станет выглядеть странной — линия фактически пропадет. В Окне данных можно увидеть, что значения элементов очень большие.



Индикатор тройного EMA, примененный к WPR без учета начала данных

Дело в том, что значения EMPTY_VALUE равны максимальному вещественному числу DBL_MAX. Поэтому без учета параметра *begin* расчеты с такими значениями генерируют тоже очень большие числа. В зависимости от специфики расчета из-за переполнения могут получаться специальные "не числа" (Not A Number, см. раздел Проверка вещественных чисел на нормальность): одно из них *-nan(ind)* подсвечено в изображении (*Окно данных* уже умеет выводить некоторые виды "не чисел", например, "inf" и "-inf", однако это пока не касается "-nan(ind)"). Как мы знаем, такие NaN-значения опасны, поскольку вычисления с их участием будут далее также давать NaN. В случае, если обойдется без NaN, по мере продвижения по барам вправо "переходный процесс" в расчете больших чисел затихает (из-за понижающего коэффициента (1 - K) в формуле EMA), и результат стабилизируется, становится адекватным. Если прокрутить график к настоящему времени, мы увидим нормальную тройную EMA.

Учет параметра *begin* — полезная практика, но она не дает гарантии, что поставщик данных (если это сторонний индикатор) корректно заполнил это свойство. Поэтому желательно предусмотреть в своем коде некоторую защиту. В данной реализации *IndTripleEMA* она реализована на начальном уровне.

Между прочим, если накладывать индикатор *IndTripleEMA* на график котировок, всегда будем получать *begin = 0*, потому что ценовые таймсерии с самого начала заполнены реальными данными, даже на самых старых барах.

5.4.15 Ожидание данных и управление видимостью (DRAW_NONE)

В предыдущей главе, в разделе Работа с массивами реальных тиков в структурах *MqlTick* был представлен скрипт *SeriesTicksDeltaVolume.mq5*, который позволяет рассчитывать дельту объемов на каждом баре. Тогда мы выводили результаты в журнал, однако намного более удобным и

логичным способом анализа такой технической информации является индикатор. В этом разделе мы такой создадим — *IndDeltaVolume.mq5*.

Здесь нам придется столкнуться с двумя факторами, которые часто встречаются при разработке индикаторов, но отсутствовали в предыдущих примерах.

Первый из них заключается в том, что тиковые данные не относятся к стандартным ценовым таймсериям, которые терминал передает в индикатор в параметрах *OnCalculate*. Значит, индикатор должен сам их запрашивать и ждать готовности, прежде чем появится возможность что-то отобразить в окне.

Второй фактор связан с тем, что значения объемов покупок и продаж, как правило, намного больше, чем их дельта, и при выводе в одном окне различить последнюю будет сложно. Однако именно дельта является показательной величиной, которую принято анализировать совместно с движением цены. Например, можно выделить 4 наиболее очевидных сочетания конфигураций баров и дельт объемов:

- Бычий бар и положительная дельта — подтверждение восходящего тренда;
- Медвежий бар и отрицательная дельта — подтверждение нисходящего тренда;
- Бычий бар и отрицательная дельта — возможен разворот вниз;
- Медвежий бар и положительная дельта — возможен разворот вверх.

Чтобы увидеть гистограмму дельт потребуется предусмотреть режим отключения "больших" гистограмм (покупок и продаж), для чего мы воспользуемся типом DRAW_NONE. Он отключает прорисовку конкретной диаграммы и её участие в подборе автоматического масштаба для окна (но оставляет буфер в *Окне данных*). Таким образом, убрав "большие" построения из рассмотрения, мы добьемся укрупнения автомасштаба под оставшуюся диаграмму дельт. Другой способ сокрытия буферов с помощью пометки их вспомогательными (режим INDICATOR_CALCULATIONS) будет рассмотрен в следующем разделе.

Напомним, что суть дельты объемов заключается в раздельном подсчете объемов покупок и продаж в тиках, после чего можно найти разницу между этими объемами. Соответственно, получается 3 таймсерии с объемами покупок, объемами продаж и сами разницы между ними. Поскольку эта информация не укладывается в шкалу цен, индикатор должен отображаться в собственном окне, а способом отображения трех таймсерий выберем гистограммы от нуля (DRAW_HISTOGRAM).

Согласно этому опишем свойства индикатора в директивах: размещение, количество буферов и диаграмм, а также их типы.

```
#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"
```

Из прежнего скрипта перенесем входные переменные. В частности, так как тики представляют собой довольно массивные данные, ограничим количество баров для расчета на истории (*BarCount*). Кроме того, в зависимости от наличия или отсутствия реальных объемов в тиках конкретного финансового инструмента, мы умеем считать дельту двумя разными способами, для чего сохранен параметр *TickType* (перечисление COPY_TICKS определено в заголовочном файле *TickEnum.mqh*, который мы уже использовали в скрипте).

```
#include <MQL5Book/TickEnum.mqh>

input int BarCount = 100;
input COPY_TICKS TickType = INFO_TICKS;
input bool ShowBuySell = true;
```

В обработчике *OnInit* переключим режим работы первых двух гистограмм между DRAW_HISTOGRAM и DRAW_NONE, в зависимости от выбора пользователем параметра *ShowBuySell* (*true* по умолчанию означает показ всех трех гистограмм). Обратите внимание, что динамическая настройка с помощью *PlotIndexSetInteger* переписывает статические настройки (в данном случае, лишь некоторые из них), внедренные в исполняемый файл с помощью директив *#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;
}
```

В данный момент может возникнуть вопрос: а где же регистрация индикаторных буферов? Мы вернемся к нему через пару абзацев, а пока создадим заготовку функции *OnCalculate*.

```
int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   if(prev_calculated == 0)
   {
      // TODO(1): инициализация, заполнение нулями
   }

   // на каждом новом баре или множестве новых баров при первом запуске
   if(prev_calculated != rates_total)
   {
      // обработать все или новые бары
      for(int i = fmax(prev_calculated, fmax(1, rates_total - BarCount));
         i < rates_total && !IsStopped(); ++i)
      {
         // TODO(2): пробуем получить данные и рассчитать i-й бар,
         // если не получится - нужно что-то предпринять!
      }
   }
   else // тики на текущем баре
   {
      // TODO(3): обновить текущий бар
   }

   return rates_total;
}
```

Основная техническая проблема находится в блоке, помеченном TODO(2). Напомним, что алгоритм получения тиков, который использовался в скрипте и будет с минимальными изменениями перенесен в индикатор, запрашивает их функцией *CopyTicksRange*. Такой вызов возвращает имеющиеся в базе тиков данные, но если их для заданного исторического бара еще нет, запрос вызывает скачивание и синхронизацию тиковых данных в асинхронном режиме (в фоне), а вызывающий код получает 0 тиков. В связи с этим, получив такой "пустой" ответ, индикатор должен прервать вычисления с признаком неуспеха (но не ошибки) и через некоторое время перезапросить тики. В нормальной ситуации открытого рынка к нам регулярно поступают тики, так что функция *OnCalculate* наверняка должна быть скоро вызвана и пересчитана с обновленной базой тиков. Но что делать в выходные, когда тиков нет?

Для корректной обработки такой ситуации MQL5 предоставляет таймер. Мы изучим его в одной из следующих глав, а пока воспользуемся как "черным ящиком". Специальная функция *EventSetTimer* позволяет "попросить" ядро вызвать нашу MQL-программу через заданное количество секунд. Точкой входа для такого вызова является зарезервированный обработчик *OnTimer* — мы видели его в общей таблице в разделе Обзор функций обработки событий. Таким образом, при возникновении задержки в получении тиковых данных следует запустить таймер с помощью *EventSetTimer* (достаточно минимального периода 1 секунда) и вернуть из *OnCalculate* ноль.

```
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): пробуем получить данные и рассчитать i-й бар,
         if(/*если данных нет*/)
         {
            Print("No data on bar ", i, ", at ", TimeToString(time[i]),
               ". Setting up timer for refresh...");
            EventSetTimer(1); // просим вызвать нас через 1 секунду
            return 0; // ничего не показываем в окне пока
         }
      }
      ...
}
```

В обработчике *OnTimer* используем функцию *EventKillTimer*, чтобы остановить таймер (если этого не сделать, система продолжит вызывать наш обработчик каждую секунду). Но помимо этого нам нужно каким-то образом запустить пересчет индикатора. Для этой цели применим другую функцию, которую нам еще предстоит освоить в главе про графики — *ChartSetSymbolPeriod* (см. раздел Переключение символа и таймфрейма). Она позволяет установить для графика с заданным идентификатором (0 означает текущий график) новое сочетание символа и таймфрейма. Однако если их не менять, передав, соответственно, *_Symbol* и *_Period* (см. раздел Предопределенные переменные), то график просто будет обновлен (индикаторы при этом пересчитываются).

```
void OnTimer()
{
   EventKillTimer();
   ChartSetSymbolPeriod(0, _Symbol, _Period); // самообновление графика
}
```

Единственный момент, на который здесь стоит дополнительно обратить внимание, заключается в том, что на открытом рынке событие таймера и самообновление графика может оказаться лишним, если следующий тик случится до вызова *OnTimer*. Поэтому мы заведем глобальную переменную (*calcDone*) для переключения признака готовности расчетов. Вначале *OnCalculate* будем её сбрасывать в *false*, а при штатном завершении расчета — взводить в *true*.

```
bool calcDone = false;

int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   calcDone = false;
   ...
         if(/*если данных нет*/)
         {
            ...
            return 0; // выходим с calcDone = false
         }
   ...
   calcDone = true;
   return rates_total;
}
```

Тогда в *OnTimer* можно инициировать автообновление графика только при *calcDone* равном *false*.

```
void OnTimer()
{
   EventKillTimer();
   if(!calcDone)
   {
      ChartSetSymbolPeriod(0, _Symbol, _Period);
   }
}
```

Теперь обратимся к вопросу, чем заменить комментарии *TODO(1,2,3)*, вместо которых должен проводиться, собственно, расчет и заполняться индикаторные буфера. Объединим все эти операции в одном классе *CalcDeltaVolume*. Таким образом, под каждое действие будет выделен отдельный метод, и мы сохраним обработчик *OnCalculate* таким же простым (вместо комментариев появятся вызовы методов).

В классе предусмотрим переменные-члены, которые примут настройки пользователи по количеству обрабатываемых баров истории и методу расчета дельты, а также три массива под индикаторные буфера. Их инициализацию проведем в конструкторе.

```
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)
   {
      // регистрируем внутренние массивы как буфера индикатора
      SetIndexBuffer(0, buy);
      SetIndexBuffer(1, sell);
      SetIndexBuffer(2, delta);
   }
```

Мы можем назначать массивы-члены в качестве буферов, поскольку собираемся затем создать глобальный объект данного класса. В принципе, для корректного отображения данных важно только, чтобы массивы, привязанные к диаграммам, существовали на момент отрисовки. Допустимо менять привязку буферов динамически (см. пример *IndSubChartSimple.mq5* в следующем разделе).

Важно отметить, что индикаторные буфера должны иметь тип *double*, однако объемы имеют тип *ulong*, в связи с чем для очень больших значений (например, на очень крупных таймфреймах) могут гипотетически возникать потери точности.

Для инициализации буферов создан метод *reset*. Большая часть элементов массивов заполняется "пустым" значением EMPTY_VALUE, а последние *limit* баров нулем, потому что в них мы будем суммировать объемы покупок и продаж, раздельно.

```
   void reset()
   {
      // заполняем массив buy, а остальные скопируем из него
      // "пустое" значение во всех элементах кроме последних limit баров с 0
      ArrayInitialize(buy, EMPTY_VALUE);
      ArrayFill(buy, ArraySize(buy) - limit, limit, 0);

      // дублируем начальное состояние в другие массивы
      ArrayCopy(sell, buy);
      ArrayCopy(delta, buy);
   }
```

Расчет на i-м историческом баре выполняет метод *createDeltaBar*. На его вход подается номер бара и ссылка на массив с временными метками баров (его мы получаем в виде параметра *OnCalculate*). i-е элементы массивов инициализируются нулем.

```
int createDeltaBar(const int i, const datetime &time[])
{
   delta[i] = buy[i] = sell[i] = 0;
   ...
```

Затем следует найти временные границы i-го бара: *prev* и *next*, причем *next* отсчитывается вправо от *prev* путем добавления значения новой для нас функции *PeriodSeconds*. Она возвращает количество секунд в текущем таймфрейме. Прибавляя это количество, мы находим теоретическое начало следующего бара. На истории, когда *i* не равно номеру последнего бара, можно было бы заменить нахождение следующей метки времени на *time[i + 1]*. Однако индикатор должен работать и на последнем, еще находящемся в процессе формирования баре, у которого нет следующего бара. Поэтому в общем случае использовать *time[i + 1]* нельзя.

```
   ...
   const datetime prev = time[i];
   const datetime next = prev + PeriodSeconds();
```

Когда мы делали аналогичный расчет в скрипте, то обошлись без функции *PeriodSeconds*, потому что не считали последний, текущий бар и могли позволить себе находить *next* и *prev* как *iTime(WorkSymbol, TimeFrame, i)* и *iTime(WorkSymbol, TimeFrame, i + 1)* соответственно.

Далее в методе *createDeltaBar* делаем запрос тиков в пределах найденных временных меток (от правой вычитаем 1 миллисекунду, чтобы не "залезть" на следующий бар). Тики поступают в массив *ticks*, обработка которого поручена вспомогательному методу *calc* — в него практически без изменений перенесен алгоритм скрипта. Выделить его в отдельный метод нас вынудило то обстоятельство, что расчет будет выполняться в двух разных ситуациях: по историческим барам (вспоминаем комментарий *TODO(2)*) и по тикам на текущем баре (комментарий *TODO(3)*). Вторую ситуацию рассмотрим чуть ниже.

```
   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;
}
```

В случае успешного запроса метод возвращает количество обработанных тиков, а в случае ошибки — код ошибки со знаком минус. Обратите внимание, что если тиков для бара пока в базе нет (что не является ошибкой, строго говоря, но не позволяет продолжать визуальную работу индикатора), метод вернет 0 (знак у 0 не меняет его значения). Поэтому в функции *OnCalculate* потребуется проверять результат метода на "меньше или равно" 0.

Метод *calc* практически состоит из рабочих строк скрипта *SeriesTicksDeltaVolume.mq5*, поэтому не станем его здесь приводить. Желающие могут освежить память, заглянув в *IndDeltaVolume.mq5*.

Для расчета дельты на постоянно обновляющемся последнем баре нам потребуется фиксировать с точностью до миллисекунды метку времени последнего обработанного тика. Тогда при следующем вызове *OnCalculate* мы сможем запросить все тики, после этой метки.

Важно напомнить об отсутствии гарантии, что система успеет вызвать наш обработчик *OnCalculate* на каждом тике в реальном режиме времени. Если мы выполняем тяжелые расчеты, или какая-то другая MQL-проограмма нагрузила терминал расчетами, или тики приходят очень быстро (например, при выходе важных новостей), события могут не попадать в очередь индикатора (в очереди хранится не более одного события каждого типа, в том числе не более одного уведомления о тике). Поэтому, если программа хочет получить все тики, она должна запрашивать их с помощью *CopyTicksRange* или *CopyTicks*.

Однако одной только временной метки последнего обработанного тика недостаточно. Дело в том, что тики могут иметь одинаковое время даже с учетом миллисекунд. Поэтому мы не имеем права прибавить к метке 1 миллисекунду, чтобы исключить "старый" тик: ведь после него могут идти "новые" тики с той же меткой.

В связи с этим следует запоминать не только метку, но и количество последних тиков с этой меткой. Тогда при следующем запросе тиков мы сможем запросить тики, начиная с запомненного времени (то есть, включая "старые" тики), но пропустить именно столько из них, сколько уже было обработано в прошлый раз.

Для реализации этого алгоритма в классе описаны две переменные *lasttime* и *lastcount*.

```
ulong lasttime; // миллисекундная метка последнего обработанного онлайн тика
int lastcount;  // количество тиков с такой меткой в тот момент
```

Из полученного от системы массива тиков найдем значения для этих переменных с помощью вспомогательного метода *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;
   }
}
```

Теперь мы можем уточнить метод *createDeltaBar*: при обработке последнего бара вызовем *updateLastTime* в первый раз.

```
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) // последний бар
      {
         updateLastTime(n, ticks);
      }
      calc(i, ticks);
   }
   ...
}
```

Имея актуальные значения *lasttime* и *lastcount*, мы можем реализовать метод для подсчета дельт на текущем баре в режиме онлайн.

```
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;
}
```

В методе *calc* для такого режима внесен дополнительный опциональный параметр *skip*, с помощью которого можно пропустить расчет на заданном количестве "старых" тиков.

```
void calc(const int i, const MqlTick &ticks[], const int skip = 0)
{
   const int n = ArraySize(ticks);
   for(int j = skip; j < n; ++j)
   ...
}
```

Класс для расчета готов, и нам осталось только вставить вызовы трех публичных методов в *OnCalculate*.

```
int OnCalculate(ON_CALCULATE_STD_FULL_PARAM_LIST)
{
   if(prev_calculated == 0)
   {
      deltas.reset(); // инициализация, заполнение нулями
   }
   
   calcDone = false;
   
   // на каждом новом баре или множестве новых баров при первом запуске
   if(prev_calculated != rates_total)
   {
      // обработать все или новые бары
      for(int i = fmax(prev_calculated, fmax(1, rates_total - BarCount));
         i < rates_total && !IsStopped(); ++i)
      {
         // пробуем получить данные и рассчитать i-й бар,
         if((deltas.createDeltaBar(i, time)) <= 0)
         {
            Print("No data on bar ", i, ", at ", TimeToString(time[i]),
               ". Setting up timer for refresh...");
            EventSetTimer(1); // просим вызвать нас через 1 секунду
            return 0; // ничего не показываем в окне пока
         }
      }
   }
   else // тики на текущем баре
   {
      if((deltas.updateLastDelta(rates_total)) <= 0)
      {
         return 0; // ошибка
      }
   }
   
   calcDone = true;
   return rates_total;
}
```

Откомпилируем и запустим индикатор. Для начала желательно выбрать таймфрейм не старше H1 и оставить количество баров в *BarCount* равным по умолчанию 100. После некоторого ожидания построения индикатора результат должен выглядеть примерно так:

Индикатор дельт объемов со всеми гистограммами, включая покупки и продажи

Теперь сравним с тем, что будет при установке параметра *ShowBuySell* в *false*.

Индикатор объемов с одной гистограммой дельт (раздельные покупки и продажи скрыты)

Итак, в данном индикаторе мы организовали ожидание загрузки тиковых данных рабочего инструмента с помощью таймера, поскольку они могут потребовать существенных ресурсов. В следующем разделе мы обратимся к мультивалютным индикаторам, работающим на уровне котировок, и потому для них будет достаточно упрощенного асинхронного запроса на обновление графика с помощью *ChartSetSymbolPeriod*. А в следующей главе мы встретимся с необходимостью реализации еще одного типа ожидания: готовности таймсерий другого индикатора.

5.4.16 Мультивалютные и мультитаймфреймовые индикаторы

До сих пор мы рассматривали индикаторы, работающие с котировками или тиками символа, являющегося текущим символом графика. Однако иногда необходимо проводить анализ нескольких финансовых инструментов или одного инструмента, отличного от текущего. В подобных случаях, как мы видели в случае анализа тиков, недостаточно стандартных таймсерий, передаваемых в индикатор через параметры *OnCalculate*. И встает задача каким-либо образом запросить "чужие" котировки, дождаться их построения, и только затем рассчитывать на их основе индикатор.

В этом смысле запрос и построение котировок для таймфрейма, отличного от текущего таймфрейма графика, не отличается от механизмов работы с другими символами. Поэтому в данном разделе мы продемонстрируем создание мультивалютных индикаторов, а мультитаймфреймовые можно организовать по аналогичному принципу.

Одной из проблем, которую потребуется в любом случае решить, является синхронизация баров по времени. В частности, для разных символов могут быть различные расписания торгов, выходные дни, и в общем случае нумерация баров на родительском графике и в котировках "чужого" символа не совпадает.

Для начала упростим задачу и ограничимся одним произвольным символом, который может отличаться от текущего. Довольно часто трейдеру желательно видеть одновременно несколько графиков различных символов (например, ведущий и ведомый в коррелированной паре). Разработаем индикатор *IndSubChartSimple.mq5* для отображения котировки выбранного пользователем символа в подокне.

IndSubChartSimple

Чтобы повторить внешний вид основного графика предусмотрим во входных параметрах не только указание символа, но и режима рисования: DRAW_CANDLES, DRAW_BARS, DRAW_LINE. Первые два требуют 4 буфера и выводят полную четверку цен *Open*, *High*, *Low*, *Close* (японскими свечами или барами), а последний — довольствуется единственным буфером для показа линии по цене *Close*. Но чтобы поддержать все режимы выбираем максимальное требуемое количество буферов.

```
#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
```

Массивы для буферов описаны по названиям типов цен.

```
double open[];
double high[];
double low[];
double close[];
```

По умолчанию включено отображение японских свечей. В этом режиме MQL5 позволяет указать не один цвет, а несколько. В директиве *#property indicator_colorN* они задаются через запятую. Если цветов два, то первый определяет цвет контуров свечей, а второй — заполнение. Если цветов три, как в нашем случае, то первый, как и в предыдущем случае, определяет цвет контуров свечей, а второй и третий — тело бычьей и тело медвежьей свечи соответственно.

В главе посвященной графикам мы познакомимся с перечислением ENUM_CHART_MODE, которое описывает 3 доступных режима работы графиков.

Элементы ENUM_CHART_MODE	Элементы ENUM_DRAW_TYPE
CHART_CANDLES	DRAW_CANDLES
CHART_BARS	DRAW_BARS
CHART_LINE	DRAW_LINE

Они соответствуют выбранным нами режимам отрисовки, что в общем-то и не удивительно, потому что для этого индикатора намеренно выбирались способы рисования, повторяющие стандартные. ENUM_CHART_MODE удобно здесь использовать, поскольку оно содержит только 3 нужных нам элемента, в отличие от ENUM_DRAW_TYPE, в котором много других способов отрисовки.

Таким образом, входные переменные получают следующие определения.

```
input string SubSymbol = ""; // Symbol
input ENUM_CHART_MODE Mode = CHART_CANDLES;
```

Для перевода ENUM_CHART_MODE в ENUM_DRAW_TYPE реализована простая функция.

```
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;
}
```

Пустая строка во входном параметре *SubSymbol* означает текущий символ графика. Однако поскольку MQL5 не позволяет редактировать входные переменные, нам придется добавить глобальную переменную для хранения реального рабочего символа и присваивать её в обработчике *OnInit*.

```
string symbol;
...
int OnInit()
{
   symbol = SubSymbol;
   if(symbol == "") symbol = _Symbol;
   else
   {
      // убеждаемся, что символ существует и выбран в Обзоре рынка
      if(!SymbolSelect(symbol, true))
      {
         return INIT_PARAMETERS_INCORRECT;
      }
   }
   ...
}
```

Попутно необходимо проверить наличие введенного пользователем символа и добавить его в окно *Обзор рынка*: этим занимается функция *SymbolSelect*, которую мы изучим в главе про символы.

Для обобщения настройки буферов и диаграмм в исходном коде выделено несколько вспомогательных функций:

- InitBuffer — настройка одного буфера
- InitBuffers — настройка всего набора буферов
- InitPlot — настройка одной диаграммы

Отдельные функции объединяют в себе несколько действий, повторяющихся при регистрации идентичных сущностей, а также открывают дорогу для дальнейшего развития данного индикатора в главе про графики: там мы поддержим интерактивное изменение настроек рисования в ответ на действия пользователя с графиком (см. полную версию индикатора *IndSubChart.mq5* в разделе Режимы отображения графика).

```
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);
      // скрываем все буфера, неиспользуемые для линейного графика
      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;
}
```

Обратите внимание, что при включении режима линейной диаграммы используется только массив *close* и ему назначается индекс 0. Остальные три массива полностью скрываются от пользователя за счет свойства INDICATOR_CALCULATIONS. В режимах свечей и баров задействованы все xtnsht массива, и их нумерация соответствует стандарту OHLC, как того требуют типы отрисовки DRAW_CANDLES и DRAW_BARS. Всем массивам назначается свойство "серийности", то есть индексации справа налево.

Функция *InitBuffers* возвращает заголовок для буферов в *Окне данных*.

В функции *InitPlot* устанавливаются все необходимые атрибуты диаграммы.

```
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);
}
```

Начальная настройка единственной диаграммы (с индексом 0) производится с помощью новых функций в обработчике *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;
}
```

Хотя в данной версии индикатора настройка выполняется единожды, это происходит динамически с учетом входного параметра *Mode*, в отличие от статической настройки, которую предоставляют директивы *#property*. И в дальнейшем, в полной версии индикатора мы сможем вызывать *InitPlot* по много раз, меняя внешнее представление индикатора "на ходу".

Заполнение буферов производится в *OnCalculate*. В простейшем случае, когда заданный символ совпадает с графиком, нам было бы достаточно написать примерно такую реализацию.

```
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) // требует уточнения (см. далее)
   {
      ArrayInitialize(open, EMPTY_VALUE);
      ArrayInitialize(high, EMPTY_VALUE);
      ArrayInitialize(low, EMPTY_VALUE);
      ArrayInitialize(close, EMPTY_VALUE);
   }

   if(_Symbol != symbol)
   {
      // в процессе разработки
      ...
   }
   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;
}
```

Однако при обработке произвольного символа параметры-массивы не содержат нужных котировок, да и общее число доступных баров наверняка отличается. Более того, при первом размещении индикатора на графике котировки "чужого" символа вообще могут быть не готовы, если для него заранее не открыт по соседству другой график. Да и загрузка котировок стороннего символа будет происходить асинхронно, из-за чего в любой момент может "прибыть" новая партия баров, требующая полного пересчета.

Поэтому создадим переменные, контролирующие количество баров на стороннем символе (*lastAvailable*), редактируемый "клон" константного аргумента *prev_calculated*, а также признак готовности котировок.

```
   static bool initialized;  // флаг готовности котировок символа
   static int lastAvailable; // количество баров для символа (и текущего таймфрейма)
   int _prev_calculated = prev_calculated; // редактируемая копия prev_calculated
```

В начале *OnCalculate* добавим проверку на одномоментное появление более одного бара: в этом нам помогает переменная *lastAvailable*, которую мы заполняем на основе значения *iBars(symbol,*

_Period) перед предыдущим штатным выходом из функции, то есть в случае успешного расчета. При обнаружении докачки истории следует сбросить *_prev_calculated* и количество баров в 0, а также убрать признак готовности, чтобы пересчитать индикатор заново.

```
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)
   {
      // докачка истории или первый старт
      _prev_calculated = 0;
      initialized = false;
      lastAvailable = 0;
   }

   // далее везде используем копию _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)
   {
      // запрос котировок и "ожидание" их готовности
      ...
      // основной расчет (заполнение буферов)
      ...
   }
   else
   {
      ... // как есть
   }
   lastAvailable = iBars(symbol, _Period);
   return rates_total;
}
```

Слово "ожидание" в комментарии не случайно взято в кавычки. Как мы помним, в индикаторах нельзя реально ждать (чтобы не тормозить интерфейсный поток терминала), и вместо этого недостаток данных должен просто приводить к выходу из функции. Таким образом "ожидание" будет заключаться в ожидании следующего события для расчета: по приходу тика или в ответ на запрос обновления графика.

Проверкой готовности котировок займется следующий фрагмент кода.

```
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, _Perioc
            (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
         {
            // асинхронный запрос на обновление графика
            ChartSetSymbolPeriod(0, _Symbol, _Period);
            return 0; // нечего показывать пока
         }
      }
      ...
```

Основную работу в нем выполняет особая функция *QuoteRefresh*. Она принимает в качестве аргументов интересующий нас символ, таймфрейм и время самого первого (старого) бара на текущем графике — более ранние даты нас не интересуют, но не факт, что на "чужом" символе имеется история на всю эту глубину. Именно поэтому удобно скрыть все сложности проверок в отдельной функции.

Функция вернет *true*, когда данные будут скачаны и синхронизированы в доступном объеме. Её внутреннее устройство рассмотрим через минуту.

Когда синхронизация выполнена, используем функцию *iBarShift* для нахождения синхронных баров и копирования их значений OHLC (функции *iOpen*, *iHigh*, *iLow*, *iClose*).

```
      ArraySetAsSeries(time, true); // обход из настоящего в прошлое
      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;
         }
      }
```

Альтернативный и на первый взгляд более эффективный способ копирования массивов цен целиком с помощью Copy-функций здесь не подходит из-за того, что в бары с равными индексами могут соответствовать на разных символах разным меткам времени. Поэтому после копирования пришлось бы анализировать даты и перемещать элементы внутри буферов, подгоняя под время на текущем графике.

Поскольку в функцию *iBarShift* последним параметром передается *true*, функция будет искать точное соответствие времени баров, и если в другом символе какой-либо бар отсутствует, мы получим -1 и отобразим на графике пустоту (EMPTY_VALUE).

После успешного полного расчета новые бары будут обсчитываться в экономном режиме, т.е. с учетом *_prev_calculated* и *rates_total*.

А теперь обратимся к функции *QuoteRefresh*. Как универсальная и полезная, она вынесена в заголовочный файл *QuoteRefresh.mqh*.

В самом начале делается проверка на то, не запрашивается ли таймсерия текущего символа и текущего таймфрейма из MQL-программы типа индикатор. Такие запросы запрещены, поскольку "родная" таймсерия, на которой запущен индикатор, уже находится в процессе построения терминалом или готова: подгонять его еще раз — чревато зацикливанием или блокировкой. Поэтому мы просто возвращаем признак синхронизации (SERIES_SYNCHRONIZED), и если она пока не готова, индикатору следует проверить данные позднее (на следующих тиках, по таймеру или как-то еще).

```
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);
   }
   ...
```

Вторая проверка касается количества баров: если оно уже равно максимальному разрешенному на графиках, докачивать что-либо не имеет смысла.

```
   if(Bars(asset, period) >= TerminalInfoInteger(TERMINAL_MAXBARS))
   {
      return (bool)SeriesInfoInteger(asset, period, SERIES_SYNCHRONIZED);
   }
   ...
```

Далее начинается фрагмент кода, который последовательно узнает у терминала начальные даты доступных котировок:

- по заданному таймфрейму (SERIES_FIRSTDATE);
- без привязки к таймфрейму (SERIES_TERMINAL_FIRSTDATE) в локальной базе терминала;
- без привязки к таймфрейму (SERIES_TERMINAL_FIRSTDATE) на сервере.

Если на каком-либо этапе запрашиваемая дата уже входит в область доступных данных, получим *true* как признак готовности. В противном случае выполняется запрос данных из локальной базы терминала или с сервера, с последующим построением таймсерии (все это делается асинхронно и автоматически в ответ на наши вызовы *CopyTime*, можно было использовать другие *Copy*-функции).

```
      datetime times[1];
      datetime first = 0, server = 0;
      if(PRTF(SeriesInfoInteger(asset, period, SERIES_FIRSTDATE, first)))
      {
         if(first > 0 && first <= start)
         {
            // прикладные данные существуют, они уже готовы или подготавливаются
            return (bool)SeriesInfoInteger(asset, period, SERIES_SYNCHRONIZED);
         }
         else
         if(PRTF(SeriesInfoInteger(asset, period, SERIES_TERMINAL_FIRSTDATE, first)))
         {
            if(first > 0 && first <= start)
            {
               // технические данные существуют в базе терминала,
               // инициируем построение таймсерии или сразу получим искомое
               return PRTF(CopyTime(asset, period, first, 1, times)) == 1;
            }
            else
            {
               if(PRTF(SeriesInfoInteger(asset, period, SERIES_SERVER_FIRSTDATE, server)
               {
                  // технические данные существуют на сервере, запросим их
                  if(first > 0 && first < server)
                     PrintFormat(
                       "Warning: %s first date %s on server is less than on terminal ",
                        asset, TimeToString(server), TimeToString(first));
                  // нельзя просить больше, чем имеет сервер - поэтому fmax
                 return PRTF(CopyTime(asset, period, fmax(start, server), 1, times)) ==
               }
            }
         }
      }

      return false;
   }
```

Индикатор готов. Откомпилируем и запустим его, например, на графике EURUSD,H1, указав в качестве дополнительного символа USDRUB. В журнале появятся примерно такие записи:

```
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
```

После индикации завершения процесса ("Done"), в подокне будут показаны свечи "чужого" графика.

Индикатор IndSubChartSimple — DRAW_CANDLES с котировками стороннего символа

Важно отметить, что из-за сокращенной торговой сессии значащие бары для USDRUB занимают лишь дневную часть каждого суточного интервала.

IndUnityPercent

Вторым индикатором, который мы создадим в рамках данного раздела, является настоящий мультивалютный (строго говоря, мультисимвольный) индикатор *IndUnityPercent.mq5*. Его идея заключается в том, чтобы отобразить относительную силу всех независимых валют (активов), входящих в состав заданных финансовых инструментов. Например, если мы торгуем корзиной из двух тикеров EURUSD и XAUUSD, то фактически в стоимости учитываются доллар, евро и золото — каждый из этих активов обладает относительной стоимостью по сравнению с другими.

В каждый момент времени существуют текущие цены, которые описываются очевидными формулами:

```
EUR / USD = EURUSD
XAU / USD = XAUUSD
```

где переменные EUR, USD, XAU — некие самостоятельные "стоимости" активов, а EURUSD и XAUUSD — константы (известные котировки).

Для нахождения переменных дополним систему еще один уравнением, ограничив сумму квадратов переменных единицей (отсюда и первое слово в названии индикатора — Unity):

```
EUR * EUR + USD * USD + XAU * XAU = 1
```

Переменных может быть гораздо больше, и их логично обозначить как x_i, причем x_0 — основная валюта (общая для всех инструментов: она обязательно должна быть).

Тогда в общем виде формулы расчета переменных запишутся следующим образом (для краткости мы опустим процесс их выведения):

```
x0 = sqrt(1 / (1 + sum(C(xi, x0)2))), i = 1..n
xi = C(xi, x0) * x0, i = 1..n
```

где n — количество переменных, $C(x_i,x_0)$ — котировка i-ой пары, включающей соответствующие переменные. Обратите внимание, что количество переменных на 1 больше, чем инструментов.

Поскольку котировки, участвующие в расчете, обычно сильно отличаются (например, как в случае EURUSD и XAUUSD), а кроме того выражаются только друг через друга (то есть без привязки к какой-либо стабильной базе), имеет смысл перейти от абсолютных значений к процентным изменениям. Таким образом, при написании алгоритмов по вышеприведенным формулам будем вместо котировки $C(x_i,x_0)$ брать отношение $C(x_i,x_0)[0]$ / $C(x_i,x_0)[1]$, где индексы в квадратных скобках означают текущий [0] и предыдущий [1] бар. Кроме того, для ускорения расчета можно избавиться от возведения в квадрат и взятия квадратного корня.

Для визуализации линий предусмотрим некое максимальное допустимое количество валют и индикаторных буферов. Разумеется, в расчете можно использовать не все, если пользователь введет меньше символов. Но повысить лимит динамически нельзя: потребуется изменить директивы и перекомпилировать индикатор.

```
#define BUF_NUM 15
#property indicator_separate_window
#property indicator_buffers BUF_NUM
#property indicator_plots BUF_NUM
```

При реализации данного индикатора решим попутно одну неприятную проблему. Поскольку предполагается множество однотипных буферов, стандартный подход предполагает их экстенсивное кодирование "размножением" (пресловутый нерекомендуемый стиль программирования "copy & paste").

```
double buffer1[];
...
double buffer15[];

void OnInit()
{
   SetIndexBuffer(0, buffer1);
   ...
   SetIndexBuffer(14, buffer15);
}
```

Это неудобно, неэффективно и чревато ошибками. Вместо этого применим ООП: создадим класс, который будет хранить массив для индикаторного буфера и отвечать за его единообразную настройку — ведь наши буфера должны быть одинаковыми (за исключением цветов и, возможно, увеличенной толщины для тех валют, которые составляют символ текущего графика, но это донастраивается позднее — после введения входных параметров пользователем).

При наличии такого класса достаточно распределить массив его объектов, и индикаторные буфера будут автоматически подключены и настроены в необходимом количестве. Схематично данный подход иллюстрируется следующим псевдокодом.

```
// код "движка" с поддержкой массива унифицированных индикаторных буферов
class Buffer
{
   static int count; // глобальный счетчик буферов
   double array[];   // массив для этого буфера
   int cursor;       // указатель присваиваемого элемента
public:
   // конструктор настраивает и подключает массив
   Buffer()
   {
      SetIndexBuffer(count++, array);
      ArraySetAsSeries(array, ...);
   }
   // перегрузка для установки номера интересующего элемента
   Buffer *operator[](int index)
   {
      cursor = index;
      return &this;
   }
   // перегрузка для записи значения в выбранный элемент
   double operator=(double x)
   {
      buffer[cursor] = x;
      return x;
   }
   ...
};

static int Buffer::count;
```

Благодаря перегрузкам операторов мы можем придерживаться привычного синтаксиса для присваивания значений элементам объекта-буфера: *buffer[i] = value*.

В коде индикатора вместо множества строк с описаниями отдельных массивов достаточно будет определить один "массив массивов".

```
// код индикатора
// конструируем 15 объектов-буферов с авто-регистрацией и настройкой
Buffer buffers[15];
...
```

Полная версия классов, реализующих данный механизм, приводится в файле *IndBufArray.mqh*. Следует отметить, что в нем обеспечена поддержка только буферов, но не диаграмм. В идеале набор классов должен быть расширен новыми, позволяющими создавать готовые объекты-диаграммы, которые занимали бы в массиве буферов необходимое их количество согласно типу конкретной диаграммы. Изучить и дополнить файл предлагается самостоятельно. В частности, в коде имеется класс-менеджер массива индикаторных буферов *BufferArray* для создания "массивов массивов" с одинаковыми значениями свойств, таких как тип ENUM_INDEXBUFFER_TYPE, направление индексации, "пустое" значение. Мы его используем в новом индикаторе следующим образом:

```
BufferArray buffers(BUF_NUM, true);
```

Здесь в первом параметре конструктора передается требуемое количество буферов, а во втором — признак индексации как в таймсерии (об этом — чуть ниже).

После этого определения мы можем в любом месте кода применять удобную нотацию для установки значения j-го бара i-го буфера (она использует двойную перегрузку оператора[] — не только в объекте-буфере, но и в массиве буферов):

```
buffers[i][j] = value;
```

Во входных переменных индикатора позволим пользователю задать перечень интересных ему символов через запятую, а также ограничим количество баров для расчета на истории, чтобы управлять длительностью загрузки и синхронизации потенциально большого набора инструментов. Если вы решите показывать всю доступную историю, следует выявить и применить наименьшее количество баров среди доступных у разных инструментов, с контролем докачки с сервера.

```
input string Instruments = "EURUSD,GBPUSD,USDCHF,USDJPY,AUDUSD,USDCAD,NZDUSD";
input int BarLimit = 500;
```

При старте программы следует произвести разбор списка символов и сформировать из них отдельный массив *Symbols* размера *SymbolCount*.

```
string Symbols[];
int Direction[]; // прямой(+1)/обратный(-1) курс к общей валюте
int SymbolCount;
```

У всех символов должна быть одна общая валюта (обычно "USD") для выявления взаимных соотношений. В зависимости от того, является эта общая валюта в конкретном символе базовой (на первом месте в паре, если речь о Forex) или валютой котирования (на втором месте в паре Forex), в расчетах должны участвовать её прямые или обратные котировки (1.0 / курс). Это направление будем хранить в массиве *Direction*.

Приведем с некоторыми сокращениями функцию *InitSymbols*, которая выполняет описанные действия. В случае успешного анализа списка она возвращает название общей валюты. Получить базовую валюту и валюту котирования любого финансового инструмента позволяет встроенная функция *SymbolInfoString*: мы изучим её в главе про финансовые инструменты.

```
string InitSymbols()
{
   SymbolCount = fmin(StringSplit(Instruments, ',', Symbols), BUF_NUM - 1);
   ArrayResize(Symbols, SymbolCount);
   ArrayResize(Direction, SymbolCount);
   ArrayInitialize(Direction, 0);

   string common = NULL; // общая валюта

   for(int i = 0; i < SymbolCount; i++)
   {
      // гарантируем наличие символа в Обзоре рынка
      if(!SymbolSelect(Symbols[i], true))
      {
         Print("Can't select ", Symbols[i]);
         return NULL;
      }

      // получаем валюты, составляющие символ
      string first, second;
      first = SymbolInfoString(Symbols[i], SYMBOL_CURRENCY_BASE);
      second = SymbolInfoString(Symbols[i], SYMBOL_CURRENCY_PROFIT);

      // подсчитываем количество включений каждой валюты
      if(first != second)
      {
         workCurrencies.inc(first);
         workCurrencies.inc(second);
      }
      else
      {
         workCurrencies.inc(Symbols[i]);
      }
   }
   ...
```

В цикле ведется учет встречаемости каждой валюты в составе всех инструментов с помощью вспомогательного шаблонного класса *MapArray*. Такой объект описан в индикаторе на глобальном уровне и требует подключения заголовочного файла *MapArray.mqh*.

```
#include <MQL5Book/MapArray.mqh>
...
// массив пар [название;количество]
// для подсчета статистики использования валют
MapArray<string,int> workCurrencies;
...
string InitSymbols()
{
   ...
}
```

Поскольку данный класс выполняет вспомогательную роль, здесь он подробно не описан. Желающие могут ознакомиться с исходным кодом. Суть в том, что при вызове его метода *inc* для нового названия валюты, оно добавляется во внутренний массив с начальным значением счетчика равным 1, а если название уже встречалось — счетчик увеличивается на 1.

Впоследствии мы находим общую валюту, как ту, у которой счетчик больше 1. При правильных настройках остальные валюты должны встретиться ровно по разу. Вот продолжение функции *InitSymbols*.

```
   ...
   // находим общую валюту на основе статистики использования валют
   for(int i = 0; i < workCurrencies.getSize(); i++)
   {
      if(workCurrencies[i] > 1) // счетчик больше 1
      {
         if(common == NULL)
         {
            common = workCurrencies.getKey(i); // получим имя i-ой валюты
         }
         else
         {
            Print("Collision: multiple common symbols");
            return NULL;
         }
      }
   }

   if(common == NULL) common = workCurrencies.getKey(0);

   // зная общую валюту, определяем "направление" каждого символа
   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;
}
```

Имея готовую функцию *InitSymbols*, мы можем написать *OnInit* (приводится с упрощениями).

```
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);

   // настройка линий по количеству валют (количество символов + 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));
   }

   // скрываем лишние буфера в Окне данных
   for(int i = SymbolCount + 1; i < BUF_NUM; i++)
   {
      PlotIndexSetInteger(i, PLOT_SHOW_DATA, false);
   }

   // единственный уровень на 1.0
   IndicatorSetInteger(INDICATOR_LEVELS, 1);
   IndicatorSetDouble(INDICATOR_LEVELVALUE, 0, 1.0);

   // Название с параметрами
   IndicatorSetString(INDICATOR_SHORTNAME,
      "Unity [" + (string)workCurrencies.getSize() + "]");

   // точность
   IndicatorSetInteger(INDICATOR_DIGITS, 5);

   return INIT_SUCCEEDED;
}
```

Теперь познакомимся с обработчиком главного события *OnCalculate*.

Важно отметить, что порядок обхода баров в главном цикле — обратный, как в таймсерии, от настоящего к прошлому. Данный подход более удобен для мультивалютных индикаторов, потому что глубина истории разных символов может оказаться разной, и имеет смысл обсчитывать бары от текущего назад, вплоть до первого обнаружения недостатка данных по любому из символов. При этом следует трактовать досрочное прерывание цикла не как ошибку и вернуть *rates_total*, чтобы отобразить на графике значения для уже посчитанных, наиболее актуальных баров.

Однако в данной упрощенной версии *IndUnityPercent* мы этого не делаем и обходимся более простым и жестким подходом: пользователь должен определить безусловную глубину запроса истории с помощью параметра *BarLimit*. Иными словами, по всем символам должны быть данные плоть до временной метки бара с номером *BarLimit* на символе графика — иначе индикатор будет пытаться скачать недостающие данные.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double& price[])
{
   if(prev_calculated == 0)
   {
      buffers.empty(); // делегируем тотальную очистку классу BufferArray
   }

   // основной цикл в направлении "как в таймсерии" от настоящего в прошлое
   const int limit = MathMin(rates_total - prev_calculated + 1, BarLimit);
   for(int i = 0; i < limit; i++)
   {
      if(!calculate(i))
      {
         EventSetTimer(1); // даем еще 1 секунду на закачку и подготовку данных
         return 0; // попробуем пересчитаться на следующем вызове
      }
   }

   return rates_total;
}
```

Расчет значений для всех буферов на i-ом баре осуществляет функция *Calculate* (см. далее). В случае недостатка данных она вернет *false*, и мы запустим таймер, чтобы дать время на построение таймсерий по всем требуемым инструментам. В обработчике таймера привычным образом отправим терминалу запрос на обновление графика.

```
void OnTimer()
{
   EventKillTimer();
   ChartSetSymbolPeriod(0, _Symbol, _Period);
}
```

В функции *Calculate* первым делом определяем диапазон дат текущего и предыдущего бара, на которых будут рассчитываться изменения.

```
bool Calculate(const int bar)
{
   const datetime time0 = iTime(_Symbol, _Period, bar);
   const datetime time1 = iTime(_Symbol, _Period, bar + 1);
   ...
```

Две даты потребовалось, чтобы вызвать далее функцию *CopyClose* в том её варианте, где указывается интервал дат. В данном индикаторе мы не можем использовать вариант с количеством баров, потому что на любом символе могут быть произвольные пропуски в барах, отличные от пропусков на других символах. Например, если на одном символе существуют бары *t* (текущий) и *t-1* (предыдущий), то для него возможно корректно рассчитать изменение *Close[t]/Close[t-1]*. Однако на другом символе бар *t* может отсутствовать, и запрос двух баров вернет "ближайшие" слева (в прошлом) бары, причем это прошлое может отстоять от "настоящего" достаточно далеко (например, соответствовать торговой сессии за предыдущий день, если символ не торгуется круглосуточно).

Чтобы такого не происходило, индикатор запрашивает котировки строго в интервале, и если он оказывается пустым для конкретного символа, это означает отсутствие изменений.

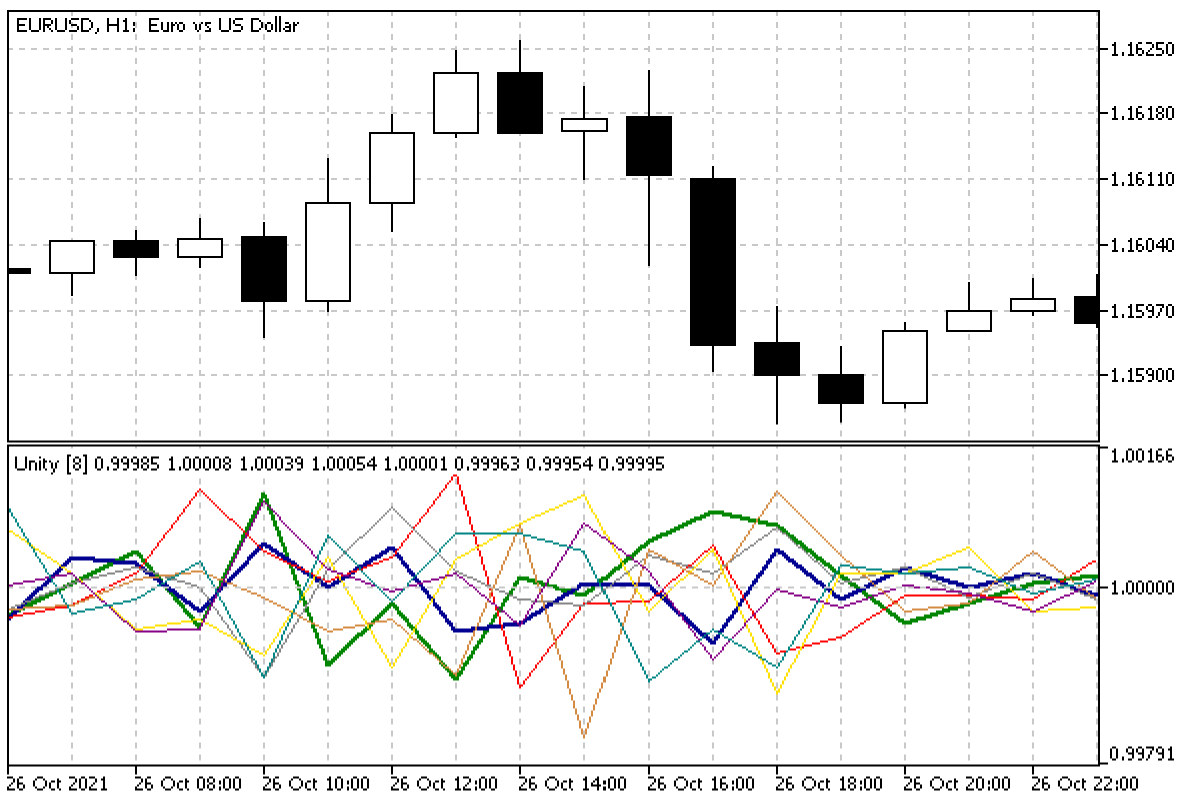
При этом возможны ситуации, когда такой запрос вернет больше 2 баров, и в этом случае всегда берутся два последних (правых) бара, как наиболее актуальные. Например, при размещении на графике USDRUB,H1 индикатор будет "видеть", что после бара в 17:00 каждого рабочего дня идет бар 10:00 следующего рабочего дня. Однако для основных валютных пар Forex, таких как EURUSD, между ними будет 16 вечерних, ночных и утренних баров H1.

```
bool Calculate(const int bar)
{
   ...
   double w[]; // приёмный массив котировок (побаровый)
   double v[]; // изменения по символам
   ArrayResize(v, SymbolCount);

   // находим изменения котировок для каждого символа
   for(int j = 0; j < SymbolCount; j++)
   {
      // пробуем получить для j-го символа как минимум 2 бара,
      // соответствующих двум барам символа текущего графика
      int x = CopyClose(Symbols[j], _Period, time0, time1, w);
      if(x < 2)
      {
         // если баров нет, пробуем получить предыдущий бар из прошлого
         if(CopyClose(Symbols[j], _Period, time0, 1, w) != 1)
         {
            return false;
         }
         // затем дублируем его как индикацию отсутствия изменений
         // (в принципе, можно было 2 раза записать любую константу)
         x = 2;
         ArrayResize(w, 2);
         w[1] = w[0];
      }

      // находим обратный курс, когда необходимо
      if(Direction[j] == -1)
      {
         w[x - 1] = 1.0 / w[x - 1];
         w[x - 2] = 1.0 / w[x - 2];
      }

      // вычисляем изменений в виде соотношения двух величин
      v[j] = w[x - 1] / w[x - 2]; // последняя / предыдущая
   }
   ...
```

Когда изменения получены, алгоритм работает по приводившимся ранее формулам и записывает значения в индикаторные буфера.

```
   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;
}
```

Посмотрим, как индикатор работает с настройками по умолчанию, на наборе основных инструментов Forex (при первом размещении может потребовать заметное время для получения таймсерий, если для инструментов не были открыты графики).



Мультисимвольный индикатор IndUnityPercent с основными валютами Forex

Расстояние между линиями двух валют в окне индикатора равно изменению соответствующей котировки в процентах (между двумя последовательными ценами *Close*). Отсюда второе слово в названии индикатора — Percent.

В следующей главе про программное использование индикаторов мы представим продвинутую версию *IndUnityPercentPro.mq5* с заменой *Copy*-функций на вызов встроенных индикаторов *iMA*, что позволит без лишних усилий реализовать сглаживание и расчет по произвольному типу цен.

5.4.17 Отслеживание формирования баров

Рассмотренный в предыдущем разделе индикатор *IndUnityPercent.mq5* пересчитывается на последнем баре на каждом тике, так как использует цены *Close*. Некоторые индикаторы и эксперты специально разрабатываются в более экономичном стиле — с однократным расчетом на каждом баре. Например, мы могли бы считать формулу Unity по ценам открытия, и тогда разумно пропускать тики. Способов определения появления нового бара можно придумать несколько:

- Запоминать время текущего 0-го бара (через параметр *time* функции *OnCalculate* — *time[0]* или в общем случае *iTime(symbol, period, 0)*) и ждать, когда оно изменится;
- Запоминать количество баров *rates_total* (или *iBars(symbol, period)*) и реагировать на увеличение на 1 (изменение на другое количество в ту или иную сторону подозрительно и может свидетельствовать о модификации истории);
- Ждать бара с тиковым объемом равным 1 (первый тик на баре).

Однако при мультивалютной природе индикатора само понятие формирования нового бара становится не таким однозначным.

На каждом символе очередной бар появляется по приходу собственных тиков, и их время обычно не совпадает. В этом случае разработчик индикатора должен определить протокол действий: стоит ли дожидаться появления баров с одинаковым временем на всех символах или пересчитывать индикатор на последних барах несколько раз — после появления нового бара на любом из символов.

В данном разделе мы представим простой класс *MultiSymbolMonitor* (см. файл *MultiSymbolMonitor.mqh*) для отслеживания формирования новых баров по заданном списку символов.

В конструктор класса можно передать требуемый таймфрейм. По умолчанию контролируется таймфрейм текущего графика, на котором запущена программа.

```
class MultiSymbolMonitor
{
protected:
   ENUM_TIMEFRAMES period;

public:
   MultiSymbolMonitor(): period(_Period) {}
   MultiSymbolMonitor(const ENUM_TIMEFRAMES p): period(p) {}
   ...
```

Для хранения списка контролируемых символов воспользуемся вспомогательным классом *MapArray* из предыдущего раздела. В этот массив будем записывать пары [имя символа;временная метка последнего бара], то есть шаблонных типов *<string,datetime>*. Для заполнения массива предусмотрен метод *attach*.

```
protected:
   MapArray<string,datetime> lastTime;
...
public:
   void attach(const string symbol)
   {
      lastTime.put(symbol, NULL);
   }
```

Для заданного массива класс умеет обновлять и проверять метки времени в методе *check*, вызывая функцию *iTime* в цикле по символам.

```
   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]) // есть ли изменения?
         {
            flags |= 1 << i;
         }

         if(refresh) // обновить метку времени
         {
            lastTime.put(symbol, dt);
         }
      }
      return flags;
   }
```

Вызывающий код должен вызывать *check* по своему усмотрению — как правило, по приходу тиков или по таймеру. Строго говоря, оба эти варианта не обеспечивают моментальной реакции на появление тиков (и новых баров) на других инструментах: ведь событие *OnCalculate* появляется только на тиках рабочего символа графика и если между ними произошел тик какого-то другого символа, мы об этом не узнаем вплоть до следующего "своего" тика.

Оперативный мониторинг тиков с нескольких инструментов мы рассмотрим в главе про интерактивные события на графиках (см. индикатор-шпион *EventTickSpy.mq5* в разделе Генерация пользовательских событий).

Сейчас же ограничимся проверками баров с доступной точностью и вернемся к рассмотрению метода *check*.

Каждый момент времени характеризуется собственным состоянием набора временных меток по всем символам в массиве. Например, в 12:00 может сформироваться новый бар только по самому ликвидному инструменту, а для нескольких других инструментов тики появятся через несколько миллисекунд или даже секунд. В этот промежуток в массиве обновится один элемент, а остальные будут старыми. Затем постепенно все символы получат бары 12:00.

Для всех символов, по которым время открытия последнего бара не равно сохраненному, метод взводит бит под номером символа, формируя таким образом битовую маску с изменениями. В списке должно быть не более 64 символов.

Если возвращаемое значение равно нулю — изменений не зафиксировано.

Параметр *refresh* задает, будет ли метод *check* просто фиксировать изменения (*false*) или произведет обновление состояния в соответствии с текущей рыночной ситуацией (*true*).

Метод *describe* позволяет по битовой маске получить список изменившихся символов.

```
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;
}
```

Наконец, метод *inSync* пригодится для определения того, имеют ли все символы в массиве одинаковое время последнего бара. Его имеет смысл применять только для корзины валют с одинаковыми торговыми сессиями.

```
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;
}
```

С помощью описанного класса реализуем простой мультивалютный индикатор *IndMultiSymbolMonitor.mq5*, единственная задача которого будет в детектировании новых баров по списку символов.

Поскольку никакой отрисовки для индикатора не предусмотрено, количество буферов и диаграмм равно 0.

```
#property indicator_chart_window
#property indicator_buffers 0
#property indicator_plots   0
```

Список инструментов задается в соответствующей входной переменной и затем преобразуется в массив, регистрируемый в объекте *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]);
   }
}
```

Обрабочик *OnCalculate* вызывает монитор на тиках и выводит изменения состояния в журнал.

```
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;
}
```

Чтобы проверить данный индикатор нам потребовалось бы провести за терминалом много времени онлайн. Однако MetaTrader 5 позволяет сделать это намного проще — с помощью тестера. Сделаем это в следующем разделе.

5.4.18 Тестирование индикаторов

Встроенный тестер MetaTrader 5 поддерживает два типа MQL-программ: эксперты и индикаторы. В случае индикаторов тестирование всегда проводится в визуальном окне. Но это касается только тестирования отдельно взятого индикатора. Если же индикатор создается и вызывается из эксперта программным способом, то тестирование этого эксперта вместе с индикатором (индикаторами) может проводиться и без визуализации, на усмотрение пользователя. Технологию использования индикаторов из MQL-кода мы изучим в следующей главе. Она же будет применяться и для интеграции с экспертами.

Разработчику индикатора при этом следует обратить внимание на то, что без визуализации тестер применяет для индикаторов, вызываемых из экспертов, ускоренный метод расчета — не на каждом тике, а только при запросе у индикатора данных из его индикаторных буферов (см. функцию *CopyBuffer*).

Если на текущем тике индикатор еще не был рассчитан, он рассчитывается однократно при первом обращении к его данным. При последующих возможных запросах в течение того же тика рассчитанные данных отдаются в готовом виде. Если на текущем тике не происходит чтение буферов индикатора, он не рассчитывается. Расчет индикаторов только по запросу даёт существенное ускорение при тестировании и оптимизации.

Если ваш индикатор имеет такую специфику, что он не должен пропускать тики, MQL5 позволяет потребовать у тестера включить пересчет индикатора на каждом тике. Это делается с помощью директивы:

```
#property tester_everytick_calculate
```

Словосочетание "каждый тик" (everytick) в директиве относится именно к расчету индикатора и не влияет на режим генерации тиков. Иными словами, под тиками подразумеваются генерируемые тестером изменения цен — по всем тикам, по ценам OHLC M1 или по открытию баров — эта настройка тестера остается в силе.

Для индикаторов, которые мы рассмотрели в данной главе, данное свойство некритично. Также следует отметить, что оно касается только работы в тестере стратегий: в терминале индикаторы всегда получают события *OnCalculate* на каждом поступившем тике (с поправкой на возможный пропуск тиков, если ваши вычисления в *OnCalculate* займут слишком много времени и не уложатся до появления нового тика).

Что же касается тестера, то индикаторы считаются в нем на каждом тике при любом из условий:

- в визуальном режиме;
- при наличии директивы *tester_everytick_calculate*;
- при наличии в них вызова *EventChartCustom*, или функций *OnChartEvent* или *OnTimer*.

Попробуем протестировать индикатор *IndMultiSymbolMonitor.mq5* из предыдущего раздела.

Выберем основным символом и таймфреймом графика EURUSD,H1. Способ генерации тиков поставим "на основе реальных тиков".

После запуска тестирования мы должны увидеть в журнале окна визуального режима примерно следующие записи:

```
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
```

Как нетрудно заметить, новые бары появляются на разных символах постепенно, и до появления флага "in-sync", равного *true*, проходит обычно несколько событий.

Вы можете запустить тестирование и других индикаторов данной главы. Обратите внимание, что если MQL-программа производит запрос истории тиков, в тестере обязательно должен быть выбран способ генерации "на основе реальных тиков".

Тестирование "по ценам открытия" можно использовать только для индикаторов и экспертов, которые разработаны с поддержкой данного режима, например, делают расчет только по ценам *Open* или анализируют завершенные бары, начиная с 1-го.

> Внимание! При тестировании индикаторов в тестере событие *OnDeinit* не срабатывает. Более того, не выполняется и прочая финализация, например, не вызываются деструкторы глобальных объектов.

5.4.19 Ограничения и преимущества индикаторов

Все специализированные функции, рассмотренные в данной главе, доступны только в исходных кодах индикаторов. Использовать их в других типах MQL-программ не имеет смысла: они вернут ошибку.

С другой стороны существует целый ряд функций, запрещенных в индикаторах.

- *OrderCalcMargin*
- *OrderCalcProfit*
- *OrderCheck*
- *OrderSend*
- *SendFTP*
- *WebRequest*
- *Socket****
- *Sleep*
- *MessageBox*
- *ExpertRemove*

Часть из них (с префиксом *Order*-) относится к торговым расчетам и разрешена только в экспертах и скриптах, часть предназначена для выполнения запросов в сети, которые блокируют выполнение потока до возвращения результата, а это недопустимо для индикаторов, потому что они выполняются в интерфейсном потоке терминала. По аналогичной причине запрещены функции *Sleep* и *MessageBox*.

В принципе, следует учитывать, что индикаторы в первую очередь отвечают за визуализацию данных и, как это ни странно, не подходят для массированных вычислений. В частности, если вы решите создать индикатор, обучающий в процессе работы нейронную сеть или дерево решений, это, скорее всего, отрицательно скажется на нормальном функционировании терминала.

Эффект длительного расчета демонстрируется индикатором *IndBarIndex.mq5*, который в штатном режиме предназначен для отображения в элементах своего буфера порядковых номеров баров. Однако с помощью входного параметра *SimulateCalculation*, который следует установить в *true*, можно по таймеру запустить бесконечный цикл.

```
// установка в true заморозит рисование индикаторов
// на чартах того же рабочего символа
// осторожно! не забудьте удалить индикатор после эксперимента!
input bool SimulateCalculation = false;

void OnInit()
{
   ...
   if(SimulateCalculation)
   {
      EventSetTimer(1);
   }
}
...
void OnTimer()
{
   Comment("Calculation started at ", TimeLocal());
   while(!IsStopped())
   {
      // бесконечный цикл для эмуляции расчетов
   }
   Comment("");
}
```

В таком режиме индикатор ожидаемо начинает целиком занимать 1 ядро процессора, однако проявляется и другой побочный эффект. Любые индикаторы на том же символе, на котором размещен *IndBarIndex*, перестают обновляться. Вы можете, например, применить *IndBarIndex* на EURUSD (любого таймфрейма), а потом на любом другом графике EURUSD попробовать применить обычную скользящую среднюю: она не отобразится, пока вы не удалите индикатор *IndBarIndex* с первого графика.

В связи с этим все длительные расчеты следует выносить в отдельные потоки, то есть скрипты или неторгующие эксперты, а в индикаторах только пользоваться их результатами. MQL5 API позволяет создавать новые графики или объекты с графиками, в которых можно применять tpl-шаблоны с требуемым экспертом или скриптом.

5.4.20 Создание заготовки индикатора в Мастере MQL

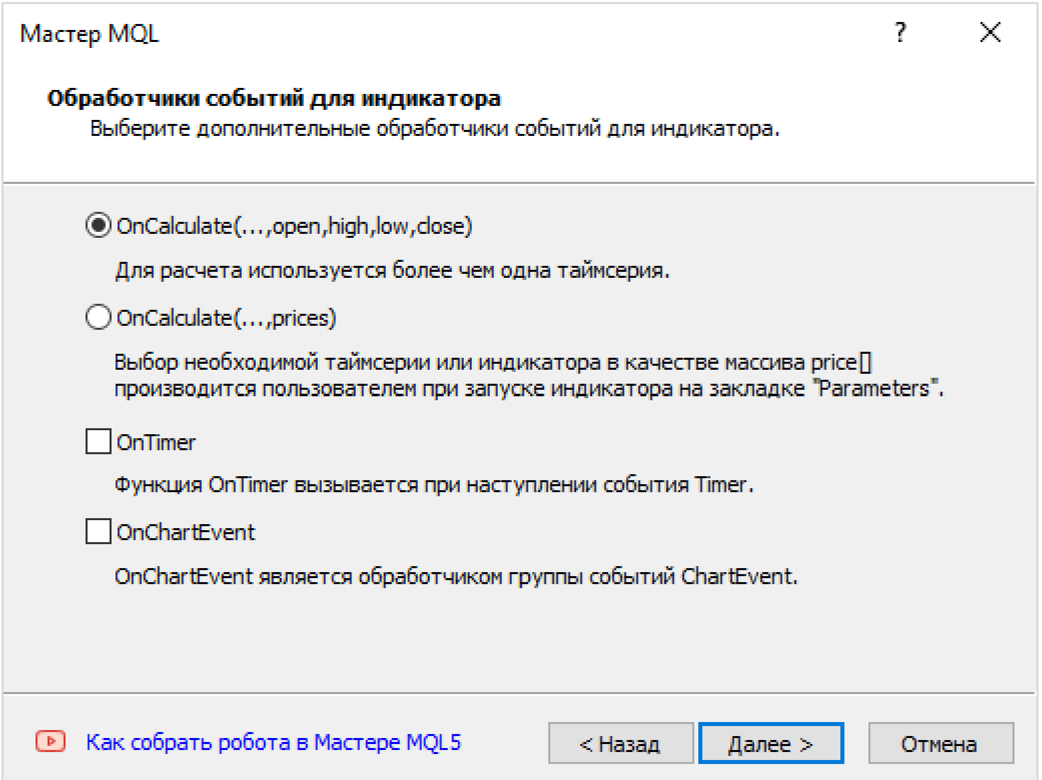
Итак, мы разобрали внутреннее устройство индикаторов и способны разобраться в том, как те или иные синтаксические конструкции в исходном коде влияют на внешнее представление и расчет индикатора. С данным уровнем подготовки можно начинать разбираться с чужим кодом и модифицировать его под свои нужды. Или попытаться создать что-то свое. Чтобы не начинать с чистого листа, можно воспользоваться Мастером MQL. В частности, с помощью него можно получить и заготовку индикатора.

Чтобы запустить Мастер, вызовите, например, контекстное меню в *Навигаторе* редактора MetaEditor для ветви *Indicators* и выполните в нем команду *Новый файл* (Ctrl+N). В первой части книги, в разделе Мастер MQL и эскиз программы мы создавали с помощью Мастера первый скрипт и видели, как выглядит этот шаг.

В данном случае (при запуске из контекстного меню) на первом шаге Мастера уже будет автоматически выбран пункт *Пользовательский индикатор*.

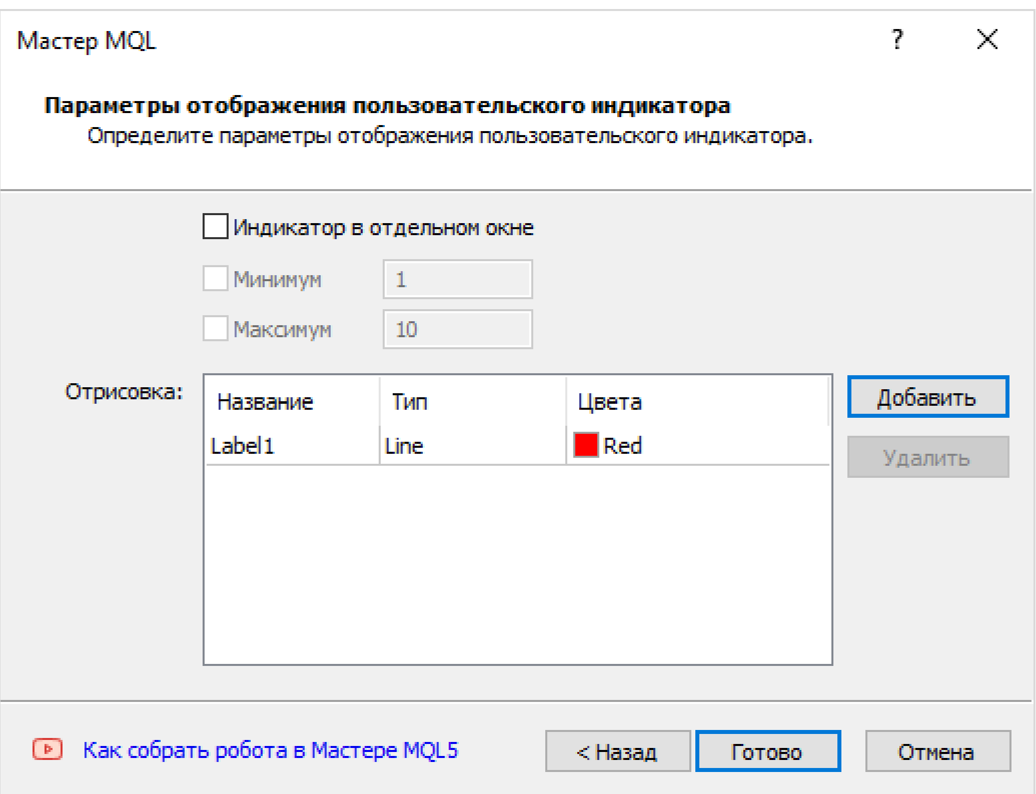
Нажав кнопку *Далее*, следует перейти ко второму шагу, где задать название файла. Здесь же можно *Добавить* входные параметры индикатора. Этот шаг не отличается от того, что было со скриптами.

На третьем шаге Мастер предлагает выбрать одну из форм обработчика *OnCalculate* и прочие опциональные обработчики событий.



Мастер MQL: выбор обработчиков событий при создании индикатора

Последний шаг позволяет определить часть графика, в котором будут отображаться линии: это может быть основное окно (по умолчанию) или отдельная панель в нижней части графика (если включить флаг *Индикатор в отдельном окне*).



Мастер MQL: выбор окна и список диаграмм при создании индикатора

С помощью кнопки *Добавить* можно составить список из нескольких графических построений и настроить их основные свойства.

Все эти термины нам уже знакомы "изнутри" и можно выбирать ту или иную опцию осознанно.

Попробуйте сгенерировать несколько вариантов индикаторов с разными включенными опциями и оцените их влияние на получающийся текст программы.

Разумеется, получив заготовку исходного кода, разработчик волен вносить произвольные правки, меняя любой из аспектов, установленных в Мастере. Это тем более актуально, что диапазон настроек Мастера — минимальный. В частности, список типов входных параметров ограничен стандартными типами MQL5, отсутствуют уровни, цветовые палитры и много другое. Кроме того, из дополнительных обработчиков событий Мастер предлагает выбрать только *OnTimer* и *OnChartEvent*, оставляя за кадром *OnBookEvent* и *OnDeinit*. Но благодаря материалу данной главы можно постепенно дополнить заготовку всем необходимым.

5.5 Использование готовых индикаторов из MQL-программ

В предыдущей главе мы научились разрабатывать собственные индикаторы. Пользователи могут размещать их на графиках и проводить с их помощью технический анализ вручную. Но это не единственный вариант применения индикаторов. MQL5 позволяет создавать экземпляры индикаторов и запрашивать их расчетные данные программным способом. Это можно сделать как из других индикаторов, комбинируя несколько простых в более сложные, так и из экспертов, реализующих автоматическую или полуавтоматическую торговлю на сигналах индикаторов.

Достаточно знать параметры индикатора, а также расположение и смысл расчетных данных в его публичных буферах, чтобы организовать построение этих новых прикладных таймсерий и получить доступ к ним.

В этой главе мы изучим функции для создания и удаления индикаторов, а также чтения их буферов. Причем это относится не только к пользовательским индикаторам, написанным на MQL5, но и к большому набору встроенных индикаторов.

Общие принципы программного взаимодействия с индикаторами включают несколько шагов:

- Создание **дескриптора** (описателя) индикатора — уникального идентификационного номера, выдаваемого системой в ответ на вызов особых функций (*iCustom* или *IndicatorCreate*), посредством которых MQL-код сообщает имя и параметры требуемого индикатора;
- Чтение данных из буферов индикатора, заданного дескриптором, с помощью функции *CopyBuffer*;
- Освобождение дескриптора (*IndicatorRelease*), если индикатор стал не нужен.

Создание и освобождение дескриптора выполняются обычно на стадиях инициализации и деинициализации программы, соответственно, а чтение буферов и их анализ производится многократно, по необходимости, например, по приходу тиков.

Во всех случаях кроме экзотических, когда требуется динамически менять настройки индикаторов по ходу выполнения программы, рекомендуется получать дескрипторы индикаторов однократно в *OnInit* или в конструкторе класса глобального объекта.

Все функции создания индикаторов имеют как минимум 2 параметра — символ и таймфрейм. Вместо символа можно передать NULL, что означает текущий инструмент, а значение 0 соответствует текущему таймфрейму, но можно пользоваться и встроенными переменными *_Symbol* и *_Period*. При необходимости можно задать произвольные символ и таймфрейм, не связанные с графиком. Таким образом, в частности, можно реализовать мультисимвольные и мультитаймфреймовые индикаторы.

Нельзя обратиться к данным индикатора сразу после создания его экземпляра, так как расчет буферов требует некоторого времени. Прежде чем читать данные, следует проверить их готовность с помощью функции *BarsCalculated* (она также принимает аргумент-дескриптор и возвращает количество просчитанных баров). В противном случае вместо данных будет получена ошибка. Хотя она не является критической в том смысле, что не приводит к остановке и выгрузке программы, но отсутствие данных сделает программу бесполезной.

Далее в данной главе будем для краткости называть создание экземпляров индикаторов и получение их дескрипторов просто "созданием индикаторов". Следует отличать его от похожего термина "создание пользовательских индикаторов", под которым мы понимали в предыдущей главе написание исходного кода индикаторов.

5.5.1 Дескрипторы и счетчики владельцев индикаторов

Итак, для программной работы с индикаторами требуется оперировать дескрипторами. Здесь можно провести параллель с дескрипторами файлов (см. раздел Открытие и закрытие файлов): там мы сообщали системе с помощью функции *FileOpen* название и режимы открытия файла, после чего дескриптор служил "пропуском" во все остальные файловые функции.

Система дескрипторов индикаторов служит нескольким целям.

Она позволяет заранее сообщить терминалу, какой индикатор запустить и какие таймсерии с его помощью рассчитать. Поскольку закачка исходных исторических данных и сам процесс расчета требуют некоторого времени (по крайней мере, при первичном запросе), а также выделения ресурсов (памяти, графики), моменты создания индикатора и его готовности разнесены. Дескриптор является связующим звеном между ними. Это — своего рода ссылка на внутренний объект терминала, хранящий набор свойств, который мы задали при создании индикатора, и актуальное состояние последнего.

Разумеется, терминалу для работы с дескрипторами необходимо поддерживать некую таблицу всех затребованных индикаторов и их свойств. Однако настоящий номер в общей таблице терминал нам не сообщает: вместо этого для каждой программы формируется свой частный список запрошенных из неё индикаторов. Записи в этом списке ссылаются на элементы общей таблицы, и дескриптор — лишь номер в списке.

Отсюда следует, что в разных программах под одними и теми же дескрипторами могут скрываться совершенно разные индикаторы, а кроме того — передавать значения дескрипторов между программами бессмысленно.

Дескрипторы являются частью системы эффективного управления ресурсами терминала за счет исключения дублирования экземпляров индикаторов с одинаковыми характеристиками, когда это возможно. Иными словами, все встроенные и пользовательские индикаторы, созданные программно, вручную или из tpl-шаблонов кэшируются.

Прежде чем создать новый экземпляр индикатора, терминал проверяет, нет ли идентичного индикатора среди находящихся в кэше. При проверке на копию применяются следующие критерии:

- Соответствие символа и периода;
- Соответствие параметров.

Для пользовательских индикаторов должны дополнительно совпадать:

- Путь на диске (как строка, без нормализации в абсолютный вид);
- График, к которому индикатор прикреплён (при создании индикатора из MQL-программы, создаваемый индикатор наследует график от создающей его программы).

Кэширование встроенных индикаторов выполняется в разрезе символа, и потому их экземпляры могут выделяться в раздельное пользование на разных графиках (с одинаковыми символом/таймфреймом).

Можно также напомнить, что создать два идентичных индикатора на одном графике вручную нельзя. Разные экземпляры программ могут запросить один и тот же индикатор, и тогда он будет создан в единственном числе и предоставляться обеим программам.

Для каждого уникального сочетания условий терминал ведет счетчик: после первого запроса на создание конкретного индикатора его счетчик равен 1, и при последующих — увеличивается на 1 (копия индикатора при этом не создается). При освобождении индикатора его счетчик уменьшается на 1. Индикатор выгружается только при обнулении счетчика, то есть когда все его владельцы явным образом откажутся от его использования.

При этом следует иметь в виду, что многократное обращение к функции-строителю индикатора с одними и теми же параметрами (включая символ/таймфрейм) в пределах одной MQL-программы не приводит к многократному увеличению счетчика ссылок — счетчик будет увеличен всего один раз. Как следствие этого, для каждого значения дескриптора достаточно одного вызова

функции освобождения (*IndicatorRelease*) — все последующие будут лишними (вернут признак ошибки, т.к. им уже нечего освобождать).

Помимо создания индикаторов с помощью *iCustom* и *IndicatorCreate* в MQL5 существует возможность получить дескриптор стороннего (уже существующего) индикатора — предназначенную для этого функцию *ChartIndicatorGet* мы изучим в главе про графики. Здесь важно отметить, что получение дескриптора таким образом тоже увеличит счетчик ссылок на него и будет препятствовать выгрузке, если дескриптор затем не освободить.

Если программа создавала подчиненные индикаторы, их дескрипторы будут автоматически освобождены (счетчик уменьшен на 1) при выгрузке этой программы, даже если функция *IndicatorRelease* не вызывалась.

5.5.2 Простой способ создания экземпляров индикаторов: iCustom

Для программного создания экземпляра индикатора MQL5 предоставляет две функции: *iCustom* и *IndicatorCreate*. Первая предполагает передачу списка параметров, который должен быть известен на момент компиляции программы. Вторая позволяет динамически формировать массив с параметрами вызываемого индикатора в процессе выполнения программы. Этот продвинутый режим будет рассмотрен в разделе Расширенный способ создания индикаторов: *IndicatorCreate*.

int iCustom(const string symbol, ENUM_TIMEFRAMES timeframe, const string pathname, ...)

Функция создает индикатор для указанного сочетания символа и таймфрейма. Значение NULL в параметре *symbol* может применяться для обозначения символа текущего графика, а значение 0 в параметре *timeframe* — для текущего периода.

В параметре *pathname* следует указать название индикатора (имя ex5-файла без расширения) и, опционально, путь. Подробнее про путь рассказано ниже.

Индикатор, на который ссылается *pathname*, должен быть откомпилирован.

Функция возвращает дескриптор индикатора или INVALID_HANDLE в случае ошибки. Дескриптор потребуется для вызова других функций, описываемых в данной главе и входящих в группу программного управления индикаторами. Дескриптор представляет собой целое число, уникально описывающее созданный экземпляр индикатора внутри вызывающей программы.

Многоточие в прототипе функции *iCustom* обозначает перечень фактических параметров для индикатора. Их типы и порядок должны соответствовать формальным параметрам (в коде индикатора), однако разрешается опускать значения, начиная с конца списка параметров. Для таких незаданных в вызывающем коде параметров внутри создаваемого индикатора будут использованы значения по умолчанию соответствующих *input*-ов.

Например, если индикатор принимает две входных переменных: период (*input int WorkPeriod = 14*) и тип цены (*input ENUM_APPLIED_PRICE WorkPrice = PRICE_CLOSE*), то можно вызвать для него *iCustom* разной степени конкретизации:

- iCustom(_Symbol, _Period, 21, PRICE_TYPICAL) — задание значений для всего списка параметров;
- iCustom(_Symbol, _Period, 21) — задание первого параметра, второй параметр опущен и получит значение PRICE_CLOSE;
- iCustom(_Symbol, _Period) — оба параметры опущены и получат значения 14 и PRICE_CLOSE;

Опустить параметр в начале или в середине списка параметров нельзя.

Если создаваемый индикатор имеет краткую форму *OnCalculate*, то последним дополнительным параметром (сверх списка входных переменных, описанных внутри индикатора) можно передавать тип цены, по которой будет строиться индикатор. Это аналог выпадающего списка *Применить к* в диалоге свойств индикатора. Кроме того, в этом дополнительном параметре можно передать дескриптор другого, созданного предварительно индикатора (см. пример далее). В таком случае, вновь создаваемый индикатор будет рассчитываться по первому буферу индикатора с указанным дескриптором. Иными словами, программист может задать расчет одного индикатора от другого.

> К сожалению, в MQL5 нет средств, чтобы программно узнать, реализован ли конкретный сторонний индикатор с использованием краткой или полной формы *OnCalculate*, то есть можно ли ему передавать дополнительный дескриптор при создании через *iCustom*. Также MQL5 не позволяет выбирать номер буфера, если индикатор, индентифицируемый дополнительным дескриптором, имеет несколько буферов.

Вернемся к параметру *pathname*.

Путем является строка, содержащая хотя бы одну обратную ('\') или прямую ('/') косую черту — это специальный символ, называемый также "слэшом" и используемый в файловой системе, как разделитель в иерархии папок и файлов. Вы можете применять как прямую, так и обратную черту, однако последняя требует "экранирования", то есть её следует писать дважды. Это связано с тем, что обратный "слэш" является управляющим символом, с помощью которого формируются многие служебные коды, такие как табуляция ('\t'), перенос на новую строку ('\n') и так далее (см. раздел Символьные типы).

Если путь начинается со слэша, он называется абсолютным и корневой папкой для него является каталог всех исходных текстов MQL5. Например, указание строки "/MyIndicator" в параметре *pathname* приведет к поиску файла *MQL5/MyIndicator.ex5*, а более длинный путь с каталогом "/Exercise/MyIndicator" сошлется на *MQL5/Exercise/MyIndicator.ex5*.

Если параметр *pathname* содержит один или более слэшей, но не начинается с него, то такой путь называется относительным, потому что он в этом случае рассматривается относительно одного из двух предопределенных мест размещения. Во-первых, файл индикатора ищется относительно той папки, где находится вызывающая MQL-программа, а если его там не удается найти, то во-вторых, поиск продолжается внутри общей папки индикаторов *MQL5/Indicators*.

В строке со слэшами тот фрагмент, что расположен правее самого правого слэша, трактуется как имя файла, а все предыдущие описывают иерархию папок. Например, путь "Folder/SubFolder/Filename" соответствует двум вложенным папкам: *SubFolder* внутри *Folder*, и внутри *SubFolder* — файл *Filename*.

Наиболее простой случай — когда в параметре *pathname* нет слэшей — задает только имя файла. Оно также рассматривается в контексте двух вышеупомянутых начальных точек поиска.

Например, советник *MyExpert.ex5* находится в папке *MQL5/Experts/Examples* и содержит вызов *iCustom(_Symbol, _Period, "MyIndicator")*. Здесь относительный путь вырожден (пуст), и присутствует только имя файла. Таким образом, поиск индикатора начинается с папки *MQL5/Experts/Examples/* и имени *MyIndicator*, что дает *MQL5/Experts/Examples/MyIndicator.ex5*. Если такой индикатор не найден в этом каталоге, поиск продолжится в корневой папке индикаторов, то есть по соединенным пути и имени *MQL5/Indicators/MyIndicator.ex5*.

Если индикатор не найден в обоих местах, функция вернет INVALID_HANDLE и установит код ошибки 4802 (ERR_INDICATOR_CANNOT_CREATE) в *_LastError*.

Более сложный случай, если *pathname* содержит не только имя, но и каталог, например, "TradeSignals/MyIndicator". Тогда указанный путь добавляется к папке вызывающей программы, в результате чего получается такая цель поиска: *MQL5/Experts/Examples/TradeSignals/MyIndicator.ex5*. Затем в случае неуспеха тот же путь добавляется к *MQL5/Indicators*, то есть ищется файл *MQL5/Indicators/TradeSignals/MyIndicator.ex5*. Если при этом в качестве разделителя решено использовать обратную косую черту, следует не забывать писать её дважды, например, *iCustom(_Symbol, _Period, "TradeSignals\\MyIndicator")*.

Для освобождения памяти компьютера от неиспользуемого больше индикатора служит функция *IndicatorRelease*, которой передается описатель этого индикатора.

Особое внимание следует уделить тестированию программы, использующей индикаторы. Если параметр *pathname* в вызове *iCustom* задан константной строкой, то соответствующий необходимый индикатор выявляется компилятором автоматически и передается тестеру вместе с тестируемой программой. В противном случае, если параметр рассчитывается в выражении или получается извне (например, через *input* от пользователя), необходимо указать в исходном коде свойство *#property tester_indicator*:

```
#property tester_indicator "indicator_name.ex5"
```

Это означает, что в составе программ можно тестировать только заранее известные пользовательские индикаторы.

Рассмотрим пример нового индикатора *UseWPR1.mq5*, который будет внутри своего обработчика *OnInit* создавать дескриптор уже известного по прошлой главе индикатора *IndWPR* (не забудьте откомпилировать *IndWPR*, так как *iCustom* загружает ex5-файлы). Получаемый в *UseWPR1* дескриптор пока никак не используется — изучим только саму возможность и проверим признак успеха. В связи с этим буфера в новом индикаторе не нужны.

```
#property indicator_separate_window
#property indicator_buffers 0
#property indicator_plots   0
```

Индикатор создаст пустое подокно, но ничего в нем не отобразит (пока) — это нормальное поведение.

Проверим несколько вариантов получения дескриптора, с разными значениями *pathname*:

1. Абсолютный путь, начинающийся с косой черты, и потому включающий всю иерархию папок (начиная от MQL5) с примерами индикаторов 5-ой главы, то есть "/Indicators/MQL5Book/p5/IndWPR";
2. Только имя "IndWPR" для поиска в той же папке, где находится и вызывающий индикатор *UseWPR1.mq5* (оба индикатора поставляются в одной папке);
3. Путь с иерархией папок примеров индикаторов относительно стандартного каталога *MQL5/Indicators*, то есть "MQL5Book/p5/IndWPR" (заметьте, что в начале нет косой черты);
4. Только имя, как в пункте 2, но для несуществующего индикатора "IndWPR NonExistent";
5. Абсолютный путь как в пункте 1, но с обратными косыми чертами без их экранирования, то есть "\Indicators\MQL5Book\p5\IndWPR";
6. Полная копия пункта 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;
}
```

Поскольку переменные с дескрипторами не используются, они объявлены локальными. Особо поясним, что хотя локальные переменные *handle* и удаляются при выходе из *OnInit*, это не влияет на дескрипторы: те продолжают существовать, пока выполняется "родительский" индикатор *UseWPR* — мы просто теряем в своем коде значения этих дескрипторов, но это не страшно, потому что они здесь нигде не используются. В примерах реальных индикаторов, которые мы рассмотрим позднее, дескрипторы, разумеется, сохраняются (как правило, в глобальных переменных) и используются.

За утечку ресурсов также не стоит переживать: при удалении индикатора *UseWPR* с графика все созданные им дескрипторы будут автоматически очищены терминалом. Более детально принципы и необходимость явного освобождения дескрипторов будут описаны в разделе об удалении экземпляров индикаторов с помощью *IndicatorRelease*.

Вышеприведенный код *OnInit* генерирует следующие записи в журнале.

```
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
```

Как мы видим, осмысленные дескрипторы 10, 11, 12, 13 получены во всех случаях кроме 4-го, с несуществующим вызываемым индикатором. При этом значение дескриптора равно -1 (INVALID_HANDLE).

Также следует обратить внимание, что при компиляции строка с вариантом N5 генерирует несколько предупреждений "неизвестная управляющая последовательность" ("unrecognized character escape sequence"). Это следствие того, что мы незаэкранировали обратную косую черту. И нам еще повезло, что инструкция выполнилась успешно, потому что если бы название какой-либо папки или файла начиналось с одной из букв в поддерживаемых управляющих последовательностях, то интерпретация последовательности нарушила бы ожидаемое прочтение названия. Например, если бы мы имели в этой же папке индикатор с именем "test" и попытались создать его через путь "MQL5Book\p5\test", то получили бы INVALID_HANDLE и ошибку 4802. Ведь '\t' — это символ табуляции, и потому терминал искал бы "MQL5Book\p5<nbsp>est". Правильная запись должна была бы быть "MQL5Book\\p5\\test". Поэтому проще пользоваться прямой косой чертой.

Также важно отметить, что хотя все успешные варианты ссылаются на один и тот же индикатор *MQL5/Indicators/MQL5Book/p5/IndWPR.ex5*, и фактически пути 1, 2, 3 и 5 эквивалентны, терминал

трактует их как разные строки, из-за чего мы получаем разные значения дескрипторов. И только вариант 6, который полностью дублирует вариант 2, возвращает идентичный дескриптор — 11.

Почему нумерация дескрипторов начинается с 10? Меньшие значения зарезервированы за системой. Как было сказано выше, для индикаторов с краткой формой *OnCalculate* последним параметром можно передавать тип цены или дескриптор другого индикатора, по буферу которого и будет рассчитываться вновь создаваемый экземпляр. Поскольку элементы перечисления ENUM_APPLIED_PRICE имеют свои значения-константы, они как раз и занимают область ниже 10. Подробнее об этом см. раздел Определение источника данных для индикатора.

В следующем примере *UseWPR2.mq5* реализуем индикатор, который создаст экземпляр *IndWPR* и проверит прогресс его расчета, пользуясь дескриптором. Но для этого следует познакомиться с новой функцией *BarsCalculated*.

5.5.3 Проверка количества просчитанных баров: BarsCalculated

Когда мы создаем сторонний индикатор — с помощью вызова *iCustom* или других функций, с которыми мы познакомимся позднее в этой главе — ему требуется некоторое время на вычисления. Как мы знаем, основным мерилом готовности данных индикатора является количество просчитанных баров, которое он возвращает из своей функции *OnCalculate*. Имея дескриптор индикатора, мы можем узнать это количество.

int BarsCalculated(int handle)

Функция возвращает количество баров, для которых рассчитаны данные в индикаторе, заданном дескриптором *handle*. В случае ошибки получим -1.

Пока данные еще не рассчитаны результат равен 0. В дальнейшем это количество следует сравнивать с размером таймсерии (например, с *rates_total*, если вызывающий индикатор проверяет *BarsCalculated* в контексте собственной функции *OnCalculate*), чтобы анализировать обработку индикатором новых баров.

В индикаторе *UseWPR2.mq5* попытаемся создать *IndWPR*, меняя период WPR во входном параметре.

```
input int WPRPeriod = 0;
```

По умолчанию он равен 0, что является некорректным значением. Оно предложено намеренно, чтобы продемонстрировать нештатную ситуацию. Напомним, что в исходном коде *IndWPR.mq5* присутствуют проверки в *OnInit* и в *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;
   ...
}
```

Таким образом, при нулевом периоде мы должны получить уведомление об ошибке и *BarsCalculated* должна всегда возвращать 0. После того как мы введем положительное значение для периода, вспомогательный индикатор должен начать рассчитываться нормально (и учитывая простоту расчета WPR — практически моментально), и *BarsCalculated* должна вернуть общее количество баров.

Теперь представим исходный код создания дескриптора в *UseWPR2.mq5*.

```
// UseWPR2.mq5
int handle; // дескриптор в глобальной переменной

int OnInit()
{
   // передаем имя и параметр
   handle = PRTF(iCustom(_Symbol, _Period, "IndWPR", WPRPeriod));
   // следующая проверка здесь бесполезна, потому что нужно выждать,
   // когда индикатор будет загружен, запустится и рассчитается
   // (здесь только для демонстрации)
   PRTF(BarsCalculated(handle));
   // успешность инициализации зависит от дескриптора
   return handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}
```

В *OnCalculate* просто выведем в журнал значения *BarsCalculated* и *rates_total*.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   // ждем, когда подчиненный индикатор рассчитается на всех барах
   if(PRTF(BarsCalculated(handle)) != PRTF(rates_total))
   {
      return prev_calculated;
   }

   // ... здесь обычно идет дальнейшая работа с использованием handle

   return rates_total;
}
```

Откомпилируем и запустим *UseWPR2* сперва с параметром 0, а затем с каким-либо допустимым значением, например, 21. Вот записи журнала для нулевого периода.

```
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
...
```

Сразу после создания дескриптора данные еще недоступны, поэтому видна ошибка INDICATOR_DATA_NOT_FOUND(4806) и результат *BarsCalculated* равен -1. Далее следует уведомление о некорректном входном параметре, что подтверждает успешную загрузку и запуск индикатора *IndWPR*. В последующем мы получаем значение *BarsCalculated*, равное 0.

Чтобы индикатор рассчитался, введем корректный входной параметр. В данном случае, *BarsCalculated* равно *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
...
```

После того как мы освоили проверку готовности подчиненного индикатора, можно приступать к чтению его данных. Займемся этим в следующем примере *UseWPR3.mq5*, где познакомимся с функцией *CopyBuffer*.

5.5.4 Получение данных таймсерии из индикатора: CopyBuffer

MQL-программа может читать данные из публичных буферов индикатора по его дескриптору. Напомним, что в пользовательских индикатрах такими буферами становятся массивы, указанные в исходном коде в вызовах функции *SetIndexBuffer*.

MQL5 API предоставляет для чтения буферов функцию *CopyBuffer*, имеющую 3 формы.

```
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[])
```

В параметре *handle* указывается дескриптор, полученный из вызова *iCustom* или других функций (см. далее разделы про *IndicatorCreate* и встроенные индикаторы). Параметр *buffer* задает индекс индикаторного буфера, из которого следует запросить данные. Нумерация ведется, начиная с 0.

Полученные элементы запрошенной таймсерии попадают в массив *array*, заданный по ссылке.

Три варианта функции различаются способом задания диапазона временных меток (*start/stop*) или номеров (*offset*) и количества (*count*) баров, для которых получаются данные. Принципы работы с этими параметрами полностью соответствуют тому, что мы изучали в Обзоре Copy-функций для получения массивов котировок. В частности, отсчет элементов копируемых данных в *offset* и *count* ведется от настоящего к прошлому, то есть стартовая позиция, равная 0, означает текущий бар. А в приемном массиве *array* элементы физически располагаются в порядке от прошлого к настоящему (однако эту адресацию можно поменять на логическом уровне на обратную с помощью *ArraySetAsSeries*).

В принципе *CopyBuffer* является аналогом функций для чтения встроенных таймсерий типа *CopyOpen*, *CopyClose* и других. Основное отличие заключается в том, что таймсерии с котировками формирует сам терминал, а таймсерии в индикаторных буферах рассчитывают пользовательские или встроенные индикаторы. Кроме того, конкретную пару символа и таймфрейма, которые определяют и идентифицируют таймсерию, в случае индикаторов мы задаем заранее — в функции создания дескриптора вроде *iCustom*, а в *CopyBuffer* эта информация передается опосредованно — через дескриптор *handle*.

При копировании заранее неизвестного количества данных в качестве массива-приемника желательно использовать динамический массив — тогда функция *CopyBuffer* распределит размер принимающего массива под размер копируемых данных. Если необходимо многократно копировать заранее известное количество данных, то лучше это делать в статически выделенный буфер (локальный с модификатором *static* или фиксированного размера в глобальном контексте), чтобы избежать повторных операций выделения памяти.

Если в качестве принимающего массива выступает индикаторный буфер (массив, предварительно зарегистрированный в системе функцией *SetIndexBufer*), то индексация в таймсерии и приемном буфере совпадают (при условии запроса для той же пары символ/таймфрейм). В этом случае легко реализовать частичное заполнение приемника (в частности, это используется для обновления последних баров, см. пример далее). Если же символ или таймфрейм запрашиваемой таймсерии не совпадают с символом и/или таймфреймом текущего графика, то функция вернет не больше элементов, чем минимальное количество баров в этих двух: источнике и приемнике.

Если в качестве аргумента *array* передан обычный массив (не буфер), то функция заполнит его, начиная с первых элементов — целиком (в случае динамического) или частично (в случае статического с превышением размера). Поэтому если необходимо произвести частичное копирование значений индикатора в произвольное место другого массив, то для этих целей необходимо использовать промежуточный массив, в который копируется требуемое количество элементов, а уже из него они переносятся в окончательное место назначения.

Функция возвращает количество скопированных элементов или -1 в случае ошибки, включая и временное отсутствие готовых данных.

Поскольку индикаторы, как правило, напрямую или опосредованно зависят от ценовых таймсерий, их расчет стартует не ранее, чем будут синхронизированы котировки. В связи с этим следует учитывать технические особенности организации и хранения таймсерий в терминале и быть готовым, что запрашиваемые данные появятся не сразу. В частности, мы можем получить 0 или количество, меньше запрошенного — все такие случаи следует обрабатывать согласно обстоятельствам, например, ждать построения или сообщать о проблеме пользователю.

> Если запрашиваемые таймсерии еще не построены или их необходимо загрузить с сервера, то функция ведет себя по-разному в зависимости от типа MQL-программы, из которой она вызывается.
>
> При запросе еще неготовых данных из индикатора, функция сразу же вернет -1, но при этом сам процесс загрузки и построения таймсерий будет инициирован.
>
> При запросе данных из эксперта или скрипта, будет инициирована загрузка с сервера и/или начнется построение нужной таймсерии, если данные можно построить из локальной истории, но они еще не готовы. Функция вернет то количество данных, которые будут готовы к моменту истечения таймаута (45 секунд), отведенного на синхронное выполнение функции (вызывающий код при этом ожидает завершения работы функции).

Обратите внимание, что функция *CopyBuffer* может читать данные из буферов вне зависимости от режима их работы — INDICATOR_DATA, INDICATOR_COLOR_INDEX, INDICATOR_CALCULATIONS — два последних скрыты от пользователя.

Также важно отметить, что в вызываемом индикаторе может быть установлен сдвиг таймсерии с помощью свойства PLOT_SHIFT и он влияет на смещение считываемых данных с помощью *CopyBuffer*. Например, если линии индикатор сдвинуты в будущее на N баров то в параметрах *CopyBuffer* (первой формы) нужно задавать *offset* равным (- N), то есть с минусом, так как текущий бар таймсерии имеет индекс 0, а индексы будущих баров со сдвигом уменьшаются на единицу на каждом баре. В частности, такая ситуация возникает с индикатором *Gator*, потому что его нулевая диаграмма сдвинута вперед на значение параметр *TeethShift*, а первая диаграмма — на значение параметра *LipsShift*. На максимальное из них и нужно делать поправку. Пример будет представлен в разделе Чтение данных из диаграмм, имеющих сдвиг.

К сожалению, MQL5 не предоставляет программных средств узнать свойство PLOT_SHIFT у стороннего индикатора, поэтому при необходимости придется запрашивать эту информацию у пользователя через входную переменную.

Работать с *CopyBuffer* из кода экспертов нам придется в главе про эксперты, а пока ограничимся индикаторами.

Продолжим развивать пример со вспомогательным индикатором *IndWPR*. На этот раз в версии *UseWPR3.mq5* предусмотрим индикаторный буфер и заполним его данными из *IndWPR* с помощью *CopyBuffer*. Для этого применим директивы с количеством буферов и настройками отрисовки.

```
#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"
```

В глобальном контексте опишем входной параметр с периодом WPR, массив для буфера и переменную с дескриптором.

```
input int WPRPeriod = 14;

double WPRBuffer[];

int handle;
```

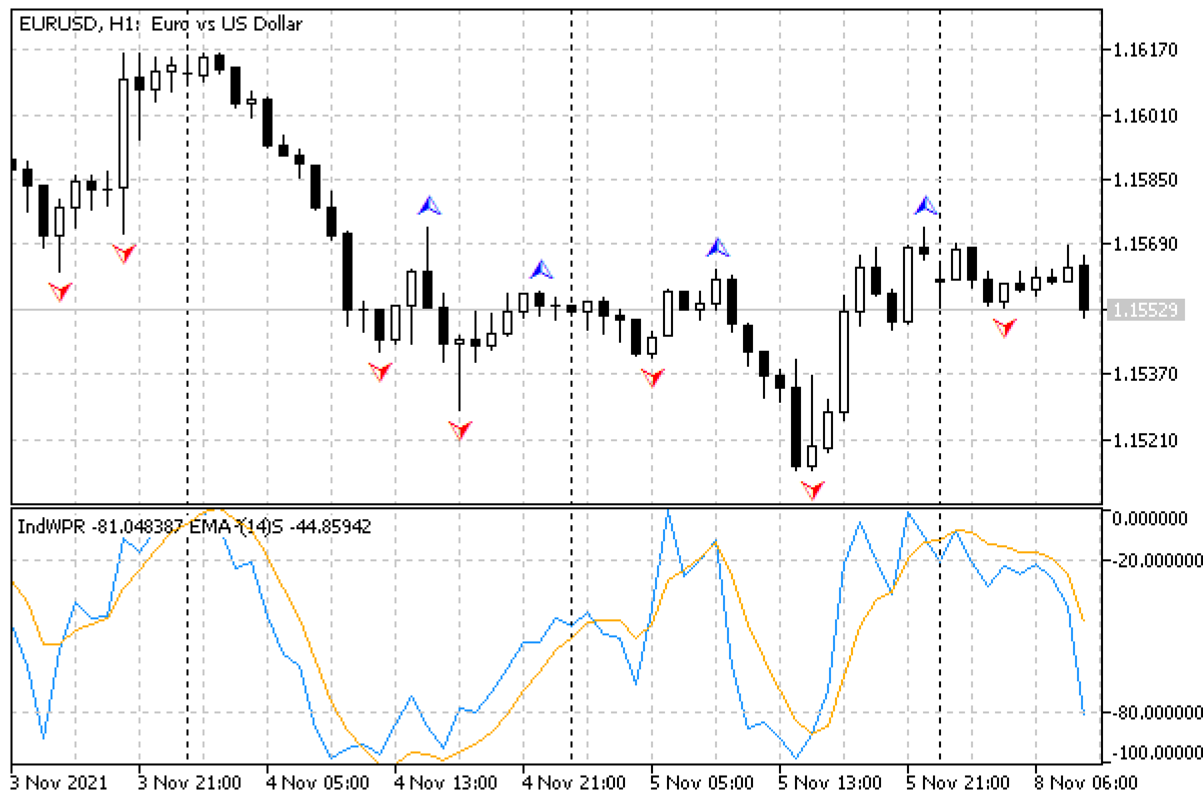
Обработчик *OnInit* практически не изменится: добавился только вызов *SetIndexBuffer*.

```
int OnInit()
{
   SetIndexBuffer(0, WPRBuffer);
   handle = iCustom(_Symbol, _Period, "IndWPR", WPRPeriod);
   return handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}
```

В *OnCalculate* будем копировать данные без преобразований.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   // ждем готовности расчета для всех баров
   if(BarsCalculated(Handle) != rates_total)
   {
      return prev_calculated;
   }

   // копируем таймсерию подчиненного индикатора целиком или на новых барах в наш буф
   const int n = CopyBuffer(handle, 0, 0, rates_total - prev_calculated + 1, WPRBuffe
   // в случае отсутствия ошибок наши данные готовы для всех баров rates_total
   return n > -1 ? rates_total : 0;
}
```

Если откомпилировать и запустить *UseWPR3*, мы получим фактически копию исходного WPR за исключением настройки уровней, точности отображения чисел и заголовка. В качестве проверки работоспособности самого механизма это сойдет, но обычно новые индикаторы, строящиеся на одном или нескольких вспомогательных индикаторах, предлагают некую собственную идею и преобразование данных. Поэтому разработаем другой индикатор, выдающий торговые сигналы на покупку и продажу (с позиции трейдинга не стоит рассматривать их как образец — это задача по программированию). Суть его композиции можно пояснить с помощью следующей картинки.



Индикаторы IndWPR, IndTripleEMA, IndFractals

Используем выход WPR из зон перекупленности и перепроданности как рекомендацию, соответственно, продавать и покупать. Чтобы сигналы не реагировали на случайные флуктуации, применим к WPR тройное скользящее среднее и будем проверять уже её значение на пересечение границ верхней и нижней зон.

В качестве фильтра к этим сигналам будем проверять, какой фрактал был последним перед этим моментом: фрактал-вершина означает разворот цены вниз и подтверждает продажу, а фрактал-впадина означает разворот вверх и потому подкрепляет покупку. По определению фракталы появляются с отставанием на количество баров, равное порядку фракталов.

Новый индикатор доступен в файле *UseWPRFractals.mq5*.

Нам потребуется три буфера: два сигнальных и еще один для фильтра. Последний мы могли бы оформить в режиме INDICATOR_CALCULATIONS. Вместо этого сделаем его стандартным INDICATOR_DATA, но со стилем DRAW_NONE — таким образом он не будет мешаться на графике, но его значения видны в Окне данных.

Сигналы будем отображать на основном графике (на ценах *Close*, по умолчанию), поэтому используем директиву *indicator_chart_window*. Это не мешает нам вызывать индикаторы типа WPR, которые предназначены для отдельного окна, поскольку все подчиненные индикаторы способны рассчитаться без визуализации. При необходимости мы можем вывести их на график, но поговорим об этом в главе про графики (см. *ChartIndicatorAdd*).

```
#property indicator_chart_window
#property indicator_buffers 3
#property indicator_plots   3
// настройки отрисовки буферов
#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"
```

Во входных переменных предусмотрим возможность указать период WPR, период усреднения (сглаживания) и порядок фракталов — это все параметры подчиненных индикаторов. Кроме того заведем переменную *Offset* с номером бара, на котором будут анализироваться сигналы. Значение 0 (по умолчанию) означает текущий бар и анализ в потиковом режиме (внимание: сигналы на последнем баре могу перерисовываться — некоторые трейдеры этого не любят). Если сделать *Offset* равным 1, будем анализировать уже сформированные бары, и такие сигналы не меняются.

```
input int PeriodWPR = 11;
input int PeriodEMA = 5;
input int FractalOrder = 1;
input int Offset = 0;
input double Threshold = 0.2;
```

Переменная *Threshold* определяет размер зон перекупленности и перепроданности как часть от ±1.0 (в каждую из сторон). Например, если следовать классическим настройкам WPR с уровнями -20 и -80 на шкале от 0 до -100, то *Threshold* должен быть равен 0.4.

Для индикаторных буферов заведены следующие массивы.

```
double UpBuffer[];   // верхний сигнал значит перекупленность, то есть продажа
double DownBuffer[]; // нижний сигнал значит перепроданность, то есть покупку
double Filter[];     // направление фильтра по фракталам +1 (вверх/покупка), -1 (вниз
```

Дескрипторы индикаторов сохраним в глобальных переменных.

```
int handleWPR, handleEMA3, handleFractals;
```

Всю настройку выполним, как обычно, в *OnInit*. Поскольку функция *CopyBuffer* использует индексацию от настоящего к прошлому, мы для единообразия чтения данных взводим флаг "серийности" (*ArraySetAsSeries*) у всех массивов.

```
int OnInit()
{
   // привязка буферов
   SetIndexBuffer(0, UpBuffer);
   SetIndexBuffer(1, DownBuffer);
   SetIndexBuffer(2, Filter, INDICATOR_DATA); // вариант: INDICATOR_CALCULATIONS
   ArraySetAsSeries(UpBuffer, true);
   ArraySetAsSeries(DownBuffer, true);
   ArraySetAsSeries(Filter, true);

   // сигналы стрелочки
   PlotIndexSetInteger(0, PLOT_ARROW, 234);
   PlotIndexSetInteger(1, PLOT_ARROW, 233);

   // подчиненные индикаторы
   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;
}
```

В вызовах *iCustom* следует обратить внимание, как создается *handleEMA3*. Поскольку эта средняя должна считаться по WPR, мы передаем дескриптор *handleWPR* (полученный в предыдущем вызове *iCustom*) последним параметром, после фактических параметров индикатора *IndTripleEMA*. При этом мы обязаны указать полный список входных параметров *IndTripleEMA* (напомним, что там параметрами являются *int InpPeriodEMA* и *BEGIN_POLICY InpHandleBegin* — второй параметр мы использовали для изучения пропуска начальных баров и сейчас он нам в принципе не важен, но мы обязаны его передать, в данном случае просто как 0). Если бы мы опустили второй параметр в вызове, как несущественный в текущем контексте применения, то переданный следом дескриптор *handleWPR* был бы принят в вызванном индикаторе за *InpHandleBegin*. В результате *IndTripleEMA* применился бы к обычной цене *Close*.

Когда нам не нужно передавать дополнительный дескриптор, синтаксис вызова *iCustom* позволяет опускать произвольное количество последних параметров, при этом они получат значения по умолчанию из исходного кода.

В обработчике *OnCalculate* ожидаем готовности индикаторов WPR и фракталов, а затем рассчитываем сигналы на всей истории или последнем баре с помощью вспомогательной функции *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) // первый запуск
   {
      ArrayInitialize(UpBuffer, EMPTY_VALUE);
      ArrayInitialize(DownBuffer, EMPTY_VALUE);
      ArrayInitialize(Filter, 0);

      // ищем сигналы по всей истории
      for(int i = rates_total - FractalOrder - 1; i >= 0; --i)
      {
         MarkSignals(i, Offset, data);
      }
   }
   else // онлайн
   {
      for(int i = 0; i < rates_total - prev_calculated; ++i)
      {
         UpBuffer[i] = EMPTY_VALUE;
         DownBuffer[i] = EMPTY_VALUE;
         Filter[i] = 0;
      }

      // ищем сигналы на новом баре или каждом тике (если Offset == 0)
      if(rates_total != prev_calculated
      || Offset == 0)
      {
         MarkSignals(0, Offset, data);
      }
   }

   return rates_total;
}
```

Интересующая нас в первую очередь работа с функцией *CopyBuffer* скрыта в *MarkSignals*. Значения сглаженного WPR будем читать в массив *wpr[2]*, а фракталы — в массивы *peaks[1]* и *hollows[1]*.

```
int MarkSignals(const int bar, const int offset, const double &data[])
{
   double wpr[2];
   double peaks[1], hollows[1];
   ...
```

Заполняем локальные массивы с помощью трех вызовов *CopyBuffer*. Обратите внимание, что нам не нужны напрямую показания *IndWPR*, потому что он участвует в расчетах *IndTripleEMA*. Мы читаем в массив *wpr* данные через дескриптор *handleEMA3*. Также важно, что во фрактальном индикаторе существует 2 буфера, и потому функция *CopyBuffer* вызывается два раза с разными индексами 0 и 1 для массивов *peaks* и *hollows*, соответственно. Фрактальные массивы читаются с отступом *FractalOrder*, потому что фрактал может сформироваться только на баре, у которого не только слева, но и справа имеется предопределенное количество баров.

```
   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)
   {
      ...
```

Далее берем с предыдущего бара буфера *Filter* прежнее направление фильтра (в начале истории там 0, но при появлении фрактала вверх или вниз мы пишем туда +1 или -1, это видно в исходном коде чуть ниже) и изменяем его соответствующим образом при обнаружении любого нового фрактала.

```
      int filterdirection = (int)Filter[bar + 1];
      
      // последний фрактал задает разворотное движение
      if(peaks[0] != EMPTY_VALUE)
      {
         filterdirection = -1; // sell
      }
      if(hollows[0] != EMPTY_VALUE)
      {
         filterdirection = +1; // buy
      }
      
      Filter[bar] = filterdirection; // запоминаем текущее направление
```

Наконец, анализируем переход сглаженного WPR из верхней или нижней зоны в среднюю, с учетом ширины зон, заданной во *Threshold*.

```
         // переводим 2 величины WPR в диапазон [-1,+1]
         const double old = (wpr[0] + 50) / 50;     // +1.0 -1.0
         const double last = (wpr[1] + 50) / 50;    // +1.0 -1.0

         // отскок от верха вниз
         if(filterdirection == -1
         && old >= 1.0 - Threshold && last <= 1.0 - Threshold)
         {
            UpBuffer[bar] = data[bar];
            return -1; // продажа
         }

         // отскок от низа вверх
         if(filterdirection == +1
         && old <= -1.0 + Threshold && last >= -1.0 + Threshold)
         {
            DownBuffer[bar] = data[bar];
            return +1; // покупка
         }
      }
      return 0; // сигнала нет
   }
```

Ниже представлен скриншот получившегося индикатора на графике.

Сигнальный индикатор UseWPRFractals на основе WPR, EMA3 и фракталов

5.5.5 Поддержка множества символов и таймфреймов

До сих пор во всех примерах индикаторов мы создавали дескрипторы для той же пары символа и таймфрейма, что и на текущем графике. Однако такого ограничения не существует. Мы можем создавать вспомогательные индикаторы на любых символах и таймфреймах. Разумеется, при этом необходимо дожидаться готовности "чужих" таймсерий, как мы уже делали ранее, например, по таймеру.

Реализуем мультитаймфреймовый индикатор WPR (см. файл *UseWPRMTF.mq5*), которому также можно назначить расчет на произвольном символе (отличном от графика).

Станем выводить значения WPR заданного периода для всех стандартных таймфреймов из перечисления ENUM_TIMEFRAMES. Количество таймфреймов равно 21, поэтому индикатор будет всегда отображаться на последних 21 барах. Самый правый нулевой бар будет содержать WPR для M1, следующий — WPR для M2, и так далее вплоть до 20-го бара с WPR для месячного таймфрейма. Для облегчения восприятия, раскрасим диаграммы разным цветом: минутные таймфреймы — красным, часовые — зеленым, а дневные и более старшие — синим.

Поскольку в индикаторе можно будет задать рабочий символ, и создать несколько копий для разных символов на одном графике, выберем стиль отрисовки DRAW_ARROW и предусмотрим входной параметр для назначения символа — таким образом, можно будет отличить показания для разных символов. Цветовая раскраска требует дополнительный буфер.

```
#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"
```

Значения WPR преобразуем в диапазон [-1,+1]. Масштаб подокна выберем с некоторым запасом от диапазона. Уровни со значениями ±0.6 соответствуют стандартным -20 и -80 до преобразования 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
```

Во входных переменных: период WPR, рабочий символ и код выводимой "стрелки". Когда символ оставлен пустым, используется символ текущего графика.

```
input int WPRPeriod = 14;
input string WorkSymbol = ""; // Symbol
input int Mark = 0;

const string _WorkSymbol = (WorkSymbol == "" ? _Symbol : WorkSymbol);
```

Для удобства кодирования набор таймфреймов перечислен в массиве *TF*.

```
#define TFS 21

ENUM_TIMEFRAMES TF[TFS] =
{
   PERIOD_M1,
   PERIOD_M2,
   PERIOD_M3,
   ...
   PERIOD_D1,
   PERIOD_W1,
   PERIOD_MN1,
};
```

Дескрипторы индикаторов для каждого таймфрейма хранятся в массиве *Handle*.

```
int Handle[TFS];
```

Настройку индикаторных буферов и получение дескрипторов выполним в *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;
}
```

Расчет в *OnCalculate* — по привычной схеме: ожидание готовности данных, инициализация, заполнение на новых барах. Непосредственную работу с дескрипторами выполняют вспомогательные функции *IsDataReady* и *FillData* (см. ниже).

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   // ожидание готовности подчиненных индикаторов
   if(!IsDataReady())
   {
      EventSetTimer(1); // если не готово, откладываем расчет
      return prev_calculated;
   }
   if(prev_calculated == 0) // инициализация
   {
      ArrayInitialize(WPRBuffer, EMPTY_VALUE);
      ArrayInitialize(Colors, EMPTY_VALUE);
      // постоянные цвета для последних TFS баров
      for(int i = 0; i < TFS; ++i)
      {
         Colors[i] = i < 11 ? 0 : (i < 18 ? 1 : 2);
      }
   }
   else // подготовка нового бара
   {
      for(int i = prev_calculated; i < rates_total; ++i)
      {
         WPRBuffer[i] = EMPTY_VALUE;
         Colors[i] = 0;
      }
   }

   if(prev_calculated != rates_total) // новый бар
   {
      // чистим метку на самом старом баре, сдвинувшемся влево за TFS баров
      WPRBuffer[TFS] = EMPTY_VALUE;
      // обновляем раскраску баров
      for(int i = 0; i < TFS; ++i)
      {
         Colors[i] = i < 11 ? 0 : (i < 18 ? 1 : 2);
      }
   }

   // копируем данные из подчиненных индикаторов в свой буфер
   FillData();
   return rates_total;
}
```

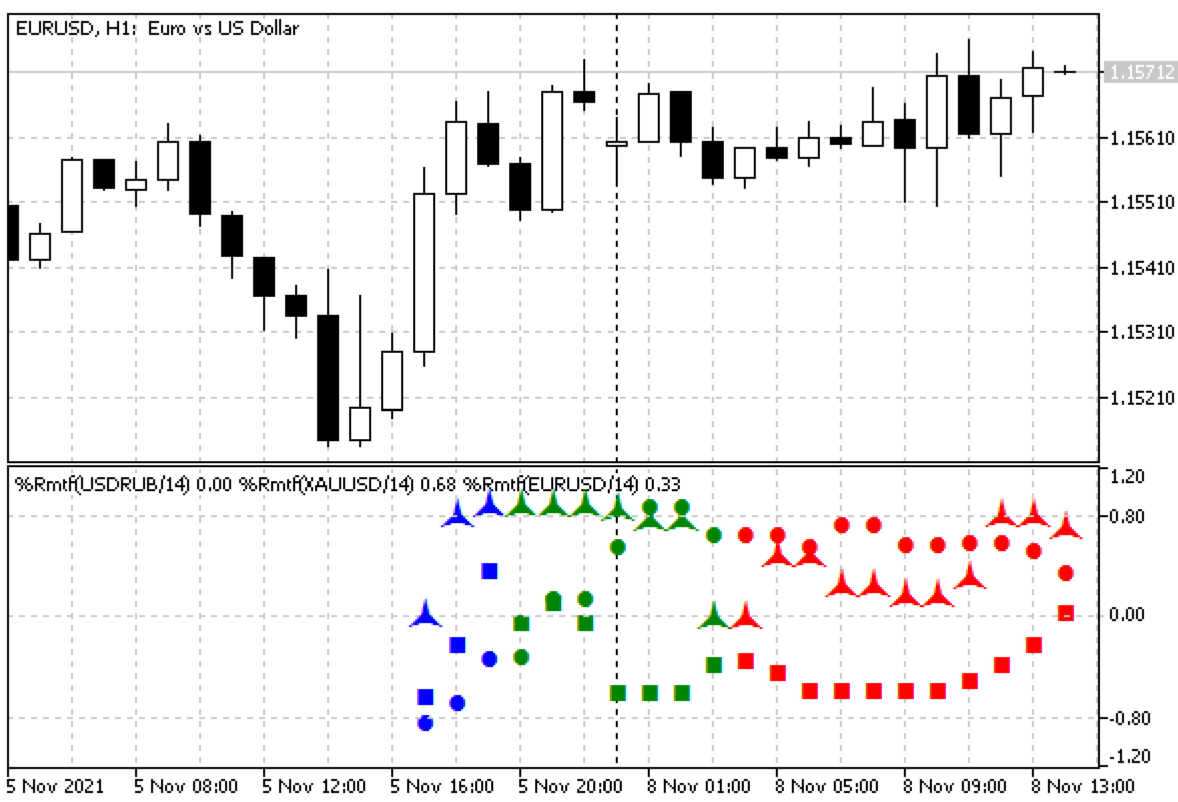
При необходимости инициируем пересчет по таймеру.

```
void OnTimer()
{
   ChartSetSymbolPeriod(0, _Symbol, _Period);
   EventKillTimer();
}
```

А вот и сами функции *IsDataReady* и *FillData* (с некоторыми сокращениями).

```
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];
      // берем последнее актуальное значение (буфер 0, индекс 0)
      if(CopyBuffer(Handle[i], 0, 0, 1, data) == 1)
      {
         WPRBuffer[i] = (data[0] + 50) / 50;
      }
   }
}
```

Откомпилируем индикатор и посмотрим, как он выглядит на графике. Например, создадим три копии для EURUSD, USDRUB и XAUUSD.

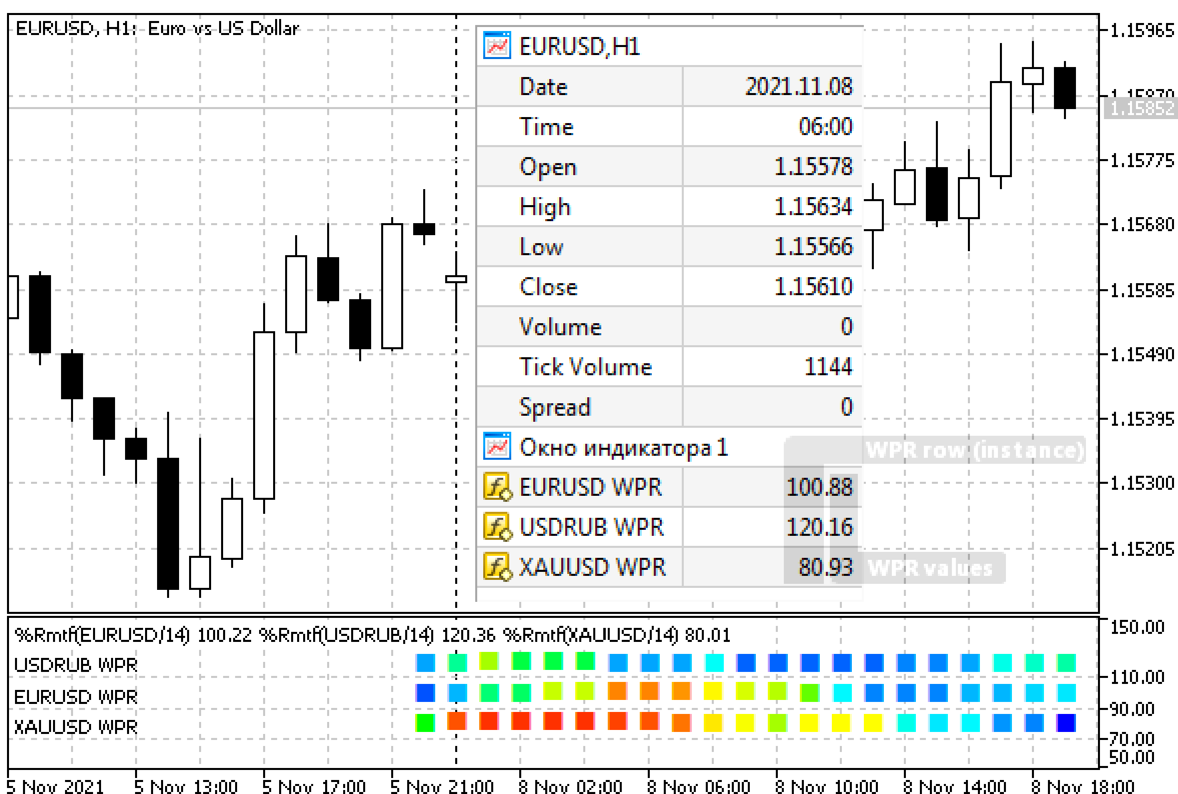


Три экземпляра мультитаймфреймового WPR для разных рабочих символов

При первом расчете индикатор может потребовать заметного времени на подготовку таймсерий по всем таймфреймам.

Точно такой же в плане расчетной части индикатор *UseWPRMTFDashboard.mq5* оформлен в виде популярной у трейдеров панели. В нем для каждого символа предполагается задать собственный вертикальный отступ в параметре *Level*, на котором в виде линейки маркеров выводятся значения WPR всех таймфреймов, причем значения кодируются цветом. В данной версии значения WPR нормируются в диапазон [0..1], поэтому использование линеек на уровнях, отстоящих друг от друга на несколько десятков (например, на 20, как на скриншоте ниже) позволяет разместить в подокне несколько экземпляров индикатора без наложений (80, 100, 120 и т.д.). Каждая копия — для своего рабочего символа. Кроме того, за счет того что *Level* больше 1.0, а значения WPR — меньше, они видны в значениях в *Окне данных* раздельно: слева и справа от десятичной точки.

Подписи для линеек меток обеспечиваются уровнями, динамически добавляемыми в *OnInit*.



Панель из трех линейчатых мультитаймфреймовых WPR для разных рабочих символов

Изучить исходный код *UseWPRMTFDashboard.mq5* и сравнить его с *UseWPRMTF.mq5* предлагается самостоятельно. Для генерации палитры цветовых оттенков используется файл *ColorMix.mqh*.

После того как мы изучим встроенные индикаторы, в том числе *iWPR*, можно заменить пользовательский *IndWPR* на встроенный *iWPR*.

Об эффективности и ресурсоемкости составных индикаторов

Продемонстрированный выше подход с генерацией множества вспомогательных индикаторов не является эффективным по быстродействию и по потреблению ресурсов. Это в первую очередь пример интеграции MQL-программ и обмена данными между ними. Но как и любую технологию, её следует использовать уместно.

Каждый из двух созданных индикаторов рассчитывает WPR на всех барах таймсерии, а в вызывающий индикатор затем берется только одно последнее значение. Мы зря расходуем и память, и время процессора.

Если имеется исходный код вспомогательных индикаторов или известен принцип их работы, наиболее оптимальным является перенос расчетного алгоритма внутрь главного индикатора (или эксперта) и его применение для ограниченной, ближайшей истории минимально необходимой глубины.

В некоторых случаях можно обойтись без обращения к старшим таймфреймам, выполняя эквивалентные расчеты на текущем таймфрейме: например, вместо размаха цены на 14 дневных барах (что требует построения полной таймсерии D1) можно взять размах на 14*24 барах H1, при условии круглосуточной торговли и запуска индикатора на графике H1.

Вместе с тем, когда в торговой системе используется коммерческий индикатор (без исходного

кода), получить из него данные можно только по открытым программным интерфейсам. В этом случае, создание дескриптора и затем чтение данных из индикаторного буфера через *CopyBuffer* — единственный, но при этом удобный, универсальный способ. Просто всегда следует иметь в виду, что вызов функций API — это более "дорогая" операция, чем манипуляции с собственным массивом внутри MQL-программы и вызовы локальных функций. Если требуется держать открытыми много терминалов, в каждом, вероятно, по набору таких неоптимизированных MQL-программ, и плюс ко всему это происходит с ограничениями на ресурсы, то вероятно падение быстродействия.

5.5.6 Обзор встроенных индикаторов

Как известно, терминал содержит большой набор популярных индикаторов, которые доступны также и через программное API — без необходимости реализовывать их алгоритмы на MQL5. Создаются они с помощью встроенных функций аналогичных *iCustom*. Например, мы ранее в обучающих целях создали собственные версии WPR и тройной скользящей средней EMA. Однако соответствующие индикаторы можно использовать прямо "из коробки": для них существуют функции *iWPR* и *iTEMA*. Эти и другие технические индикаторы перечислены в нижеприведенной таблице.

Все встроенные индикаторы принимают в качестве первых двух параметров строку с рабочим символом и таймфрейм, а также возвращают целое число — дескриптор индикатора. В общем виде прототип всех функций выглядит так:

```
int iFunction(const string symbol, ENUM_TIMEFRAMES timeframe, ...)
```

Вместо многоточия следуют специфические параметры конкретного индикатора. Их количество и типы отличаются. Некоторые индикаторы не имеют параметров.

Например, WPR имеет один параметр, как и в нашей самодельной версии — период: *int iWPR(const string symbol, ENUM_TIMEFRAMES timeframe, int period)*. А встроенный индикатор фракталов, в отличие от нашей версии, не имеет специальных параметров: *int iFractals(const string symbol, ENUM_TIMEFRAMES period)*. В данном случае порядок фракталов жестко задан и равен 2, то есть перед экстремумом (вершиной или впадиной) и за ним должно находиться как минимум по два бара с менее выраженными ценами *High* и *Low*, соответственно.

Допустимо вместо символа задавать значение NULL — оно означает рабочий инструмент текущего графика, а значение 0 в параметре *timeframe* соответствует текущему таймфрейму графика, поскольку это также и значение PERIOD_CURRENT в перечислении ENUM_TIMEFRAMES (см. раздел Символы и таймфреймы).

Также следует иметь в виду, что разные типы индикаторов имеют разное количество буферов. Например, у скользящей средней или WPR только один буфер, а у фракталов — два. Количество буферов также отмечено в таблице в отдельной колонке.

Функция	Название индикатора	Параметры	Буферы
iAC	Accelerator Oscillator	—	1*
iAD	Accumulation / Distribution	ENUM_APPLIED_VOLUME volume	1*
iADX	Average Directional Index	int period	3*

Функция	Название индикатора	Параметры	Буферы
iADXWilder	Average Directional Index by Welles Wilder	int period	3*
iAlligator	Alligator	int jawPeriod, int jawShift, int teethPeriod,int teethShift, int lipsPeriod, int lipsShift, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price	3
iAMA	Adaptive Moving Average	int period, int fast, int slow, int shift, ENUM_APPLIED_PRICE price	1
iAO	Awesome Oscillator	—	1*
iATR	Average True Range	int period	1*
iBands	Bollinger Bands	int period, int shift, double deviation, ENUM_APPLIED_PRICE price	3
iBearsPower	Bears Power	int period	1*
iBullsPower	Bulls Power	int period	1*
iBWMFI	Market Facilitation Index by Bill Williams	ENUM_APPLIED_VOLUME volume	1*
iCCI	Commodity Channel Index	int period, ENUM_APPLIED_PRICE price	1*
iChaikin	Chaikin Oscillator	int fast, int slow, ENUM_MA_METHOD method, ENUM_APPLIED_VOLUME volume	1*
iDEMA	Double Exponential Moving Average	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	Force Index	int period, ENUM_MA_METHOD method, ENUM_APPLIED_VOLUME volume	1*
iFractals	Fractals	—	2
iFrAMA	Fractal Adaptive Moving Average	int period, int shift, ENUM_APPLIED_PRICE price	1
iGator	Gator Oscillator	int jawPeriod, int jawShift, int teethPeriod,int teethShift, int	4*

Функция	Название индикатора	Параметры	Буферы
		lipsPeriod, int lipsShift, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price	
iIchimoku	Ichimoku Kinko Hyo	int tenkan, int kijun, int senkou	5
iMomentum	Momentum	int period, ENUM_APPLIED_PRICE price	1*
iMFI	Money Flow Index	int period, ENUM_APPLIED_VOLUME volume	1*
iMA	Moving Average	int period, int shift, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price	1
iMACD	Moving Averages Convergence-Divergence	int fast, int slow, int signal, ENUM_APPLIED_PRICE price	2*
iOBV	On Balance Volume	ENUM_APPLIED_VOLUME volume	1*
iOsMA	Moving Average of Oscillator (MACD histogram)	int fast, int slow, int signal, ENUM_APPLIED_PRICE price	1*
iRSI	Relative Strength Index	int period, ENUM_APPLIED_PRICE price	1*
iRVI	Relative Vigor Index	int period	1*
iSAR	Parabolic Stop And Reverse System	double step, double maximum	1
iStdDev	Standard Deviation	int period, int shift, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price	1*
iStochastic	Stochastic Oscillator	int Kperiod, int Dperiod, int slowing, ENUM_MA_METHOD method, ENUM_APPLIED_PRICE price	2*
iTEMA	Triple Exponential Moving Average	int period, int shift, ENUM_APPLIED_PRICE price	1
iTriX	Triple Exponential Moving Averages Oscillator	int period, ENUM_APPLIED_PRICE price	1*
iVIDyA	Variable Index Dynamic Average	int momentum, int smooth, int shift, ENUM_APPLIED_PRICE price	1
iVolumes	Volumes	ENUM_APPLIED_VOLUME volume	1*
iWPR	Williams Percent Range	int period	1*

В правой колонке звездочкой * обозначены индикаторы с собственным окном (они выводятся под основным графиком).

Наиболее часто используются параметры задающие периоды индикатора (*period*, *fast*, *slow* и прочие вариации), а также сдвиг линий *shift* — когда он положителен, диаграммы сдвигаются вправо, когда отрицателен — влево, на заданное количество баров.

Многие параметры имеют типы прикладных перечислений ENUM_APPLIED_PRICE, ENUM_APPLIED_VOLUME, ENUM_MA_METHOD. Мы уже знакомились с ENUM_APPLIED_PRICE в разделе Перечисления. Представим здесь его и другие типы более формально, в виде таблиц с описаниями.

Идентификатор	Описание	Значение
PRICE_CLOSE	Цена закрытия бара	1
PRICE_OPEN	Цена открытия бара	2
PRICE_HIGH	Максимальная цена на баре	3
PRICE_LOW	Минимальная цена на баре	4
PRICE_MEDIAN	Медианная цена, (high+low)/2	5
PRICE_TYPICAL	Типичная цена, (high+low+close)/3	6
PRICE_WEIGHTED	Средневзвешенная цена, (high+low+close+close)/4	7

Индикаторы, работающие с объемами, могут оперировать тиковыми объемами (фактически, это счетчик тиков) или реальными объемами (они, как правило, доступны только для биржевых инструментов). Оба типа сведены в перечисление ENUM_APPLIED_VOLUME.

Идентификатор	Описание	Значение
VOLUME_TICK	Тиковый объем	0
VOLUME_REAL	Торговый объем	1

В основе многих технических индикаторов лежат методы сглаживания (или усреднения — от английского "Moving Average") таймсерий. Терминал поддерживает 4 наиболее распространенных метода сглаживания, которые задаются в MQL5 с помощью элементов перечисления ENUM_MA_METHOD.

Идентификатор	Описание	Значение
MODE_SMA	Простое усреднение	0
MODE_EMA	Экспоненциальное усреднение	1
MODE_SMMA	Сглаженное усреднение	2
MODE_LWMA	Линейно-взвешенное усреднение	3

Для индикатора Стохастик, пример использования которого мы рассмотрим в следующем разделе, существует два варианта расчета: по ценам *Close* или по ценам *High/Low*. Это отражено в специальном перечислении ENUM_STO_PRICE.

Идентификатор	Описание	Значение
STO_LOWHIGH	Построение по ценам Low/High	0
STO_CLOSECLOSE	Построение по ценам Close/Close	1

Назначение и нумерация буферов для тех индикаторов, которые имеют больше одного буфера, приведено в следующей таблице.

Индикаторы	Константы	Описания	Значение
ADX, ADXW			
	MAIN_LINE	Основная линия	0
	PLUSDI_LINE	Линия +DI	1
	MINUSDI_LINE	Линия –DI	2
iAlligator			
	GATORJAW_LINE	Линия челюстей	0
	GATORTEETH_LINE	Линия зубов	1
	GATORLIPS_LINE	Линия губ	2
iBands			
	BASE_LINE	Основная линия	0
	UPPER_BAND	Верхняя граница	1
	LOWER_BAND	Нижняя граница	2
iEnvelopes, iFractals			
	UPPER_LINE	Верхняя линия	0
	LOWER_LINE	Нижняя линия	1
iGator			
	UPPER_HISTOGRAM	Верхняя гистограмма	0
	LOWER_HISTOGRAM	Нижняя гистограмма	2
iIchimoku			
	TENKANSEN_LINE	Линия Tenkan-sen	0
	KIJUNSEN_LINE	Линия Kijun-sen	1
	SENKOUSPANA_LINE	Линия Senkou Span A	2
	SENKOUSPANB_LINE	Линия Senkou Span B	3
	CHIKOUSPAN_LINE	Линия Chikou Span	4
iMACD, iRVI, iStochastic			
	MAIN_LINE	Основная линия	0
	SIGNAL_LINE	Сигнальная линия	1

Формулы расчета всех индикаторов приведены в документации MetaTrader 5.

Полную техническую информацию по вызову функций индикаторов, включая примеры исходных кодов, можно найти в документации по MQL5. Далее мы приведем несколько примеров.

5.5.7 Использование встроенных индикаторов

В качестве простого вводного примера использования встроенного индикатора возьмем обращение к *iStochastic*. Прототип этой функции-индикатора следующий:

```
int iStochastic(const string symbol, ENUM_TIMEFRAMES timeframe,
  int Kperiod, int Dperiod, int slowing,
  ENUM_MA_METHOD method, ENUM_STO_PRICE price)
```

Как мы видим, помимо стандартных параметров *symbol* и *timeframe*, стохастик имеет несколько специфических параметров:

- *Kperiod* — количество баров для расчета линии %K;
- *Dperiod* — период первичного сглаживания для линии %D;
- *slowing* — период вторичного сглаживания (замедления);
- *method* — метод устреднения (сглаживания);
- *price* — способ расчета стохастика.

Попробуем создать собственный индикатор *UseStochastic.mq5*, который скопирует к себе в буфера значения стохастика. Поскольку в стохастике 2 буфера, мы также зарезервируем два — они называются "главная" и "сигнальная" линия.

```
#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
```

Во входных переменных предусмотрим все требуемые параметры.

```
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;
```

Далее опишем массивы для индикаторных буферов и глобальную переменную для дескриптора.

```
double MainBuffer[];
double SignalBuffer[];

int Handle;
```

Инициализацию проведем в *OnInit*.

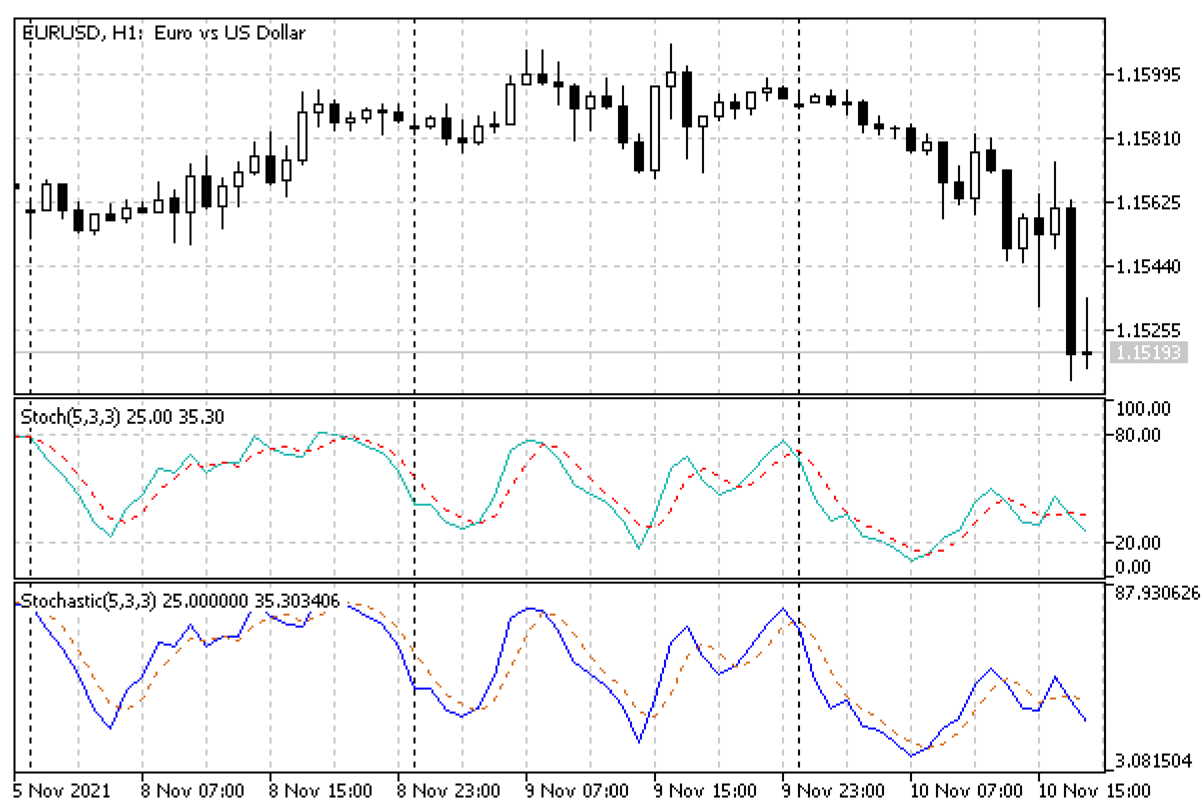
```
int OnInit()
{
   IndicatorSetString(INDICATOR_SHORTNAME,
      StringFormat("Stochastic(%d,%d,%d)", KPeriod, DPeriod, Slowing));
   // привязка массивов в качестве буферов
   SetIndexBuffer(0, MainBuffer);
   SetIndexBuffer(1, SignalBuffer);
   // получаем дескриптор Stochastic
   Handle = iStochastic(_Symbol, _Period,
      KPeriod, DPeriod, Slowing, Method, StochasticPrice);
   return Handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}
```

Далее остается дело за малым — в *OnCalculate* читать данные с помощью функции *CopyBuffer* по мере готовности дескриптора.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &data[])
{
   // ждем расчета стохастика на всех барах
   if(BarsCalculated(Handle) != rates_total)
   {
      return prev_calculated;
   }

   // копируем данные в наши два буфера
   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;
}
```

Обратите внимание, что мы два раза вызываем *CopyBuffer*: для каждого буфера отдельно (0 и 1 во втором параметре). Попытка прочитать буфер с несуществующим индексом, например, 2 — сгенерировала бы ошибку, и мы не получили бы никаких данных.

Наш индикатор не особо полезен, в том смысле, что не привносит ничего дополнительного к оригинальному стохастику и не анализирует его показания. Зато мы можем убедиться, что линии стандартного индикатора терминала и созданного на MQL5 совпадают (уровни и настройку точности также можно было бы легко добавить, как мы делали с полностью пользовательскими индикаторами, но тогда трудно было бы отличить копию от оригинала).



Стандартный стохастик и пользовательский на базе функции iStochastic

Чтобы продемонстрировать кэширование индикаторов терминалом, добавим в функцию *OnInit* пару строк.

```
double array[];
Print("This is very first copy of iStochastic with such settings=",
   !(CopyBuffer(Handle, 0, 0, 10, array) > 0));
```

Здесь применена хитрость, связанная с известной нам особенностью, что сразу после создания индикатора он требует некоторого времени на расчет, и прочитать данные из буфера сразу после получения дескриптора нельзя. Это действительно так, но при условии так называемого "холодного" старта — если индикатора с заданными параметрами еще не существует в кэше, в памяти терминала. Если же готовый двойник найдется, то мы сможет моментально обратиться к буферу.

Откомпилировав новый индикатор, следует разместить две его копии на двух чартах одного и того же символа и таймфрейма. Первый раз в журнал выведется сообщение с флагом *true* (это первая копия), а во второй раз (и последующие разы, если графиков много) будет уже *false*. Также можно сперва вручную набросить на график стандартный индикатор "Stochastic Oscillator" (с настройками по умолчанию или теми, которые затем будут применены в *UseStochastic*), а затем запустить *UseStochastic*: мы также должны получить *false*.

Попробуем теперь придумать нечто оригинальное на базе стандартного индикатора. Следующий индикатор *UseM1MA.mq5* предназначен для расчета средних побаровых цен на таймфреймах M5 и выше (преимущественно, внутридневных). Он аккумулирует цены баров M1, попадающих в диапазон временных меток каждого конкретного бара на рабочем (более старшем) таймфрейме. Это позволяет намного более точно оценить эффективную цену бара, чем стандартные типы цен (*Close*, *Open*, *Median*, *Typical*, *Weighted*, и т.д.). Дополнительно предусмотрим возможность

усреднения таких цен с некоторым периодом, но здесь следует быть готовым, что особо гладкой линии не получится.

Индикатор будет выводиться в главном окне и содержать единственный буфер. Настройки можно менять с помощью 3-х параметров:

```
input uint _BarLimit = 100; // BarLimit
input uint BarPeriod = 1;
input ENUM_APPLIED_PRICE M1Price = PRICE_CLOSE;
```

BarLimit задает количество баров ближайшей истории для расчета. Он важен, потому что графики на высоких таймфреймах могут потребовать очень большого количества баров при сопоставлении с минутным M1 (например, один день D1 при круглосуточной торговле содержит, как известно, 1440 баров M1). Это может привести к загрузке дополнительных данных и ожиданию синхронизации. Поэкспериментируйте со щадящей настройкой по умолчанию (100 баров рабочего таймфрейма), прежде чем ставить в этом параметре 0, что означает безлимитную обработку.

Однако даже при установке *BarLimit* в 0 вероятен расчет индикатора не на всю видимую историю старшего таймрфейма: если в терминале стоит ограничение количества баров в графике, то оно будет сказываться также и на запросах баров M1. Иными словами, глубина анализа определяется временем, на которое в историю уходит максимально разрешенное количество баров M1.

BarPeriod задает количество баров старшего таймфрейма, для которых выполняется усреднение. По умолчанию здесь стоит 1, что позволяет увидеть эффективную цену каждого бара в отдельности.

Наконец, в параметре *M1Price* указывается тип цены в расчетах по барам M1.

В глобальном контексте описаны массив под буфер, дескриптор и флаг самообновления, который нам потребуется для ожидания построения таймсерии "чужого" таймфрейма M1.

```
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;
```

Кроме того, здесь формируется название индикатора и период усреднения *P*. Функция *PeriodSeconds*, возвращающая количество секунд внутри одного бара текущего таймфрейма, позволяет рассчитать число баров M1 внутри одного текущего бара: *PeriodSeconds() / 60* (60 секунд — длительность бара M1).

Привычная инициализация выполняется в *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;
}
```

Для получения средней цены на баре старшего таймфрейма мы применяем простую скользящую, вызывая *iMA* с режимом MODE_SMA.

Функция *OnCalculate* далее приводится с упрощениями. При первом запуске или изменении истории мы очищаем буфер и заполняем переменную *BarLimit* (она требуется, потому что входные переменные нельзя редактировать, а мы хотим трактовать значение 0, как максимально доступное количество баров для расчета). При последующих вызовах чистятся элементы буфера только на последних барах, начиная с *prev_calculated* и не более чем *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;
      }
   }
```

Прежде чем читать данные из созданного индикатора *iMA*, необходимо дождаться их готовности: для этого сравниваем *BarsCalculated* с количеством баров M1.

```
   if(BarsCalculated(Handle) != iBars(_Symbol, PERIOD_M1))
   {
      if(prev_calculated == 0)
      {
         EventSetTimer(1);
         PendingRefresh = true;
      }
      return prev_calculated;
   }
   ...
```

Если данные не готовы, запускаем таймер, чтобы попытаться прочитать их еще раз через секунду.

Далее мы попадаем в основную расчетную часть алгоритма и потому должны остановить таймер, если он еще запущен. Такое может быть, если очередное событие тика пришло быстрее, чем 1 секунда, и *iMA* по M1 уже рассчиталась. Логично было бы просто вызвать соответствующую функцию *EventKillTimer*. Однако в её поведении есть нюанс: она не чистит очередь событий для MQL-программы типа индикатора, и если в очереди уже размещено событие таймера, то обработчик *OnTimer* будет однократно вызван. Чтобы избежать лишнего обновления графика мы контролируем процесс с помощью собственной переменной *PendingRefresh* и присваиваем ей здесь *false*.

```
   ...
   PendingRefresh = false; // данные готовы, таймер сработает вхолостую
   ...
```

Вот как это все организовано в обработчике *OnTimer*:

```
void OnTimer()
{
   EventKillTimer();
   if(PendingRefresh)
   {
      ChartSetSymbolPeriod(0, _Symbol, _Period);
   }
}
```

Но вернемся в *OnCalculate* и представим основной рабочий цикл.

```
   for(int i = fmax(prev_calculated - 1, (int)(rates_total - BarLimit));
      i < rates_total; ++i)
   {
      static double result[1];
      
      // получим последний бар M1, соответствующий i-му бару текущего таймфрейма
      const datetime dt = time[i] + PeriodSeconds() - 60;
      const int bar = iBarShift(_Symbol, PERIOD_M1, dt);
      
      if(bar > -1)
      {
         // запросим значение MA на M1
         if(CopyBuffer(Handle, 0, bar, 1, result) == 1)
         {
            Buffer[i] = result[0];
         }
         else
         {
            Print("CopyBuffer failed: ", _LastError);
            return prev_calculated;
         }
      }
   }
   
   return rates_total;
}
```

Работу индикатора иллюстрирует следующее изображение на EURUSD,H1. Голубая линия соответствует настройкам по умолчанию. Каждое значение получено усреднением PRICE_CLOSE по 60 барам M1. Оранжевая линия дополнительно включает сглаживание по 5 барам H1, с ценами M1 PRICE_TYPICAL.

Два экземпляра индикатора UseM1MA на EURUSD,H1

В книге представлена упрощенная версия *UseM1MASimple.mq5*. Мы оставили за кадром специфику усреднения последнего (неполного) бара, обработку пустых баров (для которых на M1 нет данных) и корректную установку свойства PLOT_DRAW_BEGIN, а также контроль за появлением кратковременных отставаний в расчете средней при появлении новых баров. Полная версия доступна в файле *UseM1MA.mq5*.

В качестве последнего примера построения индикаторов на основе стандартных разберем усовершенствование индикатора *IndUnityPercent.mq5*, который был представлен в разделе Мультивалютные и мультитаймфреймовые индикаторы. Первая версия использовала для вычислений цены *Close*, получая их с помощью *CopyBuffer*. В новой версии *UseUnityPercentPro.mq5* заменим этот способ на чтение показаний индикатора *iMA*. Это позволит реализовать новые возможности:

- усреднять цены на заданном периоде;
- выбирать метод усреднения;
- выбирать тип цены для расчета;

Изменения в исходном коде — минимальные. Добавим 3 новых параметра и глобальный массив для дескрипторов *iMA*:

```
input ENUM_APPLIED_PRICE PriceType = PRICE_CLOSE;
input ENUM_MA_METHOD PriceMethod = MODE_EMA;
input int PricePeriod = 1;
...
int Handles[];
```

Во вспомогательной функции *InitSymbols*, которая вызывается из *OnInit* для парсинга строки со списком рабочих инструментов, добавим выделение памяти под новый массив (его размер *SymbolCount* определяется как раз из списка).

```
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);
   }
}
```

В конце этой же функции создадим дескрипторы требуемых подчиненных индикаторов.

В функции *Calculate*, где выполняется основной расчет, заменим вызовы вида:

```
CopyClose(Symbols[j], _Period, time0, time1, w);
```

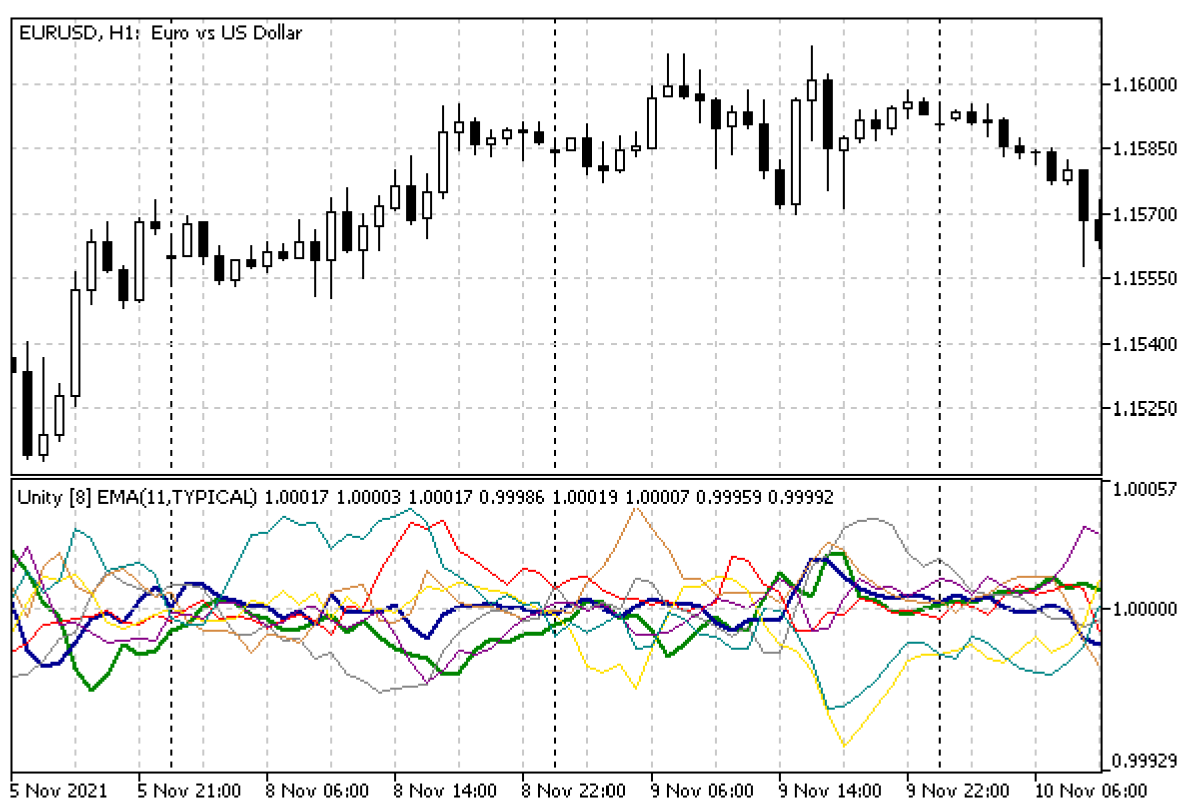
на вызовы:

```
CopyBuffer(Handles[j], 0, time0, time1, w); // j-й дескриптор, 0-й буфер
```

Для наглядности мы также дополнили краткое название индикатора указанием трех новых параметров.

```
   IndicatorSetString(INDICATOR_SHORTNAME,
      StringFormat("Unity [%d] %s(%d,%s)", workCurrencies.getSize(),
      StringSubstr(EnumToString(PriceMethod), 5), PricePeriod,
      StringSubstr(EnumToString(PriceType), 6)));
```

Вот что получилось в результате.



Мультисимвольный индикатор UseUnityPercentPro с основными парами Forex

Здесь показана корзина из 8 основных валют Forex (настройка по умолчанию), с усреднением на 11 барах и расчету по цене *typical*. Две толстые линии соответствуют относительной стоимости валют текущего графика: EUR помечено синим цветом, USD — зеленым.

5.5.8 Расширенный способ создания индикаторов: IndicatorCreate

Создание индикатора с помощью функции *iCustom* или одной из тех, что составляют набор встроенных индикаторов, требует знания перечня параметров на стадии кодирования. Однако на практике часто возникает необходимость в написании достаточно гибких программ, чтобы в них можно было заменить один индикатор на другой.

Например, при оптимизации эксперта в тестере имеет смысл подбирать не только период скользящего среднего, но и алгоритм его расчета. Разумеется, если строить алгоритм на единственном индикаторе *iMA*, можно предусмотреть задание в настройках его методов ENUM_MA_METHOD. Но кто-то, вероятно, хотел бы расширить выбор за счет переключения между двойной экспоненциальной, тройной экспоненциальной и фрактальной скользящей средней. На первый взгляд, для этого можно было бы использовать *switch* с вызовом, соответственно, *iDEMA*, *iTEMA* и *iFrAMA*. Однако, как быть с включением в этот список пользовательских индикаторов?

Хотя название индикатора достаточно просто заменить в вызове *iCustom*, список параметров может существенно отличаться. В общем случае в эксперте может потребоваться генерация сигналов на сочетании любых, не известных заранее индикаторов, а не только скользящих средних.

Для подобных случаев в MQL5 существует универсальный метод создания произвольного технического индикатора с помощью функции *IndicatorCreate*.

int IndicatorCreate(const string symbol, ENUM_TIMEFRAMES timeframe, ENUM_INDICATOR indicator, int count = 0, const MqlParam ¶meters[] = NULL)

Функция создает экземпляр индикатора для указанной пары символа и таймфрейма. Тип индикатора задается с помощью параметра *indicator*. Его тип — перечисление ENUM_INDICATOR (см. далее) содержит идентификаторы для всех встроенных индикаторов, а также вариант для *iCustom*. Количество параметров индикатора и их описания передаются, соответственно, в аргументе *count* и массиве структур *MqlParam* (см. ниже).

Каждый элемент этого массива описывает соответствующий входной параметр создаваемого индикатора, поэтому содержимое и порядок следования элементов должны отвечать прототипу функции встроенного индикатора или, в случае пользовательского индикатора, описаниям входных переменных в его исходном коде.

Нарушение этого правила может привести к ошибке на стадии выполнения программы (см. пример далее) и невозможности создать дескриптор. А в худшем случае переданные параметры будут интерпретированы неправильно, и индикатор поведет себя не так, как ожидалось, однако из-за отсутствия ошибок заметить это нелегко. Исключением является передача пустого массива или не передача его вовсе (поскольку аргументы *count* и *parameters* являются опциональными): в этом случае индикатор будет создан с настройками по умолчанию. Также для пользовательских индикаторов можно опускать произвольное количество параметров с конца списка.

Структура *MqlParam* специально разработана для передачи входных параметров при создании индикатора с помощью *IndicatorCreate* или для получения информации о параметрах стороннего индикатора (выполняющегося на графике) с помощью *IndicatorParameters*.

```
struct MqlParam
{
   ENUM_DATATYPE type;          // тип входного параметра
   long          integer_value; // поле для хранения целочисленного значения
   double        double_value;  // поле для хранения значения double или float
   string        string_value;  // поле для хранения значения строкового типа
};
```

Фактическое значение параметра должно быть задано в одном из полей *integer_value*, *double_value*, *string_value*, в соответствии со значением первого поля *type*. В свою очередь для описания поля *type* используется перечисление ENUM_DATATYPE, содержащее идентификаторы для всех встроенных типов MQL5.

Идентификатор	Тип данных
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

Если какой-либо параметр индикатора имеет тип перечисления, для его описания в поле *type* необходимо применять значение TYPE_INT.

Перечисление ENUM_INDICATOR, используемое в третьем параметре *IndicatorCreate* для указания типа индикатора, содержит следующие константы.

Идентификатор	Индикатор
IND_AC	Accelerator Oscillator
IND_AD	Accumulation/Distribution
IND_ADX	Average Directional Index
IND_ADXW	ADX by Welles Wilder
IND_ALLIGATOR	Alligator
IND_AMA	Adaptive Moving Average
IND_AO	Awesome Oscillator
IND_ATR	Average True Range
IND_BANDS	Bollinger Bands®
IND_BEARS	Bears Power

Идентификатор	Индикатор
IND_BULLS	Bulls Power
IND_BWMFI	Market Facilitation Index
IND_CCI	Commodity Channel Index
IND_CHAIKIN	Chaikin Oscillator
IND_CUSTOM	Custom indicator
IND_DEMA	Double Exponential Moving Average
IND_DEMARKER	DeMarker
IND_ENVELOPES	Envelopes
IND_FORCE	Force Index
IND_FRACTALS	Fractals
IND_FRAMA	Fractal Adaptive Moving Average
IND_GATOR	Gator Oscillator
IND_ICHIMOKU	Ichimoku Kinko Hyo
IND_MA	Moving Average
IND_MACD	MACD
IND_MFI	Money Flow Index
IND_MOMENTUM	Momentum
IND_OBV	On Balance Volume
IND_OSMA	OsMA
IND_RSI	Relative Strength Index
IND_RVI	Relative Vigor Index
IND_SAR	Parabolic SAR
IND_STDDEV	Standard Deviation
IND_STOCHASTIC	Stochastic Oscillator
IND_TEMA	Triple Exponential Moving Average
IND_TRIX	Triple Exponential Moving Averages Oscillator
IND_VIDYA	Variable Index Dynamic Average
IND_VOLUMES	Volumes
IND_WPR	Williams Percent Range

Важно отметить, что если в качестве типа индикатора передается значение IND_CUSTOM, то первый элемент массива параметров должен иметь поле *type* со значением TYPE_STRING, а поле *string_value* должно содержать имя (путь) пользовательского индикатора.

В случае успеха функция *IndicatorCreate* возвращает дескриптор созданного индикатора, а в случае неудачи — INVALID_HANDLE. Код ошибки будет находиться в *_LastError*.

Напомним, что для тестирования MQL-программ, создающих пользовательские индикаторы, имена которых не известны на стадии компиляции (что имеет место и при использовании *IndicatorCreate*), необходимо явным образом обеспечивать их привязку с помощью директивы:

```
#property tester_indicator "indicator_name.ex5"
```

Это дает возможность тестеру отправить требуемые вспомогательные индикаторы на агенты тестирования, но ограничивает процесс только заранее известными индикаторами.

Рассмотрим несколько примеров. Начнем с простого применения *IndicatorCreate* в качестве альтернативы уже известных функций, а затем для демонстрации гибкости нового подхода создадим универсальный индикатор-обертку для визуализации произвольных встроенных или пользовательских индикаторов.

Первый пример — *UseEnvelopesParams1.mq5* — создает встроенную копию индикатора *Envelopes*. Для этого описываем два буфера и две диаграммы, массивы под них и входные параметры, повторяющие параметры *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; // дескриптор подчиненного индикатора
```

Обработчик *OnInit* мог бы выглядеть следующим образом, если использовать функцию *iEnvelopes*.

```
int OnInit()
{
   SetIndexBuffer(0, UpBuffer);
   SetIndexBuffer(1, DownBuffer);

   Handle = iEnvelopes(WorkPeriod, Shift, Method, Price, Deviation);
   return Handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}
```

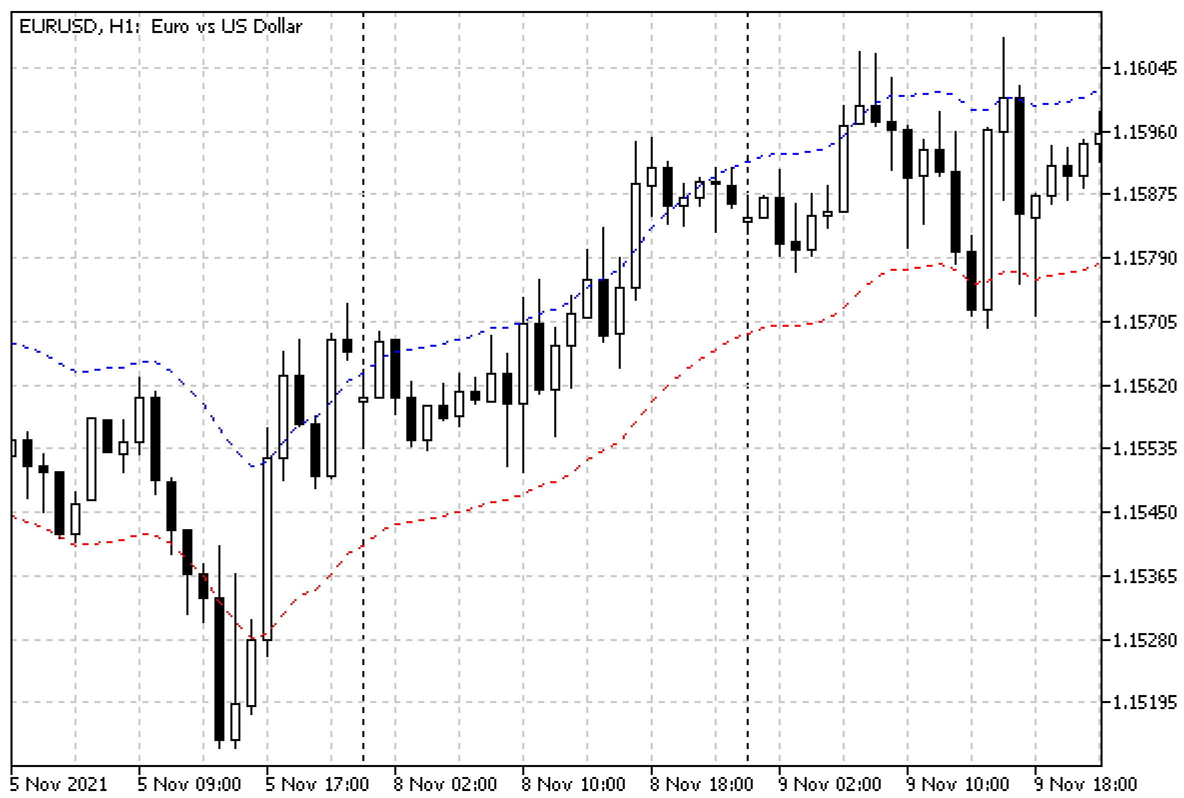
Привязка буферов останется прежней, но для создания дескриптора мы сейчас пойдем другим путем. Опишем массив *MqlParam*, заполним его и вызовем функцию *IndicatorCreate*.

```
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;
}
```

Получив дескриптор, используем его в *OnCalculate* для заполнения двух своих буферов.

```
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;
}
```

Проверим, как созданный индикатор *UseEnvelopesParams1* выглядит на графике.



Индикатор UseEnvelopesParams1

Выше был представлен стандартный, но не очень изящный способ заполнения свойств. Поскольку вызов *IndicatorCreate* может потребоваться во многих проектах, имеет смысл упростить процедуру для вызывающего кода. Для этой цели разработаем класс *MqlParamBuilder* (см. файл *MqlParamBuilder.mqh*). Его задачей будет принимать с помощью некоторых методов значения параметров, определять их тип и добавлять в массив соответствующие элементы — корректно заполненные структуры.

К сожалению, MQL5 не поддерживает в полной мере концепцию, которая есть во многих других языках программирования и называется "информацией о типах времени исполнения" (Run-Time Type Information, RTTI). С помощью неё программы могут запрашивать у среды исполнения описательные мета-данные о своих составных частях — переменных, структурах, классах, функциях и т.д. К немногочисленным встроенным возможностям MQL5, которые можно отнести к разряду RTTI, являются операторы typename и offsetof. Поскольку *typename* возвращает название типа в виде строки, построим свой автодетектор типов на строках (см. файл *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;
}
```

Шаблонная функция *rtti* получает с помощью *typename* строку с именем типа-параметра шаблона и сравнивает её с элементами массива, содержащего все встроенные типы из перечисления ENUM_DATATYPE. Порядок перечисления имен в массиве соответствует значению элемента перечисления, поэтому при обнаружении совпадающей строки достаточно привести индекс к типу (ENUM_DATATYPE) и вернуть вызывающему коду. Например, вызов *rtti(1.0)* или *rtti<double>()* даст значение TYPE_DOUBLE.

Имея этот инструмент, мы можем вернуться к разработке *MqlParamBuilder*. Опишем в классе свой массив структур *MqlParam* и переменную *n*, которая будет содержать индекс последнего, заполняемого элемента.

```
class MqlParamBuilder
{
protected:
   MqlParam array[];
   int n;
   ...
```

Публичный метод для добавления очередного значения в список параметров сделаем шаблонным. Более того, реализуем его в виде перегрузки оператора '<<', который возвращает

указатель на сам объект "строителя". Это позволит записывать в массив несколько значений одной строкой, например, так: *builder << WorkPeriod << PriceType << SmoothingMode*.

Именно в этом методе мы увеличиваем размер массива, получаем рабочий индекс *n* для заполнения и сразу же обнуляем эту *n*-ую структуру.

```
...
public:
   template<typename T>
   MqlParamBuilder *operator<<(T v)
   {
      // расширяем массив
      n = ArraySize(array);
      ArrayResize(array, n + 1);
      ZeroMemory(array[n]);
      ...
      return &this;
   }
```

Там, где стоит многоточие, последует основная рабочая часть, то есть заполнение полей структуры. Можно было бы предположить, что тип параметра мы напрямую определим с помощью самодельного *rtti*. Но следует обратить внимание на один нюанс. Если мы напишем инструкцию *array[n].type = rtti(v)*, она неправильно сработает для перечислений. Каждое перечисление является самостоятельным типом со своим названием, несмотря на то, что хранится по образцу целых чисел. Для перечислений функция *rtti* вернет 0, в связи с чем, нужно явным образом заменить его на TYPE_INT.

```
      ...
      // определяем тип значения
      array[n].type = rtti(v);
      if(array[n].type == 0) array[n].type = TYPE_INT; // подразумеваем enum
      ...
```

Осталось положить само значение *v* в одно из трех полей структуры: *integer_value* типа *long* (обратите внимание, *long* — это длинное целое, отсюда и название поля), *double_value* типа *double* или *string_value* типа *string*. При этом количество встроенных типов гораздо больше, поэтому подразумевается, что все целочисленные типы (включая *int*, *short*, *char*, *color*, *datetime*, перечисления) должны попадать в поле *integer_value*, значения *float* — в поле *double_value*, и лишь для поля *string_value* существует однозначное толкование: это всегда *string*.

Для выполнения данной задачи реализуем несколько перегруженных методов *assign*: три с конкретными типами *float*, *double*, *string*, и один шаблонный для всего остального.

```
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;
   }

   // здесь обрабатываем int, enum, color, datetime и пр. совместимое с long
   template<typename T>
   void assign(const T v)
   {
      array[n].integer_value = v;
   }
   ...
```

На этом процесс заполнения структур заканчивается, и остается вопрос передачи сформированного массива в вызывающий код. Это действие поручено публичному методу с перегрузкой оператора '>>', который имеет единственный аргумент: ссылку на приемный массив *MqlParam*.

```
   // экспортируем внутренний массив наружу
   void operator>>(MqlParam &params[])
   {
      ArraySwap(array, params);
   }
```

Теперь все готово, чтобы заняться исходным кодом модифицированного индикатора *UseEnvelopesParams2.mq5*. Изменения по сравнению с первой версией коснуться только заполнения массива *MqlParam* в обработчике *OnInit*. В нем мы описываем объект "строителя", отсылаем в него через '<<' все параметры и возвращаем через '>>' готовый массив. Все в одной строке.

```
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
   */
```

Для контроля выводим массив в журнал (выше показан результат для значений по умолчанию).

Если массив заполнен не полностью, вызов *IndicatorCreate* завершится ошибкой. Например, если передать только 3 параметра из 5 требуемых *Envelopes*, получим ошибку 4002 и недействительный дескриптор.

```
   Handle = PRTF(IndicatorCreate(_Symbol, _Period, IND_ENVELOPES, 3, params));
   // Пример ошибки:
   // indicator Envelopes cannot load [4002]
   // IndicatorCreate(_Symbol,_Period,IND_ENVELOPES,3,params)=
      -1 / WRONG_INTERNAL_PARAMETER(4002)
```

Однако более длинный массив, чем в спецификации индикатора, не считается ошибкой: лишние значения просто не принимаются во внимание.

Обратите внимание, что когда типы значений отличаются от ожидаемых типов параметров, система выполняет их неявное приведение, и это не вызывает явных ошибок, хотя созданный индикатор может работать не так, как ожидалось. Например, если вместо *Deviation* отправить в индикатор строку, она будет интерпретирована как число 0, в результате чего "конверт" схлопнется: обе линии совместятся на средней, относительно которой и делается отступ на размер *Deviation* (в процентах). Похожим образом, передав вещественное число с дробной частью в том параметре, где ожидается целое число, мы спровоцируем его округление.

Но мы, разумеется, оставим правильный вариант вызова *IndicatorCreate* и получим рабочий индикатор, также как и в первой версии.

```
   ...
   Handle = PRTF(IndicatorCreate(_Symbol, _Period, IND_ENVELOPES,
      ArraySize(params), params));
   // успех:
   // IndicatorCreate(_Symbol,_Period,IND_ENVELOPES,ArraySize(params),params)=10 / ok
   return Handle == INVALID_HANDLE ? INIT_FAILED : INIT_SUCCEEDED;
}
```

Внешне новый индикатор ничем не отличается от предыдущего.

5.5.9 Гибкое создание индикаторов с помощью IndicatorCreate

После знакомства с новым способом создания индикаторов обратимся к задаче, более приближенной к реальности. *IndicatorCreate* обычно применяют в тех случаях, когда вызываемый индикатор заранее не известен. Такая необходимость, например, возникает при написании универсальных экспертов, способных торговать по произвольным сигналам, настраиваемым пользователем. И даже названия индикаторов могут задаваться пользователем.

Мы пока не готовы к разработке экспертов, а потому изучим эту технологию на примере индикатора-обертки *UseDemoAll.mq5*, способного отобразить данные любого другого индикатора.

С точки зрения пользователя это должно выглядеть так. После наложения *UseDemoAll* на график, в диалоге свойств появляется список для выбора одного из встроенных индикаторов или пользовательского, причем в последнем случае дополнительно нужно будет указать его имя в поле ввода. В другом строковом параметре можно ввести список параметров, через запятую. Типы параметров будут определяться автоматически, исходя из их написания. Например, число с десятичной точкой (10.0) будет трактоваться как double, число без точки (15) — как целое, а нечто заключенное в кавычки ("текст") — как строка.

Это далеко не все, но основные настройки *UseDemoAll*. С остальными мы будем разбираться по мере возникновения конструктивных проблем.

За основу решения возьмем перечисление ENUM_INDICATOR: в нем уже есть элементы для всех типов индикаторов, включая пользовательские (IND_CUSTOM). Правда в чистом виде оно не подходит по нескольким причинам. Во-первых, из него невозможно получить метаданные о конкретном индикаторе, такие как количество и типы аргументов, количество буферов и в какое окно индикатор выводится (в главное или в собственное). А эта информация важна для правильного создания и визуализации индикатора. Во-вторых, если мы опишем входную переменную типа ENUM_INDICATOR, чтобы пользователь мог выбирать интересующий его индикатор, в диалоге свойств это будет представлено выпадающим списком, где варианты содержат лишь имя элемента. А было бы желательно обеспечить в этом списке подсказки для пользователя (как минимум, про параметры). Поэтому опишем свое собственное перечисление *IndicatorType*. Напомним, что MQL5 позволяет для каждого элемента указать справа комментарий, который показывается в интерфейсе.

В каждом элементе перечисления *IndicatorType* будем кодировать особым образом не только соответствующий идентификатор (ID) из ENUM_INDICATOR, но и количество параметров (P), количество буферов (B) и номер рабочего окна (W). Для этих целей разработаны следующие макросы.

```
#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))
```

Макрос MAKE_IND принимает в качестве параметров все вышеозвученные характеристики и упаковывает их в разные байты одного 4-х байтового целого числа, формируя таким образом уникальный код для элемента нового перечисления. Остальные 4 макроса позволяют выполнять обратную операцию — по коду вычислить все характеристики индикатора.

Само перечисление *IndicatorType* не станем приводить здесь полностью, а покажем лишь фрагмент. С полным исходным кодом можно ознакомиться в файле *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]*
};
```

В комментариях, которые станут элементами выпадающего списка, видимого пользователю, указаны прототипы с именованными параметрами, количество буферов в квадратных скобках и пометки звездочкой тех индикаторов, которые выводятся в собственное окно. Сами идентификаторы также сделаны информативными, потому что именно они преобразуются в текст функцией *EnumToString*, которая используется для вывода сообщений в журнал.

Список параметров особенно важен, так как пользователь должен будет ввести соответствующие значения, разделенные запятыми, в зарезервированную для этого входную переменную. Мы могли бы "подсказать" и типы параметров, но для простоты решено оставить только имена со смыслом, из которого можно заключить и тип. Например, *period*, *fast*, *slow* — это целые с периодом (количеством баров), *method* — метод усреднения ENUM_MA_METHOD, *price* — тип цены ENUM_APPLIED_PRICE, *volume* — тип объема ENUM_APPLIED_VOLUME.

Для удобства пользователя (чтобы не вспоминать значения элементов перечислений) в программе будут поддержаны названия всех перечислений. В частности, идентификатор *sma* обозначит MODE_SMA, *ema* — MODE_EMA и так далее. Цена *close* превратится в PRICE_CLOSE, *open* — в PRICE_OPEN, и прочие типы цен — аналогично, по последнему слову (после подчеркивания) в идентификаторе элемента перечисления. Например, для списка параметров индикатора iMA (iMA_period_shift_method_price) можно написать такую строку: *11,0,sma,close*. Идентификаторы не надо брать в кавычки. Однако ничто не мешает, при необходимости, передать строку с таким же текстом, например, список — *1.5,"close"* — содержит вещественное число 1.5 и строку "close".

Тип индикатора, а также строки со списком параметров и, опционально, именем (если индикатор пользовательский) — это основные данные для конструктора класса *AutoIndicator*.

```
class AutoIndicator
{
protected:
   IndicatorType type;       // выбранный тип индикатора
   string symbol;            // рабочий символ (необязательный)
   ENUM_TIMEFRAMES tf;       // рабочий таймфрейм (необязательный)
   MqlParamBuilder builder; // "строитель" массива параметров
   int handle;               // дескриптор индикатора
   string name;              // имя пользовательского индикатора
   ...
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));
      // расщепляем строку на массив параметров (формируется внутри builder)
      parseParameters(parameters);
      // создаем и запоминаем дескриптор
      handle = create();
   }

   int getHandle() const
   {
      return handle;
   }
};
```

Здесь и далее опущены некоторые фрагменты, связанные с проверками входных данных на корректность. Полный исходный код прилагается к книге.

Процесс анализа строки с параметрами поручается методу *parseParameters*. В нем реализована описанная выше схема с распознаванием типов значений и их передача в объект *MqlParamBuilder*, с которым мы познакомились в предыдущем примере.

```
int parseParameters(const string &list)
{
   string sparams[];
   const int n = StringSplit(list, ',', sparams);

   for(int i = 0; i < n; i++)
   {
      // нормализация строки (убираем пробелы приводим к нижнему регистру)
      StringTrimLeft(sparams[i]);
      StringTrimRight(sparams[i]);
      StringToLower(sparams[i]);

      if(StringGetCharacter(sparams[i], 0) == '"'
      && StringGetCharacter(sparams[i], StringLen(sparams[i]) - 1) == '"')
      {
         // все, что внутри кавычек, берется как 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;
```

```
}
```

Вспомогательная функция *lookUpLiterals* обеспечивает конвертацию идентификаторов в константы стандартных перечислений.

```
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;
}
```

После того как параметры распознаны и сохранены во внутреннем массиве объекта *MqlParamBuilder*, вызывается метод *create*. Его задача, скопировать параметры в локальный массив, дополнить его именем пользовательского индикатора (в случае такового), и вызвать функцию *IndicatorCreate*.

```
   int create()
   {
      MqlParam p[];
      // заполняем массив 'p' параметрами, собранными объектом 'builder'
      builder >> p;
      
      if(type == iCustom_)
      {
         // вставляем название пользовательского индикатора в самое начало
         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);
   }
```

Метод возвращает полученный дескриптор.

Особый интерес вызывает то, каким образом в самое начало массива вставляется дополнительный строковый параметр с названием пользовательского индикатора. Сначала массиву назначается порядок индексации "как в таймсериях" (см. *ArraySetAsSeries*), в результате чего индекс последнего (физически, по расположению в памяти) элемента становится равным 0, и подсчет элементов идет справа налево. Затем массив увеличивается в размере и в добавленный элемент записывается имя индикатора. За счет обратной индексации это добавление происходит не справа от существующих элементов, а слева. В завершение мы возвращаем массиву привычный порядок индексации, и под индексом 0 оказывается бывший только что последним новый элемент со строкой.

Дополнительно класс *AutoIndicator* умеет формировать сокращенное имя встроенного индикатора из названия элемента перечисления.

```
   ...
   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;
   }
};
```

Теперь все готово, чтобы заняться непосредственно исходным кодом *UseDemoAll.mq5*. Но начнем со слегка упрощенной версии *UseDemoAllSimple.mq5*.

Прежде всего, определимся с количеством индикаторных буферов. Поскольку максимальное количество буферов среди встроенных индикаторов равно пяти (у *Ichimoku*), примем его как ограничитель. Регистрацию этого количества массивов в качестве буферов поручим уже известному нам классу *BufferArray* (см. раздел Мультивалютные и мультитаймфреймовые индикаторы, пример *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);
```

Важно напомнить, что индикатор может быть спроектирован либо для отображения в главном окне, либо в собственном. Совместить два режима MQL5 не позволяет. Однако нам заранее не известно, какой индикатор выберет пользователь, и потому требуется изобрести некий "обходной маневр". Пока разместим свой индикатор в главном окне, а с проблемой отдельного окна разберемся позже.

Чисто технически нет никаких препятствий для копированиях данных из буферов индикаторов со свойством *indicator_separate_window* в свои буфера, отображаемые в главном окне. Однако следует иметь в виду, что диапазон величин подобных индикаторов часто не совпадает с масштабом цен, и потому увидеть их на графике вряд ли получится (линии будут находиться где-то далеко за пределами видимой области, вверху или внизу), хотя значения по-прежнему будут выводиться в *Окно данных*.

С помощью входных переменных обеспечим выбор типа индикатора, названия пользовательского индикатора и списка параметров. Также добавим переменные для типа отрисовки и ширины линий. Поскольку буфера будут подключаться к работе динамически, в зависимости от количества буферов исходного индикатора, мы не описываем стили буферов статически с помощью директив и будем делать это в *OnInit* вызовами встроенных *Plot*-функций.

```
input IndicatorType IndicatorSelector = iMA_period_shift_method_price; // Built-in Ir
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
```

Для хранения дескриптора индикатора опишем глобальную переменную.

```
int Handle;
```

В обработчике *OnInit* задействуем представленный ранее класс *AutoIndicator* для парсинга входных данных, подготовки массива *MqlParam* и получения на его основе дескритора.

```
#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;
   }
   ...
```

Для настройки диаграмм опишем набор цветов и получим краткое имя индикатора из объекта *AutoIndicator*. Также вычислим количество используемых буферов *n* встроенного индикатора с помощью макроса IND_BUFFERS, а для любого пользовательского индикатора (который неизвестен заранее), за неимением лучшего решения, включим все буфера. Далее в процессе копирования данных лишние вызовы функции *CopyBuffer* просто вернут ошибку, и такие массивы можно будет заполнить пустыми значениями.

```
   ...
   static color defColors[BUF_NUM] = {clrBlue, clrGreen, clrRed, clrCyan, clrMagenta}
   const string s = indicator.getName();
   const int n = (IndicatorSelector != iCustom_) ? IND_BUFFERS(IndicatorSelector) : E
   ...
```

В цикле настроим свойства диаграмм, с учетом ограничителя n: буфера сверх него скрываются.

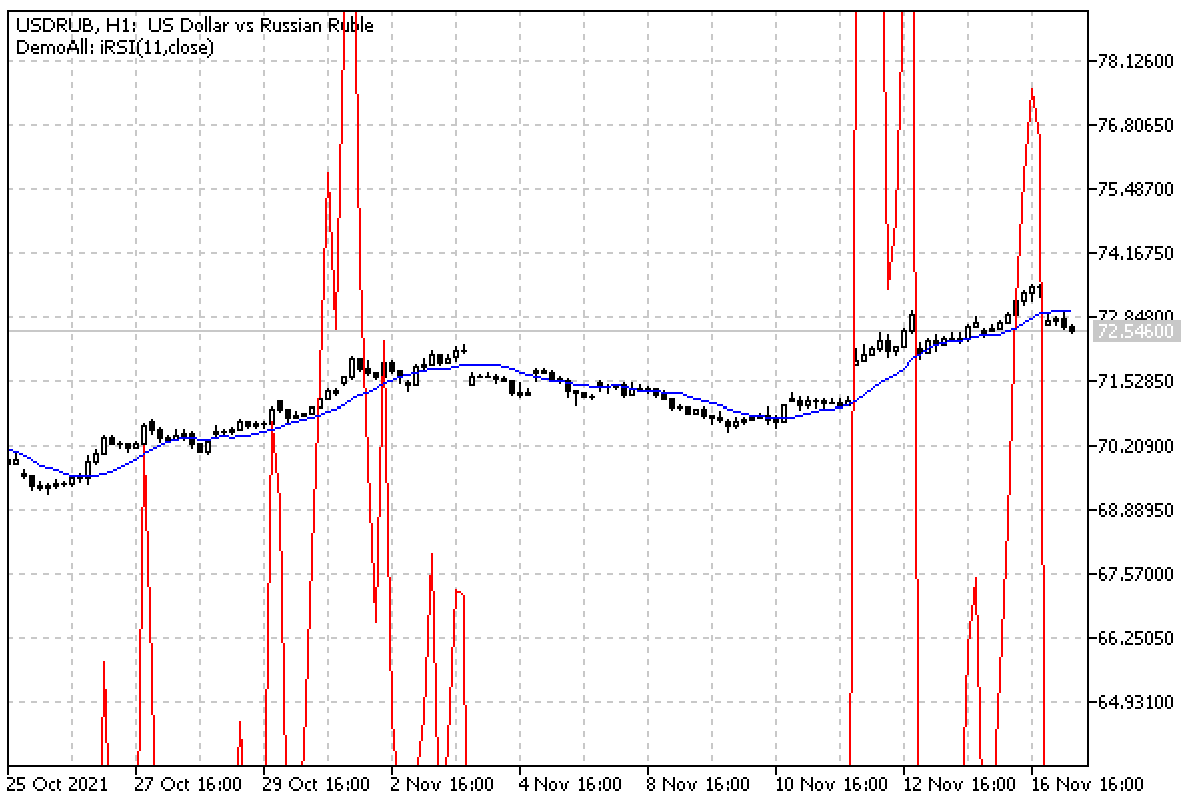
```
   for(int i = 0; i < BUF_NUM; ++i)
   {
      PlotIndexSetString(i, PLOT_LABEL, s + "[" + (string)i + "]");
      PlotIndexSetInteger(i, PLOT_DRAW_TYPE, i < n ? DrawType : DRAW_NONE);
      PlotIndexSetInteger(i, PLOT_LINE_WIDTH, DrawLineWidth);
      PlotIndexSetInteger(i, PLOT_LINE_COLOR, defColors[i]);
      PlotIndexSetInteger(i, PLOT_SHOW_DATA, i < n);
   }

   Comment("DemoAll: ", (IndicatorSelector == iCustom_ ? IndicatorCustom : s),
      "(", IndicatorParameters, ")");

   return INIT_SUCCEEDED;
}
```

В верхнем левом углу графика, в комментарии будет выводиться название индикатора с параметрами.

В обработчике *OnCalculate*, по готовности данных у дескриптора, читаем их в свои массивы.

```
int OnCalculate(ON_CALCULATE_STD_SHORT_PARAM_LIST)
{
   if(BarsCalculated(Handle) != rates_total)
   {
      return prev_calculated;
   }

   const int m = (IndicatorSelector != iCustom_) ? IND_BUFFERS(IndicatorSelector) : E
   for(int k = 0; k < m; ++k)
   {
      // заполняем свои буферы данными из индикатора с дескриптором 'Handle'
      const int n = buffers[k].copy(Handle,
         k, 0, rates_total - prev_calculated + 1);

      // в случае проблем чистим буфер
      if(n < 0)
      {
         buffers[k].empty(EMPTY_VALUE, prev_calculated, rates_total - prev_calculatec
      }
   }

   return rates_total;
}
```

Вышеописанная реализация является упрощенной и соответствует исходному файлу *UseDemoAllSimple.mq5*. Её расширением мы займемся далее, а пока проверим поведение текущей версии. На следующем изображении показаны 2 копии индикатора: синяя линия — с настройками по умолчанию (*iMA_period_shift_method_price*, параметры "*11,0,sma,close*"), а красная — *iRSI_period_price* с параметрами "*11,close*".



Два экземпляра индикатора UseDemoAllSimple с показаниями iMA и iRSI

Для демонстрации специально выбран график USDRUB, потому что значения котировок здесь более или менее совпадают с диапазоном индикатора RSI (который должен был бы выводиться в отдельном окне). На большинстве графиков других символов мы бы не заметили RSI. Если для вас важен только программный доступ к величинам, то в этом нет ничего страшного, однако при наличии требований по визуализации это проблема, которую следует решить.

Итак, следует каким-то образом обеспечить отдельное отображение индикаторов, предназначенных для подокна. В принципе, в среде MQL-разработчиков довольно часто встречается требование выводить графику одновременно в главное окно и в подокно. Мы представим один из вариантов решения, но для этого нужно сначала познакомиться с некоторыми новыми функциями.

5.5.10 Обзор функций управления индикаторами на графике

Как мы уже разобрались, индикаторы являются тем типом MQL-программ, которые сочетают в себе расчетную часть и визуализацию. И если расчеты выполняются внутри, незаметно для пользователя, то визуализация требует привязки к графику. Именно поэтому индикаторы тесно связаны с графиками и MQL5 API даже содержит группу функций, которые обеспечивают управление индикаторами на графиках. Подробно мы рассмотрим эти функции в главе про графики, а в здесь просто приведем их список.

Функция	Назначение
ChartWindowFind	Возвращает номер подокна, в котором находится текущий индикатор или индикатор с заданным именем
ChartIndicatorAdd	Добавляет на указанное окно графика индикатор с указанным дескриптором
ChartIndicatorDelete	Удаляет с указанного окна графика индикатор с указанным именем
ChartIndicatorGet	Возвращает дескриптор индикатора с указанным коротким именем на указанном окне графика
ChartIndicatorName	Возвращает короткое имя индикатора по номеру в списке индикаторов на указанном окне графика
ChartIndicatorsTotal	Возвращает количество всех индикаторов, присоединенных к указанному окну графика

В следующем разделе про Комбинирование вывода информации в главное окно и вспомогательное мы увидим пример *UseDemoAll.mq5*, использующий некоторые из этих функций.

5.5.11 Комбинирование вывода в главное окно и вспомогательное

Вернемся вновь к проблеме вывода диаграмм в главное окно и в подокно из одного индикатора, поскольку мы столкнулись с ней при разработке примера *UseDemoAllSimple.mq5*. Там мы выяснили, что индикаторы, предназначенные для отдельного окна, не подходят для визуализации на основном графике, а индикаторы для главного окна не имеют дополнительных окон. Существует несколько альтернативных подходов:

- Реализовать родительский индикатор для отдельного окна и выводить там диаграммы, а в главном окне использовать для отображения данных графические объекты. Это плохо, потому что данные из объектов нельзя считывать как из таймсерии, и множество объектов потребляет лишние ресурсы.
- Разработать для основного окна собственную виртуальную панель (класс), в которой в правильном масштабе отображать таймсерии, которые должны были бы выводиться в подокне.
- Использовать несколько индикаторов — как минимум один для главного окна и один для подокна — и обмениваться между ними данными через разделяемую память (требуется DLL), ресурсы или базу данных.
- Дублировать расчеты (использовать общий исходный код) в индикаторах для главного окна и подокна.

Мы представим один из вариантов решения, которое основывается на выходе за рамки одной MQL-программы: необходим дополнительный индикатор со свойством *indicator_separate_window*. Он у нас фактически уже есть — ведь мы создаем его расчетную часть, запрашивая дескриптор. Осталось лишь неким образом вывести его в отдельном подокне.

В новой (полной) версии *UseDemoAll.mq5* мы будем анализировать метаданные запрошенного для создания индикатора в соответствующем элементе перечисления *IndicatorType*. Напомним, что там помимо прочего закодировано и рабочее окно каждого типа встроенного индикатора. Когда индикатор требует отдельное окно, мы будем создавать таковое с помощью специальных функций MQL5, с которыми еще предстоит познакомиться.

Для пользовательских индикаторов неоткуда взять информацию о рабочем окне. Поэтому добавим входную переменную *IndicatorCustomSubwindow*, в которой пользователь сможет указать, что требуется подокно.

```
input bool IndicatorCustomSubwindow = false; // Custom Indicator Subwindow
```

В *OnInit* скроем буферы, предназначенные для подокна.

```
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);
   }
   ...
```

После этой настройки нам придется воспользоваться парой функций, которые относятся не только к работе с индикаторами, но и с графиками. Мы подробно изучим их в соответствующей главе, а вводный обзор представлен в предыдущем разделе.

Одна из функций *ChartIndicatorAdd* позволяет добавить индикатор, заданный дескриптором, в окно, причем не только в главную часть, но и в подокно. Об идентификаторах графиков и нумерации окон мы поговорим в главе про графики, а сейчас достаточно знать, что следующий

вызов функции *ChartIndicatorAdd* добавляет индикатор с дескриптором *handle* на текущий график, в новое подокно.

```
int handle = ... // получаем дескриптор индикатора, iCustom или IndicatorCreate

                    // задаем текущий график (0)
                    // |
                    // |      номер окна ставим равным текущему количеству окон
                    // |                              |
                    // |                              |        передаем сам дескриптор
                    // |                              |                          |
                    // v                              v                          v
ChartIndicatorAdd(  0, (int)ChartGetInteger(0, CHART_WINDOWS_TOTAL), handle);
```

Зная о такой возможности, нетрудно выдвинуть идею о вызове *ChartIndicatorAdd* с передачей ей дескриптора уже готового подчиненного индикатора.

Вторая функция, которая нам потребуется, называется *ChartIndicatorName*. Она возвращает краткое имя индикатора по его дескриптору. Данное имя соответствует свойству INDICATOR_SHORTNAME, установленному в коде индикатора, и может отличаться от названия файла. Имя потребуется, чтобы подчистить за собой, то есть удалить вспомогательный индикатор и его подокно, после удаления или перенастройки родительского индикатора.

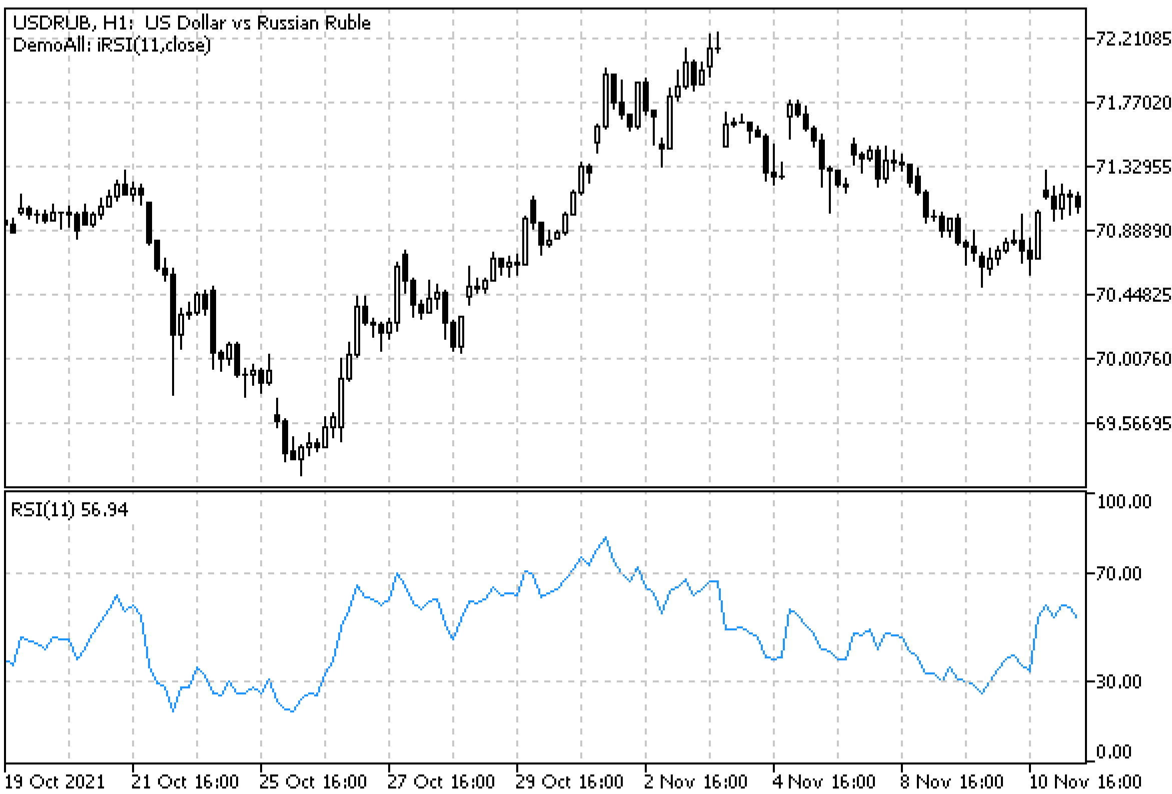
```
string subTitle = "";

int OnInit()
{
   ...
   if(subwindow)
   {
      // показываем новый индикатор в подокне
      const int w = (int)ChartGetInteger(0, CHART_WINDOWS_TOTAL);
      ChartIndicatorAdd(0, w, Handle);
      // сохраняем имя, чтобы удалить индикатор в OnDeinit
      subTitle = ChartIndicatorName(0, w, 0);
   }
   ...
}
```

В обработчике *OnDeinit* используем сохраненный заголовок *subTitle* для вызова еще одной функции, которую мы изучим позднее — *ChartIndicatorDelete* — она удаляет с графика индикатор с именем, указанном в последнем аргументе.

```
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);
   }
}
```

Здесь предполагается, что на графике работает только наш индикатор, и только в единственном экземпляре. В более общем случае для корректного удаления следует анализировать все подокна, но это потребовало бы ещё нескольких функций из тех, что будут представлены в главе про графики, поэтому ограничимся пока простым вариантом.

Если теперь запустить *UseDemoAll* и выбрать какой-нибудь индикатор из списка, помеченный звездочкой (т.е. требующий подокна), например, тот же RSI, увидим ожидаемый результат: RSI в отдельном окне.



RSI в подокне, созданном индикатором UseDemoAll

5.5.12 Чтение данных из диаграмм, имеющих сдвиг

Наш новый индикатор *UseDemoAll* почти завершен. Осталось рассмотреть один нюанс.

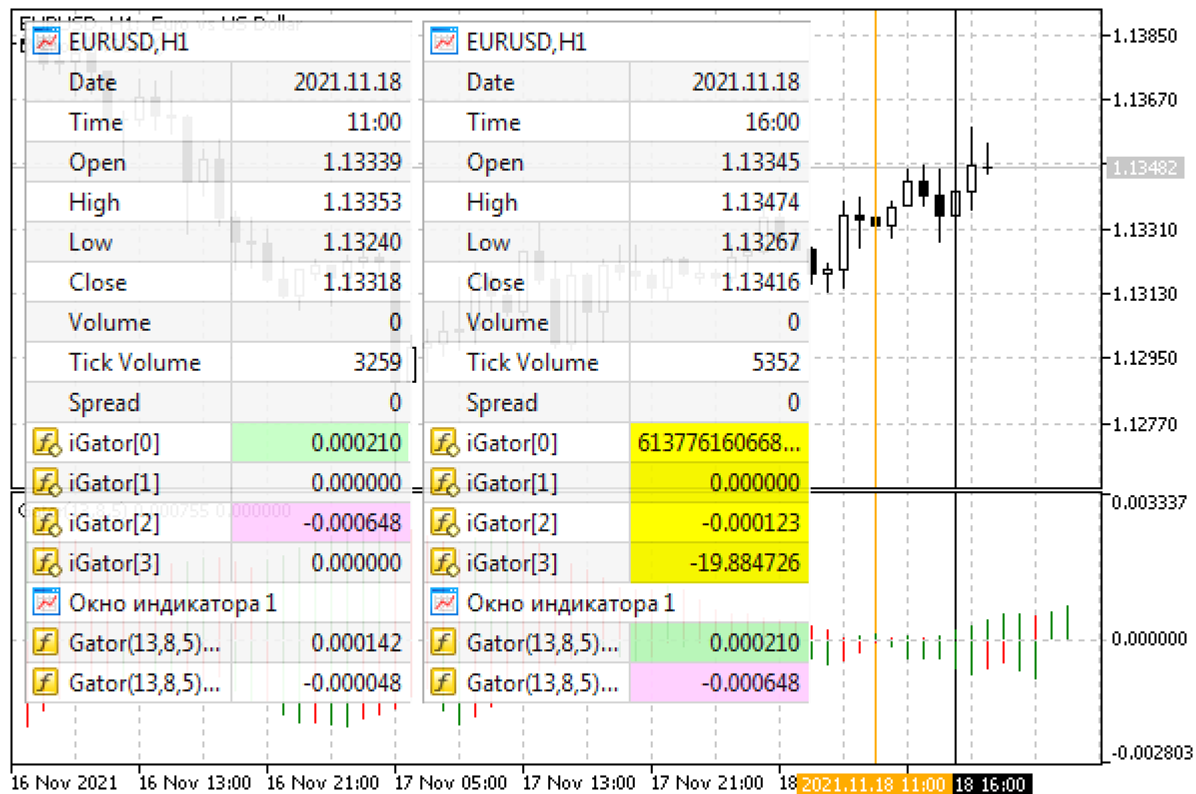
В подчиненном индикаторе некоторые диаграммы могут иметь сдвиг, задаваемый свойством PLOT_SHIFT. Например, при положительном сдвиге элементы таймсерии сдвигаются в будущее и отображаются правее бара с индексом 0. Их индексы, как это ни странно, являются отрицательными. С продвижением вправо номера уменьшаются все больше и больше: -1, -2, -3, и

т.д. Эта адресация влияет и на функцию *CopyBuffer*. Когда мы используем первую форму *CopyBuffer*, параметр *offset*, равный 0, ссылается на элемент с текущим временем в таймсерии, но если сама таймсерия сдвинута вправо, мы получим данные, начиная с элемента под номером N, где N — величина сдвига в индикаторе-источнике. При этом в элементы, расположенные в нашем буфере правее индекса N, не будут заполнены данными, и в них останется "мусор".

Чтобы продемонстрировать проблему возьмем для начала индикатор без сдвига: *Awesome Oscillator* отлично подойдет. Напомним, что *UseDemoAll* копирует все значения в свои массивы, и хотя они невидны на графике из-за разных масштабов цен и показаний индикатора, мы можем сверяться по *Окну данных*. Куда бы мы ни сдвинули курсор мыши на графике, в *Окне данных* значения индикатора в подокне и в буферах *UseDemoAll* будут совпадать. Например, на нижеприведенном изображении наглядно видно, что на часовом баре в 16:00 оба значения равны 0.001797.

Данные индикатора AO в буферах UseDemoAll

Теперь выберем в настройках *UseDemoAll* индикатор *iGator* (*Gator Oscillator*). Для простоты очистим поле с параметрами *Gator* — так он построится со своими параметрами по умолчанию. При этом сдвиг гистограмм составляет 5 баров (вперед), что хорошо видно на графике.



Данные индикатора Gator в буферах UseDemoAll без поправки на сдвиг в будущее

Черной вертикальной линией помечен часовой бар 16:00. Однако значения в *Окне данных* у индикатора *Gator* и в наших массивах, прочитанных из того же индикатора, отличаются. Желтым цветом подсвечены буфера *UseDemoAll*, содержащие мусор.

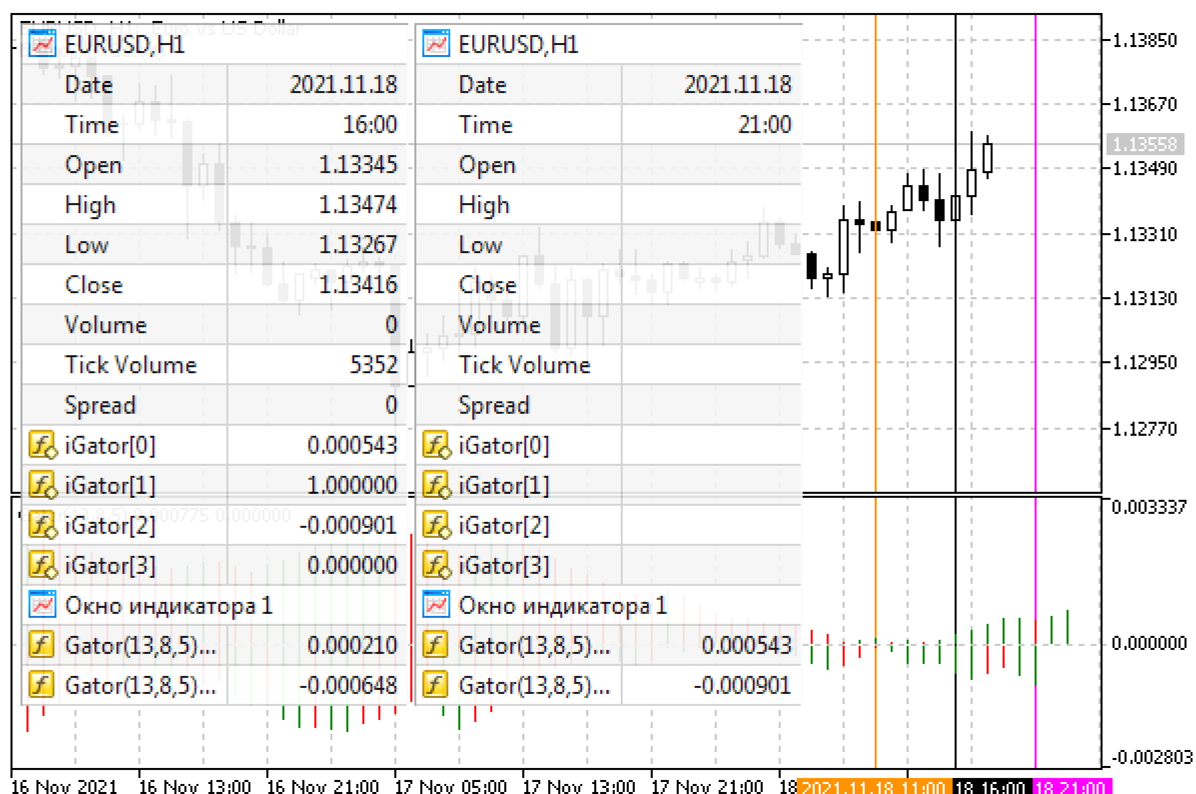
Если исследовать данные на 5 баров в прошлое, в 11:00 (оранжевая вертикальная линия), мы там обнаружим те значения, которые *Gator* выводит в 16:00. Попарные правильные значения верхней и нижней гистограммы подсвечены, соответственно, зеленым и розовым.

Чтобы решить данную проблему нам придется добавить в *UseDemoAll* входную переменную для указания пользователем сдвига диаграмм, а затем делать на неё поправку при вызове *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);
      ...
   }
}
```

К сожалению, узнать свойство PLOT_SHIFT для стороннего индикатора из MQL5 нельзя.

Проверим, как введение сдвига 5 исправляет ситуацию с индикатором Gator (с настройками по умолчанию).



Данные индикатора Gator в буферах UseDemoAll после поправки на сдвиг в будущее

Теперь показания *UseDemoAll* на баре 16:00 соответствуют актуальным данным из Gator из виртуального будущего, отстоящего на 5 баров вперед (сиреневая вертикальная линия в 21:00).

Попутно у вас может возникнуть вопрос, почему в окне *Gator* выводится только 2 буфера, а в нашем — 4. Дело в том, что цветная гистограмма *Gator* использует по одному дополнительному буферу для кодирования цвета. Поскольку цветов всего два — красный и зеленый — мы их видим в своих массивах как 0 или 1.

5.5.13 Удаление экземпляров индикаторов: IndicatorRelease

Как уже было сказано во вводной части этой главы, терминал поддерживает для каждого созданного индикатора счетчик ссылок и оставляет его в работе до тех пор, пока хотя бы одна MQL-программа или график используют его. В MQL-программе признаком необходимости индикатора является действующий дескриптор. Обычно мы запрашиваем дескриптор в процессе инициализации и пользуемся им в алгоритмах вплоть до завершения программы.

В момент выгрузки программы все созданные уникальные дескрипторы автоматически освобождаются, то есть их счетчики уменьшаются на 1 (и если достигают нуля, те индикаторы также выгружаются из памяти). Поэтому явно освобождать дескриптор не требуется.

Однако бывают ситуации, когда подчиненный индикатор становится ненужным в процессе работы программы. Тогда бесполезный индикатор продолжает потреблять ресурсы. Поэтому следует явным образом освободить дескриптор с помощью *IndicatorRelease*.

bool IndicatorRelease(int handle)

Функция удаляет указанный дескриптор индикатора и выгружает сам индикатор, если им больше никто не пользуется. Выгрузка происходит с небольшой задержкой.

Функция возвращает признак успеха (*true*) или ошибки (*false*).

После вызова *IndicatorRelease* переданный ей дескриптор становится недействительным, несмотря на то, что сама переменная сохраняет прежнее значение. Попытка использовать такой дескриптор в других индикаторных функциях вроде *CopyBuffer* завершится ошибкой 4807 (ERR_INDICATOR_WRONG_HANDLE). Во избежание недоразумений желательно сразу же после освобождения дескриптора присвоить соответствующей переменной значение INVALID_HANDLE.

Вместе с тем, если затем программа запросит дескриптор для нового индикатора, этот дескриптор, скорее всего, получит то же значение, что освобожденный до этого, но теперь будет связан с данными нового индикатора.

При работе в тестере стратегий функция *IndicatorRelease* не выполняется.

Для демонстрации применения *IndicatorRelease* подготовим специальную версию *UseDemoAllLoop.mq5*, которая будет периодически в цикле пересоздавать вспомогательный индикатор из списка, куда войдут только индикаторы для главного окна (для наглядности).

```
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; // текущая позиция внутри массива MainLoop

const string IndicatorCustom = "LifeCycle";
```

В первом элементе массива в виде исключения поставлен один пользовательский индикатор — *LifeCycle* из раздела Особенности запуска и остановки программ разных типов. Этот индикатор хоть и не отображает никаких линий, но хорош здесь тем, что выводит в журнал сообщения при вызове своих обработчиков *OnInit*/*OnDeinit*, что позволит отслеживать его жизненный цикл. Жизненные циклы остальных индикаторов аналогичны.

Во входных переменных оставим только настройки отрисовки. Вывод меток DRAW_ARROW, предлагаемый по умолчанию, является оптимальным для отображения разных типов индикаторов.

```
input ENUM_DRAW_TYPE DrawType = DRAW_ARROW; // Drawing Type
input int DrawLineWidth = 1; // Drawing Line Width
```

Для пересоздания индикаторов "на ходу" запустим в *OnInit* 5-секундный таймер, а вся прежняя инициализация (с некоторыми модификациями, описанными ниже) переместится в обработчик *OnTimer*.

```
int OnInit()
{
   Comment("Wait 5 seconds to start looping through indicator set");
   EventSetTimer(5);
   return INIT_SUCCEEDED;
}

IndicatorType IndicatorSelector; // текущий выбранный тип индикатора

void OnTimer()
{
   if(Handle != INVALID_HANDLE && ClearHandles)
   {
      IndicatorRelease(Handle);
      /*
      // дескриптор по-прежнему равен 10, но более недействителен
      // если раскомментировать фрагмент, получим указанную ниже ошибку
      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;

   // создаем дескриптор с параметрами по умолчанию
   // (т.к. передаем пустую строку в третьем аргументе конструктора)
   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(); // чистим буфера, т.к. будет отображен новый индикатор
   ChartSetSymbolPeriod(0, NULL, 0); // запрашиваем полную перерисовку
   ...
   // далее настройка диаграмм - аналогично прежней
   ...
   Comment("DemoAll: ", (IndicatorSelector == iCustom_ ? IndicatorCustom : s),
      "(default-params)");
}
```

Главное отличие заключаются в том, что тип текущего создаваемого индикатора *IndicatorSelector* теперь задается не пользователем, а последовательно выбирается из массива *MainLoop* по

индексу *Cursor*. При каждом вызове таймера этот индекс увеличивается циклически, то есть при достижении конца массива мы перескакиваем на его начало.

Для всех индикаторов строка с параметрами пуста. Так сделано для унификации их инициализации. В результате каждый индикатор будет создаваться с собственными умолчаниями.

В начале обработчика *OnTimer* делается вызов *IndicatorRelease* для предыдущего дескриптора. Однако мы предусмотрели входную переменную *ClearHandles*, чтобы отключить данную ветвь оператора *if* и посмотреть, что будет, если не чистить дескрипторы.

```
input bool ClearHandles = true;
```

По умолчанию *ClearHandles* равна *true*, то есть индикаторы будут удаляться как положено.

Наконец, еще одним дополнением настройки являются строки с очисткой буферов и запросом полной перерисовки графика — и то, и другое нужно, потому что мы подменили у себя подчиненный индикатор, поставляющий отображаемые данные.

Обработчик *OnCalculate* не претерпел изменений.

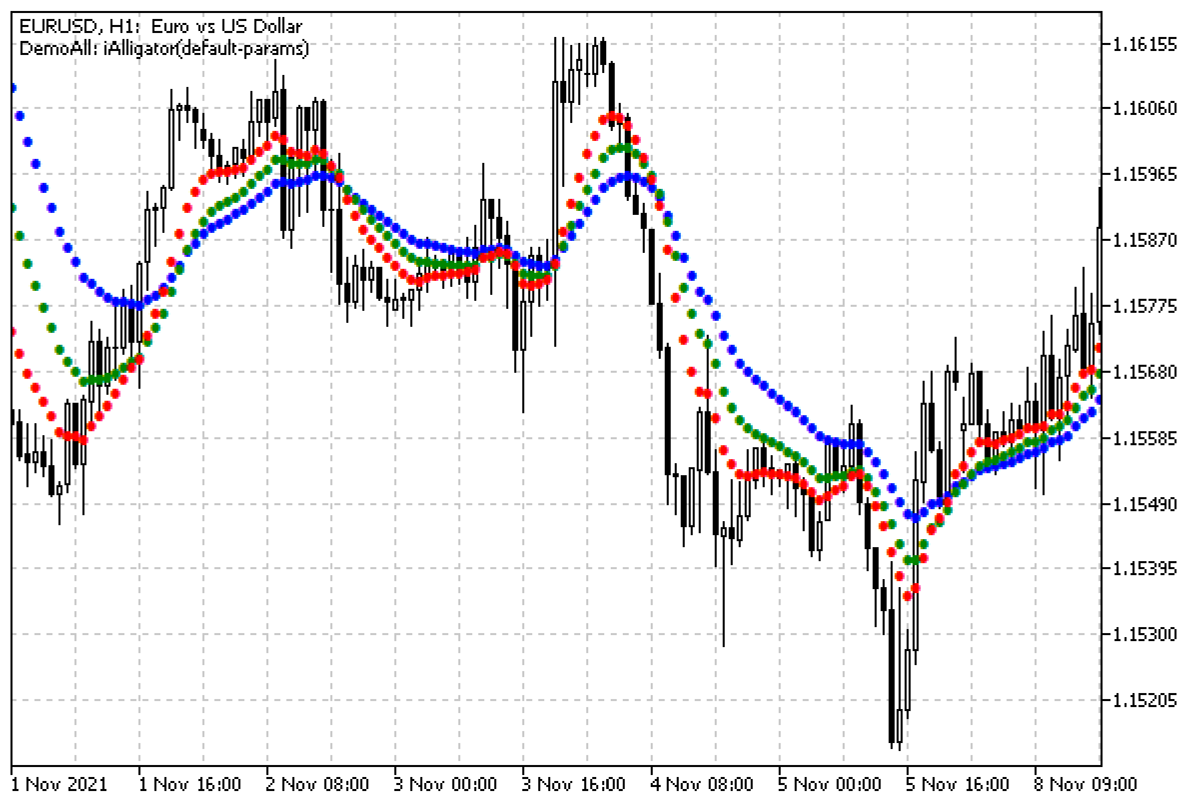
Запустим *UseDemoAllLoop* с настройками по умолчанию. В журнале появятся следующие записи (показано только начало):

```
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
...
```

Обратите внимание, что мы каждый раз получаем один и тот же "номер" дескриптора (10), потому что мы его освобождаем, прежде чем создавать новый дескриптор.

Также важно, что индикатор *LifeCycle* выгрузился вскоре после того, как мы его освободили (предполагается, что он не был добавлен на этот же график сам по себе, т.к. тогда его счетчик ссылок не обнулился бы).

На изображении ниже показан момент, когда наш индикатор визуализирует данные Alligator.



UseDemoAllLoop на шаге демонстрации Alligator

Если поменять значение *ClearHandles* на *false*, увидим совсем другую картину в журнале. Номера дескрипторов теперь будут постоянно увеличиваться, говоря о том, что индикаторы остаются в терминале и продолжают работать, потребляя ресурсы вхолостую. В частности, от индикатора *LifeCycle* не поступает сообщение о деинициализации.

```
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)
...
```

Когда индекс в цикле по массиву индикаторных типов достигнет последнего элемента и пойдет по кругу с начала, терминал начнет возвращать нашему коду дескрипторы уже существующих индикаторов (те же значения: после дескриптора 22 следует опять 10).

5.5.14 Получение настроек индикатора по его дескриптору

Иногда MQL-программе необходимо узнать параметры запущенного экземпляра индикатора. Это могут быть сторонние индикаторы на графике или дескриптор, переданный из основной программы в библиотеку или заголовочный файл. Для этой цели в MQL5 предусмотрена функция *IndicatorParameters*.

int IndicatorParameters(int handle, ENUM_INDICATOR &type, MqlParam ¶ms[])

Функция возвращает по указанному дескриптору количество входных параметров индикатора, а также их типы и сами значения.

При успешном выполнении функция заполняет переданный ей массив *params*, а тип индикатора сохраняется в параметре *type*.

В случае ошибки функция возвращает -1.

В качестве примера работы с данной функцией усовершенствуем индикатор *UseDemoAllLoop.mq5*, представленный в разделе про Удаление экземпляров индикаторов. Новую версию назовем *UseDemoAllParams.mq5*.

Как вы помните, мы там последовательно создавали в цикле некоторые встроенные индикаторы по списку и при этом оставляли список параметров пустым, что приводит к тому, что индикаторы используют некие, неизвестные нам значения по умолчанию. В связи с этим мы выводили в комментарий на графике обобщенный прототип: с названием, но без конкретных значений.

```
// UseDemoAllLoop.mq5
void OnTimer()
{
   ...
   Comment("DemoAll: ", (IndicatorSelector == iCustom_ ? IndicatorCustom : s),
      "(default-params)");
   ...
}
```

Сейчас у нас появилась возможность по дескриптору индикатора узнать его параметры и отобразить их пользователю.

```
// UseDemoAllParams.mq5
void OnTimer()
{
   ...
   // читаем параметры, примененные индикатором по умолчанию
   ENUM_INDICATOR itype;
   MqlParam defParams[];
   const int p = IndicatorParameters(Handle, itype, defParams);
   ArrayPrint(defParams);
   Comment("DemoAll: ", (IndicatorSelector == iCustom_ ? IndicatorCustom : s),
      "(" + MqlParamStringer::stringify(defParams) + ")");
   ...
}
```

Преобразование массива *MqlParam* в строку поручено выделенному классу *MqlParamStringer* (см. файл *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;
   }
};
```

Откомпилировав и запустив новый индикатор, вы можете убедиться, что в левом верхнем углу графика теперь отображается конкретный список параметров визуализируемого индикатора.

Для единственного пользовательского индикатора из списка (*LifeCycle*) первый параметр будет содержать путь и имя файла индикатора. Второй параметр описан в исходном коде как целое число. А вот третий параметр интересен тем, что он неявно описывает свойство Применить к, присущее всем индикаторам с краткой формой обработчика *OnCalculate*. В данном случае, по умолчанию, индикатор применяется к PRICE_CLOSE (значение 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
```

Согласно журналу, настройки встроенных индикаторов также соответствуют умолчаниям.

5.5.15 Определение источника данных для индикатора

Среди встроенных переменных MQL-программ существует одна, применимая только в индикаторах. Это переменная *_AppliedTo* типа *int*, которая позволяет прочитать свойство *Применить к* из диалога настроек индикатора. Кроме того, если индикатор создан с помощью вызова функции *iCustom*, в которую был передан дескриптор стороннего индикатора, то переменная *_AppliedTo* будет содержать этот дескриптор.

В следующей таблице приведено описание возможных значений переменной *_AppliedTo*.

Значение	Описание данных для расчета
0	Индикатор использует полную форму *OnCalculate*, и данные для расчета не задаются одним массивом данных
1	Цена Close
2	Цена Open
3	Цена High
4	Цена Low
5	Средняя цена = (High+Low)/2
6	Типичная цена = (High+Low+Close)/3
7	Взвешенная цена = (Open+High+Low+Close)/4
8	Данные индикатора, который был запущен на графике перед данным индикатором
9	Данные индикатора, который был запущен на графике самым первым
10+	Данные индикатора с дескриптором, содержащимся в *_AppliedTo* — этот дескриптор был передан при создании индикатора в функцию *iCustom* последним параметром

Для удобства анализа значений к книге прилагается заголовочный файл *AppliedTo.mqh* с перечислением.

5.6 Работа с таймером

Для многих прикладных задач важно иметь возможность выполнять действия по расписанию, с некоторым заданным интервалом. В MQL5 такой функционал обеспечивается таймером — системным счетчиком времени, который можно настроить таким образом, чтобы он посылал регулярные уведомления MQL-программе.

Для установки или отмены уведомлений таймера в MQL5 API имеется несколько функций: *EventSetTimer*, *EventSetMillisecondTimer*, *EventKillTimer*. А сами уведомления поступают в программу в виде событий особого типа: для них в исходном коде зарезервирован обработчик *OnTimer*. Об этой группе функций и пойдет речь в данной главе.

Напомним, что в MQL5 события могут получить только интерактивные программы, выполняющиеся на графиках, то есть индикаторы и эксперты. Скрипты и сервисы не поддерживают никакие события, в том числе и от таймера.

Вместе с тем, в главе Функции для работы со временем мы уже касались смежных тем:

- получения временных меток текущих локальных или серверных часов (*TimeLocal* / *TimeCurrent*);
- приостановки выполнения программы на заданный период с помощью *Sleep*;

- получения состояния счетчика системного времени компьютера, отсчитываемого от старта операционной системы (*GetTickCount*) или с момента запуска MQL-программы (*GetMicrosecondCount*).

Эти возможности доступны абсолютно всем типам MQL-программ.

В предыдущих главах мы уже неоднократно использовали функции таймера, несмотря на то, что их формальное описание будет дано только сейчас. Из-за того, что события таймера доступны лишь в индикаторах или экспертах, было бы затруднительно изучать его раньше, чем сами программы. После того как мы освоили создание индикаторов, тема таймеров станет логичным продолжением.

В основном мы использовали таймеры для ожидания построения таймсерий. Такие примеры можно найти в разделах Ожидание данных, Мультивалютные и мультитаймфреймовые индикаторы, Поддержка множества символов и таймфреймов, Использование встроенных индикаторов.

Кроме того, мы переключали по таймеру (каждые 5 секунд) тип подчиненного индикатора в демонстрационной "анимации" индикаторов в разделе Удаление экземпляров индикаторов.

5.6.1 Включение и отключение таймера: EventSetTimer/EventKillTimer

MQL5 позволяет включать и отключать стандартный таймер для выполнения каких-либо действий по расписанию. Для этого предназначены две функции: *EventSetTimer* и *EventKillTimer*.

bool EventSetTimer(int seconds)

Функция указывает клиентскому терминалу, что для данного эксперта или индикатора необходимо генерировать события от таймера с указанной периодичностью, которая задается в секундах (параметр *seconds*).

Функция возвращает признак успешного выполнения (*true*) или ошибки (*false*). Код ошибки можно получить в *_LastError*.

Для того чтобы обрабатывать события таймера, эксперт или индикатор должен иметь в своем коде функцию *OnTimer*. Первое событие таймера наступит не сразу после вызова *EventSetTimer*, а через *seconds* секунд.

Для каждого эксперта или индикатора, вызвавшего функцию *EventSetTimer*, создается свой собственный, выделенный таймер. Программа будет получать события только от него. Таймеры в разных программах работают независимо.

Каждая интерактивная MQL-программа, размещенная на графике, имеет отдельную очередь событий, куда складываются поступающие для неё события. Если в очереди уже есть событие *OnTimer* или оно находится в состоянии обработки, то новое событие *OnTimer* в очередь не ставится.

Если таймер больше не нужен, его следует отключить функцией *EventKillTimer*.

void EventKillTimer(void)

Функция останавливает таймер, включенный перед этим функцией *EventSetTimer* (или функцией *EventSetMillisecondTimer*, которую мы рассмотрим далее). Функцию можно вызвать и из обработчика *OnTimer*. Таким образом, в частности, можно выполнить отложенное однократное действие.

> В индикаторах вызов *EventKillTimer* не чистит очередь, поэтому после него можно получить последнее остаточное событие *OnTimer*.

При завершении работы MQL-программы таймер уничтожается принудительно, если он был создан, но не отключен функцией *EventKillTimer*.

Каждая программа может установить только один таймер. Поэтому, если требуется вызывать разные части алгоритма с разной периодичностью, следует включить таймер с периодом, являющимся наименьшим общим делителем требуемых периодов (в предельном случае, с минимальным периодом в 1 секунду), и в обработчике *OnTimer* самостоятельно отслеживать более крупные периоды. Мы рассмотрим пример этого подхода в следующем разделе.

MQL5 позволяет создавать и таймеры с периодичностью менее 1 секунды: для этого существует функция *EventSetMillisecondTimer*.

5.6.2 Событие таймера: OnTimer

Событие *OnTimer* является одним из стандартных событий, поддерживаемых программами MQL5 (см. раздел Обзор функций обработки событий). Для приема событий таймера в коде программы следует описать функцию со следующим прототипом.

```
void OnTimer(void)
```

Событие *OnTimer* периодически генерируется клиентским терминалом для эксперта или индикатора, который активизировал таймер при помощи функций *EventSetTimer* или *EventSetMillisecondTimer* (см. следующий раздел).

> Внимание! В зависимых индикаторах, создаваемых с помощью вызова *iCustom* или *IndicatorCreate* из других программ, таймер не работает, и событие *OnTimer* не генерируется. Это архитектурное ограничение MetaTrader 5.

Следует понимать, что наличие включенного таймера и обработчика *OnTimer* не делает MQL-программу многопоточной. На каждую MQL-программу выделяется не более одного потока (индикатор может даже делить поток с другими индикаторами на том же символе), поэтому вызов *OnTimer* и прочих обработчиков всегда происходит последовательно, согласно очереди событий. Если один из обработчиков, включая и *OnTimer*, начнет длительные вычисления, это приостановит выполнение всех других событий и участков кода программы.

Если необходимо организовать параллельную обработку данных, следует запустить одновременно несколько MQL-программ (возможно, экземпляров одной и той же программы на разных графиках или объектах-графиках) и обмениваться между ними командами и данными по собственному протоколу — например, используя пользовательские события.

В качестве примера создадим классы, которые позволят организовать несколько логических таймеров в одной программе. Периоды всех логических таймеров будут задаваться как множитель для базового периода, то есть периода единственного аппаратного таймера, поставляющего события в стандартный обработчик *OnTimer*. В этом обработчике мы должны вызвать некий метод нашего нового класса *MultiTimer*, который будет управлять всеми логическими таймерами.

```
void OnTimer()
{
   // вызваем метод MultiTimer чтобы проверить и вызвать зависимые таймеры, когда нуж
   MultiTimer::onTimer();
}
```

Класс *MultiTimer* и связанные с ним классы отдельных таймеров объединим в одном файле *MultiTimer.mqh*.

Базовым классом для рабочих таймеров выступит *TimerNotification*. Строго говоря, это мог бы быть интерфейс, но в него удобно вывести некоторые детали общей реализации: в частности, хранить значение счетчика *chronometer*, за счет которого мы обеспечим срабатывание таймера с неким множителем относительного периода основного таймера, а также метод для проверки момента, когда таймер должен сработать *isTimeCome*. Поэтому *TimerNotification* представляет собой абстрактный класс. В нем отсутствуют реализации двух виртуальных методов: *notify* — для действий при срабатывании таймера и *getInterval* для получения множителя, определяющего период конкретного таймера относительно периода основного таймера.

```
class TimerNotification
{
protected:
   int chronometer; // счетчик проверок таймера (вызовов isTimeCome)
public:
   TimerNotification(): chronometer(0)
   {
   }

   // рабочее событие таймера
   // чистый виртуальный метод, требуется описать в наследниках
   virtual void notify() = 0;
   // возвращает период таймера (его можно изменить на ходу)
   // чистый виртуальный метод, требуется описать в наследниках
   virtual int getInterval() = 0;
   // проверка, не пора ли таймеру сработать, и если да - вызов notify
   virtual bool isTimeCome()
   {
      if(chronometer >= getInterval() - 1)
      {
         chronometer = 0; // сбрасываем счетчик
         notify();        // уведомляем прикладной код
         return true;
      }

      ++chronometer;
      return false;
   }
};
```

Вся логика "зашита" в методе *isTimeCome*. При каждом его вызове инкрементируется счетчик *chronometer*, и если он достиг последней итерации согласно методу *getInterval*, то вызывается метод *notify* уведомления прикладного кода.

Например, если основной таймер запущен с периодом 1 секунда (*EventSetTimer(1)*), то объект-наследник *TimerNotification*, который будет возвращать 5 из метода *getInterval*, будет получать вызовы своего метода *notify* каждые 5 секунд.

Управлять такими объектами-таймерами будет, как мы уже сказали, объект-менеджер *MultiTimer*. Он нужен в единственном числе. Поэтому его конструктор объявлен защищенным, а единственный экземпляр создается статически внутри класса.

```
class MultiTimer
{
protected:
   static MultiTimer _mainTimer;

   MultiTimer()
   {
   }
   ...
```

Внутри этого класса организуем хранение массива объектов *TimerNotification* (о том, как он заполняется, — через пару абзацев). При наличии массива легко написать метод *checkTimers*, который проверяет в цикле все логические таймеры. Для доступа извне это метод дублируется публичным статическим методом *onTimer*, который мы уже видели в глобальном обработчике *OnTimer*. Поскольку единственный экземпляр менеджера создается статически, мы можем обращаться к нему из статического метода.

```
   ...
   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();
   }
   ...
```

Для добавления объекта *TimerNotification* в массив *subscribers* предусмотрен метод *bind*.

```
void bind(TimerNotification &tn)
{
   int i, n = ArraySize(subscribers);
   for(i = 0; i < n; ++i)
   {
      if(subscribers[i] == &tn) return; // уже есть такой объект
      if(subscribers[i] == NULL) break; // нашли пустой слот
   }
   if(i == n)
   {
      ArrayResize(subscribers, n + 1);
   }
   else
   {
      n = i;
   }
   subscribers[n] = &tn;
}
```

В методе сделана защита от повторного добавления объекта, а также, по возможности, проводится размещение указателя в пустом элементе массива, если такой найдется, что исключает необходимость расширять массив. Пустые элементы в массиве могут появиться, если какой-либо из объектов *TimerNotification* был удален с помощью метода *unbind* (таймеры могут использоваться эпизодически).

```
void unbind(TimerNotification &tn)
{
   const int n = ArraySize(subscribers);
   for(int i = 0; i < n; ++i)
   {
      if(subscribers[i] == &tn)
      {
         subscribers[i] = NULL;
         return;
      }
   }
}
```

Обратите внимание, что менеджер не становится владельцем объекта-таймера и не пытается вызвать для него *delete*. Если вы собираетесь регистрировать в менеджере динамически распределенные объекты таймеров, можно добавить внутрь *if*, перед обнулением, примерно такой код:

```
if(CheckPointer(subscribers[i]) == POINTER_DYNAMIC) delete subscribers[i]
```

Теперь осталось понять, как удобно организовать вызовы *bind*/*unbind*, чтобы не нагружать этими утилитарными операциями прикладной код. Если делать это "вручную", то легко где-нибудь забыть создать или наоборот удалить таймер.

Разработаем класс *SingleTimer*, производный от *TimerNotification*, в котором реализуем вызовы *bind* и *unbind* из конструктора и деструктора, соответственно. Кроме того, опишем в нем переменную *multiplier* для хранения периода таймера.

```
class SingleTimer: public TimerNotification
{
protected:
   int multiplier;
   MultiTimer *owner;

public:
   // создаем таймер с указанным множителем базового периода, опционально на паузе
   // автоматически регистрируем объект в менеджере
   SingleTimer(const int m, const bool paused = false): multiplier(m)
   {
      owner = &MultiTimer::_mainTimer;
      if(!paused) owner.bind(this);
   }

   // автоматически отключаем объект от менеджера
   ~SingleTimer()
   {
      owner.unbind(this);
   }

   // возвращаем период таймера
   virtual int getInterval() override
   {
      return multiplier;
   }

   // приостанавливаем работу этого таймера
   virtual void stop()
   {
      owner.unbind(this);
   }

   // возобновляем работу этого таймера
   virtual void start()
   {
      owner.bind(this);
   }
};
```

Второй параметр конструктора (*paused*) позволяет создать объект, но не запускать таймер моментально. Такой отложенный таймер можно затем активировать с помощью метода *start*.

> Схема подписки одних объектов на события в других является одним из популярных паттернов проектирования в ООП и называется "publisher/subscriber".

Важно отметить, что этот класс тоже является абстрактным, потому что в нем не реализован метод *notify*. На основе *SingleTimer* опишем классы таймеров с дополнительным функционалом.

Начнем с класса *CountableTimer*. Он позволяет указать количество раз, которое он должен сработать, после чего будет автоматически остановлен. С помощью него, в частности, легко организовать однократное отложенное действие. Конструктор *CountableTimer* имеет параметры для задания периода таймера, признака приостановки и количества повторов. По умолчанию,

количество повторов не ограничено, поэтому данный класс станет основой для большинства прикладных таймеров.

```
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;
      }
      // делегируем проверку времени родительскому классу,
      // увеличиваем свой счетчик, только если таймер сработал (вернул true)
      return SingleTimer::isTimeCome() && (bool)++count;
   }
   // сбрасываем свой счетчик при остановке
   virtual void stop() override
   {
      SingleTimer::stop();
      count = 0;
   }

   uint getCount() const
   {
      return count;
   }

   uint getRepeat() const
   {
      return repeat;
   }
};
```

Для того чтобы использовать *CountableTimer* мы должны описать в своей программе производный класс, вроде следующего.

```
// 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);
   }
};
```

В данной реализации метода *notify* мы просто выводим период таймера и количество его срабатываний в журнал. Кстати говоря, это фрагмент индикатора *MultipleTimers.mq5*, который мы будем использовать в качестве рабочего примера.

Второй класс, производный от *SingleTimer*, назовем *FunctionalTimer*. Его назначение — обеспечить простую реализацию таймера для тех, кому нравится функциональный стиль программирования и не хочется описывать производные классы. Конструктор класса *FunctionalTimer* будет принимать помимо периода указатель на функцию специального типа *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();
         }
      }
   }
};
```

В этой реализации метода *notify* объект вызывает функцию по указателю. Имея такой класс, мы можем описать макрос, который, будучи размещенным перед блоком с инструкциями в фигурных скобках, "сделает" его телом функции таймера.

```
// MultiTimer.mqh
#define OnTimerCustom(P) OnTimer##P(); \
FunctionalTimer ft##P(P, OnTimer##P); \
bool OnTimer##P()
```

Тогда в прикладном коде можно написать так:

```
// MultipleTimers.mq5
bool OnTimerCustom(3)
{
   Print(__FUNCSIG__);
   return true;        // продолжить работу таймера
}
```

Эта конструкция объявляет таймер с периодом 3 и набором инструкций внутри скобок (здесь просто печать в журнал). Если эта функция вернет *false*, данный таймер будет остановлен.

Рассмотрим индикатор *MultipleTimers.mq5* подробнее. Поскольку в нем не предусмотрено визуализации, укажем количество диаграмм, равное нулю.

```
#property indicator_chart_window
#property indicator_buffers 0
#property indicator_plots   0
```

Для использования классов логических таймеров включим заголовочный файл *MultiTimer.mqh* и добавим входную переменную для периода базового (глобального) таймера.

```
#include <MQL5Book/MultiTimer.mqh>

input int BaseTimerPeriod = 1;
```

Запуск базового таймера производится в *OnInit*.

```
void OnInit()
{
   Print(__FUNCSIG__, " ", BaseTimerPeriod, " Seconds");
   EventSetTimer(BaseTimerPeriod);
}
```

Напомним, что работу всех логических таймеров обеспечивает перехват глобального события *OnTimer*.

```
void OnTimer()
{
   MultiTimer::onTimer();
}
```

В дополнение к прикладному классу таймера *MyCountableTimer*, приведенному выше, опишем еще один класс приостановленного таймера *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) // выполняемся последний раз
      {
         Print("Forcing all timers to stop");
         EventKillTimer();
      }
   }
};
```

Чуть ниже мы увидим, как он запускается. Здесь же важно отметить, что после достижения заданного числа срабатываний данный таймер отключит все таймеры вызовом *EventKillTimer*.

Теперь покажем, как описаны (в глобальном контексте) объекты разных таймеров этих двух классов.

```
MySuspendedTimer st(1, 5);
MyCountableTimer t1(2);
MyCountableTimer t2(4);
```

Таймер *st* класса *MySuspendedTimer* имеет период 1 (*1 * BaseTimerPeriod*) и должен остановиться после 5 срабатываний.

Таймеры *t1* и *t2* класса *MyCountableTimer* имеют периоды 2 (*2 * BaseTimerPeriod*) и 4 (*4 * BaseTimerPeriod*), соответственно. При значении по умолчанию *BaseTimerPeriod = 1* все периоды обозначают секунды. Данные два таймера запускаются сразу после старта программы.

Также создадим два таймера в функциональном стиле.

```
bool OnTimerCustom(5)
{
   Print(__FUNCSIG__);
   st.start();         // запускаем отложенный таймер
   return false;       // а данный объект-таймер останавливаем
}

bool OnTimerCustom(3)
{
   Print(__FUNCSIG__);
   return true;        // этот таймер продолжает работать
}
```

Обратите внимание, что задача *OnTimerCustom5* одна — через 5 периодов после старта программы запустить отложенный таймер *st*, а свое выполнение прекратить. Учитывая, что отложенный таймер должен деактивировать все таймеры через 5 периодов, получим 10 секундную активность программы при настройках по умолчанию.

Таймер *OnTimerCustom3* должен успеть сработать за это время 3 раза.

Итак, у нас имеется 5 таймеров с разными периодами: 1, 2, 3, 4, 5 секунд.

Проанализируем пример того, что выводится в журнал (справа схематично показаны временные метки).

```
                                                    //   время
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                        |
```

Первое сообщение от 2-х секундного таймера поступает, как и ожидалось, примерно через 2 секунды после старта ("примерно" — потому что аппаратный таймер имеет ограничение по точности и, кроме того, на выполнение оказывает влияние прочая загрузка компьютера). Еще через секунду в первый раз срабатывает 3-х секундный таймер. Второе срабатывание 2-х секундного таймера совпадает с первым выводом от 4-х секундного. После однократного выполнения 5-ти секундного таймера в логе начинают регулярно появляться сообщения от 1-секундного таймера (его счетчик увеличивается от 0 до 4). На последней своей итерации он останавливает все таймеры.

5.6.3 Таймер повышенной точности: EventSetMillisecondTimer

Если в программе требуется более частое срабатывание таймера, чем 1 секунда, используйте вместо *EventSetTimer* функцию *EventSetMillisecondTimer*.

Нельзя одновременно запустить таймеры с разными единицами измерения: следует использовать либо одну функцию, либо другую. Тип фактически работающего таймера определяется тем, какая функция была вызвана позднее. Все особенности, присущие стандартному таймеру, остаются в силе и для таймера повышенной точности.

bool EventSetMillisecondTimer(int milliseconds)

Функция указывает клиентскому терминалу, что для данного эксперта или индикатора необходимо генерировать события таймера с периодичностью менее одной секунды. Периодичность задается в миллисекундах (параметр *milliseconds*).

Функция возвращает признак успешного выполнения (*true*) или ошибки (*false*).

При работе в тестере стратегий следует иметь в виду, что чем меньше период таймера, тем дольше будет длиться тестирование, так как возрастает количество вызовов обработчика событий таймера.

При штатной работе события таймера генерируются не чаще 1 раза в 10-16 миллисекунд, что связано с аппаратными ограничениями.

Для демонстрации работы с миллисекундным таймером расширим пример индикатора *MultipleTimers.mq5*. Поскольку активация глобального таймера отдана в ведение прикладной программы, мы легко можем поменять тип таймера, оставив классы логических таймеров без изменений. Разница будет только в том, что их множители будут применяться к базовому периоду в миллисекундах, который мы укажем в функции *EventSetMillisecondTimer*.

Для выбора типа таймера опишем перечисление и добавим новую входную переменную.

```
enum TIMER_TYPE
{
   Seconds,
   Milliseconds
};

input TIMER_TYPE TimerType = Seconds;
```

По умолчанию используем секундный таймер. В *OnInit* запускаем таймер требуемого типа.

```
void OnInit()
{
   Print(__FUNCSIG__, " ", BaseTimerPeriod, " ", EnumToString(TimerType));
   if(TimerType == Seconds)
   {
      EventSetTimer(BaseTimerPeriod);
   }
   else
   {
      EventSetMillisecondTimer(BaseTimerPeriod);
   }
}
```

Посмотрим, что выведется в журнал при выборе миллисекундного таймера.

```
                                                   //   время      мсек
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                       |
```

Последовательность генерации событий полностью совпадает с тем, что мы видели для секундного таймера, но всё происходит намного быстрее, почти мгновенно.

Из-за того, что точность системного таймера ограничена парой десятков миллисекунд, реальный интервал между событиями заметно превосходит недостижимо малую 1 миллисекунду. Кроме того, налицо разброс в размере одного "шага". Таким образом, даже при использовании миллисекундного таймера желательно не закладываться на периоды меньше нескольких десятков миллисекунд.

5.7 Работа с графиками

Большинство MQL-программ — скрипты, индикаторы, эксперты — выполняется на графиках. Только сервисы работают в фоновом режиме, без привязки к графику. Для получения и изменения свойств графиков, анализа их списка и поиска других выполняющихся программ предоставлен богатый набор функций.

Поскольку графики являются естественной средой для индикаторов, нам уже пришлось познакомиться с некоторыми из этих функций в предыдущих главах, посвященных индикаторам. В данной главе мы изучим все эти функции целенаправленно.

При работе с графиками часто используется понятие окна. Окно — это выделенная область, в которой выводятся графики цен и/или диаграммы индикаторов. Верхнее и, как правило, самое большое по размеру окно содержит графики цен, имеет номер 0 и существует всегда. Все дополнительные окна, добавляемые в нижнюю часть при размещении индикаторов, имеют номера от 1 и больше (нумерация сверху вниз). Каждое подокно существует только до тех пор, пока в нем имеется как минимум один индикатор.

Поскольку пользователь может удалить все индикаторы в произвольном подокне, в том числе и том, которое не является последним (самым нижним), номера оставшихся подокон могут уменьшаться.

Событийная модель графиков — получение и обработка уведомлений о событиях на графиках и генерация собственных событий — будут рассмотрены в отдельной главе.

Помимо обсуждаемых здесь "графиков в окнах" MetaTrader 5 также позволяет создавать "графики в объектах" — мы займемся графическими объектами в следующей главе.

5.7.1 Функции для получения основных свойств текущего графика

Во многих примерах книги нам уже приходилось использовать Предопределенные переменные, содержащие основные свойства графика и его рабочего символа. MQL-программам доступны также функции, возвращающие значения некоторых из этих переменных. Что именно использовать — переменную или функцию — не играет роли: вы можете просто придерживаться избранного стиля оформления исходных кодов.

Каждый график характеризуется рабочим символом и таймфреймом. Их можно узнать с помощью функций *Symbol* и *Period*, соответственно. Кроме того, MQL5 предоставляет упрощенный доступ к двум наиболее часто используемым свойствам символа: размеру пункта цены (*Point*) и связанному с ним количеству значащих цифр (*Digits*) после десятичной точки в цене.

string Symbol()

Функция *Symbol* возвращает имя символа текущего графика, то есть значение системной переменной *_Symbol*. Для получения символа произвольного графика существует функция *ChartSymbol*, требующая знания идентификатора того графика. Способов получения идентификаторов графиков мы коснемся чуть позже.

ENUM_TIMEFRAMES Period()

Функция *Period* возвращает значение таймфрейма (ENUM_TIMEFRAMES) текущего графика, что соответствует переменной *_Period*. Для получения таймфрейма произвольного графика предназначена функция *ChartPeriod*, и ей также требуется идентификатор в качестве параметра.

double Point()

Функция *Point* возвращает размер пункта текущего инструмента в валюте котировки, что совпадает со значением переменной *_Point*.

int Digits()

Функция возвращает количество десятичных знаков после запятой, определяющее точность измерения цены символа текущего графика, что эквивалентно переменной *_Digits*.

Прочие свойства текущего инструмента позволяют получить *SymbolInfo*-функции, которые в более общем случае обеспечивают анализ всех инструментов.

Следующий простой пример скрипта *ChartMainProperties.mq5* выводит в журнал описанные в данном разделе свойства.

```
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));
}
```

Для графика EURUSD,H1 получим следующие записи в журнале.

```
_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 Идентификация графиков

Каждый график в MetaTrader 5 "живет" в отдельном окне и обладает уникальным идентификатором. Для программистов, знакомых с принципами работы Windows, уточним, что данный идентификатор не является системным дескриптором окна (хотя MQL5 API и позволяет получить последний через свойство CHART_WINDOW_HANDLE). Как мы знаем, кроме основной рабочей области графика с котировками в нижней части окна могут появляться выделенные дополнительные области (подокна) с индикаторами, имеющими свойство *indicator_separate_window*. Все подокна являются частью графика и принадлежат одному окну Windows.

long ChartID()

Функция возвращает уникальный идентификатор текущего графика.

Во многих функциях, которые мы рассмотрим далее, требуется идентификатор графика в качестве параметра, но для текущего графика можно указывать 0 вместо вызова *ChartID*. Использовать *ChartID* имеет смысл в тех случаях, когда идентификатор пересылается между MQL-программами, например, при обмене сообщениями (пользовательскими событиями) на одном графике или на разных. Указание неверного идентификатора приведет к ошибке ERR_CHART_WRONG_ID (4101).

Идентификатор графика, как правило, остается прежним от сеанса к сеансу.

Мы продемонстрируем работу функции *ChartID* и то, как выглядят идентификаторы, в примере скрипта *ChartList1.mq5*, после изучения способа получения списка графиков.

5.7.3 Получение списка графиков

MQL-программа может получить список открытых в терминале графиков (как окон, так и объектов-графиков) с помощью функций *ChartFirst* и *ChartNext*.

long ChartFirst()
long ChartNext(long chartId)

Функция *ChartFirst* возвращает идентификатор первого графика клиентского терминала. MetaTrader 5 поддерживает некий внутренний список всех графиков, порядок в котором может отличаться от того, что мы видим на экране, например, в закладках окон, когда те максимизированы. В частности, порядок в списке может поменяться в результате перетаскивания вкладок, открепления и закрепления окон. После загрузки терминала видимый порядок закладок совпадает с внутренним представлением списка.

Функция *ChartNext* возвращает идентификатор графика, следующего за графиком с указанным идентификатором *chartId*.

В отличие от других функций для работы с графиками, значение 0 в параметре *chartId* означает не текущий график, а "начало списка". Иными словами, вызов *ChartNext(0)* эквивалентен *ChartFirst*.

Если достигнут конец списка, функция возвращает -1.

Скрипт *ChartList1.mq5* позволяет вывести в журнал список графиков. Основную работу выполняет функция *ChartList*, вызываемая из *OnStart*. В самом начале функции мы узнаем идентификатор текущего графика с помощью *ChartID* и позднее помечаем его звездочкой в списке. В конце выводится общее количество графиков.

```
void OnStart()
{
   ChartList();
}

void ChartList()
{
   const long me = ChartID();
   long id = ChartFirst();
   // long id = ChartNext(0); - аналог вызова ChartFirst()
   int count = 0, used = 0;
   Print("Chart List\nN, ID, *active");
   // продолжаем перебирать графики, пока их не останется
   while(id != -1)
   {
      const string header = StringFormat("%d %lld %s",
         count, id, (id == me ? " *" : ""));

      // поля: N, id, метка текущего графика
      Print(header);
      count++;
      id = ChartNext(id);
   }
   Print("Total chart number: ", count);
}
```

Ниже показан пример результата.

```
Chart List
N, ID, *active
0 132358585987782873
1 132360375330772909  *
2 132544239145024745
3 132544239145024732
4 132544239145024744
Total chart number: 5
```

5.7.4 Получение символа и таймфрейма произвольного графика

Два основополагающих свойства любого графика — его рабочий символ и таймфрейм. Как мы видели ранее, эти свойства для текущего графика доступны как встроенные переменные *_Symbol* и *_Period*, а также через одноименные функции *Symbol и Period*. Для определения тех же свойств у других графиков существует пара функций: *ChartSymbol* и *ChartPeriod*.

string ChartSymbol(long chartId = 0)

Функция возвращает имя символа на графике с указанным идентификатором. Если параметр равен 0, подразумевается текущий график.

Если графика не существует, возвращается пустая строка (""). При этом в *_LastError* выставляет код ошибки ERR_CHART_WRONG_ID (4101).

ENUM_TIMEFRAMES ChartPeriod(long chartId = 0)

Функция возвращает значение периода для графика с указанным идентификатором.

Если графика не существует, возвращается 0.

Скрипт *ChartList2.mq5*, аналогичный *ChartList1.mq5*, формирует список графиков с указанием символа и таймфрейма.

```
#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");
   // продолжаем перебирать графики, пока их не останется
   while(id != -1)
   {
      const string header = StringFormat("%d %lld %s %s %s",
         count, id, ChartSymbol(id), PeriodToString(ChartPeriod(id)),
         (id == me ? " *" : ""));

      // поля: N, id, символ, таймфрейм, метка текущего графика
      Print(header);
      count++;
      id = ChartNext(id);
   }
   Print("Total chart number: ", count);
}
```

Вот пример содержимого журнала после запуска скрипта на графике EURUSD,H1 (во второй строке).

```
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 позволяет не только узнавать, но и переключать символ и таймфрейм любого графика.

5.7.5 Обзор функций для работы с полным набором свойств графиков

Свойства графиков доступны для чтения и изменения через группы *ChartSet*- и *ChartGet*-функций, в каждой из которых собраны свойства определенного типа: вещественные числа (*double*), целые числа (*long*, *int*, *datetime*, *color*, *bool*, перечисления) и строки.

Все функции принимают первым параметром идентификатор графика. Значение 0 означает текущий график, то есть эквивалентно передаче результата вызова *ChartID()*. Вместе с тем, это не означает, что идентификатор текущего графика равен 0.

Константы, описывающие все свойства, формируют три перечисления ENUM_CHART_PROPERTY_INTEGER, ENUM_CHART_PROPERTY_DOUBLE, ENUM_CHART_PROPERTY_STRING, которые используются в качестве параметров функций для соответствующего типа. Сводную таблицу всех свойств можно найти в документации MQL5, на странице о свойствах графиков. В следующих разделах этой главы мы постепенно рассмотрим практически все свойства, группируя их по назначению. Исключение составят лишь свойства управления событиями на графике — их мы опишем в соответствующем разделе главы про события.

Элементам всех трех перечислений назначены такие значения, что они составляют единый перечень без пересечений (повторений). Это позволяет по конкретному значению определить тип перечисления. Например, имея константу, мы можем последовательно пытаться преобразовать её в строку с названием одного из перечислений, пока не достигнем успеха.

```
int value = ...;

ResetLastError(); // очищаем код ошибки, если она была ранее
EnumToString((ENUM_CHART_PROPERTY_INTEGER)value); // результирующая строка не важна
if(_LastError == 0) // анализируем, есть ли новая ошибка
{
   // успех - это элемент ENUM_CHART_PROPERTY_INTEGER
   return ChartGetInteger(0, (ENUM_CHART_PROPERTY_INTEGER)value);
}

ResetLastError();
EnumToString((ENUM_CHART_PROPERTY_DOUBLE)value);
if(_LastError == 0)
{
   // успех - это элемент ENUM_CHART_PROPERTY_DOUBLE
   return ChartGetDouble(0, (ENUM_CHART_PROPERTY_DOUBLE)value);
}

... // продолжаем аналогичную проверку для ENUM_CHART_PROPERTY_STRING
```

Позже мы воспользуемся этим подходом в тестовых скриптах.

Некоторые свойства (например, количество видимых баров) доступны только на чтение и не могут быть изменены. Они будут далее помечаться "r/o" (read-only).

Функции чтения свойств имеют краткую и полную форму: краткая — непосредственно возвращает затребованное значение, а полная — логический признак успеха (*true*) или ошибки (*false*), а само значение помещается в последний параметр, передаваемый по ссылке. При использовании краткой формы особенно важно проверять код ошибки в переменной *_LastError*,

потому что значение 0 (NULL), возвращаемое в случае проблем, может являться в общем случае корректным.

При обращении к некоторым свойствам необходимо указывать дополнительный параметр (*window*), который служит для указания номера окна/подокна графика. 0 означает основное окно. Подокна нумеруются, начиная с 1. Некоторые свойства применяются к графику целиком — для них предусмотрены варианты функций без параметра *window*.

Ниже приведены прототипы функций для чтения и записи целочисленных свойств. Обратите внимание, что тип значений в них — это *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)

Аналогично описаны функции для вещественных свойств. Доступных для записи вещественных свойств у подокон не существует, поэтому форма *ChartSetDouble* только одна — без параметра *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)

То же самое касается и строковых свойств, однако следует учитывать еще один нюанс: длина строки не может превышать 2045 символов (лишние символы будут обрезаны).

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)

При чтении свойств с помощью краткой формы *ChartGetInteger/ChartGetDouble* параметр *window* является необязательным и по умолчанию подразумевает основное окно (*window = 0*).

Функции для установки свойств графика (*ChartSetInteger*, *ChartSetDouble*, *ChartSetString*) являются асинхронными и служат для отправки графику команд на изменение. При успешном выполнении этих функций команда попадает в общую очередь событий графика, и возвращается результат *true*. При возникновении ошибки функции вернут *false*, а код ошибки необходимо проверять в переменной *_LastError*.

Изменение свойств графика производится отложенно, в процессе обработки очереди событий данного графика и, как правило, с некоторой задержкой, в связи с чем не стоит ожидать немедленного обновления графика после применения новых настроек. Для принудительного обновления внешнего вида и свойств графика используйте функцию *ChartRedraw*. Если требуется изменить сразу несколько свойств графика, то необходимо вызвать соответствующие функции в одном блоке кода и затем — один раз *ChartRedraw*.

В общем случае обновление графика производится терминалом автоматически в ответ на события, такие как поступление новой котировки, изменения размера окна графика или масштаба, прокрутка, добавление индикатора и т.д.

Функции получения свойств графика (*ChartGetInteger*, *ChartGetDouble*, *ChartGetString*) являются синхронными, то есть вызывающий код дожидается результата их выполнения.

5.7.6 Описательные свойства графика

Функции *ChartSetString/ChartGetString* позволяют читать и устанавливать следующие строковые свойства графиков.

Идентификатор	Описание
CHART_COMMENT	Текст комментария на графике
CHART_EXPERT_NAME	Имя эксперта, запущенного на графике (r/o)
CHART_SCRIPT_NAME	Имя скрипта, запущенного на графике (r/o)

В разделе Вывод сообщений в окно графика мы познакомились с функцией *Comment* для вывода текстового сообщения в верхний левый угол графика. Свойство CHART_COMMENT позволяет прочитать текущий комментарий графика: *ChartGetString(0, CHART_COMMENT)*. Более того, мы можем узнать комментарий на других графиках, передав в функцию их идентификаторы. А с помощью *ChartSetString* можно менять комментарии на текущем и других графиках, зная их *ID*: *ChartSetString(ID, CHART_COMMENT, "text")*.

Если на каком-либо графике запущен эксперт или/и скрипт, мы можем узнать их названия с помощью вызовов *ChartGetString(ID, CHART_EXPERT_NAME)* и *ChartGetString(ID, CHART_SCRIPT_NAME)*.

Скрипт *ChartList3.mq5*, аналогичный *ChartList2.mq5*, дополняет список графиков информацией об экспертах и скриптах. Позднее мы добавим в него и информацию об индикаторах.

```
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");
   // продолжаем перебирать графики, пока их не останется
   while(id != -1)
   {
      temp = 0; // признак MQL-программ на данном графике
      const string header = StringFormat("%d %lld %s %s %s",
         count, id, ChartSymbol(id), PeriodToString(ChartPeriod(id)),
         (id == me ? " *" : ""));
      // поля: N, id, символ, таймфрейм, метка текущего графика
      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);
}
```

Вот пример вывода этого скрипта.

```
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
```

Здесь видно, что выполняется только один наш скрипт.

5.7.7 Проверка состояния основного окна

Пара функций *ChartSetInteger/ChartGetInteger* позволяет узнать несколько характеристик состояния графика, а также изменить некоторые из них.

Идентификатор	Описание	Тип значения
CHART_BRING_TO_TOP	Активность (наличие фокуса ввода) графика поверх всех других	bool
CHART_IS_MAXIMIZED	График максимизирован	bool
CHART_IS_MINIMIZED	Графика минимизирован	bool
CHART_WINDOW_HANDLE	Windows-дескриптор окна графика (r/o)	int
CHART_IS_OBJECT	Признак того, что график является объектом "График" (OBJ_CHART) — для графического объекта возвращает *true*, а для полноценного графика значение равно *false* (r/o)	bool

Ожидаемо Window-дескриптор и признак объекта-графика доступны только на чтение. Прочие свойства, в принципе, должны редактироваться и производить соответствующие эффекты: например, вызвав *ChartSetInteger(ID, CHART_BRING_TO_TOP, true)*, вы активируете график с идентификатором ID. Однако на момент написания книги максимизация и минимизация не были доступны по некоторым причинам.

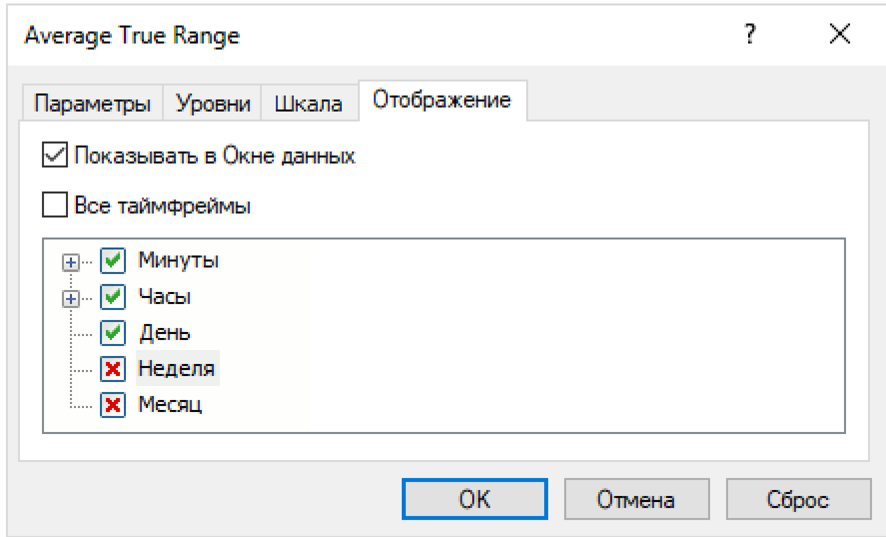
Пример применения свойств приведен в скрипте *ChartList4.mq5* в следующем разделе.

5.7.8 Получение количества и признака видимости окон/подокон

С помощью функции *ChartGetInteger* MQL-программа может узнать количество окон на графике (включая подокна), а также их видимость.

Идентификатор	Описание	Тип значения
CHART_WINDOWS_TOTAL	Общее количество окон графика, включая подокна индикаторов (r/o)	int
CHART_WINDOW_IS_VISIBLE	Видимость подокна, параметр window — номер подокна (r/o)	bool

Некоторые подокна могут быть скрыты, если размещенные в них индикаторы отключены на текущем таймфрейме в диалоге Свойств, на закладке Отображение. Сбросить все флаги нельзя: из-за особенностей хранения tpl-шаблонов такое состояние трактуется как включение всех таймфреймов. Поэтому если пользователь хочет скрыть подокно на некоторое время, нужно оставить хоть один включенный флаг на самом редко используемом таймфрейме.



Настройка видимости индикатора на разных таймфреймах

Следует отметить, что в MQL5 нет штатных средств для программного определения состояния и переключения конкретных флагов. Наиболее простой способ сымитировать подобный контроль заключается в сохранении tpl-шаблона и его анализе, с возможным последующим редактированием и загрузкой (см. раздел Работа с tpl-шаблонами).

В новой версии скрипта *ChartList4.mq5* выведем количество подокон (одно окно — основное — есть всегда), признак активности графика, признак объекта-графика, а также Windows-дескриптор.

```
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");
   }
}
```

Вот что может получиться в результате.

```
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
```

На первом графике (под индексом 0) имеется одно подокно (#1). На последнем графике нашлось два подокна (#2), причем второе из них в данный момент скрыто. Позднее, в разделе Управление индикаторами на графике мы представим полную версию *ChartList.mq5*, где включим в отчет информацию об индикаторах, находящихся в подокнах и основном окне.

Внимание! У чарта внутри объекта-графика свойство CHART_WINDOW_IS_VISIBLE всегда равно *true*, даже если визуализация объекта отключена на текущем или на всех таймфреймах.

5.7.9 Режимы отображения графика

Четыре свойства из перечисления ENUM_CHART_PROPERTY_INTEGER описывают режимы отображения графика. Все эти свойства доступны как на чтение через *ChartGetInteger*, так и на запись через *ChartSetInteger*, что позволяет менять внешний вид графика.

Идентификатор	Описание	Тип значения
CHART_MODE	Тип графика (свечи, бары или линия)	ENUM_CHART_MODE
CHART_FOREGROUND	Ценовой график на переднем плане	bool
CHART_SHIFT	Режим отступа ценового графика от правого края	bool
CHART_AUTOSCROLL	Режим автоматического перехода к правому краю графика	bool

Для режима CHART_MODE в MQL5 имеется особое перечисление ENUM_CHART_MODE. Его элементы приведены в следующей таблице.

Идентификатор	Описание	Значение
CHART_BARS	Отображение в виде баров	0
CHART_CANDLES	Отображение в виде японских свечей	1
CHART_LINE	Отображение в виде линии, проведенной по ценам Close	2

Реализуем скрипт *ChartMode.mq5*, который будет отслеживать состояние режимов и выводить в журнал сообщения при обнаружении изменений. Поскольку алгоритмы обработки свойств носят общий характер, вынесем их в отдельный заголовочный файл *ChartModeMonitor.mqh*, который будем затем подключать к разными тестам.

Основу заложим в абстрактный класс *ChartModeMonitorInterface*: в нем предоставлены перегруженные get- и set-методы для всех типов. Непосредственно проверку свойств в необходимом объеме должны будут осуществлять производные классы, переопределяя виртуальный метод *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() { }
};
```

Также в классе зарезервированы методы *print* (например, для вывода в журнал), сохранения текущего состояния (*backup*) и его восстановления (*restore*). Они объявлены не абстрактными, а с пустой реализацией, так как не являются обязательными.

Конкретные классы для свойств разных типов имеет смысл определить как единый шаблон, унаследованный от *ChartModeMonitorInterface* и принимающий параметрические типы значения (T) и перечисления (E). Например, для целочисленных свойств нужно будет установить *T=long* и *E=ENUM_CHART_PROPERTY_INTEGER*.

В объекте предусмотрен массив *data* для хранения пар [ключ,значение] со всеми запрошенными свойствами. Он имеет шаблонный тип *MapArray<K,V>*, который мы вводили ранее для индикатора *IndUnityPercent* в разделе Мультивалютные и мультитаймфреймовые индикаторы. Его особенность заключается в том, что помимо обычного доступа к элементам массива по номерам можно использовать адресацию по ключу.

Для заполнения массива в конструктор передается массив целых чисел, которые предварительно проверяются на соответствие идентификаторам заданного перечисления E с помощью метода *detect*. Все правильные свойства тут же считываются через вызов *get* и полученные значения сохраняются в карте совместно с их идентификаторами.

```
#include <MQL5Book/MapArray.mqh>

template<typename T,typename E>
class ChartModeMonitorBase: public ChartModeMonitorInterface
{
protected:
   MapArray<E,T> data; // массив-карта пар [свойство,значение]

   // метод проверяет, является ли переданная константа элементом перечисления,
   // и если да, то добавляет его в массив-карту
   bool detect(const int v)
   {
      ResetLastError();
      EnumToString((E)v); // результирующая строка не используется
      if(_LastError == 0) // важно лишь, есть ошибка или нет
      {
         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;
      // собираем текущее состояние всех свойств
      for(int i = 0; i < data.getSize(); ++i)
      {
         temp.put(data.getKey(i), get(data.getKey(i)));
      }

      // сравниваем с предыдущим состоянием, выводим различия
      for(int i = 0; i < data.getSize(); ++i)
      {
         if(data[i] != temp[i])
         {
            Print(EnumToString(data.getKey(i)), " ", data[i], " -> ", temp[i]);
         }
      }

      // сохраняем для следующего сравнения
      data = temp;
```

```
   }
   ...
};
```

Метод *snapshot* проходит в цикле по всем элементам массива и для каждого свойства запрашивает его значение. Поскольку мы хотим обнаруживать изменения, то новые данные сначала сохраняются во временный массив-карту *temp*. Затем массивы *data* и *temp* поэлементно сравниваются, и на каждое различие выводится сообщение с названием свойства, его старым и новым значением. В данном упрощенном примере просто используется журнал, но при необходимости программа может вызывать некие прикладные функции, адаптирующие поведение под среду.

Методы *print*, *backup* и *restore* реализованы максимально просто.

```
template<typename T,typename E>
class ChartModeMonitorBase: public ChartModeMonitorInterface
{
protected:
   ...
   MapArray<E,T> store; // бэкап
public:
   ...
   virtual void print() override
   {
      data.print();
   }
   virtual void backup() override
   {
      store = data;
   }

   virtual void restore() override
   {
      data = store;
      // восстанавливаем свойства графика
      for(int i = 0; i < data.getSize(); ++i)
      {
         set(data.getKey(i), data[i]);
      }
   }
```

Связка методов *backup/restore* позволяет сохранить состояние графика перед началом экспериментов с ним, а после завершения работы тестового скрипта, восстановить всё, как было.

Наконец, последний класс в файле *ChartModeMonitor.mqh* так и называется *ChartModeMonitor*. Он объединяет в себе три экземпляра *ChartModeMonitorBase*, создаваемых под имеющиеся сочетания типов свойств. Для них выделен массив *m* указателей на базовый интерфейс *ChartModeMonitorInterface*. Сам класс также является производным от него.

```
#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());
   }
};
```

Для упрощения кода здесь использован макрос CALL_ALL, который вызывает указанный метод для всех объектов из массива, причем делает это с учетом перегруженного оператора [] в классе *AutoPtr* (он применяется для разыменования умного указателя и получения прямого указателя на "охраняемый" объект).

За освобождение объектов обычно отвечает деструктор, но в данном случае было решено применить массив *AutoPtr* (этот класс был рассмотрен в разделе Шаблоны объектных типов). Это гарантирует автоматическое удаление динамических объектов при штатном освобождении массива *m*.

Более полная версия монитора с поддержкой номеров подокон поставляется в файле *ChartModeMonitorFull.mqh*.

На основе класса *ChartModeMonitor* можно легко реализовать задуманный скрипт *ChartMode.mq5*. Его задача — проверять состояние заданного набора свойств каждые полсекунды. Пока мы

используем здесь бесконечный цикл и *Sleep*, но скоро научимся реагировать на события на графиках по-другому: за счет уведомлений от терминала.

```
#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();
}
```

Набросьте скрипт на любой график и попробуйте менять режимы с помощью инструментальных кнопок — так доступны все элементы, кроме CHART_FOREGROUND, который можно переключить из диалога свойств (закладка *Общие*, флаг *График сверху*).

Кнопки инструментальной панели для переключения режимов графика

Например, следующий журнал получился в результате переключения отображения со свечей на бары, с баров на линии и обратно на свечи, а затем — включения отступа и автопрокрутки к началу.

```
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
```

Более практичным примером использования свойства CHART_MODE является усовершенствованная версия индикатора *IndSubChart.mq5* (с его упрощенной версией *IndSubChartSimple.mq5* мы познакомились в разделе Мультивалютные и мультитаймфреймовые индикаторы). Напомним, что индикатор предназначен для отображения котировок стороннего символа в подокне, и ранее нам приходилось запрашивать способ отображения (свечи, бары, линии) у пользователя через входной параметр. Сейчас необходимость в параметре отпала, потому что мы можем автоматически переключать индикатор в тот режим, который используется в основном окне.

Текущий режим хранится в глобальной переменной *mode* и первый раз присваивается во время инициализации.

```
ENUM_CHART_MODE mode = 0;

int OnInit()
{
   ...
   mode = (ENUM_CHART_MODE)ChartGetInteger(0, CHART_MODE);
   ...
}
```

Детектирование нового режима правильнее всего делать в специально предназначенном для этого обработчике события *OnChartEvent* — мы изучим его в отдельной главе. На данном этапе важно знать, что при любом изменении графика MQL-программа может получать уведомления от терминала, если в коде будет описана функция с этим предопределенным прототипом (именем и списком параметров). В частности, в её первом параметре передается идентификатор события, описывающий его суть. Нас пока интересует сам график, в связи с чем мы проверяем *eventId* на равенство CHARTEVENT_CHART_CHANGE. Это нужно, поскольку обработчик также способен отслеживать графические объекты, клавиатуру, мышь и произвольные пользовательские сообщения.

```
void OnChartEvent(const int eventId,
                  // неиспользуемые здесь параметры
                  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;
         // изменяем привязку буферов и тип отрисовки на ходу
         InitPlot(0, InitBuffers(mode), Mode2Style(mode));
         // TODO: чуть позже сделаем автонастройку цветов
         // SetPlotColors(0, mode);
         if(oldmode == CHART_LINE || newmode == CHART_LINE)
         {
            // переключение на или с режима CHART_LINE требует обновить весь график,
            // потому что меняется количество буферов
            Print("Refresh");
            ChartSetSymbolPeriod(0, _Symbol, _Period);
         }
         else
         {
            // при переключении между свечами и барами достаточно
            // просто перерисовать график на новый манер,
            // потому что данные не меняются (прежние 4 буфера со значениями)
            Print("Redraw");
            ChartRedraw();
         }
      }
   }
}
```

Вы можете проверить новый индикатор самостоятельно, набросив его на график и переключая способы отрисовки.

Это еще не все усовершенствования, сделанные в *IndSubChart.mq5*. Чуть позже, в разделе про цвета графиков мы покажем автоматическую подстройку диаграмм под цветовую схему графика.

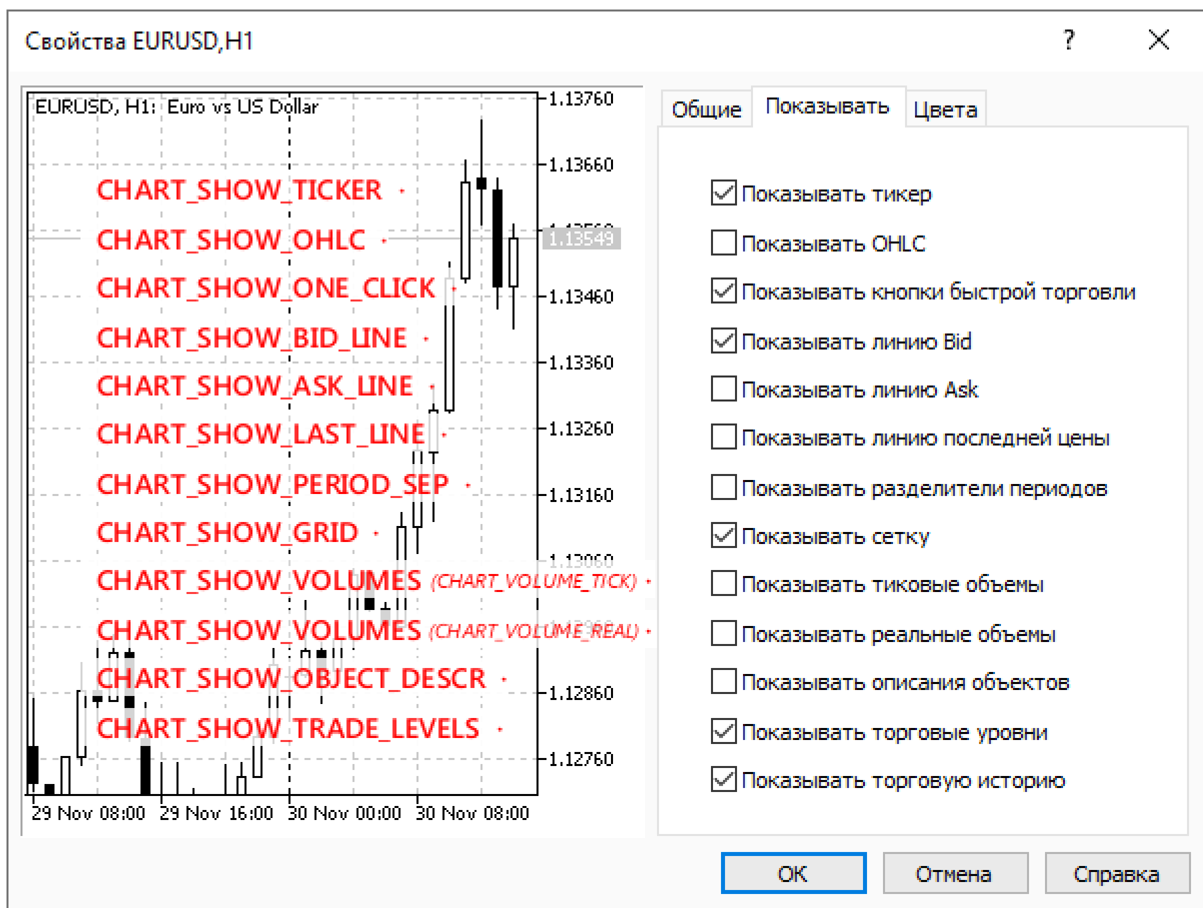
5.7.10 Управление видимостью элементов графика

Большой набор свойств в ENUM_CHART_PROPERTY_INTEGER управляет видимостью элементов графика. Почти все они имеют логический тип: *true* соответствует показу элемента, а *false* — сокрытию. Исключение составляет CHART_SHOW_VOLUMES, для которого используется перечисление ENUM_CHART_VOLUME_MODE (см. ниже).

Идентификатор	Описание	Тип значения
CHART_SHOW	Признак отрисовки ценового графика в	bool

Идентификатор	Описание	Тип значения
	целом. Если установлено значение false, то отключается отрисовка любых атрибутов ценового графика и устраняются все отступы по краям графика: шкалы времени и цены, строка быстрой навигации, метки событий календаря, значки сделок, всплывающие подсказки индикаторов и баров, подокна индикаторов, гистограммы объёмов и т.д.	
CHART_SHOW_TICKER	Отображение в левом верхнем углу тикера символа. Отключение тикера автоматически отключает OHLC (CHART_SHOW_OHLC)	bool
CHART_SHOW_OHLC	Отображение в левом верхнем углу значений OHLC. Включение OHLC автоматически включает тикер (CHART_SHOW_TICKER)	bool
CHART_SHOW_BID_LINE	Отображение значения Bid горизонтальной линией	bool
CHART_SHOW_ASK_LINE	Отображение значения Ask горизонтальной линией	bool
CHART_SHOW_LAST_LINE	Отображение значения Last горизонтальной линией	bool
CHART_SHOW_PERIOD_SEP	Отображение вертикальных разделителей между соседними периодами	bool
CHART_SHOW_GRID	Отображение сетки на графике	bool
CHART_SHOW_VOLUMES	Отображение объемов на графике	ENUM_CHART_VOLUME_MODE
CHART_SHOW_OBJECT_DESCR	Отображение текстовых описаний объектов (описания показываются не для всех типов объектов)	bool
CHART_SHOW_TRADE_LEVELS	Отображение на графике торговых уровней (уровни открытых позиций, Stop Loss, Take Profit и отложенных ордеров)	bool
CHART_SHOW_DATE_SCALE	Отображение на графике шкалы времени	bool
CHART_SHOW_PRICE_SCALE	Отображение на графике ценовой шкалы	bool

Идентификатор	Описание	Тип значения
CHART_SHOW_ONE_CLICK	Отображение на графике панели быстрой торговли (опция "Торговля одним кликом")	bool



Флаги в диалоге настроек для некоторых свойств ENUM_CHART_PROPERTY_INTEGER

Некоторые из этих свойств доступны пользователю из контекстного меню графика, некоторые — только из диалога настроек, однако есть и такие, которые можно менять только из MQL5, в частности, отображение вертикальной (CHART_SHOW_DATE_SCALE) и горизонтальной (CHART_SHOW_DATE_SCALE) шкалы, а также видимость всего графика целиком (CHART_SHOW). Последний случай следует отметить особо, потому что отключение отрисовки является идеальным решением для создания собственного интерфейса программы с использованием графических ресурсов и графических объектов — они отрисовываются всегда, независимо от значения CHART_SHOW.

К книге прилагается скрипт *ChartBlackout.mq5*, который переключает режим CHART_SHOW из текущего на обратный при каждом запуске.

```
void OnStart()
{
   ChartSetInteger(0, CHART_SHOW, !ChartGetInteger(0, CHART_SHOW));
}
```

Таким образом, можно применить его на обычном графике, чтобы полностью очистить окно, а потом применить повторно, чтобы восстановить прежний внешний вид.

Вышеупомянутое перечисление ENUM_CHART_VOLUME_MODE содержит следующие элементы.

Идентификатор	Описание	Значение
CHART_VOLUME_HIDE	Объемы скрыты	0
CHART_VOLUME_TICK	Тиковые объемы	1
CHART_VOLUME_REAL	Торговые объемы (если есть)	2

По аналогии со скриптом *ChartMode.mq5* реализуем монитор видимости элементов графика в скрипте *ChartElements.mq5*. Основное отличие заключается в другом наборе контролируемых флагов.

```
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
   };
   ...
```

Кроме того, после создания бэкапа настроек мы намеренно отключаем шкалы времени и цены программным способом (при завершении работы скрипт восстановит их из бэкапа).

```
   ...
   m.backup();

   ChartSetInteger(0, CHART_SHOW_DATE_SCALE, false);
   ChartSetInteger(0, CHART_SHOW_PRICE_SCALE, false);
   ...
}
```

Далее приведен фрагмент журнала с комментариями о выполненных действиях. Первые две записи возникли как раз из-за того, что в MQL-коде шкалы были отключены уже после создания начального бэкапа.

```
CHART_SHOW_DATE_SCALE 1 -> 0   // отключили шкалу времени в MQL5-коде
CHART_SHOW_PRICE_SCALE 1 -> 0  // отключили шкалу цен в MQL5-коде
CHART_SHOW_ONE_CLICK 0 -> 1    // отключили "Торговля в один клик"
CHART_SHOW_GRID 1 -> 0         // отключили "Сетку"
CHART_SHOW_VOLUMES 0 -> 2      // показали реальные "Объемы"
CHART_SHOW_VOLUMES 2 -> 1      // показали "Тиковые объемы"
CHART_SHOW_TRADE_LEVELS 1 -> 0 // отключили "Торговые уровни"
```

5.7.11 Отступы по горизонтали

Еще одним нюансом отображения графиков являются горизонтальные отступы от левого и правого края. Работают они слегка по-разному, но описаны в одном перечислении ENUM_CHART_PROPERTY_DOUBLE и используют тип *double*.

Идентификатор	Описание
CHART_SHIFT_SIZE	Размер отступа нулевого бара от правого края в процентах (от 10 до 50), активен только при включенном режиме CHART_SHIFT. Отступ обозначается на графике маленьким перевернутым серым треугольником на верхней рамке, в правой части окна.
CHART_FIXED_POSITION	Положение фиксированной позиции графика от левого края в процентах (от 0 до 100). Фиксированная позиция графика обозначена маленьким серым треугольником на горизонтальной оси времени и показывается только в том случае, если отключена автоматическая прокрутка к правому краю при поступлении нового тика (CHART_AUTOSCROLL). Бар, который находится на фиксированной позиции, остаётся на том же месте при увеличении и уменьшении масштаба. По умолчанию, треугольник находится в самом углу графика (в левом нижнем).

Визуальное представление свойств горизонтальных отступов

Для проверки доступа к данным свойствам подготовлен скрипт *ChartShifts.mq5*, который работает аналогично *ChartMode.mq5* и отличается только набором контролируемых свойств.

```
void OnStart()
{
   int flags[] =
   {
      CHART_SHIFT_SIZE, CHART_FIXED_POSITION
   };
   ChartModeMonitor m(flags);
   ...
}
```

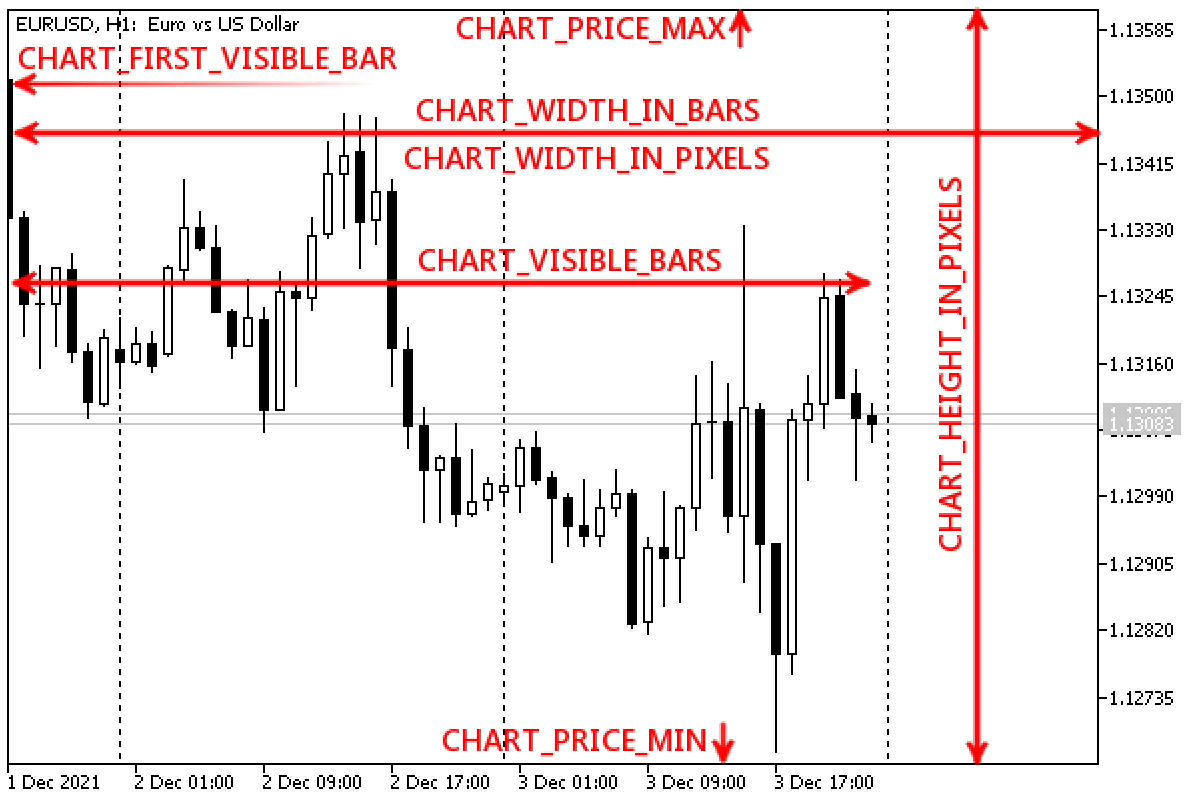
Перетаскивание мышью метки фиксированной позиции (слева внизу) приводит к такому выводу в журнал.

```
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 Горизонтальный масштаб (по времени)

Для определения масштаба и количества баров по горизонтальной оси следует использовать группу целочисленных свойств из ENUM_CHART_PROPERTY_INTEGER. Среди них только CHART_SCALE является редактируемым.

Идентификатор	Описание
CHART_SCALE	Масштаб (от 0 до 5)
CHART_VISIBLE_BARS	Количество баров, видимых в данный момент на графике (может быть меньше CHART_WIDTH_IN_BARS за счет отступа CHART_SHIFT_SIZE) (r/o)
CHART_FIRST_VISIBLE_BAR	Номер первого видимого бара на графике. Нумерация идет справа налево, как в таймсерии. (r/o)
CHART_WIDTH_IN_BARS	Ширина графика в барах (потенциальная вместимость, крайние бары слева и справа могут быть видны частично) (r/o)
CHART_WIDTH_IN_PIXELS	Ширина графика в пикселях (r/o)



Свойства ENUM_CHART_PROPERTY_INTEGER на графике

У нас все готово для реализации очередного тестового скрипта *ChartScaleTime.mq5*, позволяющего анализировать изменения этих свойств.

```
void OnStart()
{
   int flags[] =
   {
      CHART_SCALE,
      CHART_VISIBLE_BARS,
      CHART_FIRST_VISIBLE_BAR,
      CHART_WIDTH_IN_BARS,
      CHART_WIDTH_IN_PIXELS
   };
   ChartModeMonitor m(flags);
   ...
}
```

Ниже показан фрагмент журнала с комментариями о выполненных действиях.

```
Initial state:
    [key] [value]
[0]     5       4
[1]   100      35
[2]   104      34
[3]   105      45
[4]   106     715
                                  // 1) изменили масштаб на более мелкий:
CHART_SCALE 4 -> 3                // - изменилось значение свойства "масштаб"
CHART_VISIBLE_BARS 35 -> 69       // - увеличилось количество видимых баров
CHART_FIRST_VISIBLE_BAR 34 -> 68  // - увеличился номер первого видимого бара
CHART_WIDTH_IN_BARS 45 -> 90      // - увеличилось потенциальное количество баров
                                  // 2) отключили отступ у правого края
CHART_VISIBLE_BARS 69 -> 89       // - увеличилось количество видимых баров
CHART_FIRST_VISIBLE_BAR 68 -> 88  // - увеличился номер первого видимого бара
                                  // 3) уменьшили размер окна
CHART_VISIBLE_BARS 89 -> 86       // - уменьшилось количество видимых баров
CHART_WIDTH_IN_BARS 90 -> 86      // - уменьшилось потенциальное количество баров
CHART_WIDTH_IN_PIXELS 715 -> 680  // - уменьшилась ширина в пикселях
                                  // 4) нажали кнопку "End" для перехода к текущему в
CHART_VISIBLE_BARS 86 -> 85       // - уменьшилось количество видимых баров
CHART_FIRST_VISIBLE_BAR 88 -> 84  // - уменьшился номер первого видимого бара
```

5.7.13 Вертикальный масштаб (по цене и показаниям индикаторов)

Установка и анализ свойств, относящихся к масштабу по вертикали, производится с помощью элементов двух перечислений: ENUM_CHART_PROPERTY_INTEGER и ENUM_CHART_PROPERTY_DOUBLE. В следующей таблице свойства указаны вместе с типом их значений.

Некоторые свойства позволяют обращаться не только к основному окну, но и подокну, для чего в *ChartSet*- и *ChartGet*-функциях следует задействовать параметр *window* (0 означает основное окно и является значением по умолчанию для краткой формы *ChartGet*).

Идентификатор	Описание	Тип значений
CHART_SCALEFIX	Режим фиксированного масштаба	bool
CHART_FIXED_MAX	Фиксированный максимум подокна *window* или начальный максимум основного окна	double
CHART_FIXED_MIN	Фиксированный минимум подокна *window* или начальный минимум основного окна	double
CHART_SCALEFIX_11	Режим масштаба 1:1	bool
CHART_SCALE_PT_PER_BAR	Режим указания масштаба в пунктах на бар	bool
CHART_POINTS_PER_BAR	Значение масштаба в пунктах на бар	double
CHART_PRICE_MIN	Минимум значений в окне или подокне *window* (r/o)	double
CHART_PRICE_MAX	Максимум значений в окне или подокне *window* (r/o)	double
CHART_HEIGHT_IN_PIXELS	Фиксированная высота окна или подокна в пикселях, требуется параметр *window*	int
CHART_WINDOW_YDISTANCE	Дистанция в пикселях по вертикальной оси Y между верхней рамкой подокна *window* и верхней рамкой основного окна графика. (r/o)	int

По умолчанию графики поддерживают адаптивный масштаб, чтобы котировки или линии индикаторов полностью умещались по вертикали на видимом временном отрезке. Для некоторых применений желательно фиксировать масштаб, в связи с чем терминал предлагает несколько режимов. В них график можно прокручивать мышью или клавишами (Shift+стрелка) не только влево/вправо, но и вверх/вниз, а у правой шкалы появляется "движок", за который можно быстро проматывать график мышью.

Фиксированный режим устанавливается включением флага CHART_SCALEFIX и указанием требуемых максимума и минимума в полях CHART_FIXED_MAX и CHART_FIXED_MIN (в основном окне пользователь сможет передвинуть график вверх или вниз, из-за чего значения CHART_FIXED_MAX и CHART_FIXED_MIN синхронно изменятся, но вертикальный масштаб сохранится). Также пользователь сможет изменить вертикальный масштаб, нажав кнопку мыши на шкале цен и, не отпуская её, двигая вверх или вниз. В подокнах интерактивное редактирование вертикального масштаба не предусмотрено. В связи с этим мы позднее представим индикатор *SubScaler.mq5* (см. раздел про события клавиатуры), который позволит пользователю управлять диапазоном значений в подокне с помощью клавиатуры (а не из диалога настрок, с помощью полей на закладке *Шкала*).

Режим CHART_SCALEFIX_11 обеспечивает примерное видимое равенство сторон квадрата на экране: X баров в пикселях (по горизонтали) будут равны X пунктам в пикселях (по вертикали). Равенство примерное, потому что размер пикселей, как правило, не одинаков по вертикали и горизонтали.

Наконец, существует режим фиксации соотношения количества пунктов на бар — он включается опцией CHART_SCALE_PT_PER_BAR, а само требуемое соотношение задается с помощью свойства CHART_POINTS_PER_BAR. В отличие от режима CHART_SCALEFIX, пользователь не сможет интерактивно изменить масштаб мышью на графике. В этом режиме горизонтальная дистанция в один бар будет отображаться на экране в таком же отношении к заданному количеству пунктов по вертикали, как соотношение сторон графика (в пикселях). При равенстве таймфремов и размеров двух графиков, один будет выглядеть сжатым по цене по сравнению с другим в соответствии с отношением их величин CHART_POINTS_PER_BAR. Очевидно, что чем меньше таймфрейм, тем меньше размах баров, и потому при одном и том же масштабе мелкие таймфреймы выглядят более "сплюснутыми".

Программная установка свойства CHART_HEIGHT_IN_PIXELS делает невозможным редактирование размера окна/подокна пользователем. Это часто используется для окон, в которых размещаются торговые панели с предопределенным набором элементов управления (кнопок, полей ввода и т.д.). Для того чтобы убрать фиксацию размера, следует установить значение свойства в -1.

Значение CHART_WINDOW_YDISTANCE требуется для перевода абсолютных координат основного графика в локальные координаты подокна для корректной работы с графическими объектами. Дело в том, что при наступлении событий мыши координаты курсора передаются относительно основного окна графика, в то время как координаты графических объектов в подокне индикатора задаются относительно верхнего левого угла подокна.

Подготовим скрипт для анализа изменений вертикальных масштабов и размеров *ChartScalePrice.mq5*.

```
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);
   ...
}
```

Он реагирует на манипуляции с графиком следующим образом:

```
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
// уменьшили вертикальный размер окна
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
// уменьшили горизонтальный масштаб, из-за чего увеличился диапазон цен
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
// с помощью мыши расширили диапазон цен (котировки "сжались" по вертикали)
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 Цвета

MQL-программа может узнать и изменить цвета для отображения всех элементов графика. Соответствующие свойства входят в состав перечисления ENUM_CHART_PROPERTY_INTEGER.

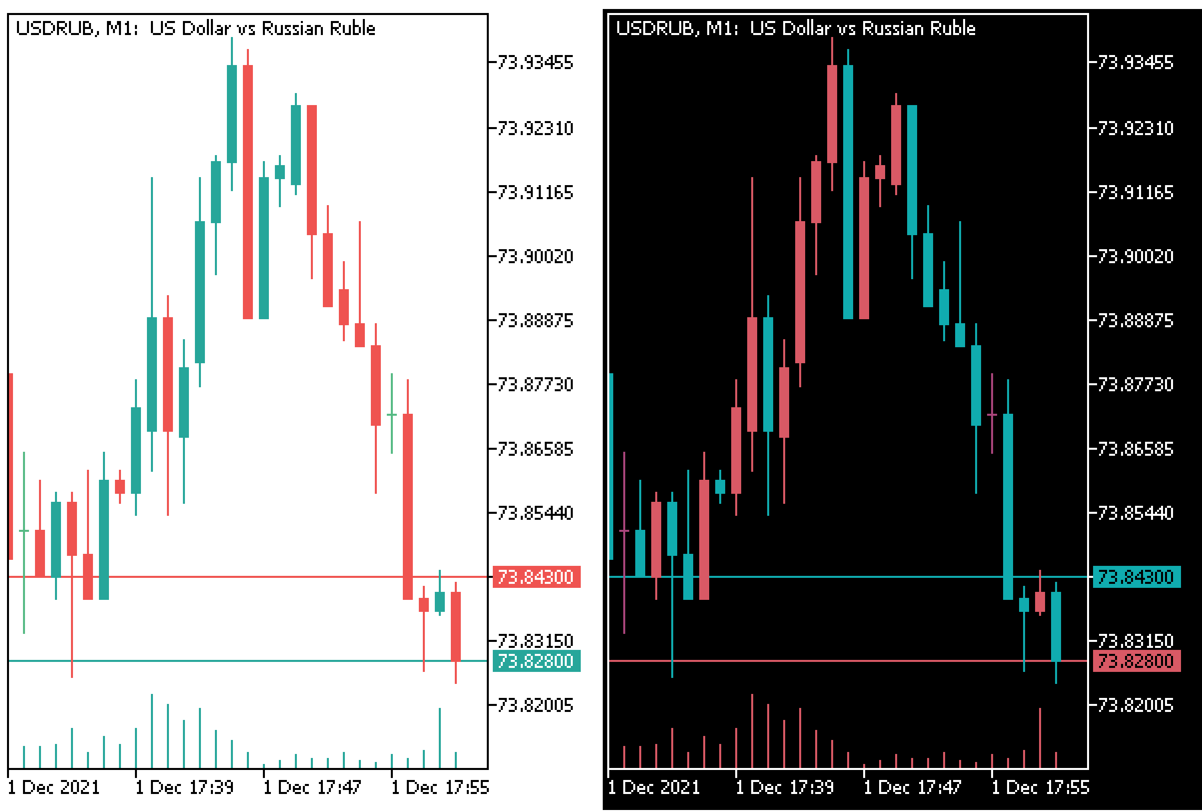
Идентификатор	Описание
CHART_COLOR_BACKGROUND	Цвет фона графика
CHART_COLOR_FOREGROUND	Цвет осей, шкалы и строки OHLC
CHART_COLOR_GRID	Цвет сетки
CHART_COLOR_VOLUME	Цвет объемов и уровней открытия позиций
CHART_COLOR_CHART_UP	Цвет бара вверх, тени и окантовки тела бычьей свечи

Идентификатор	Описание
CHART_COLOR_CHART_DOWN	Цвет бара вниз, тени и окантовки тела медвежьей свечи
CHART_COLOR_CHART_LINE	Цвет линии графика и контуров японских свечей
CHART_COLOR_CANDLE_BULL	Цвет тела бычьей свечи
CHART_COLOR_CANDLE_BEAR	Цвет тела медвежьей свечи
CHART_COLOR_BID	Цвет линии Bid-цены
CHART_COLOR_ASK	Цвет линии Ask-цены
CHART_COLOR_LAST	Цвет линии цены последней совершенной сделки (Last)
CHART_COLOR_STOP_LEVEL	Цвет уровней стоп-ордеров (Stop Loss и Take Profit)

В качестве примера работы с данными свойствами создадим скрипт *ChartColorInverse.mq5*, который будет менять все цвета графика на инверсивные, то есть для битового представления цвета в формате RGB будет выполняться операция XOR ('^', исключающее ИЛИ). Таким образом, после повторного запуска скрипта на том же графике его настройки будут восстановлены.

```
#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])));
   }
}
```

На следующем изображении совмещены образы графика до и после применения скрипта.



Инверсия цветов графика из MQL-программы

Теперь закончим доработку *IndSubChart.mq5*. Нам осталось прочитать цвета основного графика и применить их к диаграмме нашего индикатора. Для этих целей уже была запланирована функция *SetPlotColors*, вызов которой был закомментирован в обработчике *OnChartEvent* (см. последний пример в разделе Режимы отображения графика).

```
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_CC
      PlotIndexSetInteger(index, PLOT_LINE_COLOR, 1, (int)ChartGetInteger(0, CHART_CC
      PlotIndexSetInteger(index, PLOT_LINE_COLOR, 2, (int)ChartGetInteger(0, CHART_CC
   }
   else
   {
      PlotIndexSetInteger(index, PLOT_COLOR_INDEXES, 1);
      PlotIndexSetInteger(index, PLOT_LINE_COLOR, (int)ChartGetInteger(0, CHART_COLOR
   }
}
```

В этой новой функции мы получаем, в зависимости от режима отрисовки графика, либо цвета контуров и тел бычьих и медвежьих свечей, либо цвет линий, и применяем цвета к диаграммам. Разумеется, не забудем вызвать эту функцию при инициализации.

```
int OnInit()
{
   ...
   mode = (ENUM_CHART_MODE)ChartGetInteger(0, CHART_MODE);
   InitPlot(0, InitBuffers(mode), Mode2Style(mode));
   SetPlotColors(0, mode);
   ...
}
```

Индикатор готов. Попробуйте набросить его в окно и поменять цвета в диалоге свойств графика — диаграмма должна автоматически адаптироваться по новые настройки.

5.7.15 Управление мышью и клавиатурой

В данном разделе мы познакомимся с группой свойств, которые влияют на перехват графиком некоторых манипуляций с мышью и клавиатурой, которые расцениваются по умолчанию как управляющие действия. В частности, пользователи MetaTrader 5 хорошо знают, что график можно прокручивать мышью, а для выполнения наиболее востребованных команд можно вызвать контекстное меню. MQL5 позволяет отключить такое поведение графика полностью или частично. Важно отметить, что сделать это можно только программно: аналогичных опций в пользовательском интерфейсе терминала нет.

Исключение составляет лишь опция CHART_DRAG_TRADE_LEVELS (см. в таблице ниже): для неё в настройках терминала, на закладке *Графики* имеется выпадающий список, управляющий разрешением перетаскивать торговые уровни мышью.

Все свойства данной группы имеют логический тип (*true* — действие разрешено, *false* — действие отключено) и потому находятся в перечислении ENUM_CHART_PROPERTY_INTEGER.

Идентификатор	Описание
CHART_CONTEXT_MENU	Включение/отключение доступа к контекстному меню по нажатию правой клавиши мышки. Значение *false* отключает только контекстное меню графика, при этом контекстное меню для объектов на графике остается доступным. Значение по умолчанию *true*.
CHART_CROSSHAIR_TOOL	Включение/отключение доступа к инструменту "Перекрестие" по нажатию средней клавиши мышки. Значение по умолчанию *true*.
CHART_MOUSE_SCROLL	Прокрутка графика левой кнопкой мышки или колесиком. Когда прокрутка разрешена, это относится не только к прокрутке по горизонтали, но и по вертикали, однако последняя доступна только при установленном фиксированном масштабе: одном из свойств CHART_SCALEFIX, CHART_SCALEFIX_11 или CHART_SCALE_PT_PER_BAR. Значение по умолчанию *true*.
CHART_KEYBOARD_CONTROL	Разрешение на управление графиком с клавиатуры (кнопками *Home*, *End*, *PageUp/PageDown*, +/-, стрелки вверх/вниз и т.д.). Установка в значение *false* позволяет отключить прокрутку и масштабирование графика, но при этом сохраняется

Идентификатор	Описание
	возможность получать события нажатия данных клавиш в *OnChartEvent*. Значение по умолчанию *true*.
CHART_QUICK_NAVIGATION	Разрешение на работу в графике строки быстрой навигации, которая автоматически появляется в левом углу шкалы времени при двойном клике мышки или нажатии клавиш *Пробел* или *Ввод*. Таким образом можно быстро сменить символ, таймфрейм или дату первого видимого бара. По умолчанию свойство равно true, и активация быстрой навигации включена.
CHART_DRAG_TRADE_LEVELS	Разрешение на перетаскивание торговых уровней на графике с помощью мышки. По умолчанию режим перетаскивания включен (*true*).

В тестовом скрипте *ChartInputControl.mq5* установим монитор на все вышеперечисленные свойства, а кроме того предусмотрим входные переменные для произвольной установки значений пользователем. За счет того, что наш скрипт сохраняет резервную копию настроек при запуске, все измененные свойства будут восстановлены при завершении работы скрипта.

```
#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();
}
```

Например, мы можем при запуске скрипта сбросить в *false* разрешения для контекстного меню, инструмента "перекрестие", управления мышью и клавиатурой. В результате получим следующий журнал.

```
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
```

При этом вы не сможете двигать график ни мышью, ни клавиатурой, и даже вызвать контекстное меню. Поэтому для того, чтобы восстановить его работоспособность, придется набросить на график этот же или другой скрипт (напомним, что скрипт на графике может быть только один, и при нанесении нового происходит выгрузка предыдущего;). Новый экземпляр скрипта достаточно набросить, но не запускать (нажать *Отмена* в диалоге ввода входных переменных).

5.7.16 Открепление окна графика

Окна графиков в терминале могут быть откреплены от главного окна, после чего их можно двигать в произвольное место рабочего стола, включая другие мониторы. MQL5 позволяет узнать и изменить данную настройку: соответствующие свойства включены в перечисление ENUM_CHART_PROPERTY_INTEGER.

Идентификатор	Описание	Тип значения
CHART_IS_DOCKED	Окно графика закреплено (по умолчанию true). Если установить значение false, то график можно перетащить за пределы терминала	bool
CHART_FLOAT_LEFT	Левая координата открепленного графика относительно виртуального экрана	int
CHART_FLOAT_TOP	Верхняя координата открепленного графика относительно виртуального экрана	int
CHART_FLOAT_RIGHT	Правая координата открепленного графика относительно виртуального экрана	int
CHART_FLOAT_BOTTOM	Нижняя координата открепленного графика относительно виртуального экрана	int

В скрипте *ChartDock.mq5* установим слежение за данными свойствами.

```
void OnStart()
{
   const int flags[] =
   {
      CHART_IS_DOCKED,
      CHART_FLOAT_LEFT, CHART_FLOAT_TOP, CHART_FLOAT_RIGHT, CHART_FLOAT_BOTTOM
   };
   ChartModeMonitor m(flags);
   ...
}
```

Если теперь запустить скрипт, а затем открепить график с помощью контекстного меню (отжать команду-переключатель *Закреплен*) и подвигать или изменить размер, получим в журнале соответствующие записи.

```
Initial state:
    [key] [value]
[0]    51       1
[1]    52       0
[2]    53       0
[3]    54       0
[4]    55       0
                               // открепили
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
                               // изменили вертикальный размер
CHART_FLOAT_BOTTOM 472 -> 500
CHART_FLOAT_BOTTOM 500 -> 539
                               // изменили горизонтальный размер
CHART_FLOAT_RIGHT 1263 -> 1024
CHART_FLOAT_RIGHT 1024 -> 1023
                               // прикрепили обратно
CHART_IS_DOCKED 0 -> 1
```

На данном разделе завершается описание свойств, управляемых через *ChartGet*- и *ChartSet*-функции, потому подытожим материал с помощью общего скрипта *ChartFullSet.mq5*. Он отслеживает состояние всех свойств всех типов. Инициализация массива флагов производится просто заполнением последовательными индексами в цикле. Максимальное значение взято с запасом на случай появления новых свойств, а лишние несуществующие номера будут автоматически отброшены проверкой, встроенной в класс *ChartModeMonitorBase* (вспоминаем метод *detect*).

Активировав скрипт, попробуйте менять любые настройки, наблюдая за сообщениями программы в журнале.

5.7.17 Получение координат буксировки MQL-программы на график

Пользователи часто наносят MQL-программы на график с помощью мыши. Это не только удобно, но и позволяет задать некий контекст для алгоритма. Например, индикатор может быть применен

в разных подокнах или скрипт может установить отложенный ордер на той цене, где пользователь отпустил его на график. Следующая группа функций предназначена для получения координат точки, в которую была отбуксирована программа.

int ChartWindowOnDropped()

Данная функция возвращает номер подокна графика, на которое брошен мышкой текущий эксперт, скрипт или индикатор. Главное окно, как мы знаем, имеет номер 0, а подокна нумеруются, начиная с 1. Номер подокна не зависит от того, есть ли над ним скрытые подокна — их индексы остаются закрепленными за ними. Иными словами, видимый номер подокна может отличаться от его реального индекса, если на графике имеются скрытые подокна.

double ChartPriceOnDropped()
datetime ChartTimeOnDropped()

Эта пара функций возвращает координаты точки "заброски" программы в единицах цены и времени. Обратите внимание, что в подокнах могут выводиться произвольные данные, а не только цены, хотя в названии функции *ChartPriceOnDropped* фигурирует "Price".

> Внимание! Время целевой точки не округляется по размеру таймфрейма графика, поэтому даже на графиках H1, D1 можно получить значение с минутами и даже секундами.

int ChartXOnDropped()
int ChartYOnDropped()

Данные две функции возвращают экранные координаты точки по осям X и Y в пикселях. Начало координат находится в левом верхнем углу основного окна графика. Про направление осей мы рассказывали в разделе Характеристики экрана.

Координата Y всегда отсчитывается от левого верхнего угла основного графика, даже если точка "заброски" принадлежит подокну. Для перевода этого значения в координату *y* относительно подокна следует использовать свойство CHART_WINDOW_YDISTANCE (см. пример).

В скрипте *ChartDrop.mq5* выведем в журнал значения всех упомянутых функций.

```
void OnStart()
{
   const int w = PRTF(ChartWindowOnDropped());
   PRTF(ChartTimeOnDropped());
   PRTF(ChartPriceOnDropped());
   PRTF(ChartXOnDropped());
   PRTF(ChartYOnDropped());

   // для подокна пересчитаем координату y в локальную
   if(w > 0)
   {
      const int y = (int)PRTF(ChartGetInteger(0, CHART_WINDOW_YDISTANCE, w));
      PRTF(ChartYOnDropped() - y);
   }
}
```

Например, если набросить этот скрипт в первое подокно, где запущен индикатор WPR, можем получить такие результаты.

```
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
```

Несмотря на то, что скрипт набрасывался на график EURUSD,H1, мы получили временную метку с минутами и секундами.

Обратите внимание, что значение "цены" равно -50, поскольку диапазон значений WPR равен [0,-100].

Кроме того, вертикальная координата точки 312 (относительно всего окна графика) была переведена в локальную координату подокна: поскольку расстояние по вертикали от начала основного графика до подокна составило 282, значение *y* внутри подокна оказалось равным 30.

5.7.18 Перевод экранных координат во время/цену и обратно

Наличие разных принципов измерения рабочего пространства графика приводит к необходимости пересчитывать единицы измерения между собой. Для этого предназначены две следующих функции.

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)

Функция *ChartTimePriceToXY* преобразует координаты графика из представления время/цена (*time/price*) в координаты по оси X и Y в пикселях (*x/y*). Функция *ChartXYToTimePrice* выполняет обратную операцию: преобразует координаты X и Y в значения времени и цены.

Обе функции требуют указания идентификатора графика в первом параметре *chartId*. Дополнительно к этому в *ChartTimePriceToXY* передается номер подокна *window* (должен быть в пределах количества окон). При наличии нескольких подокон в каждом из них существует собственная таймсерия и шкала по вертикальной оси (называемая условно "ценой" и параметром *price*).

В функции *ChartXYToTimePrice* параметр *window* является выходным и заполняется самой функцией наравне с *time* и *price*. Это связано с тем, что пиксельные координаты являются общими для всего экрана, и исходная точка *x/y* может попасть в любое подокно.

Координаты времени, цен и экранные

Функции возвращают *true* в случае успешного выполнения.

Следует учитывать, что в обеих системах координат видимая прямоугольная область, которая соответствует котировкам или экранным координатам, ограничена. Поэтому возможны ситуации, когда при конкретных исходных данных полученное время, цены или пиксели будут находиться вне зоны видимости. В частности, могут быть получены и отрицательные значения.

Мы рассмотрим интерактивный пример пересчета в главе про события на графиках, а пока вернемся к примеру с буксировкой.

В предыдущем разделе мы видели, как можно узнать место запуска MQL-программы. Хотя физически точка окончания буксировки одна, её представление в котировочных и экранных координатах, как правило, содержит погрешность вычислений. Убедиться в этом нам как раз помогут две новых функции для перевода пикселей в цену/время и обратно.

Модифицированный скрипт называется *ChartXY.mq5*. Его можно условно разделить на 3 этапа. На первом этапе выведем координаты точки сброса, как и раньше.

```
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());
   ...
```

На втором этапе попробуем преобразовать экранные координаты *x1* и *y1* во время (*t2*) и цену (*p2*), и сравним их с теми, что получены из *OnDropped*-функций выше.

```
   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);
   ...
```

Затем выполним обратное преобразование: из полученных котировочных координат *t1* и *p1* рассчитаем экранные — *x2* и *y2*, и также сравним с исходными значениями *x1* и *y1*.

```
   int x2, y2;
   PRTF(ChartTimePriceToXY(0, w1, t1, p1, x2, y2));
   Print(x2, " ", y2);
   PRTF(x1 == x2 && y1 == y2);
   ...
```

Как мы увидим далее из примера журнала, все вышеприведенные проверки закончатся неудачей (в значениях будут небольшие расхождения). Поэтому нам требуется третий этап.

Пересчитаем еще раз экранные и котировочные координаты с суффиксом 2 в названиях переменных и сохраним их в переменных с новым суффиксом 3. Затем сравним все значения с первого и третьего этапа между собой.

```
   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);
}
```

Запустим скрипт на графике XAUUSD,H1. Вот исходные данные точки.

```
ChartWindowOnDropped()=0 / ok
ChartTimeOnDropped()=2021.11.22 18:00:00 / ok
ChartPriceOnDropped()=1797.7 / ok
ChartXOnDropped()=234 / ok
ChartYOnDropped()=280 / ok
```

Пересчет пикселей в котировки дает следующие результаты.

```
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
```

Налицо отличия как по времени, так и цене. Обратный пересчет также небезупречен в плане точности.

```
ChartTimePriceToXY(0,w1,t1,p1,x2,y2)=true / ok
232 278
x1==x2&&y1==y2=false / ok
```

Потеря точности возникает из-за квантования значений на осях согласно единицам измерения, в частности, пикселям и пунктам.

Наконец, последний этап доказывает, что полученные выше погрешности не являются артефактами функций, потому что пересчет по кругу приводит к исходному результату.

```
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
```

В псевдокоде это можно выразить следующими равенствами:

```
ChartTimePriceToXY(ChartXYToTimePrice(XY)) = XY
ChartXYToTimePrice(ChartTimePriceToXY(TP)) = TP
```

Применение функции *ChartTimePriceToXY* к результатам работы *ChartXYToTimePrice* даст исходные координаты. То же самое верно и для преобразований в другую сторону: применение *ChartXYToTimePrice* к результатам *ChartTimePriceToXY* даст совпадение.

Таким образом, следует внимательно относиться к реализации алгоритмов, использующих функции пересчета, если к ним предъявляются повешенные требования по точности.

Еще один пример использования *ChartWindowOnDropped* будет приведен в скрипте *ChartIndicatorMove.mq5* в разделе Управление индикаторами на графике.

5.7.19 Прокрутка графика по оси времени

Пользователям MetaTrader 5 известна панель быстрой навигации по графику, которая открывается по двойному нажатию мыши в левом углу шкалы времени или по нажатию клавиш *Пробел* или *Ввод*. Аналогичная возможность доступна и программно — с помощью функции *ChartNavigate*.

bool ChartNavigate(long chartId, ENUM_CHART_POSITION position, int shift = 0)

Функция осуществляет сдвиг графика *chartId* на указанное количество баров относительно предопределенной позиции графика, заданной параметром *position*. Он имеет тип перечисления ENUM_CHART_POSITION со следующими элементами.

Идентификатор	Описание
CHART_BEGIN	Начало графика (самые старые цены)
CHART_CURRENT_POS	Текущая позиция
CHART_END	Конец графика (самые свежие цены)

Параметр *shift* задает количество баров, на которое необходимо сместить график. Положительное значение смещает график вправо (к концу), отрицательное значение смещает график влево (к началу).

Функция возвращает *true* в случае успеха или *false* в результате ошибки.

Для проверки работы функции создадим простой скрипт *ChartNavigate.mq5*. С помощью входных переменных пользователь может выбрать точку отсчета и сдвиг в барах.

```
#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);
}
```

В журнал выводится номер первого видимого бара до и после перемещения.

Более практичным примером будет скрипт *ChartSynchro.mq5*, который позволяет синхронно прокручивать все графики, на которых он запущен, в ответ на ручное прокручивание пользователем одного из графиков. Таким образом можно синхронизировать окна разных таймфреймов одного и того же инструмента или анализировать параллельные движения цен на разных инструментах.

```
void OnStart()
{
   datetime bar = 0; // текущая позиция (время первого видимого бара)
   
   const string namePosition = __FILE__; // имя глобальной переменной
   
   ChartSetInteger(0, CHART_AUTOSCROLL, false); // отключаем автопрокрутку
   
   while(!IsStopped())
   {
      const bool active = ChartGetInteger(0, CHART_BRING_TO_TOP);
      const int move = (int)ChartGetInteger(0, CHART_FIRST_VISIBLE_BAR);
      
      // активный график является ведущим, а остальные - ведомыми
      if(active)
      {
         const datetime first = iTime(_Symbol, _Period, move);
         if(first != bar)
         {
            // если позиция изменилась, сохраняем её в глобальной переменной
            bar = first;
            GlobalVariableSet(namePosition, bar);
            Comment("Chart ", ChartID(), " scrolled to ", bar);
         }
      }
      else
      {
         const datetime b = (datetime)GlobalVariableGet(namePosition);
         
         if(b != bar)
         {
            // если изменилось значение глобальной переменной, подстраиваем позицию
            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("");
}
```

Выравнивание производится по дате и времени первого видимого бара (CHART_FIRST_VISIBLE_BAR). Скрипт в цикле проверяет это значение и, если он работает на активном графике, записывает в глобальную переменную. Скрипты на остальных графиках читают эту переменную и подстраивают свою позицию соответствующим образом с помощью *ChartNavigate*. В параметрах указано относительное перемещение графика (CHART_CURRENT_POS), а количество баров для смещения определяется как разница между текущим номером первого видимого бара и тем, что прочитано из глобальной переменной.

На следующем изображении показан результат синхронизации графиков H1 и M15 для EURUSD.

Пример работы скрипта синхронизации позиции графиков

После того как мы познакомимся с системой событий на графиках, мы преобразуем данный скрипт в индикатор и избавимся от бесконечного цикла.

5.7.20 Запрос перерисовки графика

В большинстве случаев графики автоматически реагируют на изменения данных и настроек терминала, соответствующим образом перестраивая изображение окна (графики цен, диаграммы индикаторов и т.д.). Однако MQL-программы слишком разнообразны и могут выполнять произвольные действия, для которых не столь просто определить, требуется ли перерисовка или нет. Кроме того анализировать на этот счет любое действие каждой MQL-программы может быть накладно в плане падения общей производительности терминала. Поэтому MQL5 API предоставляет функцию *ChartRedraw*, с помощью которой сама MQL-программа может при необходимости запросить перерисовку графика.

void ChartRedraw(long chartId = 0)

Функция вызывает принудительную перерисовку графика с указанным идентификатором (значение по умолчанию 0 — означает текущий график). Обычно она применяется после того, как программа изменяет свойства графика или размещенных на нем объектов.

Мы видели пример использования *ChartRedraw* в индикаторе *IndSubChart.mq5* в разделе Режимы отображения графика. Еще один пример будет приведен в разделе Открытие и закрытие графиков.

Данная функция влияет именно на перерисовку графика, не вызывая пересчета таймсерий с котировками и индикаторов. Последний вариант обновления (фактически, перестроения)

графика является более "жестким" и выполняется другой функцией *ChartSetSymbolPeriod* (см. следующий раздел).

5.7.21 Переключение символа и таймфрейма

Иногда MQL-программе требуется переключить текущий символ или таймфрейм графика. В частности, это является привычным функционалом для многих мультивалютных, мультитаймфреймных торговых панелей или утилит анализа торговой истории. Для этой цели в MQL5 API имеется функция *ChartSetSymbolPeriod*.

С её помощью также удобно инициировать пересчет всего графика, включая расположенные на нем индикаторы, — достаточно в качестве параметров указать текущие символ и таймфрейм. Данный прием может пригодиться для индикаторов, которые не смогли полностью рассчитаться при первом вызове *OnCalculate* и ожидают подгрузки сторонних данных ("чужих" символов, тиков или других индикаторов). Также смена символа/таймфрейма влечет за собой переинициализацию эксперта, прикрепленного к графику. Скрипт (если он выполняется периодически в цикле) при данной процедуре и вовсе пропадет с графика (будет выгружен на старом сочетании символ/таймфрейм, но не загрузится автоматически на новом сочетании).

bool ChartSetSymbolPeriod(long chartId, string symbol, ENUM_TIMEFRAMES timeframe)

Функция меняет символ и таймфрейм указанного графика с идентификатором *chartId*, на значения соответствующих параметров: *symbol* и *timeframe*. 0 в параметре *chartId* означает текущий график, NULL в параметре *symbol* — текущий символ, а 0 в параметре *timeframe* — текущий таймфрейм.

Изменения вступают в силу асинхронно, то есть функция только посылает терминалу команду и не ждет ее выполнения. Команда поступает в очередь сообщений графика и выполняется только после обработки всех предыдущих команд.

Функция возвращает *true* в случае удачного помещения команды в очередь графика или *false* в случае проблем. Информацию об ошибке можно получить в *_LastError*.

Мы видели примеры использования функции для обновления нескольких индикаторов, в частности:

- - *IndDeltaVolume.mq5* (см. Ожидание данных и управление видимостью)
- - *IndUnityPercent.mq5* (см. Мультивалютные и мультитаймфреймовые индикаторы)
- - *UseWPRMTF.mq5* (см. Поддержка множества символов и таймфреймов)
- - *UseM1MA.mq5* (см. Использование встроенных индикаторов)
- - *UseDemoAllLoop.mq5* (см. Удаление экземпляров индикаторов)
- - *IndSubChart.mq5* (см. Режимы отображения графика)

5.7.22 Управление индикаторами на графике

Как мы уже выяснили, графики являются средой выполнения и визуализации индикаторов. Их тесная связь находит дополнительное подтверждение в виде целой группы встроенных функций, обеспечивающих управление индикаторами на графиках. В одно из предыдущих глав мы уже делали обзор этих функций. Теперь мы готовы рассмотреть их детально, после того как познакомились с графиками.

Все функции объединяет тот факт, что первые два параметра являются унифицированными: это идентификатор графика (*chartId*) и номер окна (*window*). Нулевые значения параметров обозначают, соответственно, текущий график и основное окно.

int ChartIndicatorsTotal(long chartId, int window)

Функция возвращает количество всех индикаторов, присоединенных к указанному окну графика. С помощью неё можно организовать перебор всех индикаторов, присоединенных к данному графику. Количество всех окон графика можно получить из свойства CHART_WINDOWS_TOTAL функцией *ChartGetInteger*.

string ChartIndicatorName(long chartId, int window, int index)

Функция возвращает короткое имя индикатора по номеру *index* в списке индикаторов, расположенных в указанном окне графика. Под коротким именем понимается имя, заданное в свойстве INDICATOR_SHORTNAME функцией *IndicatorSetString* (если оно не задано, то по умолчанию, равно названию файла индикатора).

int ChartIndicatorGet(long chartId, int window, const string shortname)

Возвращает дескриптор индикатора с указанным коротким именем в конкретном окне графика. Можно сказать, что идентификация индикатора в функции *ChartIndicatorGet* производится именно по короткому имени, в связи с чем рекомендуется составлять его таким образом, чтобы оно содержало значения всех входных параметров. Если это по тем или иным причинам невозможно, существует другой способ идентификации экземпляра индикатора — через список его параметров, который можно получить по заданному дескриптору с помощью функции *IndicatorParameters*.

Получение дескриптора из функции *ChartIndicatorGet* увеличивает внутренний счетчик использования данного индикатора. Исполняющая система терминала держит загруженными все индикаторы, чей счетчик больше нуля. Поэтому ненужный больше индикатор необходимо явно освобождать с помощью вызова *IndicatorRelease*. В противном случае индикатор останется работать вхолостую и потреблять ресурсы.

bool ChartIndicatorAdd(long chartId, int window, int handle)

Функция добавляет в указанное окно графика индикатор с передаваемым в последнем параметре дескриптором. Индикатор и график должны иметь одинаковое сочетание символа и таймфрейма. В противном случае произойдет ошибка ERR_CHART_INDICATOR_CANNOT_ADD (4114).

Чтобы добавить индикатор в новое окно, параметр *window* должен быть на единицу больше, чем индекс последнего существующего окна, то есть равен свойству CHART_WINDOWS_TOTAL, получаемому через вызов *ChartGetInteger*. Если значение параметра превышает значение *ChartGetInteger(ID,CHART_WINDOWS_TOTAL)*, то новое окно и индикатор не будут созданы.

> Если в основное окно графика добавляется индикатор, который должен быть отрисован в отдельном подокне (например, встроенный iMACD или пользовательский индикатор с указанным свойством *#property indicator_separate_window*), то такой индикатор может казаться невидимым, хотя и будет присутствовать в списке индикаторов. Это, как правило, означает, что значения данного индикатора не попадают в отображаемый диапазон ценового графика. Значения такого "невидимого" индикатора можно наблюдать в *Окне данных* и читать с помощью функций из других MQL-программ.

Добавление индикатора на график увеличивает внутренний счетчик его использования из-за привязки к графику. Если MQL-программа хранит у себя его дескриптор, и он больше не нужен,

то стоит удалить его путем вызова *IndicatorRelease* — это фактически уменьшит счетчик, но индикатор останется на графике.

bool ChartIndicatorDelete(long chartId, int window, const string shortname)

Функция удаляет индикатор с указанным коротким именем из окна с номером *window* на графике *chartId*. Если в указанном подокне графика существует несколько индикаторов с одинаковым коротким именем, то будет удален первый по порядку.

Если на значениях удаляемого индикатора построены другие индикаторы на этом же графике, то они также будут удалены.

Удаление индикатора с графика не означает, что его расчетная часть также будет удалена из памяти терминала, если в MQL-программе остался дескриптор. Для освобождения дескриптора индикатора используйте функцию *IndicatorRelease*.

Следующая функция *ChartWindowFind* возвращает номер подокна, в котором находится индикатор. Существует 2 формы, предназначенные для поиска текущего индикатора на его графике или индикатора с заданным коротким именем на произвольном графике с идентификатором *chartId*.

int ChartWindowFind()
int ChartWindowFind(long chartId, string shortname)

Вторую форму можно использовать в скриптах и экспертах.

Первым примером для демонстрации данных функций рассмотрим полную версию скрипта *ChartList.mq5*. Напомним, что мы создали его и постепенно дорабатывали в предыдущих разделах, вплоть до раздела Получение количества и признака видимости окон/подокон. По сравнению с представленной там версией *ChartList4.mq5* добавим входные переменные для возможности перечислять только графики с MQL-программами и подавлять вывод скрытых окон.

```
input bool IncludeEmptyCharts = true;
input bool IncludeHiddenWindows = true;
```

При значении по умолчанию (*true*) параметр *IncludeEmptyCharts* предписывает включать в список все графики, включая пустые. Параметр *IncludeHiddenWindows* по умолчанию задает вывод скрытых окон. Эти настройки соответствуют предыдущей логике скриптов *ChartListN*.

Для подсчета общего количества индикаторов и индикаторов в подокнах определены переменные *indicators* и *subs*.

```
void ChartList()
{
   ...
   int indicators = 0, subs = 0;
   ...
```

Основные изменения претерпел рабочий цикл по окнам текущего графика.

```
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++;
         }
      }
      ...
```

Здесь мы добавили вызов *ChartIndicatorsTotal* и *ChartIndicatorName*. Теперь в списке будут упоминаться MQL-программы всех типов: [E] — эксперты, [S] — скрипты, [I] — индикаторы.

Вот пример генерируемых скриптом записей журнала для настроек по умолчанию.

```
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)
```

Если сбросить в *false* оба входных параметра, получим сокращенный список.

```
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)
```

В качестве второго примера рассмотрим интересный скрипт *ChartIndicatorMove.mq5*.

При наложении нескольких индикаторов на график часто через некоторое время выясняется, что порядок расположения индикаторов желательно изменить. MetaTrader 5 не имеет встроенных средств для этого, что вынуждает удалять некоторые индикаторы и добавлять их заново, причем важно сохранить и восстановить настройки. Скрипт *ChartIndicatorMove.mq5* предоставляет вариант автоматизации данной процедуры. Важно отметить, что скрипт переносит только индикаторы: если необходимо поменять порядок подокон вместе с графическими объектами (если они есть внутри), то следует воспользоваться tpl-шаблонами.

Принцип работы *ChartIndicatorMove.mq5* заключается в следующем. При нанесении скрипта на график, он определяет, в какое окно/подокно был сброшен, и начинает перечислять пользователю найденные там индикаторы с запросом подтверждения на перенос. Пользователь может согласиться или продолжить перечисление.

Направление переноса — вверх или вниз — задается во входной переменной *MoveDirection*. Для её описания подготовлено перечисление DIRECTION.

```
#property script_show_inputs

enum DIRECTION
{
   Up = -1,
   Down = +1,
};

input DIRECTION MoveDirection = Up;
```

Для того чтобы переносить индикатор не в соседнее подокно, а в следующее, то есть фактически менять подокна с индикаторами местами (что обычно и требуется), заведена входная переменная *JumpOver*.

```
input bool JumpOver = true;
```

Цикл по индикаторам целевого окна, получаемого из *ChartWindowOnDropped*, запускается в *OnStart*.

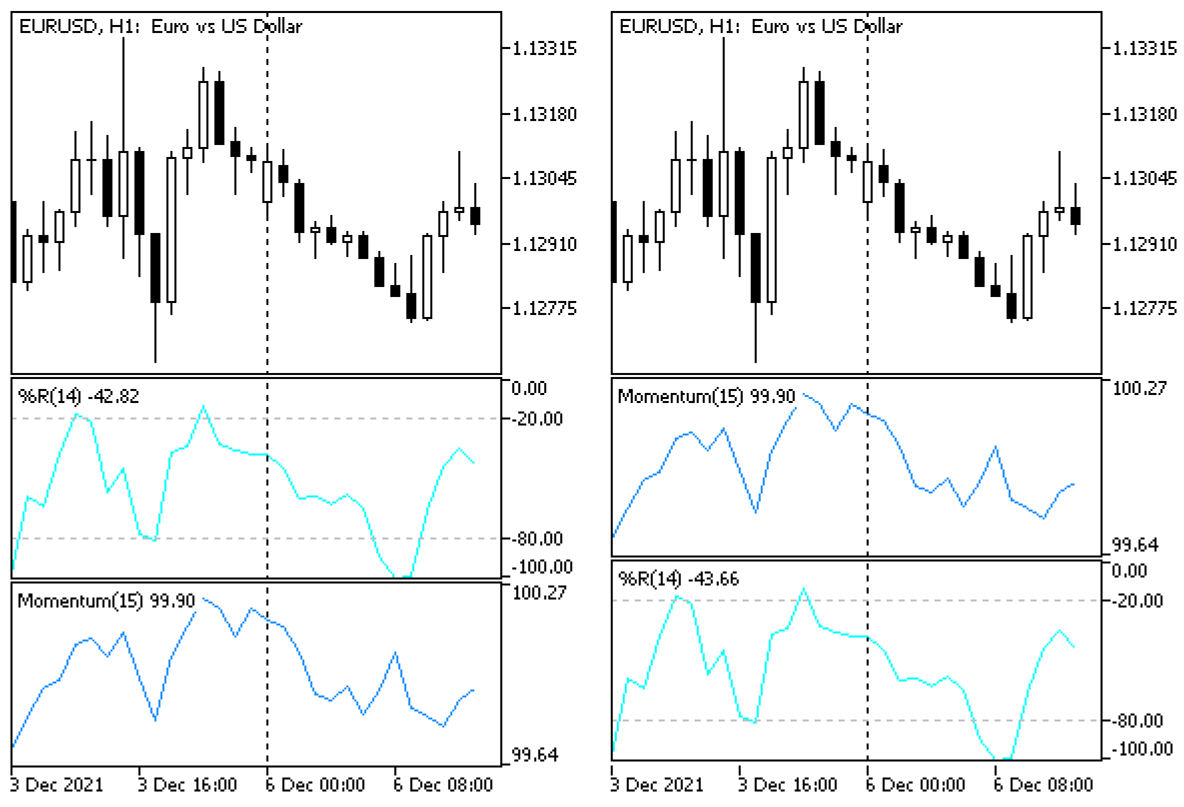
```
void OnStart()
{
   const int w = ChartWindowOnDropped();
   if(w == 0 && MoveDirection == Up)
   {
      Alert("Can't move up from window at index 0");
      return;
   }
   const int n = ChartIndicatorsTotal(0, w);
   for(int i = 0; i < n; ++i)
   {
      ...
   }
}
```

Внутри цикла определяем имя очередного индикатора, выводим сообщение пользователю и двигаем индикатор из одного окна в другое за счет последовательности таких манипуляций:

- Получаем дескриптор с помощью вызова *ChartIndicatorGet*;
- Добавляем его в окно над или под текущим через *ChartIndicatorAdd*, в соответствии с выбранным направлением, причем при переносе вниз может быть автоматически создано новое подокно;
- Удаляем индикатор из предыдущего окна с помощью *ChartIndicatorDelete*;
- Освобождаем дескриптор, потому что он нам в программе больше не нужен.

```
      ...
      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;
      }
      ...
```

На следующем изображении показан результат обмена местами подокон с индикаторами *WPR* и *Momentum*. Скрипт запускался путем отпускания на верхнем подокне с индикатором *WPR*, направление движения было выбрано вниз (*Down*), прыжок (*JumpOver*) оставлен включенным по умолчанию.



Обмен местами индикаторов в подокнах

Учтите, что если переместить индикатор из подокна в основное окно, его диаграммы, скорее всего, не будут видны из-за выхода значений за диапазон отображаемых цен. Если такое все же случилось по ошибке, вы можете с помощью скрипта перенести индикатор обратно в подокно.

5.7.23 Открытие и закрытие графиков

MQL-программа может не только анализировать список графиков, но и модифицировать его: открывать новые или закрывать имеющиеся. Для этих целей выделено 2 функции: *ChartOpen* и *ChartClose*.

long ChartOpen(const string symbol, ENUM_TIMEFRAMES timeframe)

Функция открывает новый график с указанным символом и таймфреймом, и возвращает идентификатор нового графика. Если при выполнении случилась ошибка, результат равен 0, а код ошибки можно прочитать во встроенной переменной *_LastError*.

Если параметр *symbol* равен NULL, это означает символ текущего графика (на котором выполняется MQL-программа). Значение 0 в параметре *timeframe* соответствует PERIOD_CURRENT.

Максимально возможное количество одновременно открытых графиков в терминале не может быть больше CHARTS_MAX (100).

Пример использования функции *ChartOpen* покажем в следующем разделе, после изучения функций для работы с tpl-шаблонами.

> Обратите внимание, что терминал позволяет создавать не только полноценные окна с графиками, но и объекты-графики. Они размещаются внутри обычных графиков по аналогии с прочими графическими объектами, такими как трендовые линии, каналы, ценовые метки и

т.д. Объекты-графики позволяют отображать внутри одного стандартного графика несколько мелких фрагментов ценовых рядов для альтернативных символов и таймфреймов.

bool ChartClose(long chartId = 0)

Функция закрывает график с указанным идентификатором (значение по умолчанию 0 означает текущий график).

Функция возвращает признак успеха.

В качестве примера реализуем скрипт *ChartCloseIdle.mq5*, который будет закрывать дубликаты графиков с повторяющимися сочетаниями символа и таймфрейма, если на них нет MQL-программ и графических объектов.

Для начала нам необходимо составить некий список, ведущий подсчет графиков для конкретной пары символ/таймфрейм. Эта задача поручена функции *ChartIdleList*, которая во многом похожа на то, что мы видели в скрипте *ChartList.mq5*. Сам список формируется в массиве-карте *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[];
   // собираем дубликаты пустых графиков
   if(ChartIdleList(chartCounts, duplicateChartIDs))
   {
      ...
   }
   else
   {
      Print("No idle charts.");
   }
}
```

Попутно функция *ChartIdleList* заполняет массив *duplicateChartIDs* с идентификаторами свободных графиков, подходящих под условия закрытия.

```
int ChartIdleList(MapArray<string,int> &map, ulong &duplicateChartIDs[])
{
   // перечисляем графики, пока не закончится их список
   for(long id = ChartFirst(); id != -1; id = ChartNext(id))
   {
      // пропускаем объекты
      if(ChartGetInteger(id, CHART_IS_OBJECT)) continue;

      // получаем основные свойства графика
      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);

      // подсчитываем количество индикаторов
      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     // первый раз всегда считаем новое сочетание символ/ТФ
                           // иначе считаем только пустые графики:
         || (indicators == 0           // без индикаторов
            && StringLen(expert) == 0  // без эксперта
            && StringLen(script) == 0  // без скрипта
            && objectCount == 0))      // без объектов
      {
         const int i = map.inc(key);
         if(map[i] > 1)                // дубликат
         {
            PUSH(duplicateChartIDs, id);
         }
      }
   }
   return map.getSize();
}
```

После того как список для удаления сформирован, вызываем в *OnStart* функцию *ChartClose* в цикле по списку.

```
void OnStart()
{
   ...
   if(ChartIdleList(chartCounts, duplicateChartIDs))
   {
      for(int i = 0; i < ArraySize(duplicateChartIDs); ++i)
      {
         const ulong id = duplicateChartIDs[i];
         // запрашиваем вывести график на передний план
         ChartSetInteger(id, CHART_BRING_TO_TOP, true);
         // обновляем состояние окон, прокачивая очередь с запросом
         ChartRedraw(id);
         // спрашиваем подтверждения пользователя
         const int button = MessageBox(
            "Remove idle chart: "
            + ChartSymbol(id) + "/" + PeriodToString(ChartPeriod(id)) + "?",
            __FILE__, MB_YESNOCANCEL);
         if(button == IDCANCEL) break;
         if(button == IDYES)
         {
            ChartClose(id);
         }
      }
      ...
```

Для каждого графика предварительно вызывается функция *ChartSetInteger(id, CHART_BRING_TO_TOP, true)*, чтобы показать пользователю, какое именно окно предполагается закрыть. Поскольку эта функция асинхронная (только помещает команду на активацию окна в очередь событий), требуется дополнительно вызвать *ChartRedraw*, что обрабатывает все накопившиеся сообщения. После этого пользователю выдается запрос на подтверждение действия. График закрывается только по нажатию *Да*. Нажатие *Нет* пропускает текущий график (оставляет его открытым), и цикл продолжается. Нажав кнопку *Отмена*, можно прерывать цикл досрочно.

5.7.24 Работа с tpl-шаблонами графика

Две функции MQL5 API позволяют работать с так шаблонами — файлами с расширением *tpl*, в которых сохраняется наполнение графиков, то есть все их настройки, вместе с нанесенными объектами, индикаторами и экспертом (если они есть).

bool ChartSaveTemplate(long chartId, const string filename)

Функция сохраняет текущие настройки графика в tpl-шаблон с указанным именем.

График задается идентификатором *chartId*, 0 означает текущий график.

Имя файла для сохранения шаблона (*filename*) можно указывать без расширения ".tpl": оно будет добавлено автоматически. Шаблон по умолчанию сохраняется в папку *каталог_терминала/Profiles/Templates/* и может быть использован затем для ручного применения в терминале. Однако допустимо указать не просто имя, но и путь относительно каталога MQL5, в частности, начинающийся с "/Files/". Таким образом появится возможность

открыть сохраненный шаблон функциями работы с файлами, проанализировать, и при необходимости отредактировать (см. пример *ChartTemplate.mq5* далее).

Если одноименный файл по указанному пути уже существует, его содержимое будет перезаписано.

Объединенный пример для сохранения и применения шаблона мы рассмотрим чуть позже.

bool ChartApplyTemplate(long chartId, const string filename)

Функция применяет к графику *chartId* шаблон из указанного файла.

Поиск файла шаблона осуществляется по следующим правилам:

- Если имя *filename* содержит путь (начинается с обратной "\\" или прямой "/" косой черты), то шаблон ищется относительно пути *каталог_данных_терминала/MQL5.*
- Если пути в имени нет, шаблон ищется в том же месте, где расположен исполняемый EX5-файла, в котором происходит вызов функции.
- Если шаблон не найден в первых двух местах, он ищется в стандартной папке для шаблонов *каталог_терминала/Profiles/Templates/*.

Обратите внимание, что *каталог_данных_терминала* означает папку, в которой хранятся изменяемые файлы и ее расположение может зависеть от типа операционной системы, имени пользователя и настроек безопасности компьютера. В общем случае она отличается от папки *каталог_терминала*, хотя в некоторых случаях (например, при работе под учетной записью из группы администраторов) они могут совпадать. Расположение папок *каталог_данных_терминала* и *каталог_терминала* можно узнать с помощью функции *TerminalInfoString* (см. константы TERMINAL_DATA_PATH и TERMINAL_PATH, соответственно).

Вызов *ChartApplyTemplate* фактически отдает команду, которая поступает в очередь сообщений графика и выполняется только после обработки всех предыдущих команд.

Загрузка шаблона приводит к остановке всех MQL-программ, выполняющихся на графике, включая и ту, которая инициировала загрузку. Если шаблон содержит индикаторы и эксперт, будут запущены их новые экземпляры.

> В целях безопасности при применении к графику шаблона с экспертом могут ограничиваться права на торговлю. Если у MQL-программы, которая вызывает функцию *ChartApplyTemplate*, отсутствуют права на торговлю, то эксперт, загруженный при помощи шаблона, также не будет иметь прав на торговлю вне зависимости от настроек шаблона. Если у MQL-программы, которая вызывает функцию *ChartApplyTemplate*, есть права на торговлю, а в настройках шаблона права отсутствуют, то советник, загруженный при помощи шаблона, не будет иметь прав на торговлю.

Пример скрипта *ChartDuplicate.mq5* позволяет создать копию текущего графика.

```
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);
   }
}
```

Сначала с помощью *ChartSaveTemplate* создается временный tpl-файл, затем открывается новый график (вызов *ChartOpen*), и наконец функция *ChartApplyTemplate* применяет этот шаблон к новому графику.

Однако во многих случаях перед программистом стоит более сложная задача: не просто применить шаблон, а предварительно отредактировать его.

Использование шаблонов позволяет изменять многие свойства графиков, которые недоступны с помощью остальных функций MQL5 API, например, видимость индикаторов в разрезе таймфреймов, порядок подокон индикаторов вместе с нанесенными на них объектами и т.д.

Формат tpl-файла идентичен chr-файлам, используемым терминалом для хранения графиков между сеансами (в папке *каталог_терминала/Profiles/Charts/имя_профиля*).

Tpl-файл представляет собой текстовый файл с особым синтаксисом. Свойства в нем могут представлять собой пару "ключ=значение", записываемую на одной строке, или своего рода группы, содержащие по несколько свойств "ключ=значение". Такие группы далее будем называть контейнерами, потому что помимо отдельных свойств они могут также содержать и другие, вложенные контейнеры.

Контейнер начинается строкой вида "<tag>", где *tag* — один из предопределенных типов контейнеров (см. далее), а заканчивается парной строкой вида "</tag>" (названия тегов должны совпадать). Иными словами, формат похож в некотором смысле на XML (без заголовка), в котором все лексические единицы должны записываться на отдельных строках, а свойства тегов указываются не их атрибутами (как в XML — внутри открывающей части "<tag attribute1=value1...>"), а во внутреннем тексте тега.

Перечень поддерживаемых тегов включает, в частности:

- chart — корневой контейнер с основными свойствами графика и всеми подчиненными контейнерами;
- expert — контейнер с общими свойствами эксперта, например, разрешением на торговлю (внутри chart);
- window — контейнер со свойствами окна/подокна и его подчиненными контейнерами (внутри chart);
- object — контейнер со свойствами графического объекта (внутри window);

- indicator — контейнер со свойствами индикатора (внутри window);
- graph — контейнер со свойствами диаграммы индикатора (внутри indicator);
- level — контейнер со свойствами уровня индикатора (внутри indicator);
- period — контейнер со свойствами видимости объекта или индикатора на конкретном таймфрейме (внутри object или indicator);
- inputs — контейнер с настройками (входными переменными) пользовательских индикаторов и экспертов.

Возможный перечень свойств в парах "ключ=значение" довольно обширен и не имеет официальной документации. При необходимости можно разобраться с этими особенностями платформы самостоятельно.

Вот фрагменты из одного tpl-файла (отступы в форматировании сделаны для наглядного представления вложенности контейнеров).

```
<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>
```

Для работы с tpl-файлами подготовлен заголовочный файл *TplFile.mqh*, с помощью которого можно анализировать и модифицировать шаблоны. В нем реализовано 2 класса:

- *Container* — для чтения и хранения элементов файла с учетом иерархии (вложенности), а также записи в файл после возможной модификации;
- *Selector* — для последовательного обхода элементов иерархии (объектов Container) в поисках совпадения с неким запросом, который записывается в виде строки, похожей на xpath-селектор ("/path/element[attribute=value]").

Объекты класса *Container* создаются с помощью конструктора, который принимает первым параметром дескриптор tpl-файла для чтения, а вторым — название тега. По умолчанию

название тега равно NULL, что подразумевает корневой контейнер (весь файл целиком). Таким образом, контейнер сам заполняет себя содержимым в процессе чтения файла (см. метод *read*).

Свойства текущего элемента, то есть пары "ключ=значение", расположенные непосредственно внутри данного контейнера, предполагается складывать в карту *MapArray<string,string> properties*. Вложенные контейнеры попадают в массив *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;       // элемент готов
               }

               PUSH(children, new Container(handle, subtag)).read(verbose);
            }
            else
            {
               string pair[];
               if(StringSplit(text, '=', pair) == 2)
               {
                  properties.put(pair[0], pair[1]);
               }
            }
         }
```

```
      }
      return false;
   }
   ...
};
```

В методе *read* мы построчно читаем и анализируем файл. Если встречается открывающий тег вида "<tag>", мы создаем новый объект-контейнер и продолжаем чтение в нем. Если встречается закрывающий тег вида "</tag>" с совпадающим именем, возвращаем признак успеха (*true*) — контейнер сформирован. В остальных строках считываем пары "ключ=значение" и складываем в массив *properties*.

Для поиска элементов в шаблоне разработан класс *Selector*. В его конструктор передается строка с иерархией искомых тегов. Например, строка "/chart/window/indicator" соответствует графику, в котором есть окно/подокно, а в нем, в свою очередь, любой индикатор. Результатом поиска будет первое совпадение. Данный запрос, как правило, найдет график котировок, потому что он хранится в шаблоне как индикатор с именем "Main" и идет в начале файла, до прочих подокон.

Более практичны запросы с указанием названия и значения конкретного атрибута. В частности, модифицированная строка "/chart/window/indicator[name=Momentum]" будет искать только индикатор *Momentum*. Данный поиск отличается от вызова *ChartWindowFind*, потому что здесь имя указано без параметров, в то время как *ChartWindowFind* использует короткое имя индикатора, в которое обычно включаются значения параметров, а они могут варьироваться.

Для встроенных индикаторов свойство *name* содержит непосредственно название, а для пользовательских в нем будет написано "Custom Indicator". Ссылка на пользовательский индикатор дается в свойстве *path* в виде пути к исполняемому файлу, например, "Indicators\MQL5Book\IndTripleEMA.ex5".

Итак, обратимся к внутреннему устройству класса *Selector*.

```
class Selector
{
   const string selector;
   string path[];
   int cursor;
public:
   Selector(const string s): selector(s), cursor(0)
   {
      StringSplit(selector, '/', path);
   }
   ...
```

В конструкторе мы раскладываем запрос *selector* на отдельные составляющие и сохраняем их в массиве *path*. Текущий компонент пути, который проверяется на соответствие в шаблоне, задается переменной *cursor*. В начале поиска мы находимся в корневом контейнере (рассматриваем tpl-файл целиком), и *cursor* равен 0. По мере нахождения совпадений *cursor* должен будет увеличиваться (см. метод *accept* ниже).

В классе перегружен оператор [], с помощью которого можно получить i-й фрагмент пути. Здесь также учитывается, что во фрагменте, в квадратных скобках, может быть задана пара "[ключ=значение]".

```
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];
}
...
```

Метод *accept* проверяет, совпадает ли название элемента (*tag*) и его свойства (*properties*) с теми данными, что указаны в пути селектора для текущей позиции курсора. Запись *this[cursor]* использует вышеприведенную перегрузку оператора [].

```
bool accept(const string tag, MapArray<string,string> &properties)
{
   const string name = this[cursor];
   if(!(name == "" && tag == NULL) && (name != tag))
   {
      return false;
   }

   // если в запросе есть параметр, проверяем его среди свойств
   // NB! пока поддерживается только один атрибут, а нужно много "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! поддерживается только проверка на равно '=', а должно быть '>', '<',
      string kv[];   // ключ и значение
      if(StringSplit(prop, '=', kv) == 2)
      {
         const string value = properties[kv[0]];
         if(kv[1] != value)
         {
            return false;
         }
      }
   }

   cursor++;
   return true;
}
...
```

Метод вернет *false*, если название тега не совпадает с текущим фрагментом пути, а также если во фрагменте было указано значение какого-либо параметра и оно не равно или отсутствует в

массиве *properties*. В остальных случаях мы получим совпадение условий, в результате чего курсор будет продвинут вперед (*cursor*++), а метод вернет *true*.

Процесс поиска будет успешно завершен, когда курсор достигнет последнего фрагмента в запросе, поэтому нам нужен метод для определения этого момента — *isComplete*.

```
    bool isComplete() const
    {
       return cursor == ArraySize(path);
    }
    
    int level() const
    {
       return cursor;
    }
```

Также во время анализа шаблона могут возникать ситуации, когда мы прошли по иерархии контейнеров часть пути (то есть нашли несколько совпадений), после чего очередной фрагмент запроса не совпал. В этом случае требуется "вернуться" на предыдущие уровни запроса, для чего реализован метод *unwind*.

```
    bool unwind()
    {
       if(cursor > 0)
       {
          cursor--;
          return true;
       }
       return false;
    }
 };
```

Теперь все готово для организации поиска в иерархии контейнеров (которые мы получаем после чтения tpl-файла) с помощью объекта *Selector*. Все необходимые действия будет выполнять метод *find* в классе *Container*. Он принимает в качестве входного параметра объект *Selector* и рекурсивно вызывает сам себя, пока имеются совпадения согласно методу *Selector::accept*. Достижение конца запроса означает успех, и метод *find* вернет текущий контейнер в вызывающий код.

```
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;
}
...
```

Обратите внимание, что по мере продвижения по дереву объектов метод *find* выводит в журнал название тега текущего объекта и количество вложенных объектов, причем делает это с отступом, пропорциональным уровню вложенности объектов. Если элемент подходит под запрос, запись в журнале дополняется словом "accepted".

Также важно отметить, что данная реализация возвращает первый подходящий элемент и не продолжает поиск других кандидатов, а в принципе это может быть полезно для шаблонов, потому что в них часто встречается несколько однотипных тегов внутри одного контейнера. Например, в окне может быть много объектов, и MQL-программа может быть заинтересована в анализе всего списка объектов. Этот аспект предлагается доработать факультативно.

Для упрощения вызова поиска добавлен одноименный метод, принимающий строковый параметр и создающий объект *Selector* локально.

```
Container *find(const string selector)
{
   Selector s(selector);
   return find(&s);
}
```

Поскольку мы собираемся редактировать шаблон, следует предусмотреть методы модификации контейнера, в частности, для добавления пары "ключ=значение" и нового вложенного контейнера с заданным тегом.

```
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);
}
```

А после редактирования необходимо будет записать содержимое контейнеров обратно в файл (тот же самый или другой). Вспомогательный метод *save* сохраняет объект в описанном выше tpl-формате: начинает с открывающего тега "<tag>", продолжает выгрузкой всех свойств "ключ=значение" и вызывает *save* у вложенных объектов, после чего завершает закрывающим тегом "</tag>". Дескриптор файла для сохранения передается в параметре.

```
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;
}
```

Высокоуровневый метод записи всего шаблона в файл называется *write*. Его входной параметр (дескриптор файла) может быть равен 0, что означает запись в тот же файл, откуда производилось чтение. Однако файл для этого должен быть открыт с правами на запись.

Важно отметить, что при перезаписи текстового Unicode-файла MQL5 не пишет начальную метку UTF (так называемый BOM, Byte Order Mark), в связи с чем мы вынуждены делать это сами. В противном случае (без метки), терминал не станет читать и применять наш шаблон.

Если вызывающий код передаст в параметре *h* другой файл, открытый исключительно на запись в формате Unicode, MQL5 запишет BOM автоматически.

```
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! Пишем BOM вручную, потому что MQL5 не делает это при перезаписи
      ushort u[1] = {0xFEFF};
      FileSeek(h, SEEK_SET, 0);
      FileWriteString(h, ShortArrayToString(u));
   }

   bool result = save(h);

   if(rewriting)
   {
      // NB! MQL5 не позволяет уменьшить размер файла,
      // поэтому заполняем лишнюю концовку пробелами
      while(FileTell(h) < FileSize(h) && !IsStopped())
      {
         FileWriteString(h, " ");
      }
   }
   return result;
}
```

Для демонстрации возможностей новых классов рассмотрим задачу скрытия окна конкретного индикатора. Как известно, пользователь может этого добиться, сбросив флаги видимости для таймфреймов в диалоге свойств индикатора (закладка *Отображение*). Программным способом сделать это напрямую не получится. Здесь и приходит на помощь возможность редактировать шаблон.

В шаблоне видимость индикатора для таймфреймов указывается в контейнере <indicator>, внутри которого для каждого видимого таймфрейма записывается свой контейнер <period>. Например, видимость на таймфрейме M15 выглядит так:

```
<period>
period_type=0
period_size=15
</period>
```

Внутри контейнера <period> используются свойства *period_type* и *period_size*. *period_type* — это единица измерения, одна из следующих:

- 0 — минуты;
- 1 — часы;
- 2 — недели;
- 3 — месяцы;

period_size — это количество единиц измерения в таймфрейме. Следует отметить, что дневной таймфрейм обозначается, как 24 часа.

Когда в контейнере <indicator> нет ни одного вложенного контейнера <period>, индикатор выводится на всех таймфреймах.

К книге прилагается скрипт *ChartTemplate.mq5*, который добавляет на график индикатор Momentum (если он еще отсутствует) и делает его видимым на единственном месячном таймфрейме.

```
void OnStart()
{
   // если Momentum(14) еще нет на графике, добавляем его
   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);
      // не обязательно здесь, потому что скрипт скоро завершит работу,
      // однако явно декларирует, что дескриптор больше не потребуется в коде
      IndicatorRelease(momentum);
   }
   ...
```

Далее сохраняем шаблон текущего графика в файл, который затем открываем на запись и чтение. Можно было бы выделить под запись отдельный файл.

```
   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));
      // альтернатива - другой файл, открытый только на запись
      // int writer = PRTF(FileOpen(filename + "w.tpl",
      //    FILE_WRITE | FILE_TXT | FILE_SHARE_READ | FILE_SHARE_WRITE));
```

Получив дескриптор файла, создаем корневой контейнер *main* и читаем в него весь файл (вложенные контейнеры и все их свойства будут прочитаны автоматически).

```
      Container main(handle);
      main.read();
```

Затем определим селектор для поиска индикатора *Momentum*. В принципе, более строгий подход потребовал бы также проверять заданный период (14), но в наших классах не реализована поддержка запроса нескольких свойств одновременно (эта возможность оставлена для самостоятельной проработки).

С помощью селектора выполняем поиск, распечатываем найденный объект (просто для справки) и добавляем в него вложенный контейнер <period> с настройками для отображения месячного таймфрейма.

```
      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");
      }
```

Наконец, записываем модифицированный шаблон в тот же файл, закрываем его и применяем на графике.

```
      main.write(); // или main.write(writer);
      FileClose(handle);
      
      PRTF(ChartApplyTemplate(0, "/Files/" + filename));
   }
}
```

При запуске скрипта на чистом графике увидим такие записи в жрунале.

```
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
```

Здесь видно, что прежде чем найти нужный индикатор (с пометкой "accepted"), алгоритм нашел индикатор в предыдущем, основном окне, но он не подошел, потому что его имя не равно искомому "Momentum".

Если теперь открыть список индикаторов на графике, там будет *Momentum*, а в его диалоге свойств, на закладке *Отображение* — единственный включенный таймфрейм *Месяц*.

К книге прилагается расширенная версия файла *TplFileFull.mqh*, которая поддерживает разные операций сравнения в условиях отбора тегов и их множественную выборку в массивы. Пример использования можно посмотреть в скрипте *ChartUnfix.mq5*, снимающем фиксацию размеров всех подокон графика.

5.7.25 Сохранение изображения графика

В MQL-программах часто возникает необходимость документировать текущее стояние самой программы и торгового окружения. Как правило, для этого используется вывод различных аналитических или финансовых показателей в журнал, но некоторые вещи более наглядно представить изображением графика, например, на момент совершения сделки. MQL5 API включает функцию, которая позволяет сохранять изображение графика в файл.

bool ChartScreenShot(long chartId, string filename, int width, int height,
  ENUM_ALIGN_MODE alignment = ALIGN_RIGHT)

Функция делает снимок указанного графика в формате GIF, PNG или BMP в зависимости от расширения в строке с именем файла *filename* (максимум 63 символа). Скриншот помещается в каталог *MQL5/Files*.

Параметры *width* и *height* задают ширину и высоту изображения в пикселях.

Параметр *alignment* влияет на то, какая часть графика попадет в файл. Значение ALIGN_RIGHT (по умолчанию) означает, что снимок делается для самых свежих цен (это можно себе представить так, что терминал незаметно делает переход по нажатию *End* перед снимком). Значение ALIGN_LEFT обеспечивает попадание на изображение баров, начиная с первого, видимого слева в данный момент. Таким образом, если необходимо снять скриншот графика с определенной позиции, то необходимо сначала позиционировать график вручную или при помощи функции *ChartNavigate*.

Функция *ChartScreenShot* возвращает *true* в случае успеха.

Протестируем работу функции в скрипте *ChartPanorama.mq5*. Его задача — сохранить копию графика от текущего левого видимого бара вплоть до текущего времени. Сместив предварительно начало графика назад на нужную глубину истории, можно получить довольно протяженную панораму. При этом не надо задумываться, какую ширину изображения выбрать. Однако следует иметь в виду, что слишком длинная история потребует огромного изображения, потенциально превышающего возможности графического формата или того или иного программного обеспечения.

Высота изображения будет автоматически определяться равной текущей высоте графика.

```
void OnStart()
{
   // точная ширина шкалы цен не известна, берем эмпирическим путем
   const int scale = 60;

   // вычисляем общую высоту, включая зазоры между окнами
   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);

   // вычисляем общую ширину на основе количества пикселей в одном баре,
   // и также включая сдвиг графика от правого края и ширину шкалы
   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);

   // записываем файл с картинкой в формате PNG
   const string filename = _Symbol + "-" + PeriodToString() + "-panorama.png";
   if(ChartScreenShot(0, filename, width, height, ALIGN_LEFT))
   {
      Print("File saved: ", filename);
   }
}
```

Мы могли бы использовать и режим ALIGN_RIGHT, но тогда нужно было бы принудительно отключать сдвиг от правого края, потому что он вычисляется заново для картинки, в зависимости от её размера, и результат будет выглядеть совсем не так, как на экране (отступ справа станет слишком большим, так как он задается в процентах от ширины).

Ниже показан пример журнала после запуска скрипта на графике XAUUSD,H1.

```
Gutter=2, total=2
Height=440
Shift=74
Width=2086
File saved: XAUUSD-H1-panorama.png
```

С учетом навигации на не очень отдаленную историю получился следующий скриншот (представлен в виде уменьшенной в 4 раза копии).

Панорама графика

5.8 Графические объекты

Пользователи MetaTrader 5 хорошо знакомы с понятием графических объектов: трендовых линий, ценовых меток, каналов, уровней Фибоначчи, геометрических фигур и многих других визуальных элементов, с помощью которых выполняется аналитическая разметка графика. Язык MQL5 позволяет создавать, редактировать, удалять графические объекты программно. Это может пригодиться, например, если в индикаторе желательно отображать некие данные одновременно в подокне и на основном окне — поскольку платформа поддерживает вывод индикаторных буферов только в одно окно, во втором мы можем генерировать объекты. По разметке из графических объектов легко организовать полуавтоматическую торговлю экспертами. Кроме того, объекты часто используются для построения собственного графического интерфейса MQL-программ: кнопок, полей ввода, флагов. Такими программами можно управлять, не открывая диалог свойств, да и создаваемые на MQL панели могут обладать гораздо большей гибкостью, что стандартные входные переменные.

Каждый объект существует в контексте конкретного графика. Именно поэтому функции, которые мы рассмотрим в этой главе, имеют общую черту — первый параметр задает идентификатор графика. Кроме того каждый графический объект характеризуется именем, уникальным в пределах одного графика, включая все подокна. Изменение имени графического объекта сводится к удалению объекта со старым именем и созданию такого же объекта с новым именем. Создать два объекта с одинаковым именем не получится.

Функции, задающие свойства графических объектов, а также операции создания (*ObjectCreate*) и перемещения (*ObjectMove*) объектов на графике фактически служат для отправки асинхронных команд графику. При успешном выполнении этих функций команда попадает в общую очередь событий графика. Визуальное изменение свойств графических объектов производится в процессе обработки очереди событий данного графика. В связи с этим внешнее представление графика может отобразить измененное состояния объектов с некоторой задержкой после вызова функций.

В общем случае обновление графических объектов на графике производится терминалом автоматически по событиям изменения графика — поступлению новой котировки, изменению размера окна и т.д. Для принудительного обновления графических объектов используйте

функцию для запроса перерисовки графика (*ChartRedraw*). Это особенно важно после массового создания или модификации объектов.

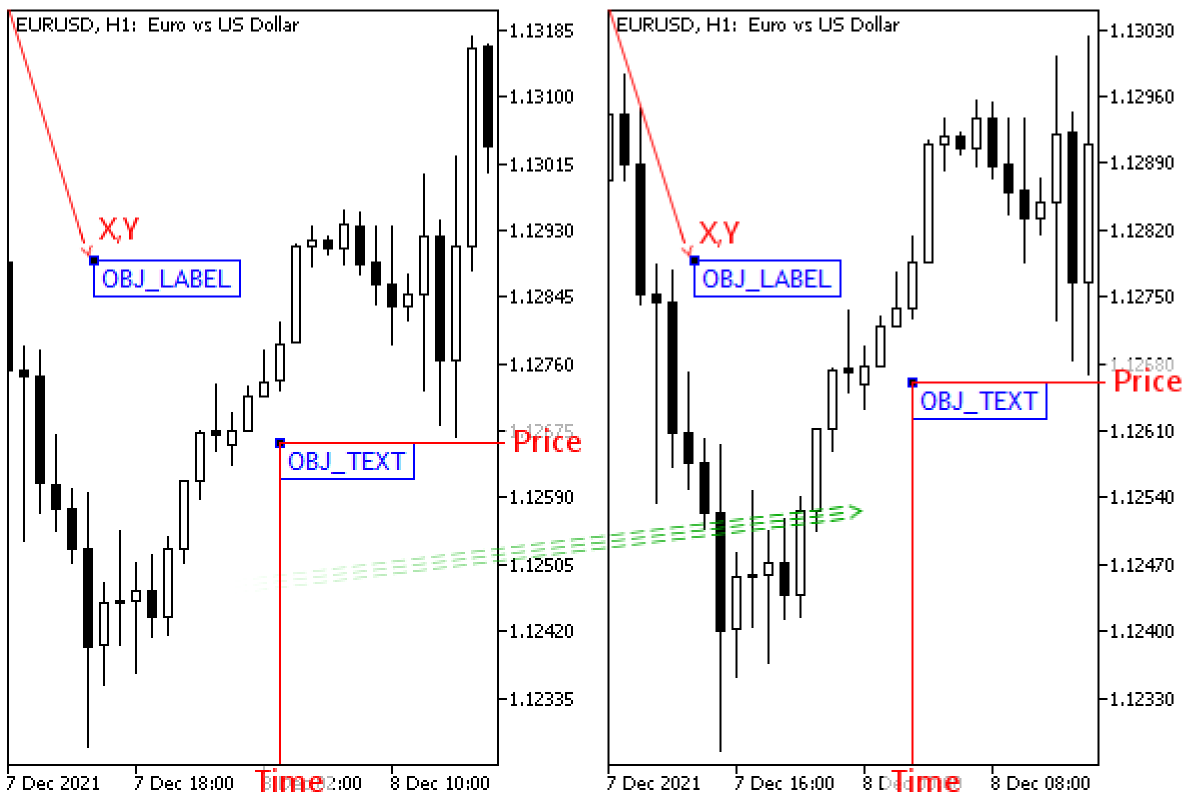
Объекты являются источником программных событий, таких как создание, удаление, изменение их свойств, нажатия мышью: все аспекты возникновения и обработки событий рассматриваются в отдельной главе, одновременно с событиями в общем контексте окна.

Изложение мы начнем с теоретических основ и постепенно перейдем к практике.

5.8.1 Типы объектов и особенности указания их координат

Как мы знаем из главы про графики, в окне существует две системы координат: экранные (пиксельные) и котировочные (время и цена). В связи с этим общий набор поддерживаемых типов объектов делится на две большие группы: тех объектов, которые имеют привязку к экрану, и тех, которые имеют привязку к графику цен. Первые всегда остается на месте относительно одного из углов окна (какой именно угол является опорным, определяет пользователь или программист в свойствах объекта). Вторые прокручиваются вместе с рабочей областью окна.

На следующем изображении приведены для сравнения два объекта с текстовыми надписями: один с привязкой к экрану (OBJ_LABEL), а другой — к графику цен (OBJ_TEXT). Их типы, приведенные в скобках, а также свойства, с помощью которых задаются координаты, мы изучим в соответствующих разделах этой главы. Важно отметить, что при прокрутке графика цен текст OBJ_TEXT двигается синхронно с ним, в то время как надпись OBJ_LABEL остается на одном и том же месте.



Две разные системы координат для объектов

Также объекты отличаются по количеству точек привязки. Например, одиночная ценовая метка ("стрелка") требует одной точки время/цена, а трендовая линия — двух таких точек. Имеются

типы объектов и с большим количеством точек привязки, такие как равноудаленный канал, треугольник или волны Эллиотта.

При выделении объекта (например, в диалоге *Список объектов*, двойным или одиночным щелчком мыши на графике, в зависимости от настройки терминала — закладка *Графики*, опция *Выделять объекты одиночным кликом мыши*) его точки привязки обозначаются маленькими квадратами контрастным цветом. Именно точки привязки используются для перетаскивания объекта, изменения его размеров и ориентации.

Все поддерживаемые типы объектов описаны в перечислении ENUM_OBJECT. Ознакомиться с ним целиком можно в документации MQL5. Мы же будем рассматривать его элементы постепенно, по частям.

5.8.2 Объекты с привязкой ко времени и цене

В следующей таблице указаны объекты с привязкой ко времени и цене, их идентификаторы в перечислении ENUM_OBJECT и количество точек привязки.

Идентификатор	Название	Точки привязки
Одиночные прямые линии		
OBJ_VLINE	Вертикальная (только координата времени)	1
OBJ_HLINE	Горизонтальная (только координата цены)	1
OBJ_TREND	Трендовая	2
OBJ_ARROWED_LINE	Со стрелкой на конце	2
Периодически повторяющиеся вертикальные линии		
OBJ_CYCLES	Циклическая	2
Каналы		
OBJ_CHANNEL	Равноудаленный	3
OBJ_STDDEVCHANNEL	Стандартного отклонения	2
OBJ_REGRESSION	Линейной регрессии	2
OBJ_PITCHFORK	Вилы Эндрюса	3
Инструменты Фибоначчи		
OBJ_FIBO	Уровни	2
OBJ_FIBOTIMES	Временные зоны	2
OBJ_FIBOFAN	Веер	2
OBJ_FIBOARC	Дуги	2
OBJ_FIBOCHANNEL	Канал	3

Идентификатор	Название	Точки привязки
OBJ_EXPANSION	Расширение	3
	Инструменты Ганна	
OBJ_GANNLINE	Линия	2
OBJ_GANNFAN	Веер	2
OBJ_GANNGRID	Сетка	2
	Волны Эллиота	
OBJ_ELLIOTWAVE5	Импульсная	5
OBJ_ELLIOTWAVE3	Корректирующая	3
	Фигуры	
OBJ_RECTANGLE	Прямоугольник	2
OBJ_TRIANGLE	Треугольник	3
OBJ_ELLIPSE	Эллипс	3
	Одиночные знаки и метки	
OBJ_ARROW_THUMB_UP	Знак "хорошо" (палец кверху)	1*
OBJ_ARROW_THUMB_DOWN	Знак "плохо" (палец книзу)	1*
OBJ_ARROW_UP	Стрелка вверх	1*
OBJ_ARROW_DOWN	Стрелка вниз	1*
OBJ_ARROW_STOP	Знак "стоп"	1*
OBJ_ARROW_CHECK	Знак "птичка" (флажок)	1*
OBJ_ARROW_LEFT_PRICE	Левая ценовая метка	1
OBJ_ARROW_RIGHT_PRICE	Правая ценовая метка	1
OBJ_ARROW_BUY	Знак "покупка" (синяя стрелка вверх)	1
OBJ_ARROW_SELL	Знак "продажа" (красная стрелка вниз)	1
OBJ_ARROW	Произвольный знак Wingdings	1*
	Текст и графика	
OBJ_TEXT	Текст	1*
OBJ_BITMAP	Картинка	1*
	События	

Идентификатор	Название	Точки привязки
OBJ_EVENT	Временная метка внизу основного окна (только координата времени)	1

Сноской со звездочкой помечены те объекты, для которых разрешено выбирать точку привязки на объекте (например, в одном из углов объекта или в середине одной из сторон). Способы выбора могут отличаться для разных типов — подробности будут изложены в разделе Определение точки привязки на объекте. Точки привязки требуются потому, что объекты имеют некоторый размер, и без них возникала бы неоднозначность позиционирования.

5.8.3 Объекты с привязкой к экранным координатам

В следующей таблице приведены названия и ENUM_OBJECT-идентификаторы объектов с привязкой к экранным координатам. Практически все они, за исключение объекта-графика, предназначены для создания пользовательского интерфейса программ. В частности, здесь есть такие базовые элементы управления как кнопка и поле ввода, а также надписи и панели для визуальной группировки объектов. На их основе можно создавать более сложные элементы управления (например, выпадающие списки или флажки-переключатели). Вместе с терминалом в виде набора заголовочных файлов поставляется библиотека классов с готовыми элементами управления (см. каталог *MQL5/Include/Controls*).

Идентификатор	Название	Настройка точки привязки
OBJ_LABEL	Надпись	Да
OBJ_RECTANGLE_LABEL	Прямоугольная панель	
OBJ_BITMAP_LABEL	Панель с картинкой	Да
OBJ_BUTTON	Кнопка	
OBJ_EDIT	Поле ввода	
OBJ_CHART	Объект-график	

Для всех этих объектов требуется определение угла привязки в окне графика. По умолчанию их координаты задаются относительно верхнего левого угла окна.

В типах из данного списка также используется точка привязки на объекте, причем только одна. В некоторых объектах её можно выбирать, а в некоторых — она жестко задана. Например, прямоугольная панель, кнопка, поле ввода и объект-график всегда привязываются за свой левый верхний угол. А для надписи или панели с картинкой доступно множество вариантов. Выбор осуществляется из перечисления ENUM_ANCHOR_POINT, описанного в разделе Определение точки привязки на объекте.

> В интерфейсе MetaTrader 5 объект Надпись (OBJ_LABEL) называется "Текстовая метка", а панель с картинкой (OBJ_BITMAP_LABEL) — "Графическая метка". Учитывая ценовые метки из предыдущего раздела, получается, что термин "метка" используется для объектов с разными способами привязки, разных размеров и назначения.

> В рамках данной книги мы будем называть метками мелкие знаки и выноски — все они относятся к типам объектов с координатами времени и цены.

Надпись (OBJ_LABEL) обеспечивает вывод текста без возможности его редактирования. Для редактирования используйте поле ввода (OBJ_EDIT).

5.8.4 Создание объектов

Для создания объекта требуется некий минимальный набор атрибутов, общих для всех типов. Дополнительные свойства, специфические для каждого типа, можно задать или изменить позднее, уже у существующего объекта. К числу обязательных атрибутов относятся идентификатор графика, где следует создать объект, имя объекта, номер окна/подокна, а также две координаты для первой точки привязки: время и цена.

Несмотря на то, что существует группа объектов, позиционируемая в экранных координатах, при их создании все равно требуется передать два значения, как правило, нулевые, потому что они не имеют значения.

В общем виде прототип функции *ObjectCreate* — следующий:

```
bool ObjectCreate(long chartId, const string name, ENUM_OBJECT type, int window,
  datetime time1, double price1, datetime time2 = 0, double price2 = 0, ...)
```

Значение 0 для *chartId* подразумевает текущий график. Имя *name* должно быть уникальным в пределах всего графика, включая подокна, и не должно превышать 63 символа.

Типы объектов для параметра *type* мы приводили в предыдущих разделах: это элементы перечисления ENUM_OBJECT.

Как мы знаем, нумерация окон/подокон для параметра *window* начинается с 0, что означает основное окно графика. Если указан больший индекс для подокна, оно должно существовать, так как в противном случае функция завершится с ошибкой и вернет *false*.

Напомним, что возвращенный признак успеха — *true* — означает лишь, что команда на создание объекта успешно помещена в очередь. Результат её выполнения сразу неизвестен — это оборотная "сторона медали" асинхронного вызова, который применен для повышения быстродействия.

Для проверки результата выполнения можно использовать функцию *ObjectFind* или любые ObjectGet-функции, запрашивающие свойства объекта. Но следует иметь в виду, что такие функции дожидаются выполнения всей очереди команд графика и только потом выдают актуальный результат (состояние объекта). Данный процесс может потребовать некоторого времени, в течение которого код MQL-программы будет приостановлен. Иными словами функции проверки состояния объектов являются синхронными, в отличие от функций создания и модификации.

Дополнительные точки привязки, начиная со второй, опциональные. Допустимое количество точек привязки — до 30-ти — предусмотрено для будущего использования, а в текущих типах объектов используется не более 5.

Важно отметить, что вызов функции *ObjectCreate* с именем уже существующего объекта просто меняет точку/точки привязки (если координаты были изменены по сравнению с предыдущим вызовом). Это удобно использовать для написания унифицированного кода без ветвления на условия по наличию или отсутствию объекта. Иными словами, безусловный вызов *ObjectCreate*

гарантирует наличие объекта, если нам не важно, был ли он до того или нет. Однако есть и нюанс. Если при вызове *ObjectCreate* тип объекта или номер подокна отличаются от уже существующего объекта, то они остаются прежними. И важно, что ошибки не возникает.

При вызове *ObjectCreate* можно оставить все точки привязки со значениями по умолчанию (т.е. нулевыми), при условии что после этой инструкции будут вызваны ObjectSet-функции с соответствующими свойствами OBJPROP_TIME и OBJPROP_PRICE.

> Порядок указания точек привязки может быть важен для некоторых типов объектов. Например, для каналов линейной регрессии OBJ_REGRESSION и стандартного отклонения OBJ_STDDEVCHANNEL обязательно должно выполняться условия *time1* < *time2*. В противном случае канал не построится нормально, хотя объект будет создан без ошибок.

В качестве примера работы функции возьмем скрипт *ObjectSimpleShowcase.mq5*, который создает на последних барах графика несколько объектов разных типов, требующих единственной точки привязки.

Все примеры работы с объектами будут использовать заголовочный файл *ObjectPrefix.mqh*, в котором находится определение строки с общим префиксом для имен объектов. Таким образом, нам будет удобнее при необходимости очищать графики от "своих" объектов.

```
const string ObjNamePrefix = "ObjShow-";
```

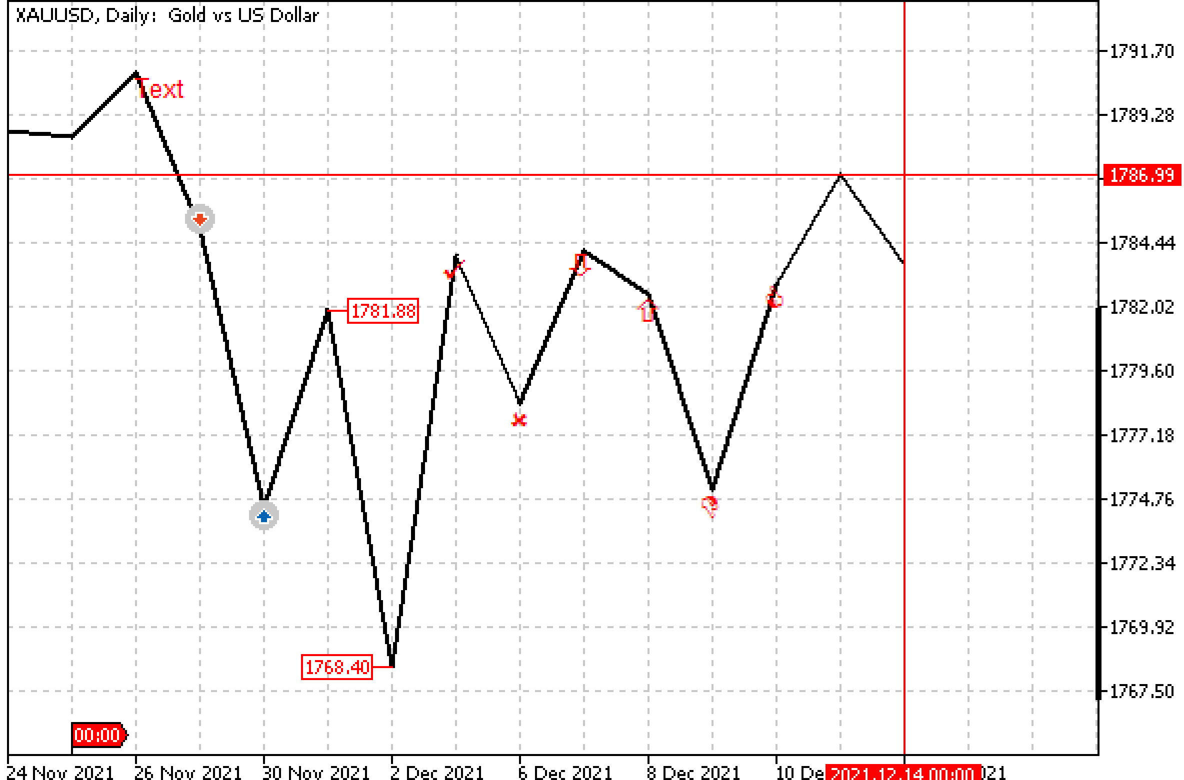
В функции *OnStart* определен массив с типами объектов.

```
void OnStart()
{
   ENUM_OBJECT types[] =
   {
      // прямые линии
      OBJ_VLINE, OBJ_HLINE,
      // метки (стрелки и другие знаки)
      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, // см. пример ObjectWingdings.mq5

      // текст
      OBJ_TEXT,
      // флажок события (как в календаре) у нижней границы окна
      OBJ_EVENT,
   };
```

Далее, в цикле по его элементам создаем объекты в основном окне, передавая время и цену закрытия i-го бара.

```
   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);
}
```

Вот что может получиться в результате запуска скрипта.



Объекты простых типов в точках закрытия последних баров

Здесь специально выставлен режим отрисовки линий по цене *Close* и выведена сетка. Размер, цвет и прочие атрибуты объектов мы научимся настраивать позднее. В частности, точки привязки большинства значков расположены по умолчанию в середине верхней стороны, поэтому они визуально смещены под линию. Однако значок продажи выведен над линией, потому что в нем точка привязки всегда в середине нижней стороны.

> Обратите внимание, что объекты, созданные программным путем по умолчанию не выводятся в список объектов в одноименном диалоге. Для того чтобы их увидеть там, следует нажать кнопку *Все*.

5.8.5 Удаление объектов

Для удаления объектов MQL5 API предлагает две функции. Для массового удаления объектов, удовлетворяющих условиям по префиксу в имени, типу или номеру подокна, следует применять *ObjectsDeleteAll*. Если удаляемые объекты требуется выбирать по каким-то другим признакам

(например, по устаревшей координате даты и времени) или это единичный объект, используйте функцию *ObjectDelete*.

Функция *ObjectsDeleteAll* имеет две формы: с параметром для префикса имени и без него.

int ObjectsDeleteAll(long chartId, int window = -1, int type = -1)
int ObjectsDeleteAll(long chartId, const string prefix, int window = -1, int type = -1)

Функция удаляет все объекты на графике с идентификатором *chartId*, с учетом подокна, типа и начальной подстроки в названии.

Значение 0 в *chartId* обозначает, как обычно, текущий график.

Значения по умолчанию (-1) в параметрах *window* и *type* задают, соответственно, все подокна и все типы объектов.

Если префикс пустой, будут удаляться объекты с любым именем.

Функция выполняется синхронно, то есть блокирует вызвавшую MQL-программу до своего завершения, и возвращает количество удаленных объектов. Поскольку функция дожидается выполнения всех команд, которые были в очереди графика перед её вызовом, действие может занять некоторое время.

bool ObjectDelete(long chartId, const string name)

Функция удаляет объект с указанным именем на графике с *chartId*.

В отличие от *ObjectsDeleteAll*, *ObjectDelete* выполняется асинхронно, то есть отправляет графику команду на удаление объекта и сразу возвращает управление MQL-программе. Результат *true* означает успешное размещение команды в очереди. Для проверки результата выполнения можно использовать функцию *ObjectFind* или любые *ObjectGet*-функции, запрашивающие свойства объекта.

В качестве примера рассмотрим скрипт *ObjectCleanup1.mq5*. Его задача заключается в удалении объектов с "нашим" префиксом, которые генерируются скриптом *ObjectSimpleShowcase.mq5* из предыдущего раздела.

В простейшем случае мы могли бы написать так:

```
#include "ObjectPrefix.mqh"

void OnStart()
{
   const int n = ObjectsDeleteAll(0, ObjNamePrefix);
   PrintFormat("%d objects deleted", n);
}
```

Но мы добавим для разнообразия возможность удаления собственными средствами с помощью многократного вызова *ObjectDelete*. Разумеется, такой подход не имеет смысла, когда *ObjectsDeleteAll* отвечает всем требованиям. Однако так бывает не всегда: когда объекты нужно отбирать по специальным условиям, то есть не только по префиксу и типу, *ObjectsDeleteAll* уже не поможет.

Позднее, когда мы познакомимся с функциями чтения свойств объектов, мы дополним пример. А пока введем лишь входную переменную для переключения в "расширенный" режим удаления (*UseCustomDeleteAll*).

```
#property script_show_inputs
input bool UseCustomDeleteAll = false;
```

В функции *OnStart* будем в зависимости от выбранного режима вызывать стандартную *ObjectsDeleteAll* или свою собственную реализацию *CustomDeleteAllObjects*.

```
void OnStart()
{
   const int n = UseCustomDeleteAll ?
      CustomDeleteAllObjects(0, ObjNamePrefix) :
      ObjectsDeleteAll(0, ObjNamePrefix);

   PrintFormat("%d objects deleted", n);
}
```

Сначала набросаем эскиз этой функции, а затем усовершенствуем её.

```
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: цикл по объектам в обратном порядке внутреннего списка графика
   // чтобы сохранить нумерацию по мере удаления с хвоста
   for(int i = n - 1; i >= 0; --i)
   {
      const string name = ObjectName(chart, i, window, type);
      if(StringLen(prefix) == 0 || StringFind(name, prefix) == 0)
      // дополнительные проверки, которые не обеспечивает ObjectsDeleteAll,
      // например, по координатам, цвету или точке привязки
      ...
      {
         // отдаем команду удалить конкретный объект
         count += ObjectDelete(chart, name);
      }
   }
   return count;
}
```

Здесь мы видим несколько новых функций, которые будут описаны в следующем разделе (*ObjectsTotal*, *ObjectName*). Их суть должна быть в целом понятна: первая функция возвращает количество объектов на графике, а вторая — название объекта под указанным номером.

Стоит также отметить, что цикл по объектам идет по убыванию индексов. Если бы мы делали это привычным образом, то удаление объектов в начале списка привело бы к нарушению нумерации. Строго говоря, даже текущий цикл не гарантирует полностью удаления, если предположить, что другая MQL-программа начнет добавлять объекты параллельно с нашим удалением. Действительно, новый "чужой" объект может добавиться в начало списка (он формируется по алфавитному порядку имен объектов) и увеличит остальные индексы, вытолкнув "наш" очередной, подлежащий удалению, объект за текущий индекс *i*. Чем больше "чужих" объектов добавится в начало, тем больше вероятность упустить удаление собственных.

Поэтому для повышения надежности можно было бы после цикла проверить, что количество оставшихся объектов равно разнице между начальным количеством и количеством удаленных объектов. Хотя и это не дает 100% гарантии, так как другие программы могли параллельно удалять объекты. Мы оставим эти нюансы для самостоятельной проработки.

В текущей реализации наш скрипт должен удалить все объекты с "нашим" префиксом вне зависимости от переключения режима *UseCustomDeleteAll*. В журнале должны появиться примерно такие записи:

```
ObjectSimpleShowcase (XAUUSD,H1) 14 objects of various types created
ObjectCleanup1 (XAUUSD,H1) 14 objects deleted
```

Давайте познакомимся с функциями *ObjectsTotal* и *ObjectName*, которые уже были только что использованы, а потом вернемся к версии скрипта *ObjectCleanup2.mq5*.

5.8.6 Поиск объектов

Для поиска объектов на графике существует 3 функции. Первые две — *ObjectsTotal* и *ObjectName* — позволяют организовать перебор объектов по имени, и далее, если требуется, использовать имя каждого объекта для анализа его прочих свойств (о том, как это делается, мы расскажем в следующем разделе). Третья функция *ObjectFind* позволяет по известному имени проверить существование объекта. В принципе, то же самое можно сделать, просто запросив какое-либо свойство через *ObjectGet*-функцию: если объекта с переданным именем нет, мы получим ошибку в *_LastError*, но это менее удобно, чем вызов *ObjectFind*. Кроме того она дополнительно сразу возвращает номер окна, в котором находится объект.

int ObjectsTotal(long chartId, int window = -1, int type = -1)

Функция возвращает количество объектов на графике с идентификатором *chartId* (0 — текущий график). В расчет принимаются только объекты в подокне с номером *window* (0 — основное окно, -1 — основное окно и все подокна), а также объекты конкретного типа, указанного в параметре *type* (-1 означает по умолчанию все типы). Значением *type* может быть элемент перечисления ENUM_OBJECT.

Функция выполняется синхронно, то есть блокирует выполнение вызывающей MQL-программы вплоть до получения результата.

string ObjectName(long chartId, int index, int window = -1, int type = -1)

Функция возвращает имя объекта под номером *index* на графике с идентификатором *chartId*. При составлении внутреннего списка, внутри которого и ищется объект, в расчет принимаются указанные номер подокна (*window*) и тип объектов (*type*). Список отсортирован по названиям объектов в лексикографическом порядке, то есть, в частности, по алфавиту с учетом регистра.

Как и *ObjectsTotal*, *ObjectName* в процессе своего выполнения дожидается выборки всей очереди команд графика и затем возвращает имя объекта из актуализированного списка объектов.

В случае ошибки будет получена пустая строка, в *_LastError* занесен код OBJECT_NOT_FOUND (4202).

Для проверки работы этих двух функций создадим скрипт *ObjectFinder.mq5*, который выводит в журнал все объекты на всех графиках. В нем применены уже известные нам функции перебора графиков (*ChartFirst* и *ChartNext*), а также функции получения свойств графиков (*ChartSymbol*, *ChartPeriod*, *ChartGetInteger*).

```
#include <MQL5Book/Periods.mqh>

void OnStart()
{
   int count = 0;
   long id = ChartFirst();
   // цикл по графиках
   while(id != -1)
   {
      PrintFormat("%s %s (%lld)", ChartSymbol(id), PeriodToString(ChartPeriod(id)), i
      const int win = (int)ChartGetInteger(id, CHART_WINDOWS_TOTAL);
      // цикл по окнам
      for(int k = 0; k < win; ++k)
      {
         PrintFormat("  Window %d", k);
         const int n = ObjectsTotal(id, k);
         // цикл по объектам
         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);
}
```

Для каждого графика определяем количество подокон (*ChartGetInteger(id, CHART_WINDOWS_TOTAL)*) и вызываем для каждого подокна *ObjectsTotal* и, во внутреннем цикле, — *ObjectName*. Далее по имени находим тип объекта и выводим их вместе в журнал.

Ниже приведен вариант возможного результата работы скрипта (с сокращениями).

```
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
```

Здесь, в частности, видно, что на графике XAUUSD D1 находятся объекты, сгенерированные скриптом *ObjectSimpleShowcase.mq5*. На некоторых графиках и в некоторых подокнах объектов нет.

int ObjectFind(long chartId, const string name)

Функция ищет объект по имени на указанном с помощью идентификатора графике и возвращает в случае успеха номер окна, где он найден.

Если объект не найден, функция возвращает отрицательное число. Как и предыдущие функции данного раздела, *ObjectFind* использует синхронный вызов.

Пример использования данной функции мы увидим в скрипте *ObjectCopy.mq5* в следующем разделе.

5.8.7 Обзор функций доступа к свойствам объектов по типам значений

Объекты обладают свойствами различных типов, которые можно читать и устанавливаться с помощью нескольких *ObjectGet*- и *ObjectSet*-функций. Как мы знаем, данный принцип уже применялся для графиков (см. раздел Обзор функций для работы с полным набором свойств графиков).

Все такие функции принимают в качестве первых трех параметров идентификатор графика, название объекта и идентификатор свойства, который должен быть элементом одного из перечислений ENUM_OBJECT_PROPERTY_INTEGER, ENUM_OBJECT_PROPERTY_DOUBLE или ENUM_OBJECT_PROPERTY_STRING. Конкретные свойства мы изучим постепенно в следующих разделах. Их полные сводные таблицы можно найти в документации по MQL5, на странице Свойства объектов.

Следует отметить, что идентификаторы свойств во всех трех перечислениях не пересекаются, что позволяет объединять в единый унифицированный код их совместную обработку. Мы воспользуется этим в примерах.

Некоторые свойства доступны только на чтение и будут помечаться "r/o" (read-only).

Также как и в случае API для графиков, функции чтения свойств имеют краткую и полную форму: краткая — непосредственно возвращает затребованное значение, а полная — логический признак успеха (*true*) или ошибки (*false*), а само значение помещается в последний параметр, передаваемый по ссылке. Отсутствие ошибки при вызове краткой формы следует проверять с помощью встроенной переменной *_LastError*.

При обращении к некоторым свойствам необходимо указывать дополнительный параметр (*modifier*), который служит для указания номера значения или уровня, если свойство является многозначным. Например, если у объекта несколько точек привязки, то модификатор позволяет выбрать конкретную из них.

Ниже приведены прототипы функций для чтения и записи целочисленных свойств. Обратите внимание, что тип значений в них — это *long*, что позволяет хранить свойства не только типов *int* или *long*, но также *bool*, *color*, *datetime* и различных перечислений (см. далее).

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)

Аналогично описаны функции для вещественных свойств.

```
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)
```

Наконец, четверка таких же функций существует и для строк.

```
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)
```

В целях повышения быстродействия все функции установки свойств объекта (*ObjectSetInteger*, *ObjectSetDouble*, *ObjectSetString*) являются асинхронными и фактически только отправляют графику команды на изменение объекта. При их успешном выполнении команды попадают в общую очередь событий графика, о чем сигнализирует возвращаемый результат *true*. При возникновении ошибки функции вернут *false*, а код ошибки необходимо проверять в переменной *_LastError*.

Изменение свойств объектов производится отложено, в процессе обработки очереди событий графика, то есть, с некоторой задержкой. Для принудительного обновления внешнего вида и свойств объектов на графике, в особенности после изменения сразу множества объектов, используйте функцию *ChartRedraw*.

Функции получения свойств графика (*ObjectGetInteger*, *ObjectGetDouble*, *ObjectGetString*) являются синхронными, то есть вызывающий код дожидается результата их выполнения. При этом выполняются все команды в очереди графика, чтобы получить актуальное значение свойств.

Вернемся к примеру скрипта по удалению объектов, точнее, к его новой версии *ObjectCleanup2.mq5*. Напомним, что в функции *CustomDeleteAllObjects* мы хотели реализовать возможность отбирать объекты на основе их свойств. Допустим, что таковыми свойствами должны быть цвет и точка привязки. Для их получения следует использовать функцию *ObjectGetInteger* и пару элементов перечисления ENUM_OBJECT_PROPERTY_INTEGER: OBJPROP_COLOR и OBJPROP_ANCHOR. Мы подробно рассмотрим их позднее.

С учетом этой информации код дополнился бы следующими проверками (здесь для простоты цвет и точка привязки заданы константами *clrRed* и ANCHOR_TOP — на самом деле мы предусмотрим для них входные переменные):

```
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);
      // условие на имя и дополнительные свойства, например, цвет и точку привязки
      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;
}
```

Обратим внимание на строки с *ObjectGetInteger*.

Их запись длинна и содержит некоторую тавтологию, потому что конкретные свойства привязаны к *ObjectGet*-функциям известных типов. Кроме того, по мере увеличения количества условий могут показаться избыточными повторения идентификатора графика и имени объекта.

Чтобы упростить запись обратимся к технологии, которую мы опробовали в файле *ChartModeMonitor.mqh* в разделе Режимы отображения графика. Её суть в том, чтобы описать класс-посредник с перегрузками методов для чтения и записи свойств всех типов. Назовем новый заголовочный файл *ObjectMonitor.mqh*.

Класс *ObjectProxy* почти полностью повторяет для объектов структуру класса *ChartModeMonitorInterface* для графиков. Основное отличие заключается в наличии виртуальных методов для установки и получения идентификатора графика и имени объекта.

```
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) { }
};
```

Реализуем эти методы в классе-наследнике (позже дополним иерархию классов монитором свойств объектов подобно монитору свойств графика).

```
class ObjectSelector: public ObjectProxy
{
protected:
   long host; // идентификатор графика
   string id; // идентификатор объекта
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;
   }
};
```

Мы разделили абстрактный интерфейс *ObjectProxy* и его минимальную реализацию в *ObjectSelector*, потому что позднее вам может потребоваться реализовать, например, массив посредников для множества однотипных объектов. Тогда достаточно в новом классе "мультиселектора" хранить массив имен или их общий префикс и обеспечивать возврат одного из них из метода *name* за счет вызова перегруженного оператора []: *multiSelector[i].get(OBJPROP_XYZ)*.

Теперь вернемся к скрипту *ObjectCleanup2.mq5* и опишем две входных переменных для задания цвета и точки привязки, как дополнительные условия для отбора объектов, подлежащих удалению.

```
// ObjectCleanup2.mq5
...
input color CustomColor = clrRed;
input ENUM_ARROW_ANCHOR CustomAnchor = ANCHOR_TOP;
```

Передадим эти значения в функцию *CustomDeleteAllObjects*, а новые проверки условий в цикле по объектам можно сформулировать более компактно благодаря классу-посреднику.

```
#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;
}
```

Важно отметить, что имя объекта (и неявный идентификатор текущего графика 0) мы указываем лишь раз — при создании объекта *ObjectSelector*. Далее все свойства запрашиваются методом *get* с единственным параметром, описывающим требуемое свойство, а соответствующую *ObjectGet*-функцию компилятор выберет сам.

Дополнительная проверка на код ошибки 4203 (OBJECT_WRONG_PROPERTY) позволяет отсеять объекты, которые не имеют запрашиваемого свойства, такого как OBJPROP_ANCHOR, например. Таким способом, в частности, можно сделать выборку, в которую попадут все типы стрелок (без необходимости, по отдельности запрашивать разные типы OBJ_ARROW_XYZ), но будут исключены из обработки линии и "события".

Это легко проверить, запустив на графике сначала скрипт *ObjectSimpleShowcase.mq5* (он создаст 14 объектов разных типов), а затем *ObjectCleanup2.mq5*. Если включить режим *UseCustomDeleteAll*, на графике останется 5 неудаленных объектов: OBJ_VLINE, OBJ_HLINE, OBJ_ARROW_BUY, OBJ_ARROW_SELL и OBJ_EVENT. Первые два и последний не имеют свойства OBJPROP_ANCHOR, а стрелки покупки и продажи не проходят по цвету (предполагается, что цвет всех остальных созданных объектов по умолчанию — красный).

Однако *ObjectSelector* задуман не только ради вышеприведенного простого применения. На самом деле он является основой для создания монитора свойств отдельного объекта, аналогичного

тому, что был реализован для графиков. Поэтому заголовочный файл *ObjectMonitor.mqh* содержит нечто более интересное.

```
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) { }
};
```

Данный набор методов должен напомнить вам *ChartModeMonitorInterface* из *ChartModeMonitor.mqh*. Единственное новшество — это метод *applyChanges*: он предназначен для копирования свойств одного объекта в другой.

Отталкиваясь от *ObjectMonitorInterface* описана базовая реализация монитора свойств для пары шаблонных типов: типа значений свойств (одного из *long*, *double*, *string*) и типа перечислений (одного из ENUM_OBJECT_PROPERTY_-что-то там).

```
template<typename T,typename E>
class ObjectMonitorBase: public ObjectMonitorInterface
{
protected:
   MapArray<E,T> data;  // массив пар [свойство,значение], текущее состояние
   MapArray<E,T> store; // бэкап (заполняется по требованию)
   MapArray<E,T> change;// зафиксированные изменения между двумя состояниями
   ...
```

Конструктор *ObjectMonitorBase* имеет два параметра: имя объекта и массив флагов с идентификаторами свойств, подлежащих наблюдению в указанном объекте. Большая часть этого кода почти 1 в 1 повторяет *ChartModeMonitor*. В частности, как и ранее, массив флагов передается во вспомогательный метод *detect*, основное назначение которого выявить те целочисленные константы, которые являются элементами перечисления *E*, и отсеять все остальные. Единственное дополнение, которое следует пояснить, — получение свойства с количеством уровней в объекте посредством *ObjectGetInteger(0, id, OBJPROP_LEVELS)*. Это нужно, чтобы поддержать перебор свойств со множественными значениями из-за наличия уровней (например, Фибоначчи). У объектов без уровней мы получим количество 0, и такое свойство будет привычным, скалярным.

```
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);
      }
   }
   ...
```

Разумеется, метод *detect* несколько отличается от того, что мы видели в *ChartModeMonitor*. Напомним, что для начала в нем идет фрагмент с проверкой константы *v* на принадлежность перечислению *E* с помощью вызова функции *EnumToString*: если такого элемента в перечислении нет, будет взведен код ошибки. Если же элемент есть, мы складываем значение соответствующего свойства в массив *data*.

```
// ChartModeMonitor.mqh
bool detect(const int v)
{
   ResetLastError();
   const string s = EnumToString((E)v); // результирующая строка не важна
   if(_LastError == 0)                  // анализируем код ошибки
   {
      data.put((E)v, get((E)v));
      return true;
   }
   return false;
}
```

В мониторе объектов мы вынуждены усложнить данную схему, поскольку некоторые свойства являются многозначными из-за параметра *modifier* в *ObjectGet*- и *ObjectSet*-функциях.

Поэтому мы вводим статический массив *modifiables* с перечнем тех свойств, которые поддерживают модификаторы (каждое свойство будет подробно рассмотрено в дальнейшем). Суть в том, что для подобных многозначных свойств необходимо их считывать и сохранять в массив *data* не единожды, а несколько раз.

```
// ObjectMonitor.mqh
   bool detect(const int v, const int levels)
   {
      // следующие свойства поддерживают множественность значений
      static const int modifiables[] =
      {
         OBJPROP_TIME,        // точка привязки по времени
         OBJPROP_PRICE,       // точка привязки по цене
         OBJPROP_LEVELVALUE,  // значение уровня
         OBJPROP_LEVELTEXT,   // надпись на линии уровня
         // NB: следующие свойства не генерируют ошибки при превышении
         // фактического количества уровней или файлов
         OBJPROP_LEVELCOLOR,  // цвет линии уровня
         OBJPROP_LEVELSTYLE,  // стиль линии уровня
         OBJPROP_LEVELWIDTH,  // толщина линии уровня
         OBJPROP_BMPFILE,     // файлы с изображениями
      };
      ...
```

Здесь, разумеется, тоже используется трюк с *EnumToString*, чтобы проверить существование свойства с идентификатором *v*. В случае успеха мы проверяем его наличие в списке *modifiables*, и устанавливаем соответствующий флаг *modifiable* в *true* или *false*.

```
      bool result = false;
      ResetLastError();
      const string s = EnumToString((E)v); // результирующая строка не важна
      if(_LastError == 0)                  // анализируем код ошибки
      {
         bool modifiable = false;
         for(int i = 0; i < ArraySize(modifiables); ++i)
         {
            if(v == modifiables[i])
            {
               modifiable = true;
               break;
            }
         }
         ...
```

По умолчанию любое свойство считается однозначным и потому необходимое количество чтений через *ObjectGet*-функцию или записей через *ObjectSet*-функцию равно 1 (переменная *k* ниже).

```
            int k = 1;
            // для свойств с модификаторами ставим правильное количество
            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;
            }
```

Если объект поддерживает уровни, мы ограничиваем потенциальное количество чтений/записей количеством уровней *levels* (как мы помним, оно получено в вызывающем коде из свойства OBJPROP_LEVELS).

Для свойства OBJPROP_BMPFILE, как мы скоро узнаем, допустимо только два состояния: включено (кнопка нажата, флаг проставлен) или выключено (кнопка отжата, флаг сброшен), поэтому *k = 2*.

Наконец, координаты объекта — OBJPROP_TIME и OBJPROP_PRICE — хороши тем, что генерируют ошибку при попытке прочитать/записать несуществующую точку привязки. Поэтому мы назначаем *k* некое заведомо большое значение MOD_MAX, и затем сможем прервать цикл чтения точек при ненулевом значении *_LastError*.

```
            // читаем значение свойства - одно или много
            for(int i = 0; i < k; ++i)
            {
               ResetLastError();
               T temp = get((E)v, i);
               // если i-го модификатора нет, получим ошибку и прервем цикл
               if(_LastError != 0) break;
               data.put((E)MOD_COMBINE(v, i), temp);
               result = true;
            }
         }
         return result;
      }
```

Поскольку у одного свойства может быть несколько значений, которые читаются в цикле вплоть до *k*, мы уже не можем просто написать *data.put((E)v, get((E)v))*. Нам необходимо неким образом совместить идентификатор свойства *v* и номер его модификации *i*. К счастью, количество свойств ограничено и в целочисленной константе (тип *int*) занято не более двух младших байтов. Поэтому мы можем с помощью побитовых операторов поместить *i* в верхний байт. Для этой цели разработан макрос MOD_COMBINE.

```
#define MOD_COMBINE(V,I) (V | (I << 24))
```

Разумеется, для извлечения идентификатора свойства и номера модификации предусмотрены обратные макросы.

```
#define MOD_GET_NAME(V)  (V & 0xFFFFFF)
#define MOD_GET_INDEX(V) (V >> 24)
```

Вот, например, как они используются в методе *snapshot*.

```
virtual int snapshot() override
{
   MapArray<E,T> temp;
   change.reset();

   // собираем все требуемые свойства в 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;
   // сравниваем предыдущее и новое состояние
   for(int i = 0; i < data.getSize(); ++i)
   {
      if(data[i] != temp[i])
      {
         // сохраняем различия в массив change
         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++;
      }
   }

   // сохраняем новое состояние как текущее
   data = temp;
   return changes;
}
```

В принципе, данный метод повторяет всю логику одноименного метода в *ChartModeMonitor.mqh*, однако для чтения свойств везде приходится предварительно выделять из хранимого ключа имя свойства с помощью MOD_GET_NAME и номер с помощью MOD_GET_INDEX.

Аналогичное усложнение приходится делать и в методе *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);
   }
}
```

Самым интересным нововведением *ObjectMonitorBase* является работа с изменениями.

```
   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);
            }
         }
      }
   }
```

Передав в метод *applyChanges* монитор состояния другого объекта, мы можем перенять из него все последние изменения.

Для поддержки свойств всех трех основных типов (*long*, *double*, *string*) нам требуется реализовать класс *ObjectMonitor* (аналог *ChartModeMonitor* из *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);
   }
   ...
```

Здесь также сохранена прежняя структура кода, а добавились только методы для поддержки изменений и имени (у графиков, как мы помним, имен нет).

```
...
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));
   }
}
```

На основе созданного монитора объектов легко реализовать несколько приемов, поддержка которых отсутствует в терминале. В частности, это создание копий объектов и групповое редактирование объектов.

Скрипт ObjectCopy

Копирование выделенных объектов демонстрирует скрипт *ObjectCopy.mq5*. В начале его функции *OnStart* мы заполняем массив *flags* последовательными целыми числами, которые являются кандидатами на элементы ENUM_OBJECT_PROPERTY_-перечислений разных типов. Нумерация элементов перечислений имеет ярко выраженную группировку по назначению, причем между группами встречаются большие пропуски (видимо, запас под будущие элементы), поэтому массив формируется достаточно большой: 2048 элементов.

```
#include <MQL5Book/ObjectMonitor.mqh>

#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1) - 1] = V)

void OnStart()
{
   int flags[2048];
   // заполняем массив последовательными целыми числами, которые будут
   // проверены на соответствие элементам перечислений объектных свойств,
   // некорректные значения будут отброшены в методе detect монитора
   for(int i = 0; i < ArraySize(flags); ++i)
   {
      flags[i] = i;
   }
   ...
```

Далее мы собираем в массив имена объектов, которые в данный момент выделены на графике — для этого используется свойство 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);
      }
   }
   ...
```

Наконец, в основном рабочем цикле по выбранным элементам мы считываем свойства каждого объекта, формируем название его копии и создаем под ним объект с таким же набором атрибутов.

```
      for(int i = 0; i < ArraySize(selected); ++i)
      {
         const string name = selected[i];

         // делаем с помощью монитора бэкап свойств текущего объекта
         ObjectMonitor object(name, flags);
         object.print();
         object.backup();
         // формируем корректное, подходящее имя для копии
         const string copy = GetFreeName(name);

         if(StringLen(copy) > 0)
         {
            Print("Copy name: ", copy);
            // создаем объект такого же типа OBJPROP_TYPE
            ObjectCreate(0, copy,
               (ENUM_OBJECT)ObjectGetInteger(0, name, OBJPROP_TYPE),
               ObjectFind(0, name), 0, 0);
            // меняем в мониторе имя объекта на новое
            object.name(copy);
            // восстанавливаем все свойства из бэкапа уже в новый объект
            object.restore();
         }
         else
         {
            Print("Can't create copy name for: ", name);
         }
      }
   }
```

Здесь важно отметить, что свойство OBJPROP_TYPE является одним из немногих, доступных только на чтение, и потому объект необходимо сразу создавать требуемого типа.

Вспомогательная функция *GetFreeName* пытается добавить к имени объекта строку "/Copy №x", где x — номер копии. Таким образом, запуская скрипт несколько раз, можно создавать 2-ую, 3-юю, и так далее копии.

```
string GetFreeName(const string name)
{
   const string suffix = "/Copy №";
   // проверяем, нет ли в имени суффикса копии
   const int pos = StringFind(name, suffix);
   string prefix;
   int n;

   if(pos <= 0)
   {
      // если суффикс не найден, предполагаем копию под номером 1
      const string candidate = name + suffix + "1";
      // проверяем свободно ли имя копии, и если да - возвращаем его
      if(ObjectFind(0, candidate) < 0)
      {
         return candidate;
      }
      // иначе готовимся к циклу с перебором номеров копий
      prefix = name;
      n = 0;
   }
   else
   {
      // если суффикс найден, выделяем название без него
      prefix = StringSubstr(name, 0, pos);
      // и находим в строке номер копии
      n = (int)StringToInteger(StringSubstr(name, pos + StringLen(suffix)));
   }

   Print("Found: ", prefix, " ", n);
   // пытаемся в цикле найти свободный номер копии выше n, но не более 1000
   for(int i = n + 1; i < 1000; ++i)
   {
      const string candidate = prefix + suffix + (string)i;
      // проверяем существование объекта с именем с концовкой "Copy №i"
      if(ObjectFind(0, candidate) < 0)
      {
         return candidate; // возвращаем вакантное имя копии
      }
   }
   return NULL; // слишком много копий
}
```

В принципе, терминал запоминает последние настройки конкретного типа объектов, и если их создавать один за другим, это эквивалентно копированию. Однако, настройки обычно меняются в процессе работы с разными графиками, и если по прошествии времени возникает необходимость продублировать какой-то "старый" объект, то настройки для него, как правило, приходится делать полностью. Особенно накладно это для типов объектов с большим количеством свойств, например, инструментов Фибоначчи. В таких случаях и пригодится данный скрипт.

Некоторые картинки из этой главы, на которых размещены однотипные объекты, были созданы с помощью данного скрипта.

Индикатор ObjectGroupEdit

Вторым примером использования *ObjectMonitor* выступит индикатор *ObjectGroupEdit.mq5*, который позволяет редактировать свойства сразу у группы выделенных объектов.

Представим себе, что мы выделили на графике несколько объектов (не обязательного одного и того же типа), у которых необходимо единообразно поменять то или иное свойство. Далее мы открываем диалог свойств любого из этих объектов, настраиваем его, и по нажатию *OK* изменения применяются ко всем выделенным объектам. Именно так работает наша следующая MQL-программа.

Индикатор как тип программы потребовался нам потому, что в нем задействованы события графика. Этому аспекту программирования на MQL5 будет посвящена целая глава, но с некоторыми основами мы познакомимся прямо сейчас.

Поскольку индикатор не имеет диаграмм, директивы *#property* содержат нули, а функция *OnCalculate* — практически пуста.

```
#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;
}
```

Для автоматической генерации полного набора всех свойств объектом мы опять воспользуемся массивом размером 2048 элементов с последовательными целочисленными значениями. Также предусмотрим массив для имен выделенных элементов и массив объектов-мониторов класса *ObjectMonitor*.

```
int consts[2048];
string selected[];
ObjectMonitor *objects[];
```

В обработчике *OnInit* инициализируем массив чисел и запускаем таймер.

```
void OnInit()
{
   for(int i = 0; i < ArraySize(consts); ++i)
   {
      consts[i] = i;
   }
   
   EventSetTimer(1);
}
```

В обработчике таймера сохраняем в массив имена выделенных объектов. Если список выделения поменялся, требуется перенастроить объекты-мониторы, для чего вызывается вспомогательная функция *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();
   }
}
```

Сама функция *TrackSelectedObjects* довольно проста: удаляем прежние мониторы и создаем новые. При желании можно сделать её более интеллектуальной — с сохранением неизменившейся части выделения.

```
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));
   }
}
```

Напомним, что при создании объекта-монитора он сразу снимает "слепок" всех свойств соответствующего графического объекта.

Теперь мы, наконец, добрались до той части, где в дело вступают события. Как уже было сказано в обзоре функций событий, за события на графике отвечает обработчик *OnChartEvent*. Нас в данном примере интересует конкретное событие CHARTEVENT_OBJECT_CHANGE: оно происходит, когда пользователь меняет какие-либо атрибуты в диалоге свойств объекта. Название измененного объекта передается в параметре *sparam*.

Если это имя совпадает с одним из объектов под наблюдением, мы просим монитор сделать новый слепок его свойств, то есть вызываем *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;
         }
      }
   }
}
```

Если изменения подтверждены (а иное вряд ли возможно), их количество в переменной *changes* будет больше 0. Тогда запускается цикл по всем выделенным объектам, и к каждому из них, кроме исходного, применяются обнаруженные изменения.

Поскольку мы потенциально можем изменять много объектов, вызываем запрос перерисовки графика с помощью *ChartRedraw*.

В обработчике *OnDeinit* не забываем удалить все мониторы.

```
void OnDeinit(const int)
{
   for(int j = 0; j < ArraySize(objects); ++j)
   {
      delete objects[j];
   }
   Comment("");
}
```

Вот и все: новый инструмент готов.

Данный индикатор позволил настроить общий внешний вид нескольких групп объектов-надписей в разделе Определение точки привязки на объекте.

Кстати говоря, по похожему принципу с помощью *ObjectMonitor* можно сделать еще один популярный инструмент, отсутствующий в терминале: для отмены правок свойств объектов — ведь для нас готов метод *restore*.

5.8.8 Основные свойства объектов

Все объекты обладают некоторыми универсальными атрибутами. Основные из них перечислены в следующей таблице. Позднее мы познакомимся с другими общими свойствами специального назначения (см. разделы Управление состоянием объекта, Z-порядок, Видимость объектов в разрезе таймфреймов).

Идентификатор	Описание	Тип
OBJPROP_NAME	Имя объекта	string
OBJPROP_TYPE	Тип объекта (r/o)	ENUM_OBJECT
OBJPROP_CREATETIME	Время создания объекта (r/o)	datetime
OBJPROP_TEXT	Описание объекта (текст, содержащийся в объекте)	string
OBJPROP_TOOLTIP	Текст всплывающей подсказки по наведению мыши	string

Свойство OBJPROP_NAME является идентификатором объекта. Его редактирование эквивалентно удалению старого объекта и созданию нового.

Для некоторых типов объектов, способных отображать текст (таких как надписи или кнопки), свойство OBJPROP_TEXT всегда выводится непосредственно на график, внутри объекта. Для остальных объектов (например, линий) это свойство содержит описание, которое выводится на график рядом с объектом и только в том случае, если в настройках графика включена опция Показывать описания объектов. В любом случае OBJPROP_TEXT выводится во всплывающей подсказке.

Свойство OBJPROP_CREATETIME сохраняется только до конца текущего сеанса и не записывается в chr-файлы.

Вы можете изменить название объекта программно или вручную (в диалоге свойств объекта) — при этом его время создания останется прежним. Забегая вперед, отметим, что программное переименование не вызывает на графике каких-либо событий об объектах. Как мы узнаем в следующей главе, ручное переименование вызывает три события:

- удаление объекта под старым именем (CHARTEVENT_OBJECT_DELETE),
- создание объекта под новым именем (CHARTEVENT_OBJECT_CREATE) и
- модификацию нового объекта (CHARTEVENT_OBJECT_CHANGE).

Если свойство OBJPROP_TOOLTIP не задано, для объекта показывается подсказка, автоматически формируемая терминалом. Чтобы отключить показ подсказки, следует установить для нее значение "\n" (перевод строки).

Адаптируем скрипт *ObjectFinder.mq5* из раздела Поиск объектов для вывода в журнал всех вышеуказанных свойств объектов на текущем графике. Назовем новый скрипт *ObjectListing.mq5*.

В самом начале *OnStart* создадим или модифицируем вертикальную прямую линию, расположенную на последнем баре (на момент запуска скрипта). Если в настройках графика стоит опция показывать описания объектов, то вдоль правой вертикальной линии увидим текст "Latest Bar At The Moment".

```
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");
   ...
```

Далее в цикле по подокнам запросим все объекты вплоть до *ObjectsTotal* и их основные свойства.

```
   int count = 0;
   const long id = ChartID();
   const int win = (int)ChartGetInteger(id, CHART_WINDOWS_TOTAL);
   // проходим по подокнам
   for(int k = 0; k < win; ++k)
   {
      PrintFormat("  Window %d", k);
      const int n = ObjectsTotal(id, k);
      // перебираем объекты
      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);
}
```

Получим в журнале примерно следующие записи.

```
  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
```

Нулевое значение OBJPROP_CREATETIME (1970.01.01 00:00) означает, что объект был создан не во время текущего сеанса, а раньше.

5.8.9 Координаты времени и цены

Для объектов тех типов, которые существуют в системе координат котировок, MQL5 API поддерживает пару свойств для указания привязки ко времени и ценам. В том случае, если у объекта несколько точек привязки, свойства требуют при вызове *ObjectSet*- и *ObjectGet*-функций указания параметра-модификатора, содержащего номер точки привязки.

Идентификатор	Описание	Тип значений
OBJPROP_TIME	Координата времени	datetime
OBJPROP_PRICE	Координата цены	double

Эти свойства доступны у абсолютно всех объектов, но их не имеет смысла устанавливать или читать у объектов с экранными координатами.

Для демонстрации работы с координатами разберем безбуферный индикатор *ObjectHighLowChannel.mq5*. Для заданного отрезка баров он проводит две трендовые линии. Их начальные и конечные точки на оси времени совпадают с первым и последним баром отрезка, а по оси цен значения считаются по разному для каждой из линий: для верхней берутся максимальная и минимальная цены *High*, а для нижней — максимальная и минимальная цены *Low*. По мере обновления графика, наш импровизированный канал должен двигаться вслед за ценами.

Диапазон баров задается с помощью двух входных переменных: номер начального бара *BarOffset* и количество баров *BarCount*. По умолчанию линии строятся на самых последних ценах, поскольку *BarOffset = 0*.

```
input int BarOffset = 0;
input int BarCount = 10;

const string Prefix = "HighLowChannel-";
```

Для объектов определен общий префикс имен "HighLowChannel-".

В обработчике *OnCalculate* мы отслеживаем появление новых баров по времени *iTime* 0-го бара. Как только бар формируется, производится анализ цен на заданном отрезке, берутся максимальные и минимальные значения цен каждого из двух типов (MODE_HIGH, MODE_LOW) и для них вызывается вспомогательная функция *DrawFigure*, где и происходит работа с объектами: создание и модификация координат.

```
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;
}
```

А вот и сама функция *DrawFigure*.

```
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;
}
```

После вызова *ObjectCreate*, который гарантирует наличие объекта, вызываются соответствующие ObjectSet-функции для OBJPROP_TIME и OBJPROP_PRICE во всех точках привязки (в данном случае, в двух).

На изображении ниже приведен результат работы индикатора.

Канал на двух трендовых линиях по ценам High и Low

Вы можете запустить индикатор в визуальном тестере, чтобы убедиться, как координаты линий меняются на лету.

5.8.10 Угол окна привязки и экранные координаты

Для объектов, которые используют систему координат в виде точек (пикселей) на графике, необходимо выбрать один из четырех углов окна, относительно которого будут отсчитываться значения по горизонтальной оси X и вертикальной оси Y до точки привязки на объекте. Этими аспектами управляют свойства из следующей таблицы.

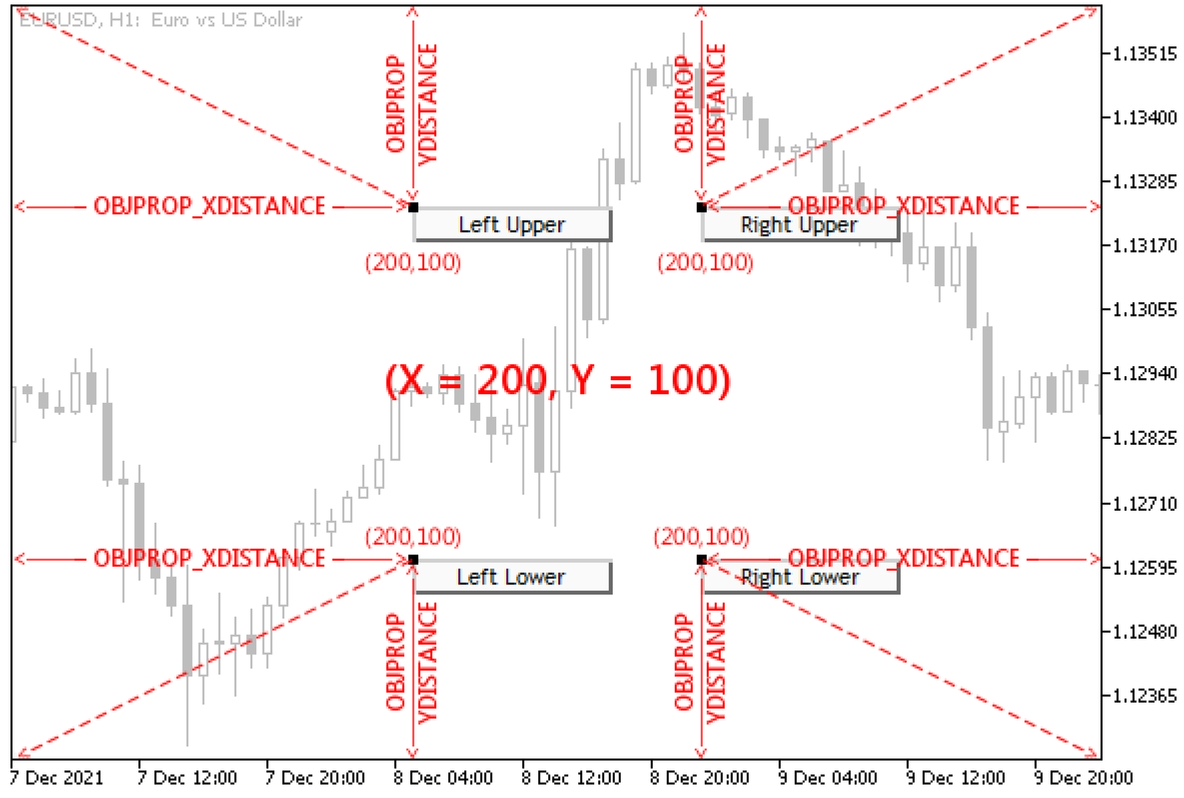
Идентификатор	Описание	Тип
OBJPROP_CORNER	Угол графика для привязки графического объекта	ENUM_BASE_CORNER
OBJPROP_XDISTANCE	Дистанция в пикселях по оси X от угла привязки	int
OBJPROP_YDISTANCE	Дистанция в пикселях по оси Y от угла привязки	int

Допустимые варианты для OBJPROP_CORNER сведены в перечисление ENUM_BASE_CORNER.

Идентификатор	Расположение центра координат
CORNER_LEFT_UPPER	Левый верхний угол окна
CORNER_LEFT_LOWER	Левый нижний угол окна
CORNER_RIGHT_LOWER	Правый нижний угол окна
CORNER_RIGHT_UPPER	Правый верхний угол окна

По умолчанию используется левый верхний угол.

На следующем рисунке показаны четыре объекта "Кнопка" с одинаковыми размерами и дистанцией до угла привязки в окне. У каждого из этих объектов отличается только сам угол привязки. Напомним, что у кнопок точка привязки одна и всегда расположена в левом верхнем углу кнопки.

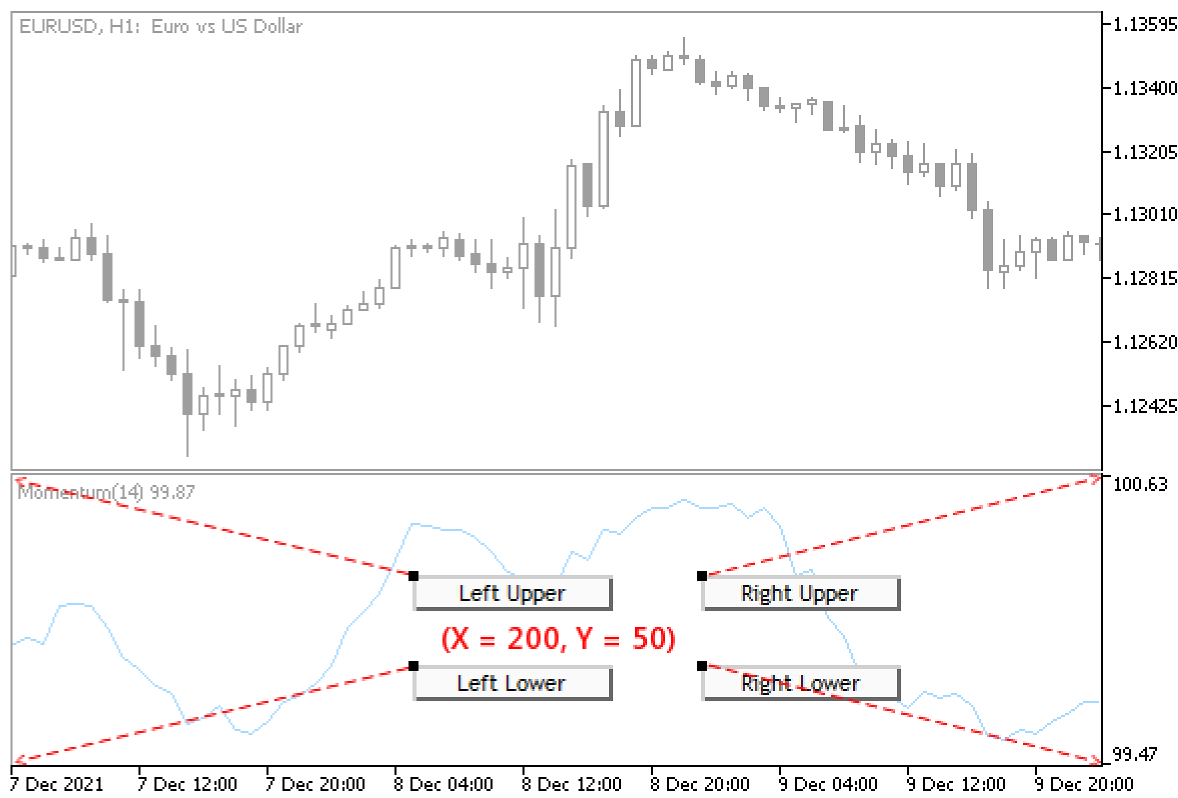


Расположение объектов с привязкой к разным углам основного окна

Все 4 объекта в данный момент выделены на графике, поэтому их точки привязки подсвечиваются контрастным цветом.

Когда мы говорим об углах окна, подразумевается конкретное окно или подокно, в котором расположен объект, а не весь график целиком. Иными словами, в объектах в подокнах координата Y отсчитывается от верхней или нижней границы этого подокна.

На следующем рисунке показаны аналогичные объекты в подокне, с привязкой к углам подокна.



Расположение объектов с привязкой к разным углам подокна

С помощью скрипта *ObjectCornerLabel.mq5* пользователь может протестировать перемещение текстовой надписи, для которой угол привязки в окне задается во входном параметре *Corner*.

```
#property script_show_inputs

input ENUM_BASE_CORNER Corner = CORNER_LEFT_UPPER;
```

Координаты периодически изменяются и выводятся в тексте самой надписи. Таким образом, надпись перемещается в окне и при достижении границы отскакивает от неё. Объект создается в том окне или подокне, куда скрипт был брошен мышью.

```
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;
   ...
```

Для корректного позиционирования мы узнаем размеры окна (и далее проверяем, не изменились ли они) и находим середину для начального размещения объекта: переменные с координатами — *x* и *y*.

Далее мы создаем и настраиваем надпись, пока без координат. Важно отметить, что мы включаем возможность выделять объект (OBJPROP_SELECTABLE) и выделяем его (OBJPROP_SELECTED), так

как это позволяет увидеть точку привязки на самом объекте, до которой и отсчитывается расстояние от угла окна (центра координат). Более подробно данные два свойства описаны в разделе Управление состоянием объекта.

```
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);
...
```

В переменных *px* и *py* будем записывать приращения координат для эмуляции движения. Сама модификации координат будет производиться в бесконечном цикле, пока его не прервет пользователь. Счетчик итераций позволит периодически, на каждой 50 итерации, случайным образом менять направление движения.

```
   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);
      }
      
      // отскок от границ окна, чтобы объект не скрылся
      if(x + px > w || x + px < 0) px = -px;
      if(y + py > h || y + py < 0) py = -py;
      // пересчитываем позиции надписи
      x += px;
      y += py;
      
      // обновляем координаты объекта и вносим их в текст
      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);
}
```

Попробуйте запустить скрипт несколько раз, указав различные углы привязки.

В следующем разделе мы дополним этот скрипт, управляя также и точкой привязки на объекте.

5.8.11 Определение точки привязки на объекте

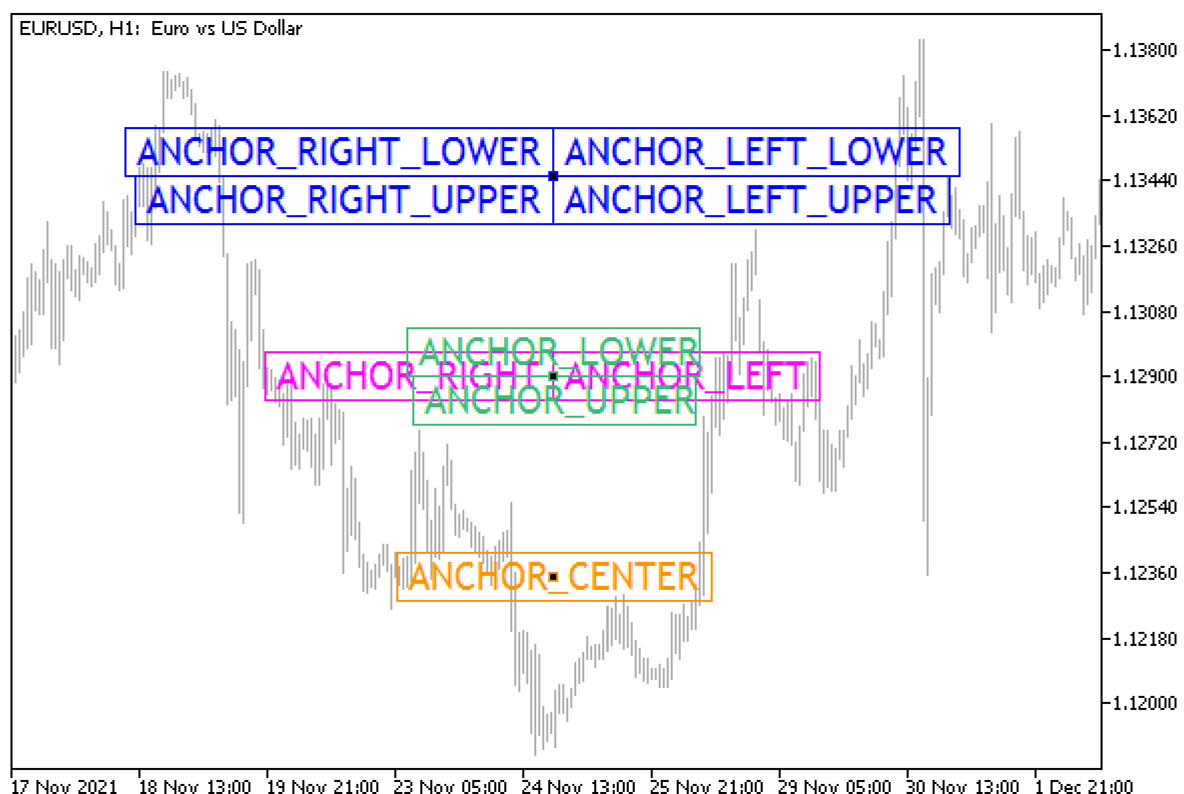
Объекты некоторых типов позволяют выбирать точку привязки. К числу таких типов относятся: текстовая метка (OBJ_TEXT) и растровая картинка (OBJ_BITMAP) с привязкой к котировкам, а также надпись (OBJ_LABEL) и панель с картинкой (OBJ_BITMAP_LABEL), позиционируемые в экранных координатах.

Для чтения и установки точки привязки следует использовать функции *ObjectGetInteger* и *ObjectSetInteger* со свойством OBJPROP_ANCHOR.

Все варианты выбора точки собраны в перечислении ENUM_ANCHOR_POINT.

Идентификатор	Расположение точки привязки
ANCHOR_LEFT_UPPER	В левом верхнем углу
ANCHOR_LEFT	Слева по центру
ANCHOR_LEFT_LOWER	В левом нижнем углу
ANCHOR_LOWER	Снизу по центру
ANCHOR_RIGHT_LOWER	В правом нижнем углу
ANCHOR_RIGHT	Справа по центру
ANCHOR_RIGHT_UPPER	В правом верхнем углу
ANCHOR_UPPER	Сверху по центру
ANCHOR_CENTER	Строго по центру объекта

Точки наглядно показаны на изображении ниже, где на график нанесено несколько объектов-надписей.



Текстовые объекты OBJ_LABEL с разными точками привязки

Верхняя группа из 4-х надписей имеет одну и ту же пару координат (X,Y), однако из-за привязки к разным углам объекта, они расположены по разные стороны от точки. Похожая ситуация во второй группе из 4-х надписей, однако там привязка сделана к серединам разных сторон объектов. Наконец, внизу отдельно показана надпись с привязкой по её центру, так что точка находится внутри объекта.

Объекты кнопка (OBJ_BUTTON), прямоугольная панель (OBJ_RECTANGLE_LABEL), поле вводе (OBJ_EDIT) и объект-график (OBJ_CHART) имеют фиксированную точку привязки в левом верхнем углу (ANCHOR_LEFT_UPPER).

Некоторые графические объекты группы одиночных ценовых меток (OBJ_ARROW, OBJ_ARROW_THUMB_UP, OBJ_ARROW_THUMB_DOWN, OBJ_ARROW_UP, OBJ_ARROW_DOWN, OBJ_ARROW_STOP, OBJ_ARROW_CHECK) имеют 2 способа привязки своих координат, задаваемые идентификаторами другого перечисления — ENUM_ARROW_ANCHOR.

Идентификатор	Расположение точки привязки
ANCHOR_TOP	Сверху по центру
ANCHOR_BOTTOM	Снизу по центру

Остальные объекты этой группы имеют предопределенные точки привязки: стрелки покупки (OBJ_ARROW_BUY) и продажи (OBJ_ARROW_SELL) — соответственно, посередине верхней и нижней стороны, а ценовые метки (OBJ_ARROW_RIGHT_PRICE, OBJ_ARROW_LEFT_PRICE) — слева и справа.

По аналогии со скриптом *ObjectCornerLabel.mq5* из предыдущего раздела создадим скрипт *ObjectAnchorLabel.mq5*. В новой версии помимо перемещения надписи будем случайным образом менять точку привязки на ней.

Угол окна для привязки будет, как и раньше, выбирать пользователь при запуске скрипта.

```
input ENUM_BASE_CORNER Corner = CORNER_LEFT_UPPER;
```

Мы выведем название угла на график в виде комментария.

```
void OnStart()
{
   Comment(EnumToString(Corner));
   ...
```

В бесконечном цикле в выбранные моменты генерируется одно из 9 возможных значений точки привязки.

```
   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);
      }
      ...
```

Название точки привязки становится текстовым содержимым надписи, вместе с текущими координатами.

```
      ObjectSetString(0, name, OBJPROP_TEXT, EnumToString(anchor)
         + "[" + (string)x + "," + (string)y + "]");
```

Прочие фрагменты кода остались практически неизменными.

Откомпилировав и запустив скрипт, обратите внимание, как надпись меняет свое положение относительно текущих координат (x,y) в зависимости от выбранной точки привязки.

Пока мы контролируем и предотвращаем выход за пределы окна непосредственно точки привязки. Однако объект имеет некоторые размеры, и потому сейчас может оказаться, что большая часть надписи обрезается. В дальнейшем, после изучения соответствующих свойств, мы займемся этой проблемой (см. пример *ObjectSizeLabel.mq5* в разделе Определение ширины и высоты объектов).

5.8.12 Управление состоянием объекта

Среди общих свойств объектов имеется несколько, управляющих состоянием объектов. Все такие свойства имеют логический тип, то есть могут быть включены (*true*) или выключены (*false*), и потому требуют использования функций *ObjectGetInteger* и *ObjectSetInteger*.

Идентификатор	Описание
OBJPROP_HIDDEN	Запрет на показ имени графического объекта в списке объектов в одноименном диалоге (вызывается из контекстного меню графика или по Ctrl+B).
OBJPROP_SELECTED	Выделенность объекта
OBJPROP_SELECTABLE	Доступность объекта для выделения

Значение *true* для OBJPROP_HIDDEN позволяет скрыть ненужный для пользователя объект из списка. По умолчанию *true* устанавливается для объектов, которые отображают события календаря, историю торговли, а также для созданных из MQL-программ. Для того чтобы увидеть такие графические объекты и получить доступ к их свойствам, нужно нажать кнопку *Все* в диалоге *Список объектов*.

Скрытый в списке объект остается видимым на графике. Чтобы скрыть объект на графике, не удаляя его, можно воспользоваться настройкой Видимости объектов в разрезе таймфреймов.

Пользователь не может выделять и менять свойства объектов, для которых OBJPROP_SELECTABLE равно *false*. Объекты, созданные программно, по умолчанию запрещены для выделения. Как мы видели в скриптах *ObjectCornerLabel.mq5* и *ObjectAnchorLabel.mq5* в предыдущих разделах, потребовалось явным образом установить OBJPROP_SELECTABLE в *true*, чтобы разблокировать возможность включить также и OBJPROP_SELECTED — таким способом мы подсветили точки привязки на объекте.

Обычно MQL-программы разрешают выделение своих объектов только в том случае, если эти объекты служат элементами управления. Например, трендовая линия с предопределенным именем, которую пользователь перемещает по желанию, может означать условие отправки торгового приказа при пересечении её ценой.

5.8.13 Приоритет объектов (Z-порядок)

Объекты на графике обеспечивают не только представление информации, но и взаимодействие с пользователем и MQL-программами посредством событий, о которых пойдет подробный рассказ в следующей главе. Одним из источников событий является манипулятор мышь. График способен, в частности, отслеживать перемещение мыши и нажатия её кнопок.

Если под мышью находится тот или иной объект, для него может выполнять специфическая обработка события. Однако объекты могут накладываться друг на друга (когда их координаты с учетом размеров перекрываются). В этом случае в дело вступает целочисленное свойство OBJPROP_ZORDER. Оно задает приоритет графического объекта на получение событий мыши. При наложении объектов друг на друга событие получит только один объект, чей приоритет выше остальных.

По умолчанию при создании объекта его Z-порядок равен нулю, но при необходимости его можно повысить.

> Важно отметить, что Z-порядок влияет только на обработку событий мыши, но не на отрисовку объектов. Объекты всегда рисуются в порядке их добавления на график. Это может служить источником недоразумений. Например, всплывающая подсказка может показываться не для того объекта, который визуально находится поверх другого, потому что перекрытый объект обладает более высоким Z-приоритетом (см. пример).

В скрипте *ObjectZorder.mq5* мы создадим 12 объектов типа OBJ_RECTANGLE_LABEL, разместив их по кругу, как на циферблате. Порядок добавления объектов соответствует часам: от 1 до 12. Для наглядности все прямоугольники получат случайный цвет (про свойство OBJPROP_BGCOLOR см. следующий раздел), а также случайный приоритет. Двигая мышью над объектами, пользователь сможет по всплывающей подсказке определить, к какому объекту она относится.

Для удобства настройки свойств объектов определим специальный класс *ObjectBuilder*, производный от *ObjectSelector*.

```
#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);
   }

   // изменение имени и графика запрещено
   virtual void name(const string _id) override = delete;
   virtual void chart(const long _chart) override = delete;
};
```

Поля с идентификаторами объекта (*id*) и графика (*host*) уже имеются в класса *ObjectSelector*. В производном мы добавляем тип объекта (*ENUM_OBJECT type*) и номер окна (*int window*).

Особенностью конструктора является то, что в нем вызывается *ObjectCreate*.

Настройка и чтение свойств полностью наследуется в виде группы *get*- и *set*-методов из *ObjectSelector*.

Как и в предыдущих тестовых скриптах, определяем окно, куда брошен скрипт, размеры окна и координаты середины.

```
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;
   ...
```

Поскольку тип объектов OBJ_RECTANGLE_LABEL поддерживает явное указание размеров в пикселях, рассчитаем ширину *dx* и высоту *dy* каждого прямоугольника как четверть окна. Их мы используем для задания свойств OBJPROP_XSIZE и OBJPROP_YSIZE, рассматриваемых в разделе Определение ширины и высоты объектов.

```
const int dx = w / 4;
const int dy = h / 4;
...
```

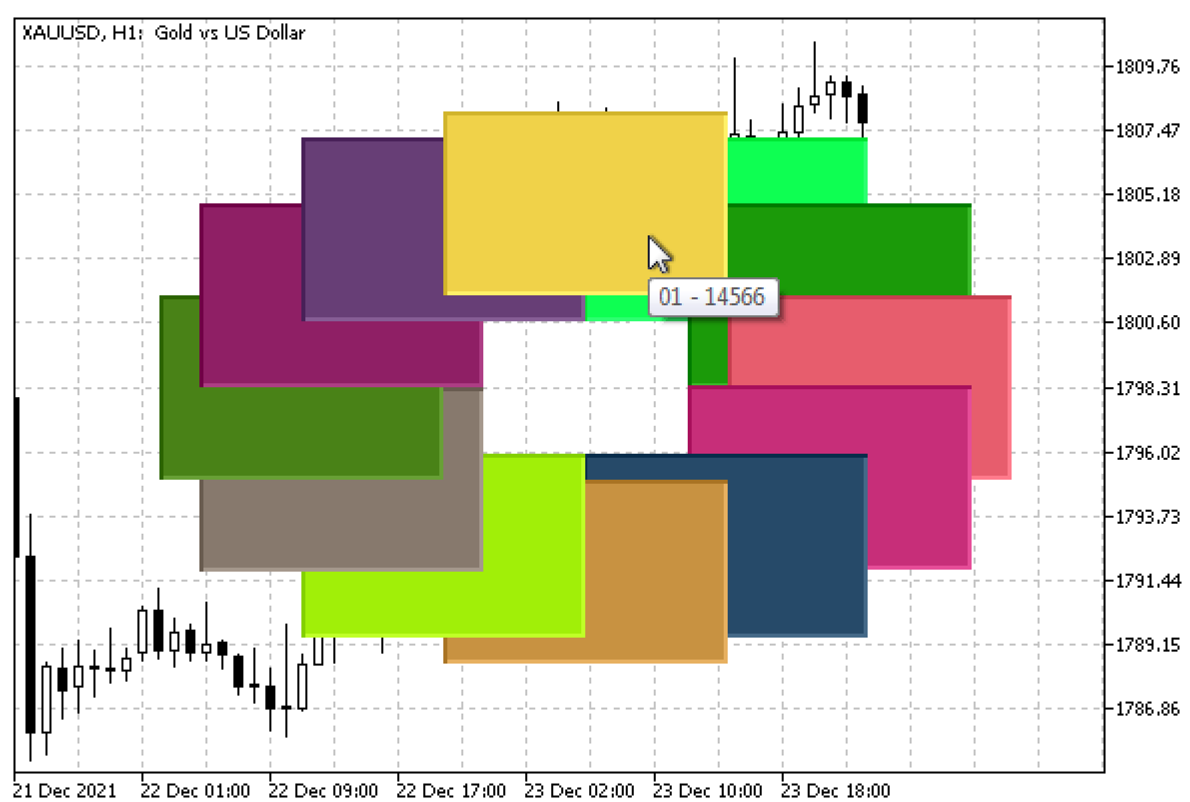
Далее в цикле создаем 12 объектов. Переменные *px* и *py* содержать смещение очередного "знака" на "циферблате" относительно центра (x,y). Приоритет *z* выбирается случайно. В название объекта и его подсказку (OBJPROP_TOOLTIP) включается строка вида "XX - YYY", XX — номер "часа" (позиция на циферблате от 1 до 12), YYY — приоритет.

```
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;
}
```

После того как вызван конструктор *ObjectBuilder*, для нового объекта *builder* цепочкой нанизаны вызовы перегруженного метода *set* для разных свойств (метод *set* возвращает указатель на сам объект).

Поскольку MQL-объект больше не нужен после создания и настройки графического объекта, тут же удаляем *builder*.

В результате выполнения скрипта на графике появятся примерно такие объекты.



Наложение объектов и всплывающие подсказки по приоритетам Z-порядка

Цвета и приоритеты будут отличаться при каждом запуске, но визуальное наложение прямоугольников всегда будет именно таким, в порядке создания от 1 (внизу) до 12 (на самом верху — здесь имеется в виду наложение объектов, а не тот факт, что 12 расположено вверху циферблата).

На изображении курсор мыши расположен в таком месте, где существуют два объекта — 01 (флюоресцирующий зелёный "лайм") и 12 (песочный). В данном случае видна подсказка для объекта 01, хотя визуально объект 12 отображается поверх объекта 01. Это происходит из-за того, что для 01 был случайно сгенерирован более высокий приоритет, чем у 12.

Одновременно выводится только одна подсказка, поэтому проверить соотношение приоритетов можно, двигая курсор мыши в другие области, где наложение объектов отсутствует, и информация в подсказке принадлежит единственному объекту под курсором.

Когда изучим обработку событий мыши в следующей главе, мы сможем улучшить данный пример и проверить влияние Z-порядка на щелчки мыши по объектам.

Для удаления созданных объектов можно воспользоваться скриптом *ObjectCleanup1.mq5*.

5.8.14 Настройка отображения объекта: цвет, стиль и рамка

Внешний вид объектов можно менять с помощью множества свойств, изучение которых мы начнем в этом разделе с цвета, стиля, ширины линий и рамок. Другие аспекты форматирования, такие как шрифт, угол наклона и выравнивание текста будут рассмотрены в следующих разделах.

Все свойства из нижеприведенной таблицы имеют типы, совместимые с целыми числами, в связи с чем управляются функциями *ObjectGetInteger* и *ObjectSetInteger*.

Идентификатор	Описание	Тип свойства
OBJPROP_COLOR	Цвет линии и основного элемента объекта (например, шрифта или заливки)	color
OBJPROP_STYLE	Стиль линии	ENUM_LINE_STYLE
OBJPROP_WIDTH	Толщина линии в пикселях	int
OBJPROP_FILL	Заливка объекта цветом (для OBJ_RECTANGLE, OBJ_TRIANGLE, OBJ_ELLIPSE, OBJ_CHANNEL, OBJ_STDDEVCHANNEL, OBJ_REGRESSION)	bool
OBJPROP_BACK	Объект на заднем плане	bool
OBJPROP_BGCOLOR	Цвет фона для OBJ_EDIT, OBJ_BUTTON, OBJ_RECTANGLE_LABEL	color
OBJPROP_BORDER_TYPE	Тип рамки для прямоугольной панели OBJ_RECTANGLE_LABEL	ENUM_BORDER_TYPE
OBJPROP_BORDER_COLOR	Цвет рамки для поля ввода OBJ_EDIT и кнопки OBJ_BUTTON	color

В отличие от большинства объектов с линиями (отдельные вертикальная и горизонтальная, трендовая, циклические, каналы и т.д.), где свойство OBJPROP_COLOR определяет цвет линии, для картинок OBJ_BITMAP_LABEL и OBJ_BITMAP оно определяет цвет рамки, а OBJPROP_STYLE — тип отрисовки рамки.

Перечисление ENUM_LINE_STYLE, используемое для OBJPROP_STYLE, мы уже встречали в главе про индикаторы, в разделе Настройка графических построений.

Следует отличать заливку, выполняемую основным цветом OBJPROP_COLOR, от цвета фона OBJPROP_BGCOLOR. И то, и другое поддерживается разными группами типов объектов — они перечислены в таблице.

Свойство OBJPROP_BACK требует отдельных пояснений. Дело в том, что объекты и индикаторы по умолчанию выводятся поверх графика цен. Пользователь может изменить такое поведение для всего графика целиком, зайдя в диалог *Настройка* графика и далее закладка *Общие*, опция *График сверху*. Этот флажок имеет и программный аналог — свойство CHART_FOREGROUND (см. Режимы отображения графика). Однако иногда желательно убрать на задний план не все объекты, а лишь избранные. Тогда для них можно установить OBJPROP_BACK в *true*. При этом объект будет перекрываться даже сеткой и разделителями периодов, если они включены на графике.

Когда включен режим заливки OBJPROP_FILL, цвет баров, попадающих внутрь фигуры, зависит от свойства OBJPROP_BACK. По умолчанию, при OBJPROP_BACK равном *false*, бары, наложившиеся на объект, рисуются инвертированным цветом по отношению к OBJPROP_COLOR (инвертированный цвет получается переключением всех битов в значении цвета на противоположные, например, для 0xFF0080 получается 0x00FF7F). При OBJPROP_BACK равном *true*, бары рисуются обычным образом, поскольку объект выводится на заднем фоне, "под" графиком (см. пример далее).

Перечисление ENUM_BORDER_TYPE содержит следующие элементы:

Идентификатор	Внешний вид
BORDER_FLAT	Плоский
BORDER_RAISED	Выпуклый
BORDER_SUNKEN	Вогнутый

Когда рамка плоская (BORDER_FLAT), она отображается линией с цветом, стилем и шириной, согласно свойствам OBJPROP_COLOR, OBJPROP_STYLE, OBJPROP_WIDTH. Выпуклый и вогнутый варианты имитируют объемные фаски по периметру в оттенках OBJPROP_BGCOLOR.

Когда цвет рамки OBJPROP_BORDER_COLOR не задан (по умолчанию, что соответствует *clrNone*), поле ввода обрамляется линией основного цвета OBJPROP_COLOR, а вокруг кнопки рисуется объемная рамка с фасками в оттенках OBJPROP_BGCOLOR.

Для проверки работы новых свойств рассмотрим скрипт *ObjectStyle.mq5*. В нем мы создадим 5 прямоугольников типа OBJ_RECTANGLE, то есть с привязкой ко времени и ценам. Они будут равномерно расположены по всей ширине окна, подсвечивая диапазон между максимальной ценой *High* и минимальной ценой *Low* в каждом из пяти временных отрезков. Для всех объектов будем настраивать и периодически менять цвет, стиль, толщину линий, а также заполнение и опцию отображения под графиком.

Вновь воспользуемся вспомогательным классом *ObjectBuilder*, производным от *ObjectSelector*. В отличие от предыдущего раздела, добавим в *ObjectBuilder* деструктор, в котором вызовем *ObjectDelete*.

```
#include <MQL5Book/ObjectMonitor.mqh>
#include <MQL5Book/AutoPtr.mqh>

class ObjectBuilder: public ObjectSelector
{
...
public:
   ~ObjectBuilder()
   {
      ObjectDelete(host, id);
   }
   ...
};
```

Это позволит возложить на этот класс не только настройку объектов, но и их автоматическое удаление по завершении работы скрипта.

В функции *OnStart* узнаем количество видимых баров, номер первого бара, а также вычисляем ширину одного прямоугольника в барах.

```
#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;
   ...
```

Зарезервируем под объекты массив умных указателей, чтобы обеспечить вызов деструкторов *ObjectBuilder*.

```
   AutoPtr<ObjectBuilder> objects[OBJECT_NUMBER];
```

Определим палитру цветов и создадим 5 объектов-прямоугольников.

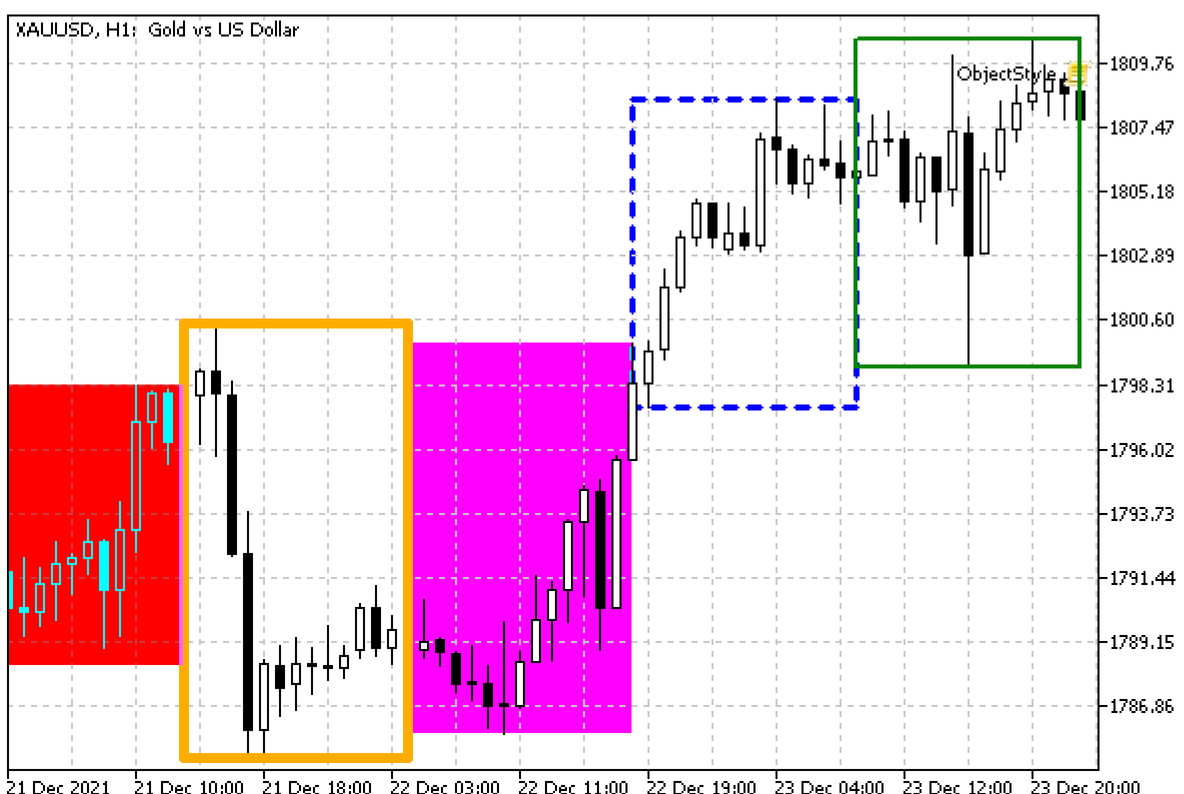
```
   color colors[OBJECT_NUMBER] = {clrRed, clrGreen, clrBlue, clrMagenta, clrOrange};

   for(int i = 0; i < OBJECT_NUMBER; ++i)
   {
      // находим индексы баров, определяющих размах цен в i-м временном поддиапазоне
      const int h = iHighest(NULL, 0, MODE_HIGH, rectsize, i * rectsize);
      const int l = iLowest(NULL, 0, MODE_LOW, rectsize, i * rectsize);
      // создаем и настриваем объект в i-м поддиапазоне
      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);
      // сохраняем в массив
      objects[i] = object;
   }
   ...
```

Здесь для каждого объекта вычисляются координаты двух точек привязки, устанавливаются начальные цвет, стиль и ширина линий.

Далее в бесконечном цикле меняем свойства объектов. При включении *ScrollLock* анимацию можно приостановить.

```
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;
   // время от времени меняем цвет/стиль/ширину/заливку/фоновое отображение
   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();
}
```

Вот как это выглядит на графике.



Прямоугольники OBJ_RECTANGLE с разными настройками отображения

Самый левый прямоугольник красного цвета имеет включенный режим заливки и находится на переднем плане — поэтому бары внутри него отображаются контрастным ярко-голубым цветом (*clrAqua*, широко известным также как *Cyan*, — это инвертированный *clrRed*). Прямоугольник

фиолетового цвета также имеет заливку, но с опцией заднего плана, поэтому бары в нем отображаются стандартным способом.

Обратите внимание, что оранжевый прямоугольник полностью перекрывает бары в начале и конце своего поддиапазона за счет большой ширины линий и отображения поверх графика.

При включенной заливке ширина линии не учитывается. При ширине границы больше 1 некоторые прерывистые стили линий не применяются.

ObjectShapesDraw

Для второго примера данного раздела вспомним о гипотетической программе рисования фигур, которую мы схематично проектировали в третьей Части, когда изучали ООП. Наш прогресс остановился на том, что в виртуальном методе рисования (а он так и назывался — draw) мы могли лишь вывести сообщение в журнал о том, что рисуем конкретную фигуру. Теперь, после знакомства с графическими объектами, у нас появилась возможность реализовать рисование.

В качестве отправной точки возьмем скрипт *Shapes5stats.mq5*. Дополненная версия будет называться *ObjectShapesDraw.mq5*.

Напомним, что помимо базового класса *Shape* у нас описано несколько классов фигур: *Rectangle*, *Ellipse*, *Triangle*, *Square*, *Circle*. Все они удачно ложатся на графические объекты типов OBJ_RECTANGLE, OBJ_ELLIPSE, OBJ_TRIANGLE. Но есть и некоторые нюансы.

Все указанные объекты имеют привязку к координатам времени и цены, в то время как наша программа рисования предполагает унифицированные оси X и Y с точечным позиционированием. В связи с этим нам потребуется особым образом настроить график для рисования и пользоваться функцией *ChartXYToTimePrice* для пересчета экранных точек во время и цену.

Кроме того, объекты OBJ_ELLIPSE и OBJ_TRIANGLE допускают произвольное вращение (в частности, малый и большой радиус эллипса могут быть повернуты), в то время как OBJ_RECTANGLE всегда имеет свои стороны ориентированными по горизонтали и вертикали. Мы для упрощения примера ограничимся стандартным положением всех фигур.

В принципе, новую реализацию следует рассматривать именно как демонстрацию графических объектов, а не программы рисования. Более правильным подходом для полноценного рисования, лишенного ограничений, которые накладывают графические объекты (поскольку они предназначены в общем-то для других целей — разметки графика), является использование графических ресурсов. Поэтому мы вернемся к переосмыслению программы рисования в главе про ресурсы.

В новом классе *Shape* избавимся от вложенной структуры *Pair* с координатами объекта: эта структура служила средством демонстрации нескольких принципов ООП, но сейчас проще вернуть изначальное описание полей *int x, y* непосредственно в класс *Shape*. Также мы добавим поле с названием объекта.

```
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);
   }
   ...
```

Поле *name* потребуется для установки свойств графического объекта, а также для его удаления с графика, что логично сделать в деструкторе.

Поскольку разные типы фигур требуют разного количества точек или характерных размеров, добавим в интерфейс *Shape* в дополнение к виртуальному методу *draw* еще один — *setup*:

```
virtual void setup(const int &parameters[]) = 0;
```

Напомним, что у нас в скрипте реализован вложенный класс *Shape::Registrator*, который занимался подсчетом количества фигур по типам. Настало время поручить ему кое-что более ответственное — работать в качестве фабрики фигур. Так называемые "фабричные" классы или методы хороши тем, что позволяют унифицированным образом создавать объекты разных классов.

Для этого добавим в *Registrator* метод для создания фигуры — с параметрами, включающими обязательные координаты первой точки, цвет и массив дополнительных параметров (каждая фигура сможет интерпретировать его по своим правилам, а в перспективе считывать из или записывать в файл).

```
virtual Shape *create(const int px, const int py, const color back,
         const int &parameters[]) = 0;
```

Метод является абстрактным виртуальным, потому что конкретные типы фигур смогут создавать только производные классы-регистраторы, описываемые в классах-наследниках *Shape*. Чтобы упростить написание производных классов-регистраторов, введем шаблонный класс *MyRegistrator* с подходящей для всех случаев реализацией метода *create*.

```
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;
   }
};
```

Здесь мы вызываем конструктор некоей заранее неизвестной фигуры T, донастраиваем её с помощью вызова *setup* и возвращаем экземпляр вызывающему коду.

Вот как это используется в классе *Rectangle*, у которого два дополнительных параметра обозначают ширину и высоту.

```
class Rectangle : public Shape
{
   static MyRegistrator<Rectangle> r;

protected:
   int dx, dy; // размеры (ширины, высота)

   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;
```

При создании фигуры её имя будет содержать не только имя класса (*typename*), но и порядковый номер экмепляра, подсчитываемый в вызове *r.increment()*.

Другие классы фигур описаны аналогично.

Теперь настало время заглянуть в метод *draw* для *Rectangle*. В нем мы переводим пару точек (x,y) и (x + dx, y + dy) в координаты время/цена с помощью *ChartXYToTimePrice* и создаем объект 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);
}
```

Разумеется, не забываем установить цвет OBJPROP_COLOR и заливку OBJPROP_FILL.

Для класса *Square* практически ничего не требуется менять: достаточно лишь установить равными *dx* и *dy*.

Для класса *Ellipse* два дополнительных параметра *dx* и *dy* определяют малый и большой радиусы, откладываемые относительно центра (x,y). Соответственно, в методе *draw* мы рассчитываем 3 точки привязки и создаем объект OBJ_ELLIPSE.

```
class Ellipse : public Shape
{
   static MyRegistrator<Ellipse> r;
protected:
   int dx, dy; // большой и малый радиусы
   ...
public:
   void draw() override
   {
      // Print("Drawing ellipse");
      int subw;
      datetime t;
      double p;

      // (x, y) центр
      // 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 — это частный случай эллипса с равными радиусами.

Наконец, треугольники у нас поддерживаются на данном этапе только равносторонние: размер стороны содержится в дополнительном поле *dx*. С их методом *draw* предлагается ознакомиться в исходном коде самостоятельно.

Новый скрипт будет, как и прежде, генерировать заданное количество случайных фигур. Их созданием занимается функция *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);
}
```

Именно здесь мы видим использование фабричного метода *create*, вызываемого у случайно выбранного объекта-регистратора под номером *n*. Если мы решим позднее добавить другие классы фигур, нам не потребуется ничего менять в логике генерации.

Все фигуры размещаются в центральной части окна и имеют размеры не больше четверти окна.

Осталось рассмотреть непосредственно вызовы функции *addRandomShape* и особую настройку графика, которую мы уже упоминали.

Для обеспечения "квадратного" представления точек на экране установим режим CHART_SCALEFIX_11. Кроме того выберем самый плотный (сжатый) масштаб по оси времени CHART_SCALE (0), потому что в нем один бар занимает 1 пиксель по горизонтали (максимальная точность). Наконец, отключим отображение самого графика, поставив CHART_SHOW в *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();
   ...
```

Для хранения фигур зарезервируем массив умных указателей и заполним его случайными фигурами.

```
#define FIGURES 21
...
void OnStart()
{
   ...
   AutoPtr<Shape> shapes[FIGURES];

   for(int i = 0; i < FIGURES; ++i)
   {
      Shape *shape = shapes[i] = addRandomShape();
      shape.draw();
   }

   ChartRedraw();
   ...
```

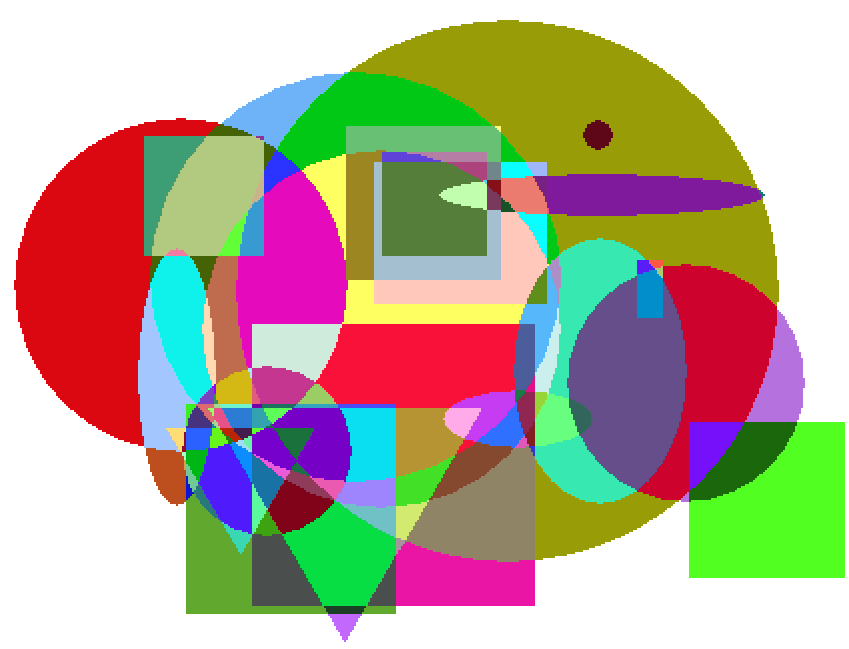
Затем запускаем бесконечный цикл, пока пользователь не остановит скрипт, в котором слегка двигаем фигуры с помощью метода *move*.

```
   while(!IsStopped())
   {
      Sleep(250);
      for(int i = 0; i < FIGURES; ++i)
      {
         shapes[i][].move(random(20) - 10, random(20) - 10);
         shapes[i][].draw();
      }
      ChartRedraw();
   }
   ...
```

В конце восстанавливаем настройки графика.

```
   // недостаточно отключить CHART_SCALEFIX_11, нужно CHART_SCALEFIX
   ChartSetInteger(0, CHART_SCALEFIX, false);
   ChartSetInteger(0, CHART_SCALE, scale);
   ChartSetInteger(0, CHART_SHOW, true);
}
```

На следующем скриншоте показано, как может выглядеть график с нарисованными фигурами.



Объекты-фигуры на графике

Особенностью отрисовки объектов является "умножение" цветов в тех местах, где они перекрываются.

Поскольку ось Y идет сверху вниз, все треугольники получаются перевернутыми, но это не критично, потому что мы в любом случае собираемся переделать программу рисования на базе ресурсов.

5.8.15 Настройки шрифта

Во всех типах объектов поддерживается возможность задать для них текст (OBJPROP_TEXT). Многие из них выводят указанный текст непосредственно на графике, для остальных он становится информативной частью всплывающей подсказки.

Когда текст отображается внутри объекта (т.е. для типов OBJ_TEXT, OBJ_LABEL, OBJ_BUTTON, OBJ_EDIT), допустимо выбрать название и размер шрифта. Для объектов остальных типов применение настроек шрифта не имеет силы: их описания всегда выводятся стандартным шрифтом графика.

Идентификатор	Описание	Тип
OBJPROP_FONTSIZE	Размер шрифта в пикселях	int
OBJPROP_FONT	Шрифт	string

Задать размер шрифта в типографских пунктах здесь нельзя.

Тестовый скрипт *ObjectFont.mq5* создает объекты с текстом и изменяет название и размер шрифта. Воспользуемся в нем классом *ObjectBuilder* из предыдущего скрипта.

В начале *OnStart* вычисляется середина окна, как в экранных координатах, так в осях время/цена — это требуется, потому что объекты разных типов, участвующие в тесте, используют разные системы координат.

```
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;
   ...
```

Перечень тестируемых типов объектов указан в массиве *types*. Для некоторых из них, в частности, OBJ_HLINE и OBJ_VLINE настройки шрифта не будут иметь эффекта, хотя текст описаний появится на экране (чтобы это гарантировать мы включаем режим 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; // курсор
   ...
```

Переменная *t* будет использоваться для последовательного переключения с одного типа на другой.

В массиве *fonts* собраны наиболее популярные стандартные шрифты 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; // курсор
...
```

Их мы будем перебирать с помощью переменной *f*.

Внутри демонстрационного цикла поручаем *ObjectBuilder* создать объект текущего типа *types[t]* в середине окна (для унификации координаты задаются в обеих системах координат, чтобы не делать различия в коде в зависимости от типа: неподдерживаемые объектом координаты просто не будут иметь эффекта).

```
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);
   ...
```

Далее настраиваем текст и шрифт (размер выбирается случайно).

```
   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);
   ...
```

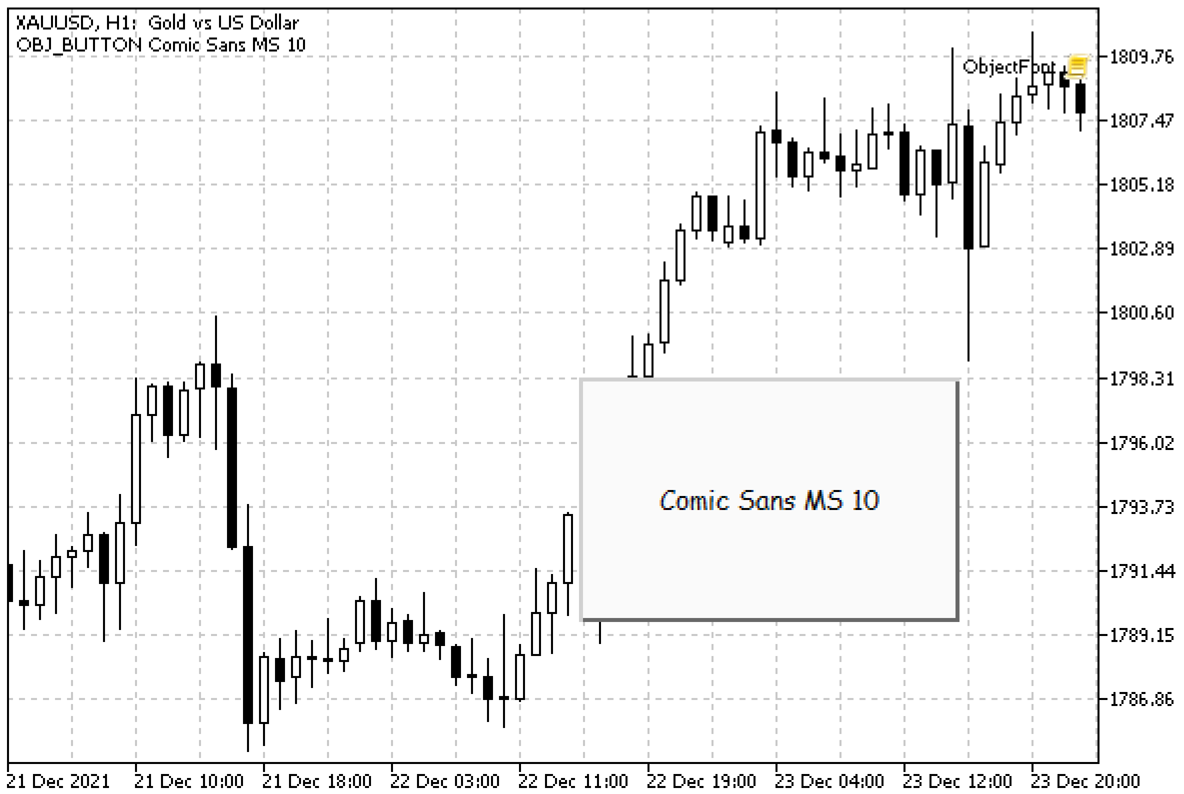
Для следующего прохода передвигаем курсоры в массивах типов объектов и названий шрифтов.

```
   t = ++t % ArraySize(types);
   f = ++f % ArraySize(fonts);
   ...
```

Наконец, обновляем график, ждем 1 секунду и удаляем объект, чтобы создать очередной.

```
        ChartRedraw();
        Sleep(1000);
        delete object;
      }
  }
```

На изображении ниже показан момент работы скрипта.

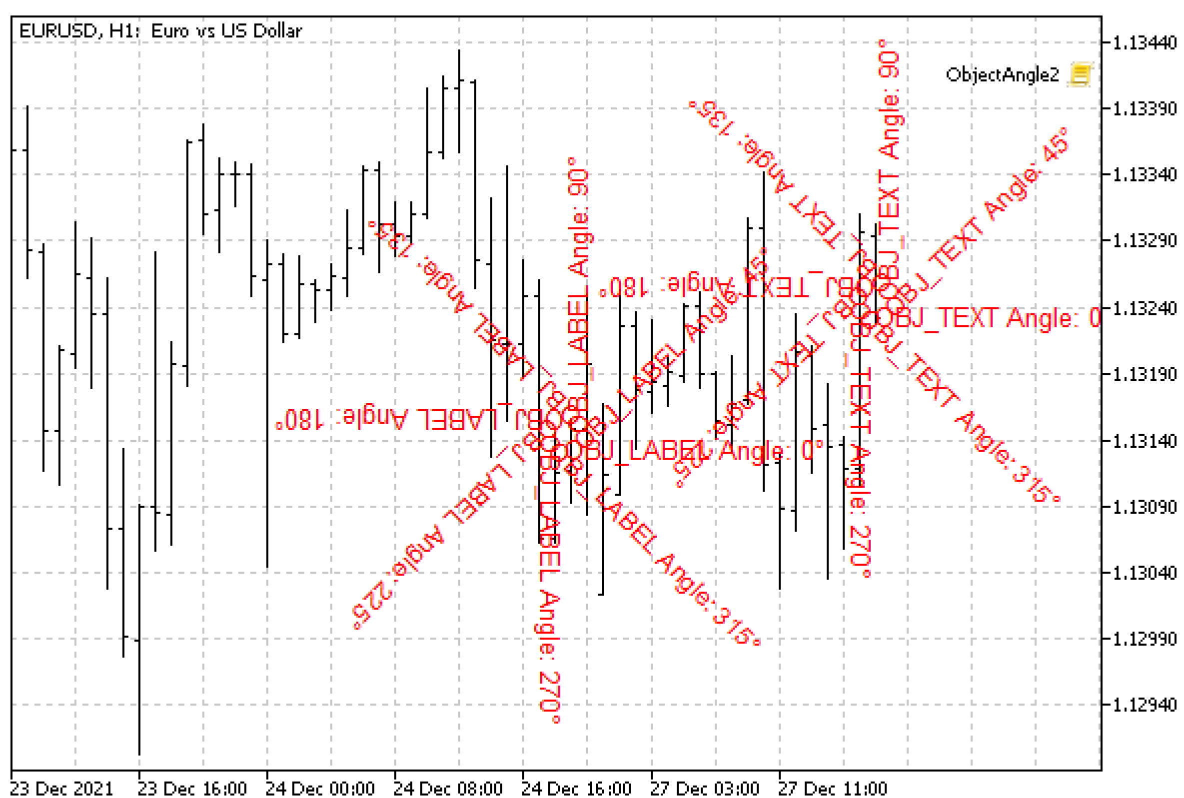


Кнопка с нестандартными настройками шрифта

5.8.16 Поворот текста на произвольный угол

Объекты текстовых типов — метка OBJ_TEXT (в котировочных координатах) и панель OBJ_LABEL (в экранных координатах) — позволяют повернуть надпись на произвольный угол. Для этой цели предусмотрено свойство OBJPROP_ANGLE типа *double*. Оно содержит угол в градусах относительно нормальной ориентации объекта. Положительные значения поворачивают объект против часовой стрелки, отрицательные — по часовой.

Однако следует иметь в виду, что углы с разницей, кратной 360 градусам, идентичны, то есть, например, +315 и -45 — это одно и то же. Поворот осуществляется вокруг точки привязки на объекте (по умолчанию, слева вверху).



Поворот объектов OBJ_LABEL и OBJ_TEXT на углы, кратные 45 градусам

Проверить влияние свойства OBJPROP_ANGLE на объект можно с помощью скрипта *ObjectAngle.mq5*. В нем создается текстовая надпись OBJ_LABEL в центре окна, после чего она начинает периодически поворачиваться на 45 градусов, пока пользователь не остановит процесс.

```
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);
}
```

В тексте выводится текущее значение угла.

5.8.17 Определение ширины и высоты объектов

Некоторые типы объектов позволяют устанавливать свои размеры в пикселях. К их числу относятся OBJ_BUTTON, OBJ_CHART, OBJ_BITMAP, OBJ_BITMAP_LABEL, OBJ_EDIT, OBJ_RECTANGLE_LABEL. Кроме того, объекты OBJ_LABEL поддерживают чтение (но не установку) размеров, поскольку надписи автоматически расширяются или сужаются под содержащийся в них текст. Попытка обратиться к свойствам у других типов объектов приведет к ошибке OBJECT_WRONG_PROPERTY (4203).

Идентификатор	Описание
OBJPROP_XSIZE	Ширина объекта по оси X в пикселях
OBJPROP_YSIZE	Высота объекта по оси Y в пикселях

Оба размера — это целые числа и потому обрабатываются функциями *ObjectGetInteger/ObjectSetInteger*.

Особая обработка размеров производится для объектов OBJ_BITMAP и OBJ_BITMAP_LABEL.

Без назначения картинки эти объекты позволяют задать произвольный размер. При этом они рисуются прозрачными (видна только рамка, если её не "спрятать" также, установив цвет *clrNone*), но получают все события, в частности, о перемещении мыши (с показом текстового описания, если оно есть, во всплывающей подсказке) и нажатиях её кнопок на объекте.

Когда картинка назначена, она по умолчанию определяет высоту и ширину объекта. Однако MQL-программа может установить меньшие размеры и выбрать фрагмент картинки для показа —

подробнее об этом в разделе Кадрирование. Если высоту или ширину попытаться установить больше размера картинки, она перестает отображаться, а размеры объекта не меняются.

В качестве примера разработаем усовершенствованную версию скрипта *ObjectAnchorLabel.mq5* из раздела Определение точки привязки на объекте. Там мы перемещали надпись по окну и "разворачивали" её по достижению любой из границ окна, однако делали это только с учетом точки привязки. В связи с этим, в зависимости от расположения точки привязки на объекте, могла складываться ситуация, когда надпись почти полностью "выезжала" за пределы окна. Например, если точка привязки находилась на правой стороне объекта, то при движении влево практически весь текст уходил за левую границу окна, прежде чем точка привязки упиралась в край.

В новом скрипте *ObjectSizeLabel.mq5* мы будем учитывать размер объекта и менять направление перемещения, как только он касается края окна любой своей стороной.

Для правильной реализации такого режима следует учитывать, что каждый угол окна, используемый в качестве центра отсчета координат до точки привязки на объекте, обуславливает характерное направление обоих осей X и Y. Например, если пользователь выбрал во входной переменной ENUM_BASE_CORNER Corner левый верхний угол, то X увеличивается слева направо, а Y — сверху вниз. Если же центром считается правый нижний угол, то X увеличивается справа налево от него, а Y — снизу вверх.

Различное взаимное сочетание угла привязки в окне и точки привязки на объекте требует разной корректировки расстояний между краями объекта и границами окна. В частности, когда выбран один из правых углов и одна из точек привязки на правой стороне объекта, то поправка у правой границы окна не требуется, а у противоположной — левой — мы должны учитывать ширину объекта (чтобы его габариты не вышли налево за пределы окна).

Это правило о внесении поправки на размер объекта можно обобщить:

- на границе окна, прилегающей к углу привязки, поправка нужна при нахождении точки привязки на дальней стороне объекта по отношению к этому углу;
- на границе окна, противоположной углу привязки, поправка нужна при нахождении точки привязки на ближней стороне объекта по отношению к этому углу.

Иными словами, если в названии угла (в элементе ENUM_BASE_CORNER) и точки привязки (в элементе ENUM_ANCHOR_POINT) встречается общее слово (например, RIGHT), поправка нужна на дальней стороне окна (то есть удаленной от выбранного угла). Если же в сочетании сторон ENUM_BASE_CORNER и ENUM_ANCHOR_POINT обнаруживаются противоположные направления (например, LEFT и RIGHT), поправка нужна у ближайшей стороны окна. Данные правила работают одинаково для горизонтальной и вертикальной оси.

Дополнительно следует учитывать, что точка привязки может быть в середине какой-либо стороны объекта. Тогда в перпендикулярном направлении требуется отступ от границ окна в половину размера объекта.

Особый случай представляет точка привязки в центре объекта. Для неё следует всегда иметь запас расстояния в любом направлении, равный половине размера объекта.

Описанная логика реализована в специальной функции *GetMargins*. Она принимает на вход выбранный угол и точку привязки, а также размеры объекта (*dx* и *dy*). Возвращает функция структуру с 4-мя полями, содержащими размеры дополнительных отступов, которые следует отложить от точки привязки в направлении ближних и дальних границ окна, чтобы объект не

вышел за пределы видимости. Отступы резервируют расстояние согласно габаритам и относительному расположению самого объекта.

```
struct Margins
{
   int nearX; // добавка по X между точкой объекта и смежной с углом границей окна
   int nearY; // добавка по Y между точкой объекта и смежной с углом границей окна
   int farX;  // добавка по X между точкой объекта и противоположной углу границе окн
   int farY;  // добавка по Y между точкой объекта и противоположной углу границе окн
};

Margins GetMargins(const ENUM_BASE_CORNER corner, const ENUM_ANCHOR_POINT anchor,
   int dx, int dy)
{
   Margins margins = {}; // по умолчанию нулевые поправки
   ...
   return margins;
}
```

Для унификации алгоритма введены следующие макроопределения направлений (сторон):

```
#define LEFT 0x1
#define LOWER 0x2
#define RIGHT 0x4
#define UPPER 0x8
#define CENTER 0x16
```

С их помощью определены битовые маски (комбинации), описывающие элементы перечислений ENUM_BASE_CORNER и ENUM_ANCHOR_POINT.

```
const int corner_flags[] = // флаги для элементов ENUM_BASE_CORNER
{
   LEFT | UPPER,
   LEFT | LOWER,
   RIGHT | LOWER,
   RIGHT | UPPER
};

const int anchor_flags[] = // флаги для элементов ENUM_ANCHOR_POINT
{
   LEFT | UPPER,
   LEFT,
   LEFT | LOWER,
   LOWER,
   RIGHT | LOWER,
   RIGHT,
   RIGHT | UPPER,
   UPPER,
   CENTER
};
```

Каждый из массивов *corner_flags* и *anchor_flags* содержит ровно столько элементов, сколько имеется в соответствующем перечислении.

Далее идет основной фрагмент функции. Прежде всего "разбираемся" с самым простым вариантом: центральной точкой привязки.

```
if(anchor == ANCHOR_CENTER)
{
   margins.nearX = margins.farX = dx / 2;
   margins.nearY = margins.farY = dy / 2;
}
else
{
   ...
}
```

Для анализа остальных ситуаций воспользуемся битовыми масками из вышеприведенных массивов путем прямой адресации по полученным значениями *corner* и *anchor*.

```
const int mask = corner_flags[corner] & anchor_flags[anchor];
...
```

Если угол и точка привязки на одной стороне по горизонтали, сработает следующее условие, и будет сделана поправка на ширину объекта у дальней границы окна.

```
if((mask & (LEFT | RIGHT)) != 0)
{
   margins.farX = dx;
}
...
```

Если они не на одной стороне, то могут быть на противоположных, а может быть и случай, что точка привязки в середине горизонтальной стороны (сверху или снизу). Проверка на точку привязки в середине делается с помощью выражения *(anchor_flags[anchor] & (LEFT | RIGHT)) == 0* — тогда поправка равна половине ширины объекта.

```
else
{
   if((anchor_flags[anchor] & (LEFT | RIGHT)) == 0)
   {
      margins.nearX = dx / 2;
      margins.farX = dx / 2;
   }
   else
   {
      margins.nearX = dx;
   }
}
...
```

Иначе, при противоположной ориентации угла и точки привязки делаем поправку в ширину объекта у ближней границы окна.

Аналогичные проверки выполняются для оси 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;
         }
      }
```

Теперь функция *GetMargins* готова, и можно приступать к основному коду скрипта в функции *OnStart*. Как и ранее, мы определяем размер окна, рассчитываем начальные координаты в центре, создаем объект OBJ_LABEL и выделяем его для наглядности.

```
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);
   ...
```

Для анимации в бесконечном цикле предусмотрены переменные *pass* (счетчик итераций) и *anchor* (точка привязки, которая будет периодически выбираться случайным образом).

```
   int pass = 0;
   ENUM_ANCHOR_POINT anchor = 0;
   ...
```

Но по сравнению с *ObjectAnchorLabel.mq5* сделан и ряд изменений.

Мы не станем генерировать случайные перемещения объекта. Вместо этого зададим постоянную скорость 5 пикселей по диагонали.

```
   int px = 5, py = 5;
```

Для хранения размера надписи зарезервируем две новых переменных.

```
int dx = 0, dy = 0;
```

Результат подсчета дополнительных отступов будем сохранять в переменной *m* типа *Margins*.

```
Margins m = {};
```

Далее следует непосредственно цикл перемещения и модификации объекта. В нем на каждой 75-ой итерации (одна итерация 100 мсек, см. далее) мы случайно выбираем новую точку привязки, формируем из неё новый текст (содержимое объекта) и ждем, когда изменения применятся к объекту (вызываем *ChartRedraw*). Последнее необходимо, потому что размер надписи автоматически подгоняется под содержимое, а нам важен новый размер, чтобы корректно посчитать отступы в вызове *GetMargins*.

Размеры мы получаем с помощью вызовов *ObjectGetInteger* со свойствами OBJPROP_XSIZE и OBJPROP_YSIZE.

```
for( ;!IsStopped(); ++pass)
{
   if(pass % 75 == 0)
   {
      // ENUM_ANCHOR_POINT состоит из 9 элементов: случайно выберем один
      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);
   }
   ...
```

После того как точка привязки и все расстояния известны, выполняем перемещение объекта. Если он "упрется" в стенку, меняем направление движения на противоположное (*px* на *-px* или *py* на *-py*, в зависимости от стороны).

```
// отскок от границ окна, объект полностью видимый
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;
}

// рассчитываем новую позицию надписи
x += px;
y += py;
...
```

Остается обновить состояние самого объекта: вывести текущие координаты в надпись, и назначить их свойствам OBJPROP_XDISTANCE и 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);
...
```

После изменения объекта вызываем *ChartRedraw* и ждем 100 мсек, чтобы обеспечить достаточно плавную анимацию.

```
ChartRedraw();
Sleep(100);
...
```

В концовке цикла мы снова проверяем размер окна, так как пользователь может его изменить в процессе работы скрипта, а также повторяем запрос размеров.

```
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);
```

```
}
```

Некоторые прочие нововведения скрипта *ObjectSizeLabel.mq5* мы опустили ради краткости — желающие могут обратиться к коду. В частности, были использованы отличительные цвета для надписи: каждый конкретный цвет соответствует собственной точке привязки, что делает более заметными моменты её переключения. Кроме того, вы можете нажать *Delete* во время работы скрипта: это удалит выделенный объект с графика, и скрипт автоматически завершится.

5.8.18 Видимость объектов в разрезе таймфреймов

Пользователи MetaTrader 5 знают, что в диалоге свойств объекта присутствует закладка Отображение, на которой можно с помощью переключателей выбрать, на каких таймфреймах объект будет отображаться, а на каких будет скрыт. В частности, можно временно скрыть объект полностью — на всех таймфреймах.

В MQL5 имеется аналогичное программное свойство — OBJPROP_TIMEFRAMES — оно управляет видимостью объекта на таймфрейма. Значением данного свойства может быть любая комбинация специальных целочисленных флагов: каждый флаг (константа) содержит бит, соответствующий одному таймфрейму (см. таблицу). Для установки/получения свойства OBJPROP_TIMEFRAMES следует использовать функции *ObjectSetInteger/ObjectGetInteger*.

Константа	Значение	Видимость на таймфреймах
OBJ_NO_PERIODS	0	Объект невидим на всех таймфреймах
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
OBJ_PERIOD_H6	0x00008000	H6

Константа	Значение	Видимость на таймфреймах
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	Все таймфреймы

Флаги можно комбинировать с помощью оператора побитового ИЛИ ("|"), например, суперпозиция флагов OBJ_PERIOD_M15 | OBJ_PERIOD_H4 означает, что объект будет видимым на 15-минутном и 4-часовом таймфреймах.

Обратите внимание, что каждый флаг можно получить сдвигом 1 влево на количество бит, равное номеру константы в таблице. Это облегчает динамическую генерацию флагов, когда алгоритм оперирует множеством таймфреймов, а не одним конкретным.

Мы воспользуемся такой возможностью в тестовом скрипте *ObjectTimeframes.mq5*. Его задача — создать на графике множество больших текстовых надписей с названиями таймфреймов, причем каждое название должно выводиться только на соответствующем таймфрейме. Например, крупная надпись "D1" будет видна только на дневном графике, а при переключении на H4 — увидим надпись "H4".

Для получения краткого названия таймфрейма, без префикса "PERIOD_", реализована простая вспомогательная функция.

```
string GetPeriodName(const int tf)
{
   const static int PERIOD_ = StringLen("PERIOD_");
   return StringSubstr(EnumToString((ENUM_TIMEFRAMES)tf), PERIOD_);
}
```

Получение списка всех таймфреймов из перечисления ENUM_TIMEFRAMES поручим функции EnumToArray, которую мы представили в разделе о конвертации Перечислений.

```
#include "ObjectPrefix.mqh"
#include <MQL5Book/EnumToArray.mqh>

void OnStart()
{
   ENUM_TIMEFRAMES tf = 0;
   int values[];
   const int n = EnumToArray(tf, values, 0, USHORT_MAX);
   ...
```

Все надписи будут выводиться в центре окна на момент запуска скрипта. Изменение размера окна после завершения работы скрипта приведет к тому, что созданные надписи перестанут быть отцентрированными. Это следствие того, что MQL5 поддерживает привязку только к углам окна, но не к центру. Если хочется автоматически поддерживать позицию объектов, следует

реализовать аналогичный алгоритм в индикаторе и реагировать на события изменения размера окна. В качестве альтернативы мы могли бы выводить надписи в каком-либо углу, например, правом нижнем.

```
const int centerX = (int)ChartGetInteger(0, CHART_WIDTH_IN_PIXELS) / 2;
const int centerY = (int)ChartGetInteger(0, CHART_HEIGHT_IN_PIXELS) / 2;
...
```

В цикле по таймфреймам создаем для каждого из них объект OBJ_LABEL, размещаем его в середине окна, с привязкой по центру объекта.

```
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);
   ...
```

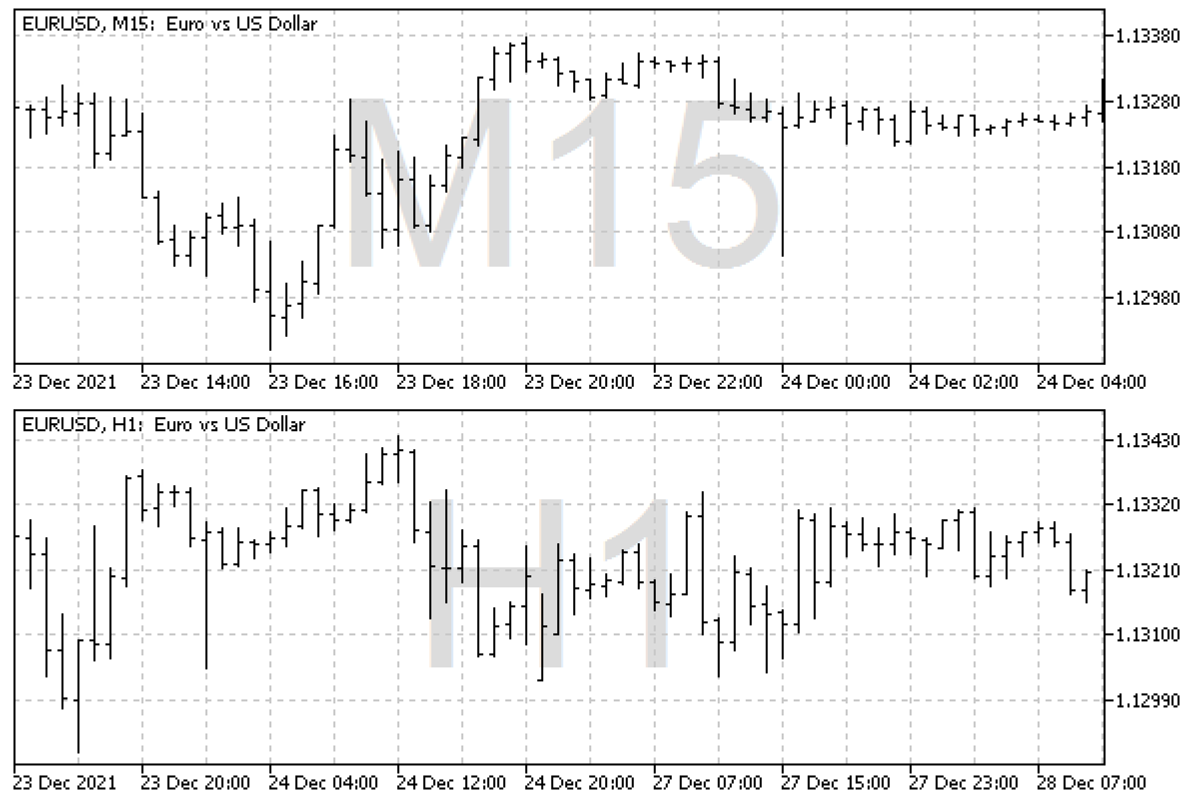
Далее устанавливаем текст (название таймфрейма), большой размер шрифта, серый цвет и свойство отображения на заднем плане.

```
   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);
   ...
```

Наконец, генерируем правильный флаг видимости для *i*-го таймфрейма и записываем его в свойство OBJPROP_TIMEFRAMES.

```
   const int flag = 1 << (i - 1);
   ObjectSetInteger(0, name, OBJPROP_TIMEFRAMES, flag);
}
```

Посмотрите, что получилось на одном и то же графике при переключении таймфреймов.



Надписи с названиями таймфреймов

Если открыть диалог *Список объектов* и включить в списке *Все* объекты, легко убедиться в наличии сгенерированных надписей для всех таймфреймов и проверить их флаги видимости.

Для удаления объектов можно запустить скрипт *ObjectCleanup1.mq5*.

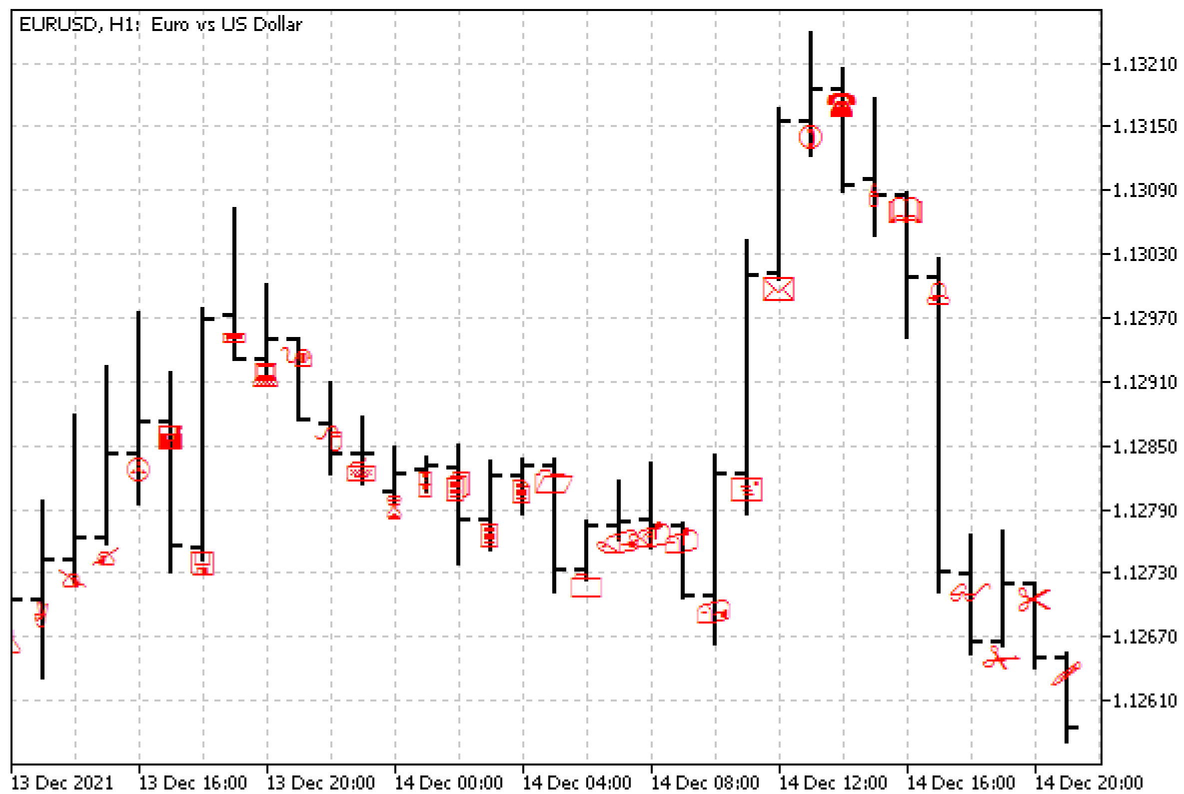
5.8.19 Назначение кода символа для метки

Как было сказано в обзоре Объектов с привязкой ко времени и цене, метка OBJ_ARROW позволяет вывести на график произвольный символ шрифта Wingdings (полный перечень доступных символов можно найти в документации MQL5). Сам код символа для объекта определяется целочисленным свойством OBJPROP_ARROWCODE.

Продемонстрировать все символы шрифта позволяет скрипт *ObjectWingdings.mq5*. В нем мы создаем в цикле метки с различными символами, размещая их по одной на баре.

```
#include "ObjectPrefix.mqh"

void OnStart()
{
   for(int i = 33; i < 256; ++i) // коды символов
   {
      const int b = i - 33; // номер бара
      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);
}
```

Как это выглядит на графике, показано на следующем скриншоте.



Символы Wingdings в метках OBJ_ARROW

5.8.20 Свойства лучей для объектов с прямыми линиями

Среди графических объектов есть несколько типов, в которых линии между точками привязки могут отображаться либо в виде отрезков (то есть строго между парой точек) либо как бесконечные прямые линии, продолжающиеся в ту или иную сторону по всей области видимости окна. К подобным объектам относятся:

- трендовая линия,

- трендовая линия по углу,
- все типы каналов (равноудаленный, стандартных отклонений, регрессии, вилы Эндрюса),
- линия Ганна,
- линии Фибоначчи,
- канал Фибоначчи,
- расширение Фибоначчи.

Для них можно раздельно включить продолжение линии влево или вправо с помощью свойств OBJPROP_RAY_LEFT и OBJPROP_RAY_RIGHT, соответственно. Кроме того, для вертикальной линии можно указать, следует ли её рисовать по всем подокнам графика или только по текущему (где находится точка привязки): за это отвечает свойство OBJPROP_RAY. Все свойства являются логическими, то есть могут быть включены (*true*) или выключены (*false*).

Идентификатор	Описание
OBJPROP_RAY_LEFT	Луч продолжается влево
OBJPROP_RAY_RIGHT	Луч продолжается вправо
OBJPROP_RAY	Вертикальная линия продолжается на все окна графика

Проверить работу лучей можно с помощью скрипта *ObjectRays.mq5*. Он создает 3 канала стандартных отклонений с различными настройками лучей.

Один конкретный объект создает и настраивает вспомогательная функция *SetupChannel*. Через её параметры задается длина канала в барах и ширина канала (девиация), а также опции показа лучей влево и вправо, и цвет.

```
#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: 0-я точка привязки должна иметь более раннее время, чем 1-я,
   // иначе канал получится вырожденным
   ObjectCreate(0, name, OBJ_STDDEVCHANNEL, 0, iTime(NULL, 0, length), 0);
   ObjectSetInteger(0, name, OBJPROP_TIME, 1, iTime(NULL, 0, 0));
   // девиация
   ObjectSetDouble(0, name, OBJPROP_DEVIATION, deviation);
   // цвет и описание
   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" : ""));
   // свойства лучей
   ObjectSetInteger(0, name, OBJPROP_RAY_RIGHT, right);
   ObjectSetInteger(0, name, OBJPROP_RAY_LEFT, left);
   // подсвечиваем объекты путем выделения
   // (кроме того, так их проще удалить пользователю)
   ObjectSetInteger(0, name, OBJPROP_SELECTABLE, true);
   ObjectSetInteger(0, name, OBJPROP_SELECTED, true);
}
```

В функции *OnStart* вызываем *SetupChannel* для 3-х разных каналов.

```
void OnStart()
{
   SetupChannel(24, 1.0, true);
   SetupChannel(48, 2.0, false, true, clrBlue);
   SetupChannel(36, 3.0, false, false, clrGreen);
}
```

В результате получим график примерно следующего вида.

Каналы с различными настройками свойств OBJPROP_RAY_LEFT и OBJPROP_RAY_RIGHT

При включении лучей появляется возможность запросить у объекта экстраполяцию значений времени и цены с помощью функций, которые мы опишем в разделе Получение времени или цены в заданных точках линий.

5.8.21 Управление нажатым состоянием объекта

Для объектов типа кнопки (OBJ_BUTTON) и панели с картинкой (OBJ_BITMAP_LABEL) терминал поддерживает специальное свойство, визуально переключающее объект из нормального (отжатого) состояния в нажатое и обратно. Для этого зарезервирована константа OBJPROP_STATE. Свойство имеет логический тип: при значении *true* объект считается нажатым, а при *false* — отжатым (по умолчанию).

Если для OBJ_BUTTON эффект объемной рамки рисует сам терминал, то для OBJ_BITMAP_LABEL программист должен указать два изображения (как файлы или ресурсы), которые обеспечат подходящее внешнее представление. Поскольку с технической точки зрения данное свойство является просто переключателем, его легко использовать не только для эффекта "нажатия" и "отжатия", но и других целей. Например, с помощью соответствующих изображений можно реализовать флаг (опцию).

Об использовании изображений в объектах будет рассказано в следующем разделе.

В принципе, состояние объектов обычно меняется в интерактивных MQL-программах, которые реагируют на действия пользователя, в частности, нажатия мыши. Мы обратимся к этой возможности в главе про события.

Сейчас протестируем свойство на простых кнопках, в статичном режиме. Скрипт *ObjectButtons.mq5* создает на графике две кнопки: одну в нажатом состоянии, другую — в отжатом.

Настройка отдельной кнопки выделена в функцию *SetupButton* с параметрами, задающими название и текст кнопки, а также её координаты, размер и состояние.

```
#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);
   // позиция и размер
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);
   ObjectSetInteger(0, name, OBJPROP_XSIZE, dx);
   ObjectSetInteger(0, name, OBJPROP_YSIZE, dy);
   // надпись на кнопке
   ObjectSetString(0, name, OBJPROP_TEXT, button);

   // нажата (true) / отжата (false)
   ObjectSetInteger(0, name, OBJPROP_STATE, state);
}
```

Затем в *OnStart* дважды вызываем эту функцию.

```
void OnStart()
{
   SetupButton("Pressed", 100, 100, 100, 20, true);
   SetupButton("Normal", 100, 150, 100, 20);
}
```

Получившиеся кнопки могут выглядеть следующим образом.

Нажатая и отжатая кнопки OBJ_BUTTON

Что интересно — вы можете щелкнуть по любой из кнопок мышью, и кнопка сменит свое состояние. Правда, мы еще не научились перехватывать уведомление об этом.

Важно отметить, что данное автоматическое переключение состояния выполняется, только если в свойствах объектов взведена опция *Отключить выделение*, но это условие по умолчанию выполняется для всех объектов, созданных программно. Напомним, что при необходимости выделение можно и разрешить — для этого надо явным образом установить свойство OBJPROP_SELECTABLE в *true*, чем мы пользовались в некоторых предыдущих примерах.

Для удаления ставших ненужными кнопок воспользуйтесь скриптом *ObjectCleanup1.mq5*.

5.8.22 Настройка изображений в объектах-картинках

Объекты типа OBJ_BITMAP_LABEL (панель с картинкой, позиционируемая в экранных координатах) позволяют отображать растровые картинки. Под растровыми картинками понимается графический формат BMP: хотя в принципе существует множество других растровых форматов (например, PNG или GIF), они в MQL5 на данный момент не поддерживаются, как и векторные.

Указать изображение для объекта позволяет строковое свойство OBJPROP_BMPFILE. Оно должно содержать имя BMP-файла или ресурса.

Поскольку данный объект поддерживает возможность двухпозиционного переключения состояния (см. OBJPROP_STATE), для него следует использовать параметр-модификатор: под индексом 0 задается картинка для состояния "включено"/"нажато", а под индексом 1 — для состояния "выключено"/"отжато". Если задать только одну картинку (без модификатора, что эквивалентно

0), она будет использоваться для обоих состояний. Напомним, что по умолчанию состояние объекта — "выключено".

Размер объекта становится равным размеру картинки, но его можно изменить, указав в свойствах OBJPROP_XSIZE и OBJPROP_YSIZE меньшие значения: при этом выводится лишь часть изображения (подробности см. в следующем разделе о кадрировании).

Длина строки OBJPROP_BMPFILE не должна превышать 63 символа. Она может содержать не только имя файла, но и путь к нему. Если строка начинается с символа-разделителя пути (прямая косая черта '/' или двойная обратная '\\'), то файл ищется относительно *каталога_данных_терминала/MQL5/*. В противном случае файл ищется относительно папки, в которой находится MQL-программа.

Например, строка "\\Images\\euro.bmp" (или "/Images/euro.bmp") ссылается на файл в каталоге *MQL5/Images/euro.bmp*. В стандартной поставке терминала в папке MQL5 действительно существует папка *Images*, а в ней — пара тестовых файлов *euro.bmp* и *dollar.bmp*, так что путь является рабочим. Если же задать строку "Images\\euro.bmp" или ("Images/euro.bmp"), то это будет предполагать, например, для скрипта, запущенного из *MQL5/Scripts/MQL5Book/*, что папка *Images* с файлом *euro.bmp* должна находиться непосредственно там, то есть общий путь получится *MQL5/Scripts/MQL5Book/Images/euro.bmp*. Такого файла в нашей книге нет, а это привело бы к ошибке загрузки изображения. Однако такая схема расположения графических файлов рядом с программой имеет и свои плюсы: проще контролировать сборку, и не возникает путаницы с перемешанными картинками разных программ.

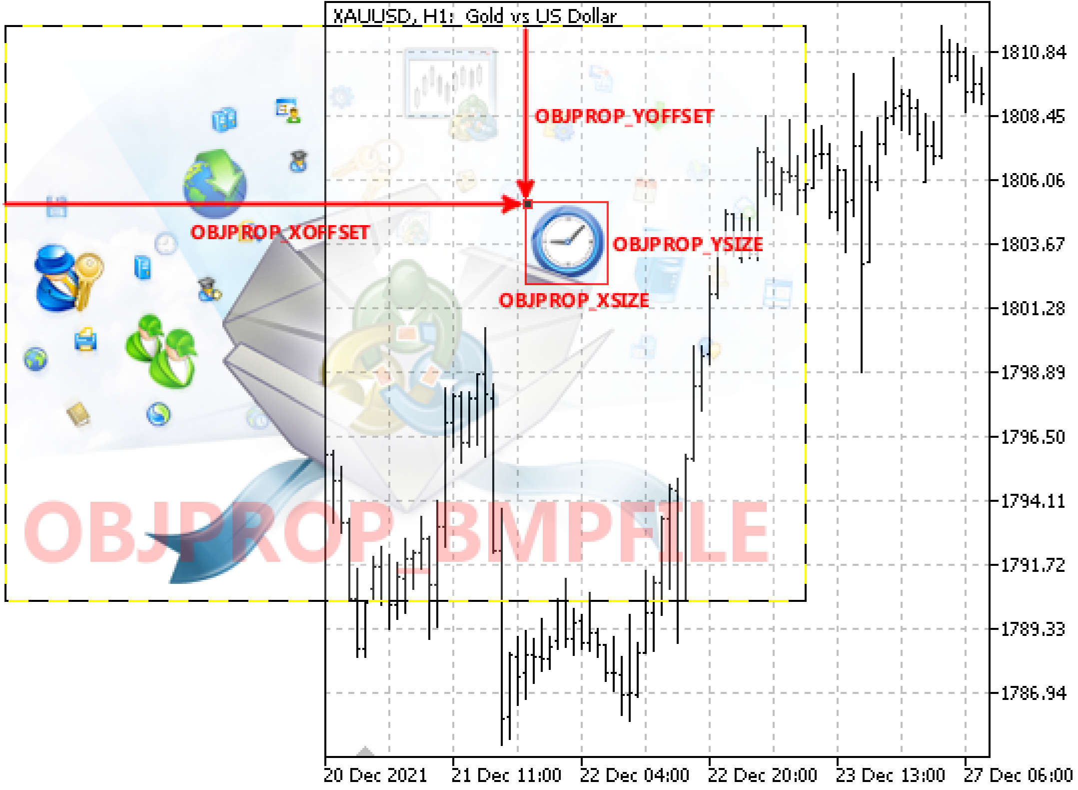
Скрипт *ObjectBitmap.mq5* создает на графике панель с картинкой и назначает ей два изображения: "\\Images\\dollar.bmp" и "\\Images\\euro.bmp".

```
#include "ObjectPrefix.mqh"

void SetupBitmap(const string button, const int x, const int y,
   const string imageOn, const string imageOff = NULL)
{
   // создаем панель
   const string name = ObjNamePrefix + "Bitmap";
   ObjectCreate(0, name, OBJ_BITMAP_LABEL, 0, 0, 0);
   // настраиваем позицию
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);
   // подключаем изображения
   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");
}
```

Как и в случае результата работы скрипта из предыдущего раздела, здесь вы тоже можете пощелкать мышью по объекту-картинке и убедиться, что она переключается с изображения доллара на евро и обратно.

5.8.23 Кадрирование (вывод части) изображения

Для графических объектов с рисунками (OBJ_BITMAP_LABEL и OBJ_BITMAP) MQL5 позволяет настроить показ лишь части изображения, заданного свойством OBJPROP_BMPFILE. Для этого необходимо установить размер объекта (OBJPROP_XSIZE и OBJPROP_YSIZE) меньше размера изображения, а координаты верхнего левого угла видимого прямоугольного фрагмента задать с помощью целочисленных свойств OBJPROP_XOFFSET и OBJPROP_YOFFSET. Данные два свойства задают, соответственно, отступ по X и по Y в пикселях от левой и верхней границы исходного изображения.



Вывод части изображения в объект

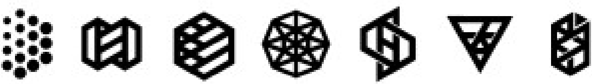
Обычно подобный прием с использованием части большого изображения применяется для иконок инструментальных панелей (наборов кнопок, меню и т.п.): единый файл со всеми иконками обеспечивает более эффективное ресурсопотребление, чем множество мелких файлов с отдельными иконками.

Тестовый скрипт *ObjectBitmapOffset.mq5* создает несколько панелей с картинками (OBJ_BITMAP_LABEL), причем для всех них указан один и тот же графический файл в свойстве OBJPROP_BMPFILE. Однако за счет свойств OBJPROP_XOFFSET и OBJPROP_YOFFSET все объекты отображают разные участки изображения.

```
void SetupBitmap(const int i, const int x, const int y, const int size,
   const string imageOn, const string imageOff = NULL)
{
   // создаем объект
   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);
   // позиция и размер
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);
   ObjectSetInteger(0, name, OBJPROP_XSIZE, size);
   ObjectSetInteger(0, name, OBJPROP_YSIZE, size);
   // смещение в исходной картинке, по которому читается i-й фрагмент
   ObjectSetInteger(0, name, OBJPROP_XOFFSET, i * size);
   ObjectSetInteger(0, name, OBJPROP_YOFFSET, 0);
   // общее изображение (файл)
   ObjectSetString(0, name, OBJPROP_BMPFILE, imageOn);
}

void OnStart()
{
   const int icon = 46; // размер одной иконки
   for(int i = 0; i < 7; ++i) // цикл по иконкам в файле
   {
      SetupBitmap(i, 10, 10 + i * icon, icon,
         "\\Files\\MQL5Book\\icons-322-46.bmp");
   }
}
```

В исходном изображении находится несколько мелких значков размером 46 на 46 пикселей. Скрипт "вырезает" их по одному и располагает по вертикали у правого края окна.

Ниже показан общий файл (*/Files/MQL5Book/icons-322-46.bmp*), и что получилось на графике.



BMP-файл с иконками

Объекты-кнопки с иконками на графике

5.8.24 Свойства поля ввода: выравнивание текста и "только чтение"

Для объектов типа OBJ_EDIT (поле ввода) MQL-программа может установить два специфических свойства, определяемых с помощью функций *ObjectSetInteger/ObjectGetInteger*.

Идентификатор	Описание	Тип значений
OBJPROP_ALIGN	Горизонтальное выравнивание текста	ENUM_ALIGN_MODE
OBJPROP_READONLY	Возможность редактирования текста	bool

Перечисление ENUM_ALIGN_MODE содержит следующие элементы.

Идентификатор	Описание
ALIGN_LEFT	Выравнивание по левой границе
ALIGN_CENTER	Выравнивание по центру
ALIGN_RIGHT	Выравнивание по правой границе

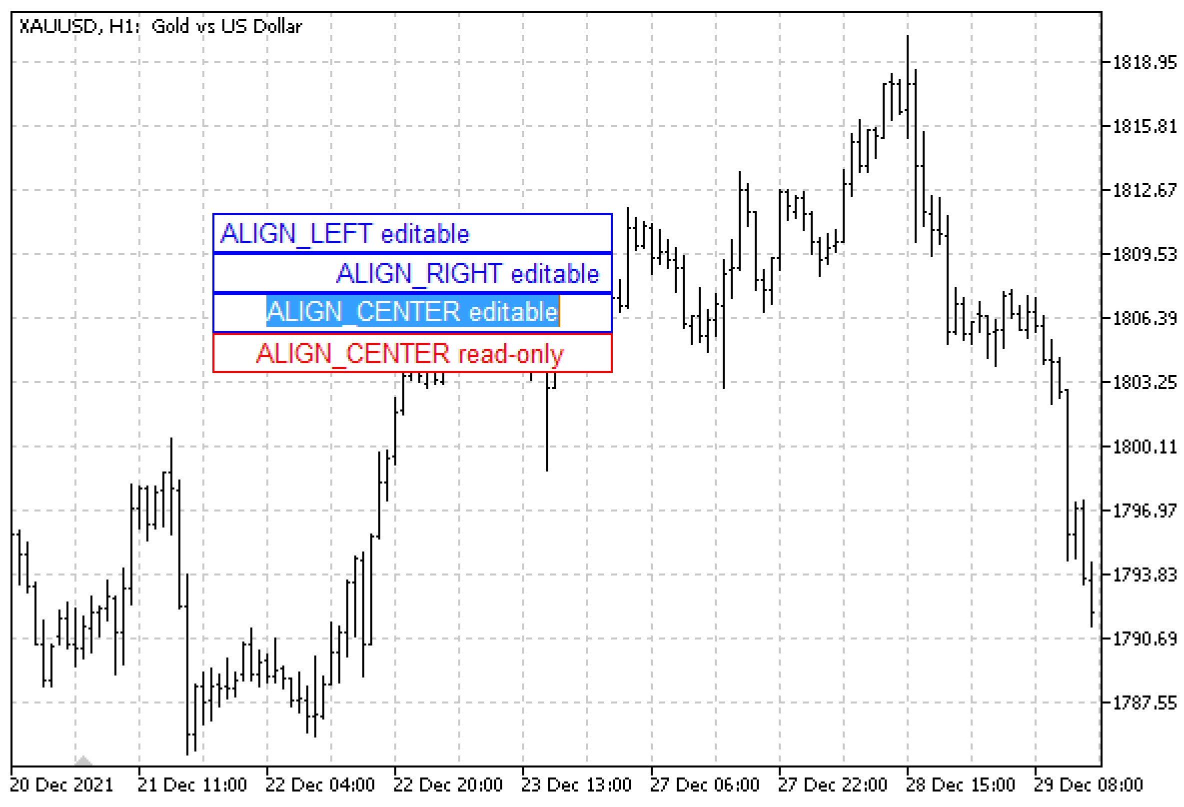
Обратите внимание, что в отличие от объектов OBJ_TEXT и OBJ_LABEL, поле ввода не подстраивает автоматически свой размер под введенный текст, поэтому для длинных строк может потребоваться явно задать свойство OBJPROP_XSIZE.

В режиме редактирования внутри поля ввода работает горизонтальная прокрутка текста.

Скрипт *ObjectEdit.mq5* создает 4 объекта OBJ_EDIT, из них 3 — редактируемые, с разными способами выравнивания текста, а 4-ый — в режиме "только чтения".

```
#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)
{
   // создаем объект с описанием свойств
   const string props = EnumToString(alignment)
      + (readonly ? " read-only" : " editable");
   const string name = ObjNamePrefix + "Edit" + props;
   ObjectCreate(0, name, OBJ_EDIT, 0, 0, 0);
   // позиция и размер
   ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x);
   ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y);
   ObjectSetInteger(0, name, OBJPROP_XSIZE, dx);
   ObjectSetInteger(0, name, OBJPROP_YSIZE, dy);
   // специфические свойства полей ввода
   ObjectSetInteger(0, name, OBJPROP_ALIGN, alignment);
   ObjectSetInteger(0, name, OBJPROP_READONLY, readonly);
   // цвета (разные в зависимости от редактируемости)
   ObjectSetInteger(0, name, OBJPROP_BGCOLOR, clrWhite);
   ObjectSetInteger(0, name, OBJPROP_COLOR, readonly ? clrRed : clrBlue);
   // содержимое
   ObjectSetString(0, name, OBJPROP_TEXT, props);
   // подсказка для редактируемых
   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);
}
```

Результат работы скрипта показан на изображении ниже.



Поля ввода в разных режимах

Вы можете щелкнуть мышью на любом редактируемом поле и изменить его содержимое.

5.8.25 Ширина канала стандартного отклонения

Канал стандартного отклонения OBJ_STDDEVCHANNEL имеет особое свойство, определяющее ширину канала как множитель для стандартного (среднеквадратического) отклонения. Свойство называется OBJPROP_DEVIATION и может принимать положительные вещественные значения (*double*). По умолчанию оно равно 1.0.

Мы уже видели пример его использования в скрипте *ObjectRays.mq5* в разделе Свойства лучей для объектов с прямыми линиями.

5.8.26 Настройка уровней в объектах с их поддержкой

Некоторые графические объекты строятся с использованием нескольких уровней (повторяющихся линий). К их числу относятся:

- вилы Эндрюса OBJ_PITCHFORK,
- инструменты Фибоначчи:
  - уровни OBJ_FIBO,
  - временные зоны OBJ_FIBOTIMES,
  - веер OBJ_FIBOFAN,
  - дуги OBJ_FIBOARC,
  - канал OBJ_FIBOCHANNEL,

- расширение OBJ_EXPANSION.

Для них MQL5 позволяет задать свойства уровней — их количество, цвет, значения, надписи.

Идентификатор	Описание	Тип
OBJPROP_LEVELS	Количество уровней	int
OBJPROP_LEVELCOLOR	Цвет линии-уровня	color
OBJPROP_LEVELSTYLE	Стиль линии-уровня	ENUM_LINE_STYLE
OBJPROP_LEVELWIDTH	Толщина линии-уровня	int
OBJPROP_LEVELTEXT	Описание уровня	string
OBJPROP_LEVELVALUE	Значение уровня	double

Для всех свойств кроме OBJPROP_LEVELS при вызове *ObjectGet-* и *ObjectSet*-функций требуется указание дополнительного параметра-модификатора с номером конкретного уровня.

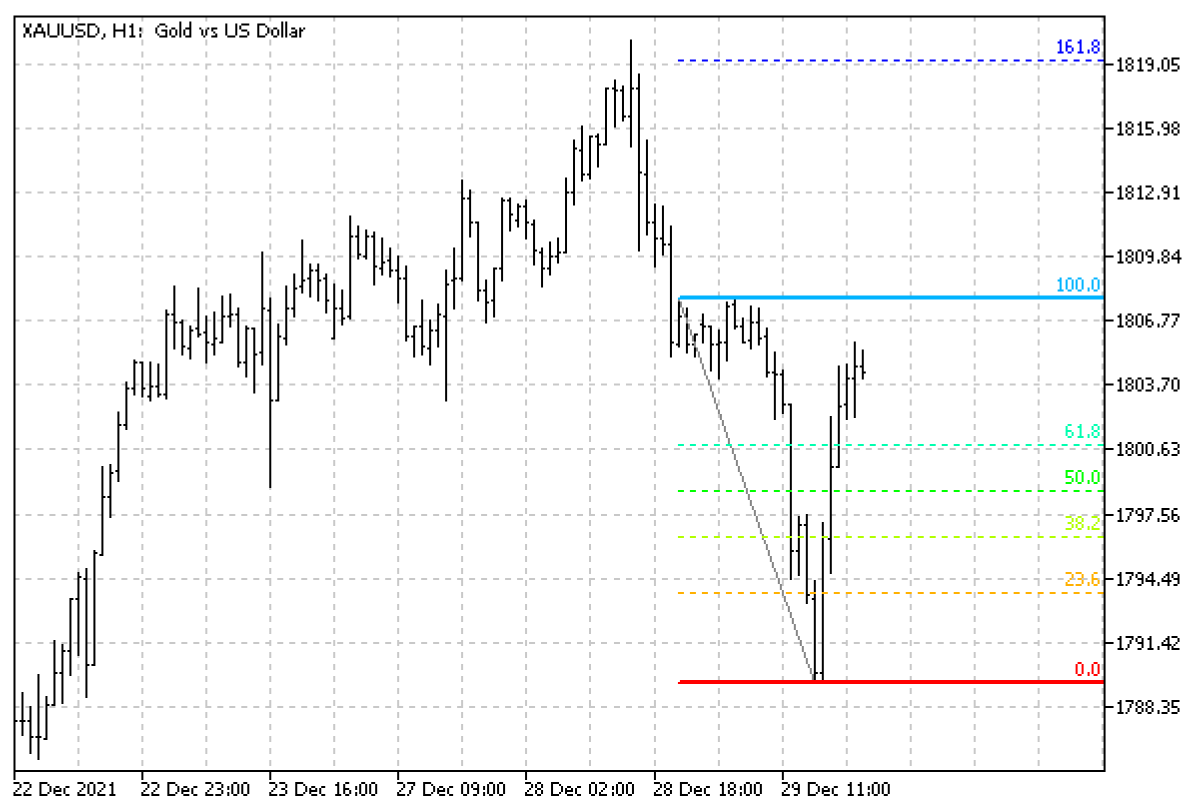
В качестве примера рассмотрим индикатор *ObjectHighLowFibo.mq5*. Для заданного диапазона баров, который определяется как номер последнего бара (*BarOffset*) и количество баров (*BarCount*) слева от него, индикатор находит максимальную цену *High* и минимальную цену *Low*, после чего создает для этих точек объект OBJ_FIBO. По мере формирования новых баров уровни Фибоначчи будут сдвигаться вправо на более актуальные цены.

```
#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;
}
```

Непосредственная настройка объекта производится во вспомогательной функции *DrawFibo*. В ней, в частности, уровни раскрашиваются в радужные цвета, а их стиль и толщина определяются на основе того, являются ли соответствующие значения "круглыми" (без дробной части).

```
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);
   // точки привязки
   for(int i = 0; i < ArraySize(t); ++i)
   {
      ObjectSetInteger(0, name, OBJPROP_TIME, i, t[i]);
      ObjectSetDouble(0, name, OBJPROP_PRICE, i, p[i]);
   }
   // общие настройки
   ObjectSetInteger(0, name, OBJPROP_COLOR, clr);
   ObjectSetInteger(0, name, OBJPROP_RAY_RIGHT, true);
   // настройки уровней
   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;
}
```

Вот вариант того, что как может выглядеть объект на графике.



Объект Фибоначчи с настройками уровней

5.8.27 Дополнительные свойства объектов Ганна, Фибоначчи и Эллиота

Для объектов Ганна, Фибоначчи и Эллиота существуют специфические настройки, свойственные только им. В зависимости от типа применяйте функции *ObjectGetInteger/ObjectSetInteger* или *ObjectGetDouble/ObjectSetDouble*.

Идентификатор	Описание	Тип
OBJPROP_DIRECTION	Тренд объектов Ганна (веер OBJ_GANNFAN и сетка OBJ_GANNGRID)	ENUM_GANN_DIRECTION
OBJPROP_DEGREE	Уровень волновой разметки Эллиота	ENUM_ELLIOT_WAVE_DEGREE
OBJPROP_DRAWLINES	Отображение линий для волновой разметки Эллиота	bool
OBJPROP_SCALE	Масштаб в пунктах на бар (свойство объектов Ганна и объекта "Дуги Фибоначчи")	double
OBJPROP_ELLIPSE	Отображение полного эллипса для объекта "Дуги Фибоначчи" (OBJ_FIBOARC)	bool

Перечисление ENUM_GANN_DIRECTION имеет следующие элементы:

Константа	Направление тренда
GANN_UP_TREND	Восходящие линии
GANN_DOWN_TREND	Нисходящие линии

ENUM_ELLIOT_WAVE_DEGREE используется для задания размера (способа маркировки) волн Эллиота.

Константа	Описание
ELLIOTT_GRAND_SUPERCYCLE	Главный Суперцикл (Grand Supercycle)
ELLIOTT_SUPERCYCLE	Суперцикл (Supercycle)
ELLIOTT_CYCLE	Цикл (Cycle)
ELLIOTT_PRIMARY	Первичный цикл (Primary)
ELLIOTT_INTERMEDIATE	Промежуточное звено (Intermediate)
ELLIOTT_MINOR	Второстепенный цикл (Minor)
ELLIOTT_MINUTE	Минута (Minute)
ELLIOTT_MINUETTE	Секунда (Minuette)
ELLIOTT_SUBMINUETTE	Субсекунда (Subminuette)

5.8.28 Объект-график

Объект-график OBJ_CHART позволяет создать внутри графика миниатюры других графиков — для других инструментов и таймфреймов. Объекты-графики включаются в общий список графиков, который мы получали программно с помощью функций *ChartFirst* и *ChartNext*. Как было сказано в разделе Проверка состояния основного окна, специальное свойство графика CHART_IS_OBJECT позволяет узнать по идентификатору, является ли он полноценным окном или объектом-графиком. В последнем случае вызов *ChartGetInteger(id, CHART_IS_OBJECT)* вернет *true*.

У объекта-графика имеется набор характерных только для него свойств.

Идентификатор	Описание	Тип
OBJPROP_CHART_ID	Идентификатор графика (r/o)	long
OBJPROP_PERIOD	Период графика	ENUM_TIMEFRAMES
OBJPROP_DATE_SCALE	Признак отображения шкалы времени	bool
OBJPROP_PRICE_SCALE	Признак отображения ценовой шкалы	bool
OBJPROP_CHART_SCALE	Масштаб (значение в диапазоне 0 – 5)	int
OBJPROP_SYMBOL	Символ	string

Идентификатор, получаемый через свойство OBJPROP_CHART_ID, позволяет управлять объектом, как обычным графиком с помощью функций из главы Работа с графиками. Вместе с тем, существуют и некоторые ограничения:

- Объект нельзя закрыть с помощью *ChartClose*;
- В объекте нельзя поменять символ/период с помощью функции *ChartSetSymbolPeriod*;
- В объекте не модифицируются свойства CHART_SCALE, CHART_BRING_TO_TOP, CHART_SHOW_DATE_SCALE и CHART_SHOW_PRICE_SCALE.

По умолчанию все свойства (за исключением OBJPROP_CHART_ID) равны соответствующим свойствам текущего окна.

Демонстрация объектов-графиков реализована в виде безбуферного индикатора *ObjectChart.mq5*. Он создает подокно с двумя объектами-графиками для того же символа, что и текущий график, но со смежными таймфреймами — выше и ниже текущего.

Объекты привязываются к правому верхнему углу подокна и имеют одинаковые предопределенные размеры:

```
#define SUBCHART_HEIGHT 150
#define SUBCHART_WIDTH  200
```

Разумеется, высота подокна должна совпадать с высотой объектов, пока мы не умеем адаптивно реагировать на события изменения размера.

```
#property indicator_separate_window
#property indicator_height SUBCHART_HEIGHT
#property indicator_buffers 0
#property indicator_plots   0
```

Настройка одного мини-графика производится в функции *SetupSubChart*, которая принимает номер объекта, его размеры и требуемый таймфрейм. Результатом *SetupSubChart* является идентификатор объекта-графика, который мы просто для справки выводим в журнал.

```
void OnInit()
{
   Print(SetupSubChart(0, SUBCHART_WIDTH, SUBCHART_HEIGHT, PeriodUp(_Period)));
   Print(SetupSubChart(1, SUBCHART_WIDTH, SUBCHART_HEIGHT, PeriodDown(_Period)));
}
```

Макросы *PeriodUp* и *PeriodDown* используют вспомогательную функцию *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;
}
```

А вот и основная рабочая функция *SetupSubChart*.

```
long SetupSubChart(const int n, const int dx, const int dy,
   ENUM_TIMEFRAMES tf = PERIOD_CURRENT, const string symbol = NULL)
{
   // создаем объект
   const string name = Prefix + "Chart-"
      + (symbol == NULL ? _Symbol : symbol) + PeriodToString(tf);
   ObjectCreate(0, name, OBJ_CHART, ChartWindowFind(), 0, 0);
   
   // привязываем к правому верхнему углу подокна
   ObjectSetInteger(0, name, OBJPROP_CORNER, CORNER_RIGHT_UPPER);
   // позиция и размер
   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);
   
   // специфические настройки графика
   if(symbol != NULL)
   {
      ObjectSetString(0, name, OBJPROP_SYMBOL, symbol);
   }
   
   if(tf != PERIOD_CURRENT)
   {
      ObjectSetInteger(0, name, OBJPROP_PERIOD, tf);
   }
   // отключаем вывод линеек
   ObjectSetInteger(0, name, OBJPROP_DATE_SCALE, false);
   ObjectSetInteger(0, name, OBJPROP_PRICE_SCALE, false);
   // добавляем индикатор MA в объект по его id просто для демо
   const long id = ObjectGetInteger(0, name, OBJPROP_CHART_ID);
   ChartIndicatorAdd(id, 0, iMA(NULL, tf, 10, 0, MODE_EMA, PRICE_CLOSE));
   return id;
}
```

Напомним, что у объекта-графика точка привязки всегда зафиксирована в левом верхнем углу объекта, поэтому при привязке к правому углу окна требуется добавлять ширину объекта (это делается за счет +1 в выражении *(n + 1) * dx* для OBJPROP_XDISTANCE).

На следующем скриншоте показан результат работы индикатора на графике XAUUSD,H1.

Два объекта-графика в подокне индикатора

Как мы видим, в мини-графиках отображаются таймфреймы M30 и H2.

Важно отметить, что вы можете добавлять в объекты-графики индикаторы и применять к ним tpl-шаблоны, в том числе и с экспертом. Однако создавать объекты внутри объектов-графиков нельзя.

> Когда объект-график скрыт из-за отключенной визуализации на текущем или на всех таймфреймах, свойство CHART_WINDOW_IS_VISIBLE для внутреннего чарта продолжает возвращать *true*.

5.8.29 Перемещение объектов

Для перемещения объектов в координатах время/цена можно использовать не только *ObjectSet*-функции изменения свойств, но и специальную функцию *ObjectMove* — она изменяет координаты указанной точки привязки объекта.

bool ObjectMove(long chartId, const string name, int index, datetime time, double price)

Параметр *chartId* задает идентификатор графика (0 — текущий график). Имя объекта передается в параметре *name*. Номер точки привязки и координаты указываются в параметрах *index*, *time* и *price*, соответственно.

Функция использует асинхронный вызов, то есть отправляет команду в очередь событий графика и не дожидается самого перемещения.

Функция возвращает признак того, была ли команда успешно поставлена в очередь (в этом случае результат равен *true*). О фактическом положении объекта следует узнавать с помощью вызовов *ObjectGet*-функций.

Модифицируем в индикаторе *ObjectHighLowFibo.mq5* функцию *DrawFibo* таким образом, чтобы задействовать *ObjectMove*. Вместо двух вызовов *ObjectSet*-функций в цикле по точкам привязки теперь будет один вызов *ObjectMove*:

```
bool DrawFibo(const string name, const datetime &t[], const double &p[],
   const color clr)
{
   ...
   for(int i = 0; i < ArraySize(t); ++i)
   {
      // было:
      // ObjectSetInteger(0, name, OBJPROP_TIME, i, t[i]);
      // ObjectSetDouble(0, name, OBJPROP_PRICE, i, p[i]);
      // стало:
      ObjectMove(0, name, i, t[i], p[i]);
   }
   ...
}
```

Функцию *ObjectMove* имеет смысл применять, где меняются обе координаты точки привязки. В некоторых случаях эффект имеет лишь одна координата (например, в каналах стандартного отклонения и линейной регрессии в точках привязки важны только начальная и конечная даты/время, а значение цен в этих точках каналы рассчитывают автоматически) — тогда единственный вызов *ObjectSet*-функции более уместен, чем *ObjectMove*.

5.8.30 Получение времени или цены в заданных точках линий

Множество графических объектов включает в свой состав одну или несколько прямых линий. MQL5 позволяет выполнять интерполяцию и экстраполяцию точек на этих линиях и получать по одной координате другую, например, цену по времени или время по цене.

Интерполяция доступна всегда: она работает "внутри" объекта, то есть между точками привязки. Экстраполяция за пределы объекта возможна только в том случае, если для него включено свойство "луча" в соответствующем направлении (см. Свойства лучей для объектов с прямыми линиями).

Функция *ObjectGetValueByTime* возвращает значение цены для указанного времени. Функция *ObjectGetTimeByValue* возвращает значение времени для указанной цены.

double ObjectGetValueByTime(long chartId, const string name, datetime time, int line)
datetime ObjectGetTimeByValue(long chartId, const string name, double value, int line)

Вычисления производятся для объекта с именем *name* на графике с идентификатором *chartId*. Параметры *time* и *value* задают известную координату, для которой следует рассчитать неизвестную. Так как объект может иметь несколько линий, одной координате будет соответствовать несколько значений, в связи с чем необходимо указать номер линии в параметре *line*.

Функция возвращает значение цены или времени для проекции точки с указанной исходной координатой относительно линии.

В случае ошибки будет получен 0, а код ошибки записан в *_LastError*. Например, попытка экстраполировать значение линии при отключенном свойстве луча генерирует ошибку OBJECT_GETVALUE_FAILED (4205).

Функции применимы для следующих объектов:

- Трендовая линия (OBJ_TREND)
- Трендовая линия по углу (OBJ_TRENDBYANGLE)
- Линия Ганна (OBJ_GANNLINE)
- Равноудаленный канал (OBJ_CHANNEL) — 2 линии
- Канал на линейной регрессии (OBJ_REGRESSION) — 3 линии
- Канал стандартного отклонения (OBJ_STDDEVCHANNEL) — 3 линии
- Линия со стрелкой (OBJ_ARROWED_LINE)

Проверим работу функции с помощью безбуферного индикатора *ObjectChannels.mq5*. Он создает два объекта с каналами стандартного отклонения и линейной регрессии, после чего запрашивает и выводит в комментарий цену верхней и нижней линий на будущих барах. Для канала стандартного отклонения свойство OBJPROP_RAY_RIGHT включено, а для канала регрессии — нет (намеренно). В связи с этим значения от второго канала получены не будут, и на экране для него всегда выводятся нули.

По мере формирования новых баров каналы будут автоматически перемещаться вправо. Длина каналов задается во входном параметре *WorkPeriod* (10 баров по умолчанию).

```
input int WorkPeriod = 10;

const string Prefix = "ObjChnl-";
const string ObjStdDev = Prefix + "StdDev";
const string ObjRegr = Prefix + "Regr";

void OnInit()
{
   CreateObjects();
   UpdateObjects();
}
```

Функция *CreateObjects* создает 2 канала и делает начальные настройки для них.

```
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: луч не включен у канала регрессии (намеренно)
}
```

Функция *UpdateObjects* перемещает каналы на последние *WorkPeriod* баров.

```
void UpdateObjects()
{
   const datetime t0 = iTime(NULL, 0, WorkPeriod);
   const datetime t1 = iTime(NULL, 0, 0);

   // мы не используем ObjectMove, потому что каналы работают
   // только с координатой времени (цена рассчитывается автоматически)
   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);
}
```

В обработчике *OnCalculate* обновляем позицию каналов на новых барах, а на каждом тике вызываем *DisplayObjectData*, чтобы получить экстраполяцию цен и вывести в комментарий.

```
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;
}
```

В функции *DisplayObjectData* узнаем цены в точках привязки на средней линии (OBJPROP_PRICE), а также запрашиваем с помощью *ObjectGetValueByTime* значения цен для верхней и нижней линий каналов через *WorkPeriod* баров в будущем.

```
void DisplayObjectData()
{
   const double p0 = ObjectGetDouble(0, ObjStdDev, OBJPROP_PRICE, 0);
   const double p1 = ObjectGetDouble(0, ObjStdDev, OBJPROP_PRICE, 1);

   // следующие равенства всегда верны вследствие алгоритма расчета каналов:
   // - средние линии обоих каналов совпадают,
   // - точки привязки всегда лежат на средней линии,
   // ObjectGetValueByTime(0, ObjStdDev, iTime(NULL, 0, 0), 0) == p1
   // ObjectGetValueByTime(0, ObjRegr, iTime(NULL, 0, 0), 0) == p1

   // пытаемся экстраполировать будущие цены по верхней и нижней линии
   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);

   // выводим все полученные цены в комментарий
   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));
}
```

Важно отметить, что из-за того, что свойство луча не включено для канала регрессии, он всегда дает нули в будущем (хотя при запросе цен внутри временного отрезка канала мы получили бы корректные значения).

Каналы и значения цен в точках их линий

Здесь для каналов длиной 10 баров экстраполяция также делается на 10 баров вперед, что дает будущие значения, показанные в строке с "dev:", примерно соответствующие правой границе окна.

5.9 Интерактивные события на графиках

Графики MetaTrader 5 не только обеспечивают визуальное представление данных и являются средой выполнения MQL-программ, но и поддерживают механизм интерактивных событий — за счет него программы могут реагировать на действия пользователя и других программ. Как уже говорилось в Обзоре функций обработки событий, для этих целей выделен особый тип событий — *OnChartEvent*.

Любой индикатор или эксперт может получать такие события при условии описания в коде одноименной функции обработки событий с предопределенной сигнатурой. В некоторых примерах индикаторов, которые мы рассмотрели ранее, нам уже пришлось воспользоваться этой возможностью. В данной главе мы рассмотрим систему событий подробно.

Событие *OnChartEvent* генерируется клиентским терминалом при следующих действиях пользователя с графиком:

- изменение размера или настроек графика;
- нажатия клавиатуры, когда окно графика находится в фокусе;
- перемещение курсора мыши;
- щелчок мыши на графике;
- щелчок мыши на графическом объекте;

- создание графического объекта;
- удаление графического объекта;
- перемещение графического объекта при помощи мыши;
- окончание редактирования текста в поле ввода графического объекта OBJ_EDIT.

MQL-программа получает перечисленные события только от графика, на котором она запущена. Как и прочие типы событий, они поступают в очередь. Все события затем обрабатываются одно за другим в порядке поступления. Если в очереди MQL-программы уже находится событие *OnChartEvent* конкретного типа или оно обрабатывается, новое событие такого же типа в очередь не ставится (отбрасывается).

Некоторые типы событий активны всегда, а некоторые по умолчанию выключены и их необходимо явным образом разрешить, установив соответствующие свойства графика с помощью вызова *ChartSetInteger*. К подобным выключенным событиям относятся, в частности, перемещения мыши и прокрутка её колесика. Все они характеризуются тем, что могут генерировать массовые потоки событий, и в целях экономии ресурсов рекомендуется их включать только при необходимости.

Помимо стандартных событий существует понятие "пользовательских событий", смысл и содержимое параметров которых назначает и интерпретирует сама MQL-программа (одна или несколько, если речь о взаимодействии комплекса программ). MQL-программа может посылать "пользовательские события" на график (в том числе, на другой) при помощи функции *EventChartCustom*. Такие события также обрабатываются функцией *OnChartEvent*.

Если на графике несколько MQL-программ с обработчиком *OnChartEvent*, они все получат одинаковый поток событий.

> Все MQL-программы работают в потоках, отличных от главного потока приложения. Главный поток терминала отвечает за обработку всех системных сообщений Windows, и в результате этой обработки в свою очередь порождает сообщения Windows для своего же приложения. Например, буксировка графика мышью порождает несколько системных сообщений WM_MOUSE_MOVE (в терминах Windows API) для последующей отрисовки окна приложения, а также посылает внутренние сообщения экспертам и индикаторам, запущенным на этом графике. При этом может возникнуть ситуация, что главный поток приложения ещё не успел обработать системное сообщение о перерисовке окна WM_PAINT (и поэтому ещё не изменил внешний вид графика), а эксперт или индикатор уже получили событие о перемещении курсора мыши. Тогда свойство графика CHART_FIRST_VISIBLE_BAR будет изменено только после отрисовки графика.

Поскольку из двух типов интерактивных MQL-программ мы пока изучили только индикаторы, все примеры данной главы будут строиться на базе индикаторов. Второй тип интерактивных MQL-программ — эксперты — будет описан в следующей Части книги, однако принципы работы с событиями в них полностью совпадают с представленными здесь.

5.9.1 Функция обработки событий OnChartEvent

Индикатор или эксперт может получать интерактивные события от терминала, если в коде описана функция *OnChartEvent* со следующим прототипом.

```
void OnChartEvent(const int event, const long &lparam, const double &dparam, const string &sparam)
```

Эта функция будет вызываться терминалом в ответ на действия пользователя или в случае генерации "пользовательского события" с помощью *EventChartCustom*.

В параметре *event* передается идентификатор события (его тип) — одно из значений перечисления ENUM_CHART_EVENT (см. таблицу).

Идентификатор	Описание
CHARTEVENT_KEYDOWN	Действие на клавиатуре
CHARTEVENT_MOUSE_MOVE	Перемещение мыши и нажатие кнопок мышки (если для графика установлено свойство CHART_EVENT_MOUSE_MOVE)
CHARTEVENT_MOUSE_WHEEL	Нажатие или прокрутка колесика мышки (если для графика установлено свойство CHART_EVENT_MOUSE_WHEEL)
CHARTEVENT_CLICK	Нажатие мышки на графике
CHARTEVENT_OBJECT_CREATE	Создание графического объекта (если для графика установлено свойство CHART_EVENT_OBJECT_CREATE)
CHARTEVENT_OBJECT_CHANGE	Изменение графического объекта через диалог свойств
CHARTEVENT_OBJECT_DELETE	Удаление графического объекта (если для графика установлено свойство CHART_EVENT_OBJECT_DELETE)
CHARTEVENT_OBJECT_CLICK	Нажатие мышки на графическом объекте
CHARTEVENT_OBJECT_DRAG	Перетаскивание графического объекта
CHARTEVENT_OBJECT_ENDEDIT	Окончание редактирования текста в графическом объекте "поле ввода"
CHARTEVENT_CHART_CHANGE	Изменение размеров или свойств графика (через диалог свойств, панель инструментов или контекстное меню)
CHARTEVENT_CUSTOM	Начальный номер события из диапазона пользовательских событий
CHARTEVENT_CUSTOM_LAST	Конечный номер события из диапазона пользовательских событий

Параметры *lparam*, *dparam*, *sparam* используются по-разному в зависимости от типа события. В общем можно сказать, что они содержат дополнительные данные, необходимые для обработки конкретного события. В следующих разделах приведены подробности для каждого типа.

Внимание! Функция *OnChartEvent* вызывается только в индикаторах и экспертах, которые непосредственно нанесены на график. Если какой-либо индикатор создан программно с помощью *iCustom* или *IndicatorCreate*, в него события *OnChartEvent* транслироваться не будут.

Кроме того, обработчик *OnChartEvent* не вызывается в тестере, даже в визуальном режиме.

Для первой демонстрации обработчика *OnChartEvent* рассмотрим безбуферный индикатор *EventAll.mq5*, который позволяет перехватывать и записывать в журнал все события.

```
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);
}
```

По умолчанию на графике могут генерироваться все типы событий, кроме четырех массовых, которые, как указано в вышеприведенной таблице, включаются специальными свойствами графика. В следующем разделе мы дополним индикатор настройками, чтобы по желанию включать те или иные типы.

Попробуйте запустить индикатор на графике, возможно, с уже имеющимися объектами, или создайте их в процессе работы индикатора.

Меняете размер или настройки графика, совершайте щелчки мышью, редактируйте свойства объектов. В журнале появятся записи следующего вида.

```
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 ''
```

Здесь мы видим события различных типов, значения их параметров станут понятны после прочтения следующих разделов.

5.9.2 Связанные с событиями свойства графика

Четыре типа событий способны генерировать очень много сообщений и потому по умолчанию отключены. Для их активации или последующего отключения следует установить соответствующие свойства графика с помощью функции *ChartSetInteger*. Все свойства имеют логический тип: *true* — включено, *false* — выключено.

Идентификатор	Описание
CHART_EVENT_MOUSE_WHEEL	Отправка на график сообщений CHARTEVENT_MOUSE_WHEEL о событиях колёсика мыши
CHART_EVENT_MOUSE_MOVE	Отправка на график сообщений CHARTEVENT_MOUSE_MOVE о перемещениях мыши
CHART_EVENT_OBJECT_CREATE	Отправка на график сообщений о создании графических объектов CHARTEVENT_OBJECT_CREATE
CHART_EVENT_OBJECT_DELETE	Отправка на график сообщений об уничтожении графических объектов CHARTEVENT_OBJECT_DELETE

Если какая-либо MQL-программа изменяет одно из этих свойств, оно влияет на все другие программы, выполняющиеся на том же графике, и продолжает действовать даже после завершения исходной программы.

По умолчанию все свойства имеют значение *false*.

Дополним индикатор *EventAll.mq5* из предыдущего раздела 4-мя входными переменными, позволяющими включить на выбор любые из данных типов событий (в дополнение к остальным, неотключаемым). Кроме того опишем 4 вспомогательных переменных, чтобы иметь возможность восстановить настройки графика после удаления индикатора.

```
input bool ShowMouseMove = false;
input bool ShowMouseWheel = false;
input bool ShowObjectCreate = false;
input bool ShowObjectDelete = false;

bool mouseMove, mouseWheel, objectCreate, objectDelete;
```

При запуске запомним текущие значения свойств и затем применим настройки, выбранные пользователем.

```
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);
}
```

Свойства восстанавливаются в обработчике *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);
}
```

Запустите индикатор с разрешенными новыми типами событий. Будьте готовы к большому количеству сообщений о перемещениях мыши. Вот фрагмент журнала:

```
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
```

Специфику информации по каждому типу события мы раскроем в соответствующих разделах далее.

5.9.3 Событие изменений графика

При изменении размеров, способа отображения котировок, масштаба или прочих настроек графика терминал посылает событие CHARTEVENT_CHART_CHANGE, которое не имеет параметров — суть изменений MQL-программа должна выяснить самостоятельно с помощью вызовов *ChartGet*-функций.

Мы уже использовали это событие в примере *ChartModeMonitor.mq5* в разделе Режимы отображения графика. Сейчас разберем еще один пример.

Как известно, MetaTrader 5 позволяет сохранять образ текущего графика в файл заданного размера (команда *Сохранить как рисунок* контекстного меню). Однако такой способ получения скриншота подходит не во всех случаях. В частности, если требуется изображение вместе со всплывающей подсказкой или при активном состоянии объекта типа "поле ввода" (когда внутри поля выделен текст и виден текстовый курсор), штатная команда не поможет, поскольку она формирует образ графика заново без учета этих и некоторых других нюансов текущего состояния окна.

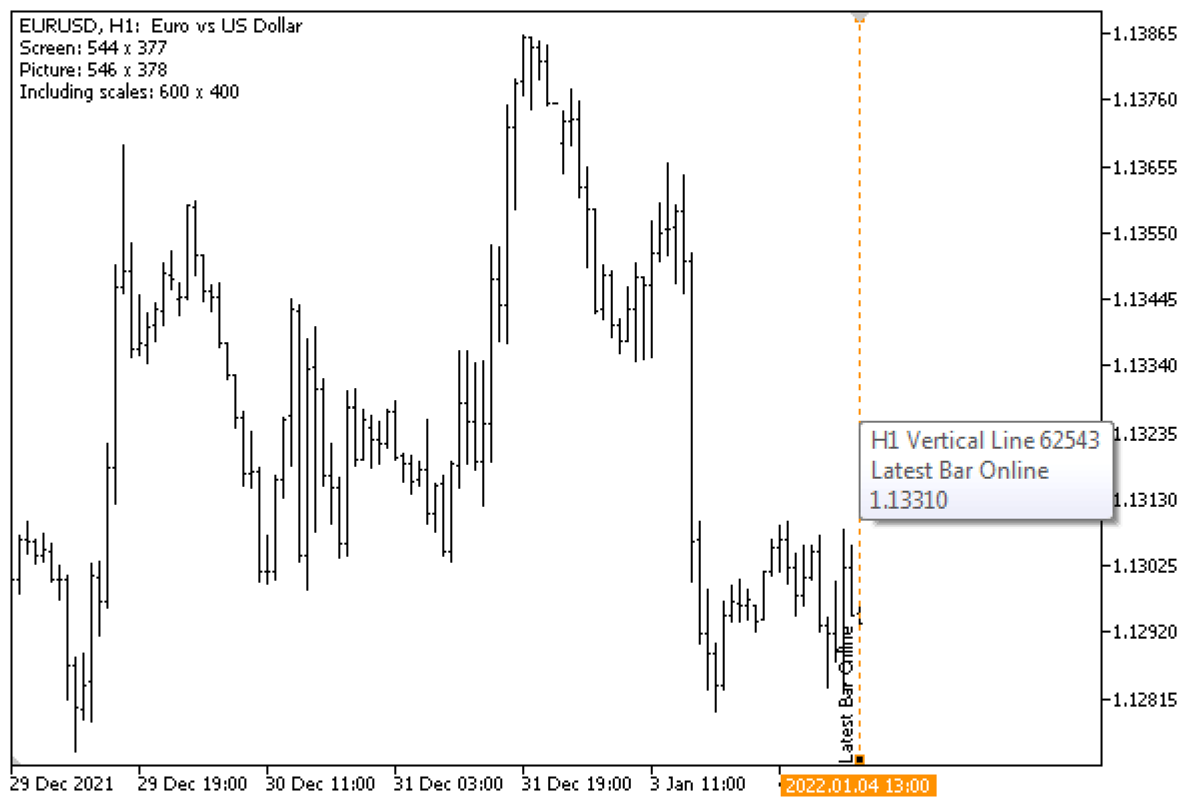
Единственной альтернативой для получения точной копии окна является использование внешних по отношению к терминалу средств (например, клавишей *PrtSc* через буфер обмена Windows), но такой способ не гарантирует необходимый размер окна. Чтобы не подбирать размер методом проб и ошибок или какими-то дополнительными программами, создадим индикатор *EventWindowSizer.mq5*, который будет на лету отслеживать настройку размера пользователем и выводить текущее значение в комментарий.

Вся работа производится в обработчике *OnChartEvent*, начиная с проверки идентификатора события на равенство CHARTEVENT_CHART_CHANGE. Размеры окна в пикселях можно получить с помощью свойств CHART_WIDTH_IN_PIXELS и CHART_HEIGHT_IN_PIXELS. Однако они возвращают размеры без учета рамок, а скриншот обычно требуется с рамками. Поэтому мы выведем в комментарий не только значения свойств (помечены словом "Screen"), но и откорректированные величины (помечены словом "Picture"): по ширине следует прибавить 2 пикселя, а по вертикали — 1 (таковы особенности отрисовки окна в терминале).

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strir
{
   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);
      // "Сырые" размеры "как есть" выводятся с пометкой "Screen",
      // поправка на (-2,-1) нужна для включения рамок - выведено с пометкой "Picture
      // поправка на (-54,-22) нужна для включения шкал - выведено с пометкой "Incluc
      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));
   }
}
```

Более того, полученные значения не учитывают шкалы времени и цен. Если они также должны быть учтены в размере скриншота, то следует внести поправку и на их размер. К сожалению, MQL5 API не предоставляет возможности узнать эти размеры, так что мы можем определить их лишь эмпирическим путем: для стандартных настроек шрифтов Windows ширина шкалы цен составляет 54 пикселя, и высота шкалы времени — 22 пикселя. В вашей версии Windows эти константы могут отличаться, поэтому их следует отредактировать или задать с помощью входных параметров.

Запустив индикатор на графике, попробуйте менять размеры окна и смотрите, как будут меняться числа в комментарии.



Скриншот окна со всплывающей подсказкой и текущими размерами в комментарии

5.9.4 События клавиатуры

MQL-программы могут получать от терминала сообщения о нажатиях клавиш на клавиатуре, обрабатывая в функции *OnChartEvent* события CHARTEVENT_KEYDOWN.

Важно отметить, что события генерируются только в активном графике, и только в том случае, когда он имеет фокус ввода.

В системе Windows фокусом называется логическое и визуальное выделение одного конкретного окна, с которым в данный момент взаимодействует пользователь. Как правило, перемещение фокуса осуществляется кликом мыши или специальными сочетаниями клавиш (*Tab*, *Ctrl+Tab*), в результате чего выбранное окно тем или иными образом подсвечивается. Например, в поле ввода появится текстовый курсор, в списке раскрашивается текущая строка альтернативным цветом и т.д.

Подобные визуальные эффекты заметны и в терминале, когда фокус получает, в частности, окно *Обзора рынка*, *Окно данных* или журнал экспертов. Однако с окнами графиков ситуация несколько иная. Отличить по внешним признакам, имеет ли видимый на переднем плане график фокус ввода или нет, не всегда представляется возможным. Гарантированно переключить фокус можно, как уже было сказано, щелчком мыши по требуемому графику (именно по графику, а не по заголовку окна или его рамке) или с помощью горячих клавиш:

- Alt+W вызывает окно со списком графиков, в котором можно выбрать один;
- Ctrl+F6 выполняет переключение на следующий график (в списке окон, где порядок соответствует, как правило, порядку закладок);
- Crtl+Shift+F6 выполняет переключение на предыдущий график;

С полным перечнем горячих клавиш MetaTrader 5 можно ознакомиться в документации — некоторые сочетания не соответствуют общим рекомендациям Microsoft (например, F10 открывает окно котировок, а не активирует главное меню).

В параметрах события CHARTEVENT_KEYDOWN содержится следующая информация:

- *lparam* — код нажатой клавиши;
- *dparam* — количество нажатий клавиши, сгенерированных за время ее удержания в нажатом состоянии;
- *sparam* — битовая маска, описывающая статус клавиш клавиатуры, преобразованная в строку.

Биты	Описание
0–7	Скан-код клавиши (зависит от аппаратной части, OEM)
8	Признак клавиши расширенной клавиатуры
9–12	Для служебных целей Windows (не использовать)
13	Состояние клавиши *Alt* (1 нажата, 0 отжата), недоступно (см. ниже)
14	Предыдущее состояние клавиши (1 если была нажата, 0 если была отжата)
15	Измененное состояние клавиши (1 если отпускается, 0 если нажимается)

Состояние клавиши *Alt* на самом деле недоступно, т.к. перехватывается терминалом, и этот бит всегда равен 0. Бит 15 также всегда равен 0 из-за контекста срабатывания данного события — в MQL-программу передаются только нажатия, но не отжатия клавиш.

Признак расширенной клавиатуры (бит 8) устанавливается, например для клавиш числового блока (на ноутбуках обычно активируется по *Fn*), клавиш типа *NumLock*, *ScrollLock*, правого *Ctrl* (в отличие от левого, основного *Ctrl*) и так далее. Подробнее об этом читайте в документации Windows.

При первом нажатии какой-либо несистемной клавиши бит 14 будет равен 0. Если же удержать клавишу нажатой, последующие автоматически генерируемые повторения события будут иметь 1 в этом бите.

Следующая структура поможет убедиться в правильности описания битов.

```
struct KeyState
{
   uchar scancode;
   bool extended;
   bool altPressed;
   bool previousState;
   bool transitionState;

   KeyState() { }
   KeyState(const ushort keymask)
   {
      this = keymask; // используем перегрузку оператора=
   }
   void operator=(const ushort keymask)
   {
      scancode = (uchar)(0xFF & keymask);
      extended = 0x100 & keymask;
      altPressed = 0x2000 & keymask;
      previousState = 0x4000 & keymask;
      transitionState = 0x8000 & keymask;
   }
};
```

В MQL-программе её можно использовать так.

```
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);
   }
}
```

В практических целях более удобно выделять битовые признаки из маски клавиш макросами.

```
#define KEY_SCANCODE(SPARAM) ((uchar)(((ushort)SPARAM) & 0xFF))
#define KEY_EXTENDED(SPARAM) ((bool)(((ushort)SPARAM) & 0x100))
#define KEY_PREVIOUS(SPARAM) ((bool)(((ushort)SPARAM) & 0x4000))
```

Вы можете запустить на графике индикатор *EventAll.mq5* из раздела Связанные с событиями свойства графика и посмотреть, какие значения параметров будут выводиться в журнал при нажатии тех или иных клавиш.

Важно отметить, что код в *lparam* — это один из виртуальных кодов клавиш клавиатуры. Их список можно увидеть в файле *MQL5/Include/VirtualKeys.mqh*, который поставляется вместе с MetaTrader 5. Вот, например, некоторые из них:

```
#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 коды символов '0' - '9' (0x30 - 0x39)
// VK_A - VK_Z ASCII коды символов 'A' - 'Z' (0x41 - 0x5A)
```

Коды называются виртуальными, потому что соответствующие клавиши могут быть по-разному расположены на разных клавиатурах или даже реализовываться через совместные нажатия вспомогательных клавиш (таких как *Fn* на ноутбуках). Кроме того, виртуальность имеет и другую сторону: одна и та же клавиша может генерировать различные символы или управляющие воздействия. Например, клавиша с английской буквой 'A' при переключении на русскоязычную раскладку станет соответствовать русской букве 'Ф'. Также каждая из буквенных клавиш "умеет" генерировать заглавную или строчную букву в зависимости от режима *CapsLock* и состояния клавиш *Shift*.

В связи с этим для получения символа из кода виртуальной клавиши в MQL5 API существует специальная функция *TranslateKey*.

short TranslateKey(int key)

Функция возвращает Unicode-символ по переданному виртуальному коду клавиши, учитывая текущий язык ввода и состояние управляющих клавиш.

В случае ошибки будет получено значение -1. Ошибка может возникнуть в том случае, если код не соответствует корректному символу, например, при попытке получить символ для клавиши *Shift*.

Напомним, что помимо полученного кода нажатой клавиши MQL-программа способна дополнительно Проверить состояние клавиатуры в части управляющих клавиш и режимов. Кстати говоря, константы вида TERMINAL_KEYSTATE_XXX, передаваемые в качестве параметра в функцию *TerminalInfoInteger*, составлены по принципу 1000 + виртуальный код клавиши. Например, TERMINAL_KEYSTATE_UP равно 1038, потому что VK_UP равно 38 (0x26).

При планировании алгоритмов с реакцией на нажатия клавиш имейте в виду, что терминал может перехватывать многие комбинации клавиш, поскольку они зарезервированы для выполнения определенных действий (ссылка на документацию приводилась выше). В частности, нажатие на пробел открывает поле быстрой навигации по оси времени. MQL5 API позволяет отчасти управлять такой встроенной обработкой клавиатуры и при необходимости отключать — см. раздел Управление мышью и клавиатурой.

Простой безбуферный индикатор *EventTranslateKey.mq5* служит демонстрацией данной функции. В его обработчике *OnChartEvent* для событий CHARTEVENT_KEYDOWN вызывается *TranslateKey*, чтобы получить допустимый символ *Unicode*. Если это удается, символ добавляется в строку

сообщения, которое выводится в комментарии графика. По нажатию *Enter* в текст вставляется перевод строки, а по нажатию *Backspace* — стирается последний символ с конца.

```
#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);
   }
}
```

Вы можете попробовать вводить символы в разных регистрах и разных языках.

Будьте внимательны. Функция возвращает знаковую величину *short*, в основном, для возможности вернуть код ошибки -1. Однако типом "широкого" двухбайтового символа принято считать беззнаковое целое *ushort*. Если приемная переменная будет описана как *ushort*, проверка с использованием -1 (например, *c!=-1*) будет выдавать предупреждение компилятора "sign mismatch" (требуется явное приведение типа), а другой вариант (*c >= 0*) вообще ошибочен, так как всегда равен *true*.

Для того чтобы в сообщение можно было вставлять пробелы между словами, в обработчике *OnInit* предварительно отключается быстрая навигация, активируемая пробелом.

```
void OnInit()
{
   ChartSetInteger(0, CHART_QUICK_NAVIGATION, false);
}
```

В качестве полноценного примера использования событий клавиатуры рассмотрим следующую прикладную задачу. Пользователям терминала хорошо известно, что масштаб главного окна графика можно менять интерактивно, не открывая диалог настроек, с помощью мыши:

достаточно нажать кнопку мыши в шкале цен и, не отпуская её, двигать вверх/вниз. К сожалению, в подокнах этот способ не работает.

Подокна всегда масштабируются автоматически, чтобы уместить всё содержимое, а чтобы поменять масштаб приходится открывать диалог и вводить значения вручную. Иногда необходимость в этом возникает, если в индикаторах в подокне наблюдаются "выбросы" — слишком большие единичные показания, которые мешают анализировать остальные данные нормального (среднего) размера. Кроме того, иногда желательно просто укрупнить картинку, чтобы разобраться с более мелкими деталями.

Чтобы решить эту проблему и позволить пользователю подстраивать масштаб подокна с помощью нажатий клавиш, реализован индикатор *SubScaler.mq5*. Сам он не имеет буферов и ничего не отображает.

SubScaler должен быть первым индикатором в подокне или, если выразиться более строго, он должен добавляться в подокно до того, как туда будет добавлен интересующий вас рабочий индикатор, масштаб которого нужно контролировать. Чтобы сделать *SubScaler* первым, его следует набросить на график (в главное окно) и тем самым создать новое подокно, куда уже затем добавить подчиненный индикатор.

В диалоге настроек рабочего индикатора важно включить опцию *Наследовать шкалу* (на закладке *Шкала*).

Когда оба индикатора запущены в подокне, можно использовать клавиши стрелок *Вверх/Вниз* для увеличения/уменьшения масштаба. Если при этом нажата клавиша *Shift*, то текущий видимый диапазон значений по вертикальной оси смещается вверх или вниз.

Увеличение масштаба означает укрупнение деталей ("наезд камеры"), так что часть данных может выйти за пределы окна. Уменьшение масштаба означает, что общая картинка становится меньше ("отъезд камеры").

Во входных параметрах задается:

- Начальный максимум — верхняя граница данных при первичном размещении на графике, по умолчанию +1000;
- Начальный минимум — нижняя граница данных при первичном размещении на графике, по умолчанию -1000;
- Коэффициент масштабирования — шаг, с которым будет меняться масштаб по нажатию клавиш, значение в диапазоне [0.01 ... 0.5], по умолчанию 0.1;

Минимум и максимум мы вынуждены спрашивать у пользователя, потому что *SubScaler* не может заранее знать рабочий диапазон значений произвольного стороннего индикатора, который будет добавлен в подокно следующим.

Когда график восстанавливается после запуска новой сессии терминала или загружается tpl-шаблон, *SubScaler* подхватывает масштаб предыдущего (сохраненного) состояния.

Теперь рассмотрим реализацию *SubScaler*.

Вышеописанные настройки задаются в соответствующих входных переменных:

```
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;
```

Кроме того переменная *Disabled* позволяет временно отключить реакцию на клавиатуру у конкретного экземпляра индикатора, чтобы настраивать несколько разных масштабов в разных подокнах (поочередно).

Поскольку входные переменные доступны в MQL5 только на чтение, мы вынуждены объявить еще одну переменную *ScaleFactor*, чтобы корректировать введенное значение в разрешенный диапазон [0.01 ... 0.5].

```
double ScaleFactor;
```

Номер текущего подокна (*w*) и количество индикаторов в нем (*n*) хранятся в глобальных переменных: все они заполняются в обработчике *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);
}
```

В функции *OnChartEvent* обрабатываем два типа событий: об изменении графика и с клавиатуры. Событие CHARTEVENT_CHART_CHANGE необходимо для того, чтобы отслеживать добавление в подокно следующего индикатора (рабочего, подлежащего изменению масштаба). Мы при этом запрашиваем текущий диапазон значений подокна (CHART_PRICE_MIN, CHART_PRICE_MAX) и определяем, не является ли он вырожденным, то есть когда и максимум, и минимум равны нулю. В этом случае необходимо применить указанные во входных параметрах начальные пределы (*FixedMinimum*, *FixedMaximum*).

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strir
{
   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;
   ...
   }
}
```

При получении события нажатия клавиатуры вызывается основная функция *Scale*, в которую передается не только *lparam*, но и состояние клавиши *Shift*, получаемое путем обращения к *TerminalInfoInteger(TERMINAL_KEYSTATE_SHIFT)*.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strir
{
  switch(id)
  {
    case CHARTEVENT_KEYDOWN:
      if(!Disabled)
         Scale(lparam, TerminalInfoInteger(TERMINAL_KEYSTATE_SHIFT));
      break;
    ...
  }
}
```

Внутри функции *Scale* первым делом получаем текущий диапазон значений в переменные *min* и *max*.

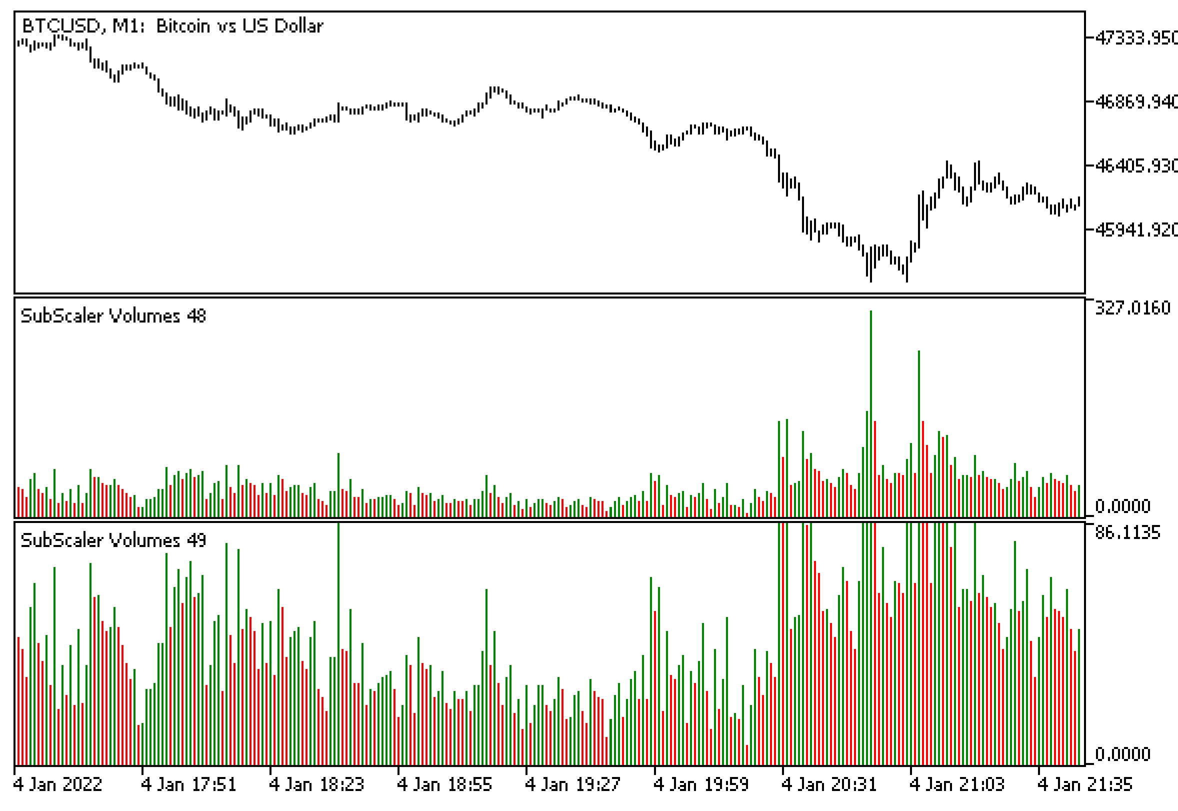
```
void Scale(const long cmd, const int shift)
{
   const double min = ChartGetDouble(0, CHART_PRICE_MIN, w);
   const double max = ChartGetDouble(0, CHART_PRICE_MAX, w);
   ...
```

Далее в зависимости от того, нажата ли в данный момент клавиша *Shift*, выполняется либо изменения масштаба, либо панорамирование, то есть сдвиг видимого диапазона значений вверх или вниз. В обоих случаях модификация производится с заданным шагом (множителем) *ScaleFactor*, относительно пределов *min* и *max*, и они назначаются свойствам индикатора INDICATOR_MINIMUM и INDICATOR_MAXIMUM, соответственно. Из-за того, что в "ведомом" индикаторе сделана настройка Наследовать шкалу, она становится рабочей и для него.

```
      if((shift & 0x10000000) == 0) // Shift не нажат - смена масштаба
      {
         if(cmd == VK_UP) // укрупняем (наезд)
         {
            IndicatorSetDouble(INDICATOR_MINIMUM, min / (1.0 + ScaleFactor));
            IndicatorSetDouble(INDICATOR_MAXIMUM, max / (1.0 + ScaleFactor));
            ChartRedraw();
         }
         else if(cmd == VK_DOWN) // мельчим (отъезд)
         {
            IndicatorSetDouble(INDICATOR_MINIMUM, min * (1.0 + ScaleFactor));
            IndicatorSetDouble(INDICATOR_MAXIMUM, max * (1.0 + ScaleFactor));
            ChartRedraw();
         }
      }
      else // Shift нажат - панорамирование/свдиг диапазона
      {
         if(cmd == VK_UP) // сдвигаем диаграммы вверх
         {
            const double d = (max - min) * ScaleFactor;
            IndicatorSetDouble(INDICATOR_MINIMUM, min - d);
            IndicatorSetDouble(INDICATOR_MAXIMUM, max - d);
            ChartRedraw();
         }
         else if(cmd == VK_DOWN) // сдвигаем диаграммы вниз
         {
            const double d = (max - min) * ScaleFactor;
            IndicatorSetDouble(INDICATOR_MINIMUM, min + d);
            IndicatorSetDouble(INDICATOR_MAXIMUM, max + d);
            ChartRedraw();
         }
      }
   }
```

При любом изменении вызывается *ChartRedraw*, чтобы обновить график.

Посмотрим, как *SubScaler* работает со стандартным индикатором объемов (любые другие индикаторы, включая пользовательские, управляются аналогично).



Различный масштаб, заданный индикаторами SubScaler в двух подокнах

Здесь в двух подокнах два экземпляра *SubScaler* применяют к объемам различный вертикальный масштаб.

5.9.5 События мыши

Мы уже имели возможность убедиться в получении событий мыши с помощью индикатора *EventAll.mq5* из раздела Связанные с событиями свойства графика. Событие CHARTEVENT_CLICK передается в MQL-программу по каждому нажатию кнопки мыши в окне, а события перемещения курсора CHARTEVENT_MOUSE_MOVE и прокрутки колесика CHARTEVENT_MOUSE_WHEEL требуют предварительного включения в настройках графика, для чего служат, соответственно, свойства CHART_EVENT_MOUSE_MOVE и CHART_EVENT_MOUSE_WHEEL (оба по умолчанию отключены).

Если под мышью находится графический объект, при нажатии кнопки генерируется не только событие CHARTEVENT_CLICK, но и CHARTEVENT_OBJECT_CLICK.

Для событий CHARTEVENT_CLICK и CHARTEVENT_MOUSE_MOVE параметры обработчика *OnChartEvent* содержат следующую информацию:

- lparam — координата X
- dparam — координата Y

Кроме того, для события CHARTEVENT_MOUSE_MOVE параметр *sparam* содержит строковое представление битовой маски, описывающей статус кнопок мыши и управляющих клавиш (*Ctrl*, *Shift*). Установка в 1 конкретного бита означает нажатие соответствующей кнопки или клавиши.

Биты	Описание
0	Состояние левой кнопки мыши
1	Состояние правой кнопки мыши
2	Состояние клавиши SHIFT
3	Состояние клавиши CTRL
4	Состояние средней кнопки мыши
5	Состояние первой дополнительной кнопки мыши
6	Состояние второй дополнительной кнопки мыши

Например, если установлен 0-й бит, получим число 1 (1 << 0), а если установлен 4-й бит, это даст число 16 (1 << 4). Одновременное нажатие кнопок или клавиш обозначается суперпозицией битов.

Для события CHARTEVENT_MOUSE_WHEEL координаты X и Y, а также флаги состояния кнопок мыши и управляющих клавиш особым образом кодируются внутри параметра *lparam*, а параметр *dparam* сообщает направление (плюс/минус) и величину прокрутки колесиком (кратную ±120).

8-байтовое целое число *lparam* совмещает в себе несколько упомянутых информационных полей.

<table>
<tr><th>Байты</th><th>Описание</th></tr>
<tr><td>0</td><td rowspan="2">Значение типа short с координатой X</td></tr>
<tr><td>1</td></tr>
<tr><td>2</td><td rowspan="2">Значение типа short с координатой Y</td></tr>
<tr><td>3</td></tr>
<tr><td>4</td><td>Битовая маска состояния кнопок и клавиш</td></tr>
<tr><td>5</td><td rowspan="3">Не используется</td></tr>
<tr><td>6</td></tr>
<tr><td>7</td></tr>
</table>

Вне зависимости от типа события координаты мыши передаются относительно всего окна, включая подокна, поэтому при необходимости их следует пересчитывать для конкретного подокна.

Для лучшего понимания CHARTEVENT_MOUSE_WHEEL воспользуйтесь индикатором *EventMouseWheel.mq5*. Он получает и раскодирует сообщения, выводя затем их описание в журнал.

```
#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);
   }
}
```

Запустите индикатор на графике и прокручивайте колёсико мыши, нажимая поочередно различные кнопки и клавиши. Вот пример результата:

```
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 События с графическими объектами

Для графических объектов, находящихся на графике, терминал генерирует несколько специализированных событий. Большинство из них относится к объектам любых типов. Событие окончания редактирования текста в поле ввода — CHARTEVENT_OBJECT_ENDEDIT — генерируется только для объектов типа OBJ_EDIT.

События щелчка мышью на объекте (CHARTEVENT_OBJECT_CLICK), буксировки мышью (CHARTEVENT_OBJECT_DRAG) и изменения свойств объека (CHARTEVENT_OBJECT_CHANGE) активны всегда, в то время как события создания объекта CHARTEVENT_OBJECT_CREATE и удаления объекта CHARTEVENT_OBJECT_DELETE требуют явного включения путем установки свойств графика: CHART_EVENT_OBJECT_CREATE и CHART_EVENT_OBJECT_DELETE.

При переименовании объекта вручную (из диалога свойств) терминал генерирует последовательность событий CHARTEVENT_OBJECT_DELETE, CHARTEVENT_OBJECT_CREATE, CHARTEVENT_OBJECT_CHANGE. При программном переименовании объекта эти события не генерируются.

Все события в объектах несут имя связанного объекта в параметре *sparam* функции *OnChartEvent*.

Кроме того, для CHARTEVENT_OBJECT_CLICK передаются координаты щелчка: X в параметре *lparam* и Y в параметре *dparam*. Координаты являются общими для всего графика, включая подокна.

Щелчок мышью на объектах работает по-разному в зависимости от типа объекта. Для некоторых, таких как эллипс, курсор должен находиться над любой точкой привязки. Для других (треугольник, прямоугольник, линии) курсор может быть над периметром объекта, а не только над точкой. Во всех таких случаях при наведении курсора мыши на интерактивный участок объекта появляется всплывающая подсказка с названием объекта.

Объекты с привязкой к экранным координатам, позволяющие формировать графический интерфейс программы, в частности, кнопка, поле ввода, прямоугольная панель, — генерируют события при щелчке мышью в любом месте внутри объекта.

При наличии нескольких объектов под курсором, событие генерируется для объекта с наибольшим Z-приоритетом. Если приоритеты объектов равны, событие приписывается тому, который был создан позднее (это соответствует их визуальному отображению, то есть более поздний перекрывает более ранний).

Проверить события в объектах поможет новая версия индикатора *EventAllObjects.mq5*. Мы в ней создадим и настроим с помощью уже известного класса *ObjectSelector* несколько объектов, а затем перехватим в обработчике *OnChartEvent* их характерные события.

```
#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);
   }
};
```

Изначально в *OnInit* создается объект-кнопка и вертикальная линия. Для линии будем отслеживать событие перемещения (буксировки), а по нажатию кнопки — создадим поле ввода, для которого будем проверять введенный текст.

```
const string ObjNamePrefix = "EventShow-";
const string ButtonName = ObjNamePrefix + "Button";
const string EditBoxName = ObjNamePrefix + "EditBox";
const string VLineName = ObjNamePrefix + "VLine";

bool objectCreate, objectDelete;

void OnInit()
{
   // запоминаем исходные настройки, чтобы восстановить в OnDeinit
   objectCreate = ChartGetInteger(0, CHART_EVENT_OBJECT_CREATE);
   objectDelete = ChartGetInteger(0, CHART_EVENT_OBJECT_DELETE);

   // устанавливаем новые свойства
   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();
}
```

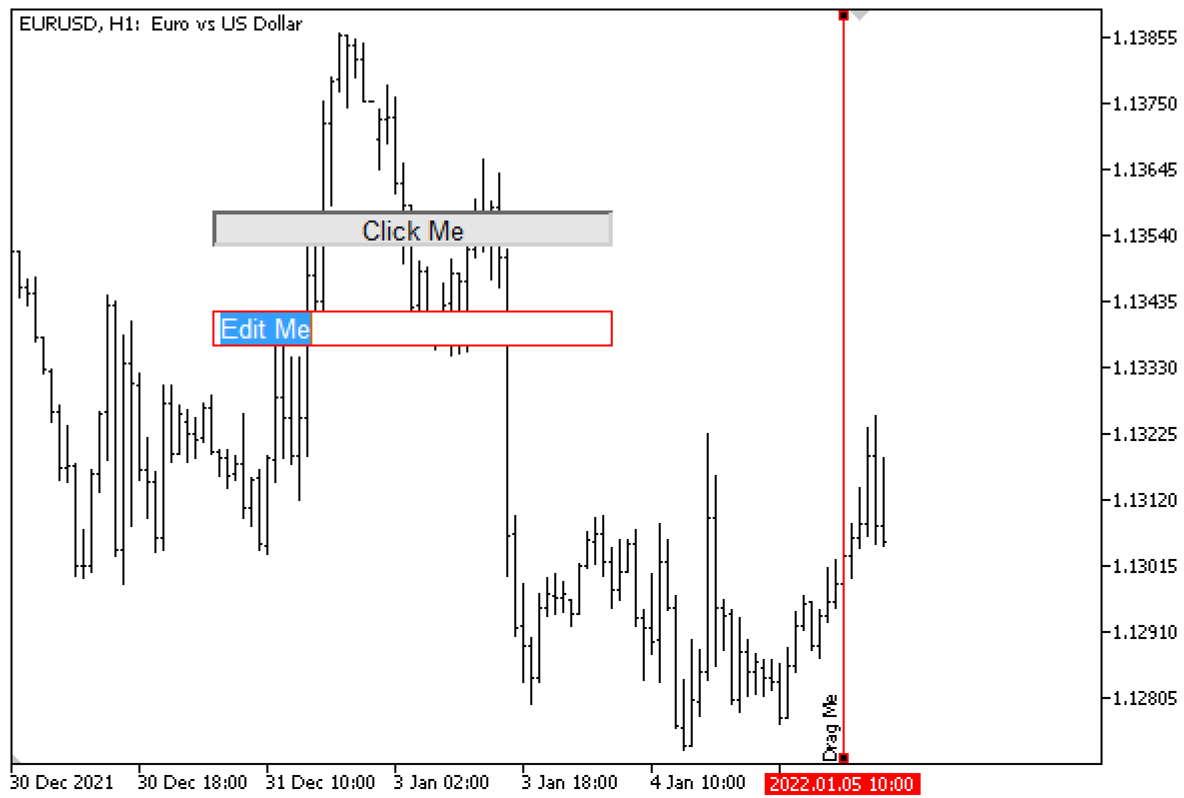
Попутно не забываем установить свойства графика CHART_EVENT_OBJECT_CREATE и CHART_EVENT_OBJECT_DELETE в *true*, чтобы получать уведомления об изменении набора объектов.

В функции *OnChartEvent* обеспечим дополнительную реакцию на требуемые события: по завершении буксировки выведем в журнал новую позицию линии, а по завершении редактирования текста в поле ввода — его содержимое.

```
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)));
   }
}
```

Обратите внимание, что когда кнопка нажимается первый раз, её состояние меняется с отжатого на нажатое, и в ответ на это мы создаем поле ввода. Если же щелкнуть кнопку еще раз, она сменит состояние обратно, в результате чего поле ввода будет удалено с графика.

Ниже приведено изображение графика в процессе работы индикатора.



Объекты, контролируемые обработчиком событий OnChartEvent

Сразу после запуска индикатора в журнале появляются такие строки:

```
CHARTEVENT_OBJECT_CREATE 0 0.000000 'EventShow-Button'
CHARTEVENT_OBJECT_CREATE 0 0.000000 'EventShow-VLine'
CHARTEVENT_CHART_CHANGE 0 0.000000 ''
```

Если затем перетащить линию мышью, увидим примерно следующее:

```
CHARTEVENT_OBJECT_DRAG 0 0.000000 'EventShow-VLine'
2022.01.05 10:00
```

Далее можно нажать кнопку и отредактировать текст в только что созданном поле ввода (по завершении редактирования нажмите *Enter* или щелкните мышью за пределами поля ввода). Это приведет к появлению таких записей в журнале (координаты и текст сообщения могут отличаться — здесь был введен текст "new message"):

```
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
```

Если после этого отжать кнопку, поле ввода удалится.

```
CHARTEVENT_OBJECT_CLICK 162 109.000000 'EventShow-Button'
CHARTEVENT_CLICK 162 109.000000 ''
CHARTEVENT_OBJECT_DELETE 0 0.000000 'EventShow-EditBox'
```

Стоит отметить, что кнопка работает по умолчанию как двухпозиционный переключатель, то есть "залипает" попеременно в нажатом или отжатом состоянии в результате клика мышью. Для обычной кнопки такое поведение излишне: чтобы просто отслеживать нажатия кнопки, следует при обработке события возвращать её в отжатое состояние вызовом *ObjectSetInteger(0, ButtonName, OBJPROP_STATE, false)*.

5.9.7 Генерация пользовательских событий

Помимо стандартных событий терминал поддерживает возможность программной генерации собственных событий, суть и информационное наполнение которых определяет MQL-программа. Мы называем их пользовательскими или настраиваемыми (custom). Такие события попадают в общую очередь событий графика и могут обрабатываться в функции *OnChartEvent* всеми заинтересованными программами.

Под пользовательские события зарезервирован специальный диапазон целочисленных идентификаторов в количестве 65536: от CHARTEVENT_CUSTOM до CHARTEVENT_CUSTOM_LAST включительно. Иными словами пользовательское событие должно иметь идентификатор CHARTEVENT_CUSTOM + n, где n находится в пределах от 0 до 65535. CHARTEVENT_CUSTOM_LAST как раз равно CHARTEVENT_CUSTOM + 65535.

Для отправки пользовательского события на график существует функция *EventChartCustom*.

bool EventChartCustom(long chartId, ushort customEventId,
  long lparam, double dparam, string sparam)

chartId — идентификатор графика-получателя события, 0 означает текущий график. *customEventId* — идентификатор события (выбирается разработчиком MQL-программы). Этот идентификатор автоматически добавляется к значению CHARTEVENT_CUSTOM и преобразуется к целому типу — именно это значение поступит в обработчик *OnChartEvent* первым аргументом. Остальные параметры *EventChartCustom* соответствуют стандартным параметрам событий в *OnChartEvent* с типами *long*, *double* и *string*, и могут содержать произвольную информацию.

Функция возвращает *true* в случае удачной постановки пользовательского события в очередь или *false* в случае ошибки (код ошибки станет доступным в *_LastError*).

Поскольку мы медленно, но верно приближаемся к наиболее сложной и важной части нашей книги, посвященной непосредственно автоматизации торговли, начнем решать прикладные задачи, которые пригодятся при разработке роботов. Сейчас, в контексте демонстрации возможностей пользовательских событий, обратимся к такому нюансу, как мультивалютный (или в более общем смысле, мультиинструментальный) анализ торгового окружения.

Чуть ранее, в главе про индикаторы, мы рассматривали мультивалютные индикаторы, но не обратили внимания на важный момент: несмотря на то, что индикаторы обрабатывали котировки разных символов, сам расчет запускался в обработчике *OnCalculate*, который срабатывает по приходу нового тика только одного символа — рабочего символа графика. Получается, что тики других инструментов по сути пропускаются. Например, если индикатор работает на символе A, по приходу его тика мы просто берем последние известные тики прочих инструментов (B, C, D), но вполне вероятно, что по каждому из них успели проскочить другие тики.

Если расположить мультивалютный индикатор на наиболее ликвидном инструменте (где тики поступают наиболее часто), это не столь критично. Однако в разные отрезки суток наиболее "быстрыми" могут становиться разные инструменты, и если аналитический или торговый алгоритм требует максимально быстрой реакции на новые котировки всех инструментов "корзины", мы оказываемся перед фактом, что текущее решение нас не устраивает.

К сожалению, стандартное событие прихода нового тика работает в MQL5 только для одного символа — рабочего символа текущего графика. Как мы знаем, в индикаторах в такие моменты вызывается обработчик *OnCalculate*, а в экспертах за это отвечает обработчик *OnTick*.

Следовательно, необходимо изобрести некий механизм, чтобы MQL-программа могла получать уведомления о тиках на всех интересующих её инструментах. В этом нам и помогут пользовательские события. Разумеется, для программ, анализирующих только один инструмент, это не нужно.

Мы сейчас разработаем пример индикатора *EventTickSpy.mq5*, который, будучи запущенным на конкретном символе X, сможет отправлять из своей функции *OnCalculate* уведомления о тике с помощью *EventChartCustom*. В результате, в обработчике *OnChartEvent*, который специально подготовлен для приема таких уведомлений, можно будет собрать уведомления из разных экземпляров индикатора с разных символов.

Данный пример будет носить демонстрационный характер. Впоследствии, при изучении мультивалютной автоторговли мы адаптируем данный прием для более удобного применения в экспертах.

Прежде всего, придумаем для индикатора номер пользовательского события. Поскольку мы собираемся посылать уведомления о тиках множества разных символов из некоторого заданного списка, здесь можно выбрать разные тактики. Например, можно выбрать один идентификатор события, а номер символа в списке и/или само название символа передавать в параметрах *lparam* и *sparam*, соответственно. Или можно взять некую константу (больше и равную CHARTEVENT_CUSTOM), а номера событий получать, прибавляя к этой константе номер символа (тогда у нас остаются свободными все параметры, в частности, *lparam* и *dparam*, и их можно использовать для передачи цен *Ask*, *Bid* или чего-то еще).

Мы остановимся на варианте, когда код события один. Объявим его в макросе TICKSPY. Это будет значение по умолчанию, которое пользователь сможет изменить, чтобы при необходимости избежать коллизий (хоть они и маловероятны) с другими программами.

```
#define TICKSPY 0xFEED // 65261
```

Это значение взято специально как довольно сильно отстоящее от первого разрешенного CHARTEVENT_CUSTOM.

При первоначальном (интерактивном) запуске индикатора пользователь должен указать перечень инструментов, тики которых он хочет отслеживать. Для этой цели опишем входную строковую переменную *SymbolList* — в ней символы указываются через запятую.

В параметре *Message* задается идентификатор пользовательского события.

Наконец, для передачи событий, как известно, требуется идентификатор приемного графика, поэтому предусмотрим параметр *Chart*. Пользователь не должен его редактировать: в первом экземпляре индикатора, запускаемого вручную, график известен неявным образом, за счет прикрепления к графику. В других копиях индикатора, которые наш первый экземпляр будет

запускать программным образом, этот параметр заполнит алгоритм с помощью вызова функции *ChartID* (см. далее).

```
input string SymbolList = "EURUSD,GBPUSD,XAUUSD,USDJPY"; // Список символов, через за
input ushort Message = TICKSPY;                          // Пользовательское сообщени
input long Chart = 0;                                    // Принимающий график (не ре
```

В параметре *SymbolList* для примера уже указан список с 4-мя распространенными инструментами. Отредактируйте его при необходимости в соответствии с вашим *Обзором рынка*.

В обработчике *OnInit* преобразуем список в массив символов *Symbols*, а затем в цикле запускаем этот же индикатор для всех символов из массива, за исключением текущего (как правило, есть такое совпадение, т.к. текущий символ уже обрабатывается этой исходной копией индикатора).

```
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();
               // запускаем этот же индикатор на другом символе с другими настройками
               // в частности, передаем наш ChartID, чтобы получать обратно уведомлен
               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'");
      }
   }
}
```

В начале *OnInit* в журнал выводится информация о запускаемом экземпляре индикатора, чтобы было понятно, что происходит.

Если бы мы выбрали вариант с раздельными кодами событий для каждого символа, то должны были бы вызывать *iCustom* следующим образом (добавляем *i* к *Message*):

```
iCustom(Symbols[i], PERIOD_CURRENT, MQLInfoString(MQL_PROGRAM_NAME), "",
   Message + i, ChartID());
```

Обратите внимание, что ненулевое значение параметра *Chart* подразумевает, что данная копия запущена программно и должна мониторить единственный символ — рабочий символ графика. Поэтому при запуске подчиненных копий нам не требуется передавать список символов.

В функции *OnCalculate*, которая вызывается при получении нового тика, отправляем на график *Chart* пользовательское событие *Message* с помощью вызова *EventChartCustom*. При этом параметр *lparam* не используется (равен 0), в параметре *dparam* мы передаем текущую (последнюю) цену *price[0]* (это *Bid* или *Last*, в зависимости от того, по какому типу цены строится график: она же — цена последнего обработанного графиком тика), а в параметре *sparam* — название символа.

```
int OnCalculate(const int rates_total, const int prev_calculated,
   const int, const double &price[])
{
   if(prev_calculated)
   {
      ArraySetAsSeries(price, true);
      if(Chart > 0)
      {
         // отправляем уведомление о тике на родительский график
         EventChartCustom(Chart, Message, 0, price[0], _Symbol);
      }
      else
      {
         OnSymbolTick(_Symbol, price[0]);
      }
   }

   return rates_total;
}
```

В исходном экземпляре индикатора, где параметр *Chart* равен 0, мы напрямую вызываем специальную функцию — своего рода мультисимвольный обработчик тиков *OnSymbolTick*. В этом случае нет необходимости вызывать *EventChartCustom*: хотя такое сообщение все равно придет на график и в эту копию индикатора, передача занимает несколько миллисекунд и зря загружает очередь.

Единственная задача *OnSymbolTick* в данной демонстрации — вывести в журнал название символа и новую цену.

```
void OnSymbolTick(const string &symbol, const double price)
{
   Print(symbol, " ", DoubleToString(price,
      (int)SymbolInfoInteger(symbol, SYMBOL_DIGITS)));
}
```

Разумеется, эта же функция вызывается и из обработчика *OnChartEvent* в приемной (исходной) копии индикатора по условию, что получено наше сообщение. Напомним, что терминал вызывает

OnChartEvent только в интерактивной копии индикатора (нанесенной на график) и не вызывает в тех копиях, которые мы создали "невидимыми" с помощью *iCustom*.

```
void OnChartEvent(const int id,
   const long &lparam, const double &dparam, const string &sparam)
{
   if(id >= CHARTEVENT_CUSTOM + Message)
   {
      OnSymbolTick(sparam, dparam);
      // ИЛИ (если использовать диапазон пользовательских событий):
      // OnSymbolTick(Symbols[id - CHARTEVENT_CUSTOM - Message], dparam);
   }
}
```

В принципе, мы могли бы не отсылать ни цену, ни название символа в своем событии, поскольку общий список символов известен в исходном индикаторе (инициировавшем процесс), и потому достаточно каким-либо образом сообщить ему номер символа из списка. Это можно было бы сделать в параметре *lparam*, или, как уже было упомянуто выше, с помощью добавления номера к базовой константе пользовательского события. Тогда исходный индикатор, принимая события, мог бы взять символ по индексу из массива и получить по нему всю информацию о последнем тике с помощью *SymbolInfoTick*, включая разные типы цен.

Запустим индикатор на графике EURUSD с настройками по умолчанию, включая тестовый список "EURUSD,GBPUSD,XAUUSD,USDJPY". Вот фрагмент журнала:

```
16:45:48.745 (EURUSD,H1) Starting for chart 0, msg=0xFEED [EURUSD,GBPUSD,XAUUSD,USDJF
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
```

Обратите внимание, что в колонке с (символом,таймфреймом) — источником записи мы вначале видим стартующие экземпляры индикаторов на 4-х запрошенных символах.

После запуска первым тиком был тик на XAUUSD, а не EURUSD. Далее тики символов поступают примерно с равной интенсивностью, перемежаясь. Тики на EURUSD помечены звездочками, так что вы можете составить представление, сколько прочих тиков было бы пропущено, если бы не уведомления.

В левой колонке для справки сохранены временные метки.

Места, где две цены у двух последовательных событий с одного и того же символа совпадают, как правило, означают, что в тике менялась цена *Ask* (мы её просто не выводим здесь).

Чуть позже, после изучения торговых MQL5 API, мы применим такой же принцип для реагирования на мультивалютные тики в экспертах.

Часть 6. Автоматизация торговли

В этой части мы займемся изучением наиболее сложной и важной составляющей MQL5 API, позволяющей автоматизировать непосредственно торговлю.

Начнем мы с описания сущностей, без которых невозможно написать корректный советник, таких как спецификации финансовых инструментов и настройки торгового счета.

Затем обратимся к встроенным торговым функциям и структурам данных, специфическим для роботов событиям и режимам работы. В частности, для экспертов ключевой особенностью является интеграция с тестером, который позволяет оценивать финансовые показатели и оптимизировать торговые стратегии. Мы познакомимся с внутренними механизмами оптимизации и её контролем через программное API.

Кроме того, тестер очень важен в плане разработки MQL-программ, поскольку он предоставляет инструменты для отладки в различных режимах — по барам или по тикам, с искусственной генерацией тиков или на истории реальных тиков, с визуальным воспроизведением потока котировок в выделенном окне или без него, то есть в ускоренном режиме, так сказать, "без присмотра".

Мы уже пробовали тестировать индикаторы в визуальном режиме, но для них разрешен лишь ограниченный набор настроек. При разработке экспертов нам станет доступен полный набор возможностей тестера.

Кроме того, мы познакомимся с новым "измерением" рыночной информации — стаканом цен и его программным интерфейсом.

Программирование на MQL5 для трейдеров — исходные коды из книги: Часть 6

Примеры из книги также доступны в публичном проекте \MQL5\Shared Projects\MQL5Book

6.1 Финансовые инструменты и обзор рынка

MetaTrader 5 предоставляет пользователям возможность анализа и торговли финансовыми инструментами (или просто символам, "тикерам"), которые формируют основу практически всех подсистем терминала. Графики, индикаторы и сама история котировок существуют в привязке к торговым символам. И конечно же, на финансовых инструментах строится основной функционал терминала — торговые приказы, сделки, контроль маржинальных требований, история торгового счета.

Брокер поставляет трейдерам через терминал оговоренный перечень символов, из которого каждый пользователь выбирает для себя наиболее интересные, формируя Обзор рынка. Напомним, что именно список в окне Обзор рынка определяет символы, по которым терминал запрашивает онлайн котировки и позволяет открывать графики с последующей подкачкой истории.

Разумеется, MQL5 API предоставляет аналогичные программные средства, позволяющие просматривать и анализировать характеристики всех символов, добавлять их в Обзор рынка или исключать оттуда.

В дополнение к стандартным символам, информация по которым поступает с торговых площадок и транслируется в терминал брокером, MetaTrader 5 дает возможность создавать пользовательские символы: их свойства и история котировок может загружаться из произвольных источников данных, рассчитываться по формулам или MQL-программами. Пользовательские символы также участвуют в Обзоре рынка, могут использоваться для тестирования стратегий и технического анализа, однако имеют и естественное ограничение — ими нельзя торговать онлайн штатными средствами MQL5 API, поскольку эти символы отсутствуют на сервере. Пользовательские символы будут рассмотрены в отдельной главе, в последней, седьмой части книги.

Чуть раньше, мы уже изучили в соответствующих главах временные ряды с ценовыми данными отдельных символов, включая подкачку истории на примере индикаторов. Весь этот функционал на самом деле предполагает, что соответствующие символы уже включены в Обзор рынка. Это особенно актуально для мультисимвольных индикаторов и экспертов, которые обращаются не только к рабочему символу графика, но и прочим символам. В данной главе мы познакомимся с тем, как Обзор рынка управляется из MQL-программ.

Кроме того, в главе про графики уже были описаны некоторые свойства символов, сделанные доступными через функции получения основных свойств текущего графика (*Point*, *Digits*), поскольку работа графика невозможна без связанного с ним символа. Теперь мы изучим большинство свойств символов, включая их спецификацию. Их полный набор можно изучить в документации MQL5 на сайте.

6.1.1 Получение списков доступных символов и Обзора рынка

Для действий с символами MQL5 API имеет несколько функций. С помощью них можно узнать общее количество доступных символов, количество символов, выбранных в *Обзор рынка*, а также их имена. Как известно, общий перечень символов, доступных в терминале, указан в виде иерархической структуры в диалоге *Символы*, который пользователь может открыть командой *Вид -> Символы* или из контекстного меню *Обзора рынка*. В этот перечень включены как символы, предоставляемые брокером, так и пользовательские символы, созданные локально. Общее количество символов позволяет узнать функция *SymbolsTotal*.

int SymbolsTotal(bool selected)

Параметр *selected* задает, запрашиваются ли только символы в *Обзоре рынка* (*true*) или все доступные символы (*false*).

В паре с *SymbolsTotal* обычно используется еще одна функция — *SymbolName*: она возвращает название символа по его сквозному номеру (группировка хранения символов по логическим папкам здесь не учитывается, см. свойство SYMBOL_PATH).

string SymbolName(int index, bool selected)

Параметр *index* задает номер запрашиваемого символа. Значение *index* должно лежать в пределах от 0 до количества символов, с учетом контекста запроса, определяемого вторым параметром *selected*: *true* ограничивает перечисление символами, выбранными в *Обзор рынка*, в то время как *false* соответствует абсолютно всем символам (по аналогии с *SymbolsTotal*). Следовательно, при вызове *SymbolName* следует всегда указывать в *selected* такое же значение, как и при предыдущем вызове *SymbolsTotal*, использованном для определения диапазона индексов.

В случае ошибки, в частности, если запрошенный индекс выходит за пределы списка, функция вернет пустую строку, а код ошибки записывается в переменную *_LastError*.

> Важно отметить, что при включенной опции *selected* пара функций *SymbolsTotal* и *SymbolName* возвращает информацию для списка символов, реально актуализируемых терминалом, то есть символов, по которым производится постоянная синхронизация с сервером, а для MQL-программ доступна история котировок. Этот список может быть больше списка, видимого в *Обзоре рынка*, куда элементы добавляются явным образом: пользователем или MQL-программой (о том, как это делается — см. раздел Редактирование списка *Обзора рынка*). Такие невидимые в окне символы автоматически подключаются терминалом, когда они необходимы для расчета кросс-курсов. Среди свойств символов имеются два, позволяющие отличить явный выбор (SYMBOL_VISIBLE) от неявного (SYMBOL_SELECT) — они будут рассмотрены в разделе о проверке состояния символа. Строго говоря, для функций *SymbolsTotal* и *SymbolName* установка *selected* в *true* соответствует расширенному набору символов с взведенным свойством SYMBOL_SELECT, а не только тех, что имеют SYMBOL_VISIBLE равное *true*.

Порядок, в котором возвращаются символы *Обзоры рынка*, соответствует окну терминала (с учетом возможной перестановки, сделанной пользователем, и без учета сортировки по какой-либо колонке, если она включена). Изменить порядок символов в *Обзоре рынка* программным образом нельзя.

Порядок в общем списке *Символов* задается самим терминалом (содержимое и сортировка *Обзора рынка* на него не влияет).

В качестве примера рассмотрим простой скрипт *SymbolList.mq5*, который выводит в журнал доступные символы. Входной параметр *MarketWatchOnly* позволяет пользователю ограничить список только символами в *Обзоре рынка* (если параметр равен *true*) или получить полный список (*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);
   // пишем список символов в Обзоре рынка или всех доступных
   for(int i = 0; i < n; ++i)
   {
      PrintFormat("%4d %s", i, SymbolName(i, MarketWatchOnly));
   }
   // намеренно запрашиваем вне диапазона, чтобы показать ошибку
   PRTF(SymbolName(n, MarketWatchOnly)); // MARKET_UNKNOWN_SYMBOL(4301)
}
```

Ниже приводится пример журнала.

```
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 Редактирование списка Обзора рынка

С помощью функции *SymbolSelect* разработчик MQL-программы может добавить конкретный символ в *Обзор рынка* или удалить оттуда.

bool SymbolSelect(const string name, bool select)

Параметр *name* содержит имя символа, над которым выполняется действие. В зависимости от значения параметра *select* символ добавляется в *Обзор рынка* (*true*) или удаляется из него. В именах символов учитывается регистр: например, "EURUSD.m" не равно "EURUSD.M".

Функция возвращает признак успеха (*true*) или ошибки (*false*). Код ошибки можно узнать в *_LastError*.

> Символ не может быть убран, если есть открытые графики с этим символом или открытые позиции по этому символу. Кроме того, нельзя удалить символ, который явным образом использован в формуле расчета синтетического (пользовательского) инструмента, добавленного в *Обзор рынка*.

Следует иметь в виду, что даже если по символу нет открытых графиков и позиций, он может опосредованно использоваться MQL-программами: например, они могут читать его историю котировок или тиков. Удаление такого символа может привести к проблемам в этих программах.

Следующий скрипт — *SymbolRemoveUnused.mq5* — способен скрыть все явно неиспользуемые символы, поэтому рекомендуется проверять его на демо-счете или предварительно сохранить текущий набор символов через контекстное меню.

```
#include <MQL5Book/MqlError.mqh>

#define PUSH(A,V) (A[ArrayResize(A, ArraySize(A) + 1) - 1] = V)

void OnStart()
{
   // запрашиваем подтверждение пользователя на удаление
   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[];
      // проходим по символам Обзора рынка в обратном порядке
      for(int i = n - 1; i >= 0; --i)
      {
         const string s = SymbolName(i, true);
         if(SymbolSelect(s, false))
         {
            // запоминаем, что удалось удалить
            PUSH(removed, s);
         }
         else
         {
            // в случае ошибки выводим причину
            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);
      ...
```

После того как пользователь даст согласие на анализ списка символов, программа пытается последовательно скрыть каждый символ, вызывая *SymbolSelect(s, false)*. Это удается только для неиспользуемых явно инструментов. Перебор символов производится в обратном порядке, чтобы не нарушать индексацию. Все успешно удаленные символы собираются в массиве *removed*. В журнал выводится статистика и сам массив.

Если *Обзор рынка* был изменен, пользователю затем предоставляется возможность восстановить в нем все удаленные символы, вызвав в цикле *SymbolSelect(removed[i], true)*.

```
        if(r > 0)
        {
           // есть возможность вернуть удаленные символы обратно в Обзор рынка
           // (в этот момент в окне отображается сокращенный список)
           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);
           }
        }
     }
  }
```

Вот каким может быть результат в журнале.

```
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
```

Примите к сведению, что хотя восстановление символов и производится в исходном порядке, в каком они были в *Обзоре рынка* относительно друг друга, добавление происходит в конец списка, после оставшихся символов. Таким образом, все "занятые" символы окажутся в начале списка, а восстановленные — после них. Такова специфика *SymbolSelect*: добавление символа всегда производится в конец списка, то есть вставить символ в конкретную позицию нельзя — перестановка элементов списка доступна только для ручного редактирования.

6.1.3 Проверка символа на существование

Вместо того чтобы просматривать весь список символов MQL-программа может проверить наличие конкретного символа по его имени. Для этой цели предназначена функция *SymbolExist*.

bool SymbolExist(const string name, bool &isCustom)

В параметре *name* следует передать название интересующего вас символа. Передаваемый по ссылке параметр *isCustom* будет установлен функцией согласно признаку, является ли указанный символ стандартным (*false*) или пользовательским (*true*).

Функция возвращает *false*, если символ не найден ни среди стандартных, ни среди пользовательских символов.

Частичным аналогом данной функции является запрос свойства SYMBOL_EXIST.

Разберем простой скрипт *SymbolExists.mq5* для проверки данной функции. В его параметре пользователь может указать интересующее его название, которое передается в *SymbolExist*, а результат выводится в журнал. Если на вход подать пустую строку, будет проверен рабочий символ текущего графика. По умолчанию параметр имеет значение "XYZ", что предположительно не соответствует ни одному из доступных символов.

```
#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
}
```

При запуске скрипта 2 раза — сначала со значением по умолчанию, а потом с пустой строкой на графике EURUSD — получим следующие записи в журнале.

```
Symbol 'XYZ' is missing
Symbol 'EURUSD' is standard
```

Если вы уже имеете пользовательские символы или создадите новый с какой-нибудь простой расчетной формулой, то сможете убедиться в заполнении переменной custom. Например, если в терминале открыть окно *Символы* и нажать кнопку *Создать символ*, то в поле *Формула синтетического инструмента* можно ввести "SP500/FTSE100" (названия индексов могут отличаться у вашего брокера), а в поле с названием *Символа* — "GBPUSD.INDEX". По нажатию кнопки *OK* будет создан пользовательский инструмент, для которого можно открыть график и на нем наш скрипт должен вывести:

```
Symbol 'GBPUSD.INDEX' is custom
```

При настройке собственного символа не забудьте задать не только формулу, но и достаточно "мелкие" значение размера пункта и шага изменения цены (тика) — в противном случае ряд синтетических котировок может получиться "ступенчатым" или даже выродиться в прямую линию.

6.1.4 Проверка актуальности данных по символу

В силу распределенности клиент-серверной архитектуры, клиентские и серверные данные могут время от времени отличаться. Например, это может происходить сразу после начала сеанса работы терминала, при потере подключения или при большой нагрузке на ресурсы компьютера. Также символ, скорее всего, останется некоторое время несинхронизированным сразу после его добавления в Обзор рынка. MQL5 API позволяет проверить актуальность котировочных данных по конкретному символу с помощью функции *SymbolIsSynchronized*.

bool SymbolIsSynchronized(const string name)

Функция возвращает *true*, если локальные данные по символу с именем *name* синхронизированы с данными на торговом сервере.

В разделе Получение характеристик массивов котировок было представлено, среди прочих свойств таймсерий, свойство SERIES_SYNCHRONIZED, которое возвращает более узкий по своему смыслу признак синхронизированности: он распространяется на конкретное сочетание символа и таймфрейма. В отличие от этого свойства, функция *SymbolIsSynchronized* возвращает признак синхронизированности общей истории символа.

Построение всех таймфреймов запускается только после завершения скачивания истории. Из-за многопоточной архитектуры и параллельных вычислений в терминале может сложиться ситуация, когда *SymbolIsSynchronized* вернет *true*, а для какого-либо таймфрейма на том же символе свойство SERIES_SYNCHRONIZED будет временно равно *false*.

Посмотрим, как новая функция работает в индикаторе *SymbolListSync.mq5*. Он предназначен для периодической проверки всех символов из *Обзора рынка* на синхронизированность. Период проверки задается пользователем в параметре *SyncCheckupPeriod* в секундах. Он обуславливает запуск таймера в *OnInit*.

```
#property indicator_chart_window
#property indicator_plots 0

input int SyncCheckupPeriod = 1; // SyncCheckupPeriod (seconds)

void OnInit()
{
   EventSetTimer(SyncCheckupPeriod);
}
```

В обработчике *OnTimer* в цикле вызываем *SymbolIsSynchronized* и собираем в общую строку все несинхронизированные символы, после чего они выводятся в комментарии и в журнале.

```
void OnTimer()
{
   string unsynced;
   const int n = SymbolsTotal(true);
   // проверяем все символы в Обзоре рынка
   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");
   }
}
```

Например, если добавить в *Обзор рынка* отсутствовавший ранее символ (Brent), получим запись такого вида:

```
Unsynced symbols:
Brent
```

При нормальных условиях большую часть времени (пока запущен индикатор) в журнале не должно быть подобных сообщений. Однако во время проблем со связью может генерироваться поток предупреждений.

6.1.5 Получение последнего тика по символу

В главе про таймсерии, в разделе Работа с массивами реальных тиков, уже была представлена встроенная структура *MqlTick*, содержащая поля со значениями цен и объемов для конкретного символа, известными на момент каждого изменения котировок. В режиме онлайн MQL-программа может запросить последние полученные цены и объемы с помощью функции *SymbolInfoTick*, использующей эту же структуру.

bool SymbolInfoTick(const string symbol, MqlTick &tick)

Для символа с заданным именем *symbol* функция заполняет переданную по ссылке структуру *tick*. В случае успеха возвращается *true*.

Как известно, индикаторы и эксперты автоматически вызываются терминалом по приходу нового тика, если в них описаны соответствующие обработчики *OnCalculate* и *OnTick*. Однако информация о сути изменений цены, объеме последней сделки и времени генерации тика не передаются напрямую в обработчики — более подробную информацию позволяет получить как раз функция *SymbolInfoTick*.

Кроме того, тиковые события генерируются только для символа графика, в связи с чем мы уже рассматривали вариант получения собственного мультисимвольного события для тиков на основе пользовательских событий. В этом случае *SymbolInfoTick* дает возможность считывать информацию о тиках на сторонних символах по получению уведомлений.

Возьмем за основу индикатор *EventTickSpy.mq5* и преобразуем его в *SymbolTickSpy.mq5*, который будет запрашивать на каждом "мультивалютном" тике структуру *MqlTick* для соответствующего символа, а затем рассчитывать и выводить на график все спреды.

Добавим новый входной параметр *Index*. Он потребуется для нового способа рассылки уведомлений: мы будем посылать в пользовательском событии только номер изменившегося символа (см. далее).

```
#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)
```

Также добавим массив *Spreads* для хранения спредов по символам и переменную *SelfIndex*, для того чтобы запомнить позицию символа текущего графика в списке (если он там встретится — а это, как правило, так). Последнее нужно, чтобы вызывать нашу функцию "обработки" нового тика из *OnCalculate* в исходной копии индикатора — для неё проще и правильнее взять готовый индекс для *_Symbol* явным образом, а не отсылать через событие самим себе.

```
int Spreads[];
int SelfIndex = -1;
```

Введенные структуры данных инициализируем в *OnInit*. В остальном *OnInit* остался без изменений, включая запуск подчиненных экземпляров индикатора на сторонних символах (здесь эти строки опущены).

```
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;
         }
   ...
}
```

В обработчике *OnCalculate* генерируем пользовательское событие на каждом тике, если копия индикатора работает на чужом символе (при этом идентификатор графика *Chart*, на который следует отсылать уведомления, не равен 0). Обратите внимание, что в событии заполняется только параметр *lparam*, равный *Index* (*dparam* равен 0, *sparam* — NULL). Если *Chart* равно 0, значит мы в главной копии индикатора, работающей на символе графика *_Symbol*, и если он найден во входном списке символов, вызываем напрямую *OnSymbolTick* с соответствующим индексом *SelfIndex*.

```
int OnCalculate(const int rates_total, const int prev_calculated, const int, const dc
{
   if(prev_calculated)
   {
      if(Chart > 0)
      {
         EventChartCustom(Chart, Message, Index, 0, NULL);
      }
      else if(SelfIndex > -1)
      {
         OnSymbolTick(SelfIndex);
      }
   }

   return rates_total;
}
```

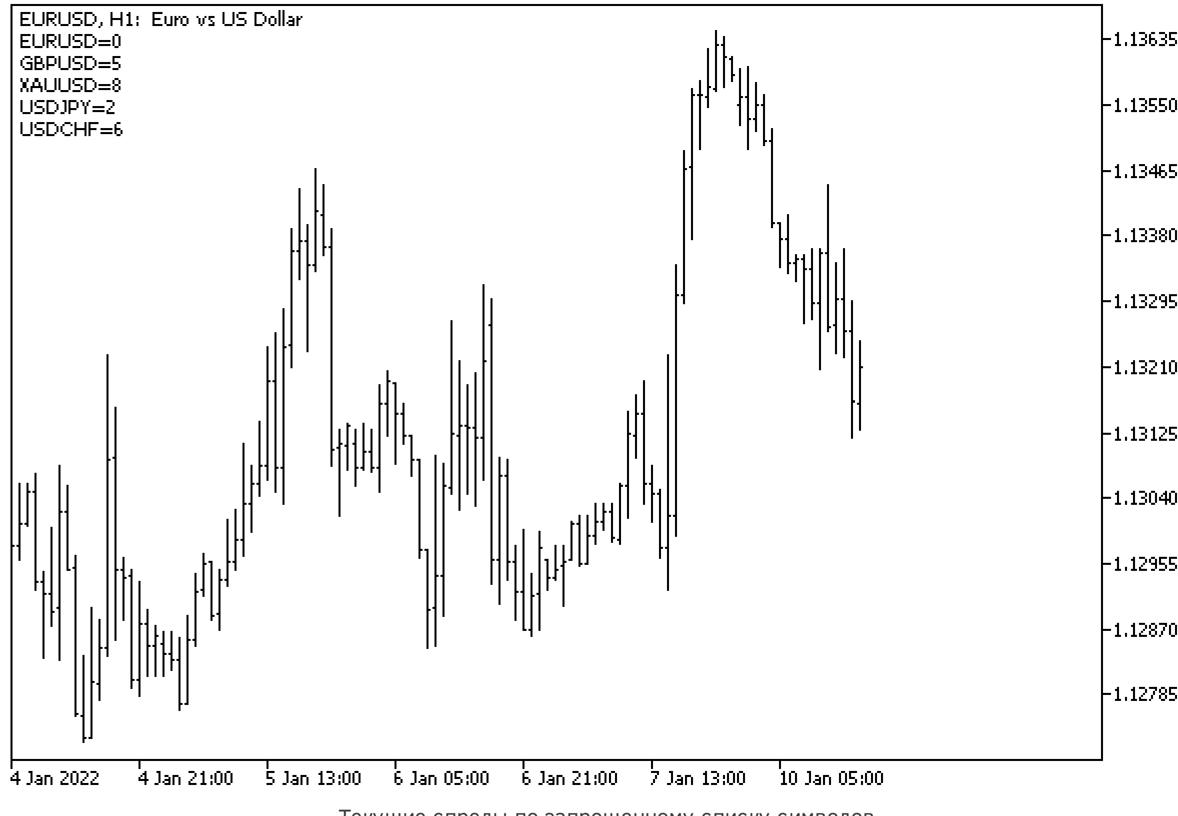
В приемной части алгоритма событий в *OnChartEvent* также вызываем *OnSymbolTick*, но на этот раз номер символа из списка получаем в *lparam* (то, что было отослано как параметр *Index* из другой копии индикатора).

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strir
{
   if(id == CHARTEVENT_CUSTOM + Message)
   {
      OnSymbolTick((int)lparam);
   }
}
```

Сама функция *OnSymbolTick* запрашивает полную информацию по тику с помощью *SymbolInfoTick* и вычисляет спред как разницу между ценами *Ask* и *Bid*, деленную на размер пункта (свойство SYMBOL_POINT мы рассмотрим позднее).

```
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);
   }
}
```

Новый спред обновляет соответствующую ячейку в массиве *Spreads*, после чего весь массив выводится на график в комментарии. Вот как это выглядит.



Текущие спреды по запрошенному списку символов

Вы можете сравнить в реальном времени соответствие информации в комментарии и в окне *Обзора рынка*.

6.1.6 Расписания торговых и котировочных сессий

Чуть позже, в следующих главах, мы обратимся к функциям MQL5 API, позволяющим автоматизировать торговые операции, но прежде следует изучить технические особенности платформы, от которых зависит успех вызова этих API. В частности, некоторые ограничения накладывают спецификации финансовых инструментов. В этой главе мы постепенно рассмотрим их программный анализ в полном объеме, а начнем с такого нюанса как сессии.

При торговле финансовыми инструментами следует учитывать, что многие международные рынки, такие как биржи, имеют предопределенные часы работы, только в течение которых доступна информация и торговля. Несмотря на то, что терминал постоянно подключен к серверу брокера, попытка заключить сделку вне рабочего расписания, закончится неудачей. В связи с этим, для каждого символа терминал хранит расписание сессий — временных отрезков внутри суток, когда можно выполнять те или иные действия.

Как известно, существует 2 основных типа сессий: котировочная и торговая. Во время котировочной сессии в терминал поступают (могут поступать) свежие котировки. Во время торговой сессии разрешено отправлять торговые приказы и совершать сделки. В течение суток может быть несколько сессий каждого типа, с перерывами (например, утренняя и вечерняя). Очевидно, что длительность котировочных сессий больше или равна торговым.

В любом случае времена сессий, то есть часы открытия и закрытия, переводятся терминалом из локального часового пояса биржи в часовой пояс брокера (серверное время).

MQL5 API позволяет узнать котировочные и торговые сессии каждого инструмента с помощью функций *SymbolInfoSessionQuote* и *SymbolInfoSessionTrade*. Эта важная информация позволяет, в частности, проверять в программе, открыт ли рынок в данный момент, прежде чем отправлять на сервер торговый приказ. Таким образом, мы предотвращаем получение неизбежного ошибочного результата и не нагружаем сервер зря. Следует учитывать, что в случае массированных ошибочных запросов на сервер из-за некорректно реализованной MQL-программы, сервер может начать "игнорировать" ваш терминал, отказываясь некоторое время выполнять последующие команды (даже корректные).

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)

Функции работают по одинаковому принципу. Для заданного символа *symbol* и дня недели *dayOfWeek* они заполняют переданные по ссылке параметры *from* и *to* временем открытия и закрытия сессии под номером *sessionIndex*. Индексация сессий начинается с 0. Структура ENUM_DAY_OF_WEEK была описана в разделе Перечисления.

Для запроса количества сессий отдельных функций не существует: вместо этого следует вызывать *SymbolInfoSessionQuote* и *SymbolInfoSessionTrade* с увеличивающимся индексом *sessionIndex*, пока функция не вернет признак ошибки (*false*). Когда сессия с указанным номером существует, и выходные аргументы *from* и *to* получили корректные значения, функции возвращают признак успеха (*true*).

Согласно документации MQL5, в полученных значениях *from* и *to* типа *datetime* следует игнорировать дату и учитывать только время. Это связано с тем, что информация представляет собой внутридневное расписание. Однако в этом правиле есть важное исключение.

Поскольку рынок потенциально открыт 24 часа в сутки, как в случае Forex, или биржа на другом конце света, где дневные рабочие часы совпадают со сменой дат в "таймзоне" вашего брокера, окончание сессий может иметь время, равное или превышающее 24 часа. Например, если начало сессий Forex равно 00:00, то окончание — 24:00. Вместе с тем, с точки зрения типа *datetime*, 24 часа — это 00 часов 00 минут уже на следующий день.

Более запутанной ситуация становится для бирж, расписание которых сдвинуто относительно временной зоны вашего брокера на несколько часов таким образом, что начало сессии приходится на один сутки, а окончание — на другие. Из-за этого в переменную *to* записывается не только время, но и лишние сутки, которые нельзя игнорировать, поскольку иначе внутридневное время *from* окажется больше внутридневного времени *to* (например, сессия может длиться с 21:00 сегодня до 8:00 завтра, то есть 21 > 8). При этом записанная естественным образом проверка на вхождение текущего времени внутрь сессии ("время икс больше начала и меньше окончания") окажется некорректной (например, условие x >= 21 && x < 8 не выполняется при x = 23, хотя сессия на самом деле активна).

Таким образом, мы приходим к выводу, что игнорировать дату в параметрах *from/to* нельзя, и этот нюанс следует учитывать в алгоритмах (см. пример).

Для демонстрации возможностей функций вернемся к примеру скрипта *EnvPermissions.mq5*, который был представлен в разделе Разрешения. Один из типов разрешений (или ограничений, если угодно) относится как раз к доступности торговли. Ранее в скрипте были учтены настройки терминала (TERMINAL_TRADE_ALLOWED) и настройки конкретной MQL-программы

(MQL_TRADE_ALLOWED). Теперь мы можем добавить к ним проверку сессий, чтобы определить торговые разрешения, действующие в определенный момент для конкретного символа.

Новая версия скрипта называется *SymbolPermissions.mq5*. Она также не является окончательной: в одной из следующих глав мы изучим ограничения, налагаемые настройками торгового счета.

Напомним, что в скрипте реализован класс *Permissions*, который предоставляет централизованное описание всех типов разрешений/ограничений, применимых к MQL-программам. Среди прочего в классе имеются методы для проверки доступности торговли: *isTradeEnabled* и *isTradeOnSymbolEnabled*. Первый из них относится к глобальным разрешениям и останется практически без изменений:

```
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);
   }
   ...
```

После проверок свойств терминала и MQL-программы идет обращение к *isTradeOnSymbolEnabled*, где и производится анализ спецификации символа. Раньше этот метод был практически пустым.

Кроме рабочего символа, передаваемого в параметре symbol, функция *isTradeOnSymbolEnabled* принимает текущее время (*now*) и требуемый режим торгов (*mode*). О последнем мы подробнее поговорим в следующих разделах (см. Разрешения на торговлю). Сейчас лишь отметим, что значение по умолчанию SYMBOL_TRADE_MODE_FULL дает максимальную свободу (разрешены все торговые операции).

```
static bool isTradeOnSymbolEnabled(string symbol, const datetime now = 0,
   const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
   // проверка сессий
   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;
      }
   }
   // проверка режима торговли для символа
   return found && (SymbolInfoInteger(symbol, SYMBOL_TRADE_MODE) == mode);
}
```

Если время *now* не указано (равно 0 по умолчанию), мы считаем, что сессии нас не интересуют. Это выражается в том, что переменная *found* с признаком того, что была найдена подходящая сессия (то есть, содержащая заданное время), сразу же устанавливается в *true*. Если же параметр *now* задан, функция попадает в блок анализа торговых сессий.

Для выделения времени без учета даты из значений типа *datetime* мы описываем константу *day*, равную количеству секунд в сутках. Выражение вроде *now % day* вернет остаток от деления полной даты и времени на длительность одного дня, что даст только время (старшие разряды в *datetime* будут нулевыми).

Функция *TimeDayOfWeek* возвращает день недели для переданного значения *datetime* и находится в заголовочном файле *MQL5Book/DateTime.mqh*, который мы уже использовали ранее (см. Дата и время).

Далее в цикле *while* вызываем функцию *SymbolInfoSessionTrade*, постоянно инкрементируя индекс сессии *i*, пока не будет найдена подходящая сессия или функция не вернёт *false* (сессий больше нет). Таким образом программа может получить полный перечень сессий по дням недели, аналогичный тому, что выводится в терминале в окне *Спецификации* символа.

Очевидно, что подходящей сессией является та, для которой время начала *from* и окончания *to* содержат заданную величину *time* между собой. Именно здесь мы учитываем нюанс, связанный с возможной круглосуточной торговлей: *from* и *to* сравниваются с *time* "как есть", без отбрасывания суток (*from % day* или *to % day*).

Как только *found* станет равен *true*, выходим из цикла. В противном случае цикл закончится по превышению допустимого количества сессий (функция *SymbolInfoSessionTrade* вернет *false*), и подходящая сессия так и не будет найдена.

Если согласно расписанию сессий торговля сейчас разрешена, мы дополнительно проверяем режим торговли по символу (SYMBOL_TRADE_MODE). Например, торговля по символу может быть полностью запрещена ("индикатив") или находиться в режиме "только закрытие позиций".

Приведенный выше код имеет некоторые упрощения по сравнению с окончательным вариантом в файле *SymbolPermissions.mq5*. В нем дополнительно реализован механизм для маркировки источника ограничения, вызвавшего отключение торговли. Все такие источники сведены в перечисление TRADE_RESTRICTIONS.

```
enum TRADE_RESTRICTIONS
{
   TERMINAL_RESTRICTION = 1,
   PROGRAM_RESTRICTION = 2,
   SYMBOL_RESTRICTION = 4,
   SESSION_RESTRICTION = 8,
};
```

В данный момент ограничение может исходить из 4-х инстанций: терминала, программы, символа и расписания сессий. Позднее мы добавим другие варианты.

Для регистрации факта обнаружения ограничения в классе *Permissions* заведена переменная *lastFailReasonBitMask*, позволяющая собирать в себе битовую маску из элементов перечисления с помощью вспомогательного метода *pass* (бит взводится, когда проверяемое условие *value* ложно, равно *false*).

```
static uint lastFailReasonBitMask;
static bool pass(const bool value, const uint bitflag)
{
   if(!value) lastFailReasonBitMask |= bitflag;
   return value;
}
```

Вызов метода *pass* с конкретным флагом делается на соответствующих этапах проверки. Например, метод *isTradeEnabled* в полном виде выглядит так:

```
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);
}
```

За счет этого при отрицательном результате вызова *TerminalInfoInteger(TERMINAL_TRADE_ALLOWED)* или *MQLInfoInteger(MQL_TRADE_ALLOWED)* будут взведены флаги TERMINAL_RESTRICTION или PROGRAM_RESTRICTION, соответственно.

Метод *isTradeOnSymbolEnabled* также устанавливает свои флаги при обнаружении проблем, включая сессионные.

```
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
}
```

В результате MQL-программа, использующая запрос *Permissions::isTradeEnabled*, может после получения запрета уточнить его суть с помощью методов *getFailReasonBitMask* и *explainBitMask*: первый возвращает маску взведенных флагов запретов "как есть", а второй формирует понятное для пользователя текстовое описание ограничений.

```
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;
}
```

С помощью вышеописанного класса *Permissions* в обработчике *OnStart* делается проверка доступности торговли для всех символов *Обзора рынка* (в текущий момент, *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 following symbols and origins:");
      Print(disabled);
   }
}
```

Если для какого-то символа торговля запрещена, мы увидим в журнале пояснение.

```
Trade is disabled for following symbols and origins:
USDRUB=SESSION_RESTRICTION
SP500m=SYMBOL_RESTRICTION
```

В данном случае рынок закрыт для "USDRUB", а для символа "SP500m" торговля отключена (более строго, не соответствует режиму SYMBOL_TRADE_MODE_FULL).

Предполагается, что при запуске скрипта алготрейдинг был разрешен на глобальном уровне в терминале. В противном случае мы дополнительно увидим в логе запреты TERMINAL_RESTRICTION и PROGRAM_RESTRICTION.

6.1.7 Маржинальные коэффициенты по символу

Среди характеристик спецификации символа, которые доступны в MQL5 API и подробно рассматриваемых в следующих разделах, есть несколько связанных с маржинальными (залоговыми) требованиями — они, как известно, предъявляются при открытии и поддержании торговых позиций. В связи с тем, что терминал обеспечивает торговлю на разных рынках, разными типами инструментов, эти требования могут существенно варьироваться. В обобщенном виде это выражается в применении поправочных коэффициентов маржи, задаваемых индивидуально для символов и разных типов торговых операций. Для пользователя коэффициенты выводятся в терминале в окне *Спецификации*.

Как мы увидим далее, коэффициент (если он применяется) умножается на величину маржи из свойств символа. Для получения коэффициента маржи программным способом существует функция *SymbolInfoMarginRate*.

bool SymbolInfoMarginRate(const string symbol, ENUM_ORDER_TYPE orderType, double &initial, double &maintenance)

Для указанного символа и типа приказа (ENUM_ORDER_TYPE) функция заполняет переданные по ссылке параметры *initial* и *maintenance* коэффициентами маржи — соответственно, начальной и

поддерживающей. Полученные коэффициенты следует умножить на величину маржи соответствующего типа (как её запросить, рассказано в разделе о маржинальных требованиях), чтобы получить размер средств, который будет зарезервирован на счете при размещении приказа типа *orderType*.

Функция возвращает *true* в случае удачного выполнения.

В качестве примера используем простой скрипт *SymbolMarginRate.mq5*, который выводит коэффициенты маржи для символов *Обзора рынка* или всех доступных, в зависимости от параметр *MarketWatchOnly*. Тип операции можно указать в параметре *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);
   }
}
```

Ниже приведен фрагмент журнала.

```
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
```

Вы можете сравнить полученные значения со спецификациями символов в терминале.

6.1.8 Обзор функций получения свойств символа

Полная спецификация каждого символа может быть получена путем запроса его свойств: для этой цели в MQL5 API предусмотрено 3 функции — *SymbolInfoInteger*, *SymbolInfoDouble*, *SymbolInfoString*, — каждая из которых ответственна за свойства конкретного типа. Сами свойства описаны как элементы трех перечислений — ENUM_SYMBOL_INFO_INTEGER, ENUM_SYMBOL_INFO_DOUBLE и ENUM_SYMBOL_INFO_STRING, соответственно. Аналогичный технический прием использован в уже знакомых нам API графиков и объектов.

В любую из функций передаются имя символа и идентификатор запрашиваемого свойства.

Каждая из функций представлена в 2-х формах — сокращенной и полной. Сокращенная непосредственно возвращает запрошенное свойство, а полная записывает его в выходной параметр, переданный по ссылке. Например, для свойств, совместимых с целым типом, функции имеют такие прототипы:

```
long SymbolInfoInteger(const string symbol, ENUM_SYMBOL_INFO_INTEGER property)
bool SymbolInfoInteger(const string symbol, ENUM_SYMBOL_INFO_INTEGER property, long &value)
```

Вторая форма возвращает логический признак успеха (*true*) или ошибки (*false*). К наиболее возможным причинам, по которым функция может вернуть *false*, относятся неправильное имя символа (MARKET_UNKNOWN_SYMBOL, 4301) или неверный идентификатор запрошенного свойства (MARKET_WRONG_PROPERTY, 4303) — их можно уточнить с помощью *_LastError*.

Как и ранее, свойства в перечислении ENUM_SYMBOL_INFO_INTEGER относятся к различным типам, совместимым с целым числом: *bool*, *int*, *long*, *color*, *datetime* и специальные перечисления (все они будут рассмотрены в отдельных разделах).

Для свойств с типом вещественных чисел определены следующие 2 формы функции *SymbolInfoDouble*.

```
double SymbolInfoDouble(const string symbol, ENUM_SYMBOL_INFO_DOUBLE property)
bool SymbolInfoDouble(const string symbol, ENUM_SYMBOL_INFO_DOUBLE property, double &value)
```

Наконец, для строковых свойств аналогичные функции выглядят так:

```
string SymbolInfoString(const string symbol, ENUM_SYMBOL_INFO_STRING property)
bool SymbolInfoString(const string symbol, ENUM_SYMBOL_INFO_STRING property, string &value)
```

Свойства различных типов, наиболее востребованные при будущей разработке экспертов, логически сгруппированы в описаниях следующих разделов этой главы.

На основе вышеописанных функций создадим универсальный класс *SymbolMonitor* (файл *SymbolMonitor.mqh*) для получения любых свойств символа. Его основу будет составлять набор перегруженных методов *get* для 3 перечислений.

```
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);
   }
   ...
```

Еще три похожих метода дают возможность избавиться от типа перечисления в первом параметре и выбирать компилятором нужную перегрузку за счет второго фиктивного параметра (его тип здесь всегда совпадает с типом результата). Мы воспользуемся этим в будущих классах-шаблонах.

```
   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);
   }
   ...
```

Таким образом, создав объект с нужным именем символа, можно единообразно запрашивать его свойства любых типов. Для запроса и вывода в журнал всех свойств одного типа мы могли бы реализовать примерно такой метод.

```
// проект (черновик)
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);
   }
}
```

Однако из-за того, что в свойствах типа *long* на самом деле "скрываются" значения других типов, которые желательно отображать специфическим образом (например, для перечислений — вызывать *EnumToString*, для даты и времени — *TimeToString* и т.д.), имеет смысл определить еще одну тройку перегруженных методов, которые возвращали бы строковое представление свойства. Назовем их *stringify*. Тогда в приведенном наброске *list2log* можно будет использовать *stringify* вместо приведения значений к *(string)*, а сам метод избавится от одного шаблонного параметра.

```
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));
   }
}
```

Для вещественного и строкового типов реализация *stringify* выглядит довольно прямолинейно.

```
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);
}
```

А вот для ENUM_SYMBOL_INFO_INTEGER все немного сложнее. Конечно, когда свойство имеет тип *long* или *int* его достаточно привести к *(string)*. Все остальные случаи требуется индивидуально анализировать и преобразовывать внутри оператора *switch*.

```
string stringify(const ENUM_SYMBOL_INFO_INTEGER property) const
{
   const long v = SymbolInfoInteger(name, property);
   switch(property)
   {
      ...
   }
   
   return (string)v;
}
```

Например, если свойство имеет логический тип, его удобно представить строкой "true" или "false" (таким образом, оно будет визуально отличаться от простых чисел 1 и 0). Немного забегая вперед, для примера скажем, что среди свойств имеется SYMBOL_EXIST, эквивалентное функции *SymbolExist*, то есть возвращающее логический признак того, существует ли указанный символ. Для его обработки и прочих логических свойств имеет смысл реализовать вспомогательный метод *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;
}
```

Для свойств, являющихся перечислениями, наиболее подходящим решением будет шаблонный метод, использующий функцию *EnumToString*.

```
template<typename E>
static string enumstr(const long v)
{
   return EnumToString((E)v);
}
```

Например, одно из свойств SYMBOL_SWAP_ROLLOVER3DAYS определяет, в какой день недели на открытые позиции по символу начисляется тройной своп, и это свойство имеет тип ENUM_DAY_OF_WEEK. Значит, для его обработки мы можем написать внутри *switch*:

```
      case SYMBOL_SWAP_ROLLOVER3DAYS:
         return enumstr<ENUM_DAY_OF_WEEK>(v);
```

Особый случай представляют свойства, значения которых являются комбинациями битовых флагов. В частности, по каждому символу брокером устанавливаются разрешения на приказы

конкретных типов, такие как рыночные, лимитные, стоплосс, тейкпрофит и другие (мы рассмотрим эти разрешения отдельно). Каждый тип приказов обозначается константой с взведенным одним битом, так что их суперпозиция (объединение битовым оператором ИЛИ '|') хранится в свойстве SYMBOL_ORDER_MODE, и в отсутствие ограничений взведены все биты одновременно. Для подобных свойств мы определим в нашем заголовочном файле собственные перечисления, например:

```
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,
};
```

Здесь для каждой встроенной константы, такой как SYMBOL_ORDER_MARKET, описан соответствующий элемент, идентификатор которого совпадает с константой, но предваряется символом подчеркивания, чтобы избежать конфликта имен.

Для представления комбинаций флагов из таких перечислений в виде строки мы реализуем еще один шаблонный метод *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;
}
```

Его суть похожа на *enumstr*, но функция *EnumToString* вызывается для каждого взведенного бита в значении свойства, после чего полученные строки "склеиваются".

Теперь обработка SYMBOL_ORDER_MODE в операторе *switch* возможна по похожей схеме:

```
         case SYMBOL_ORDER_MODE:
            return maskstr<SYMBOL_ORDER>(v);
```

Приведем полный код метода *stringify* для ENUM_SYMBOL_INFO_INTEGER. Со всеми свойствами и перечислениями мы поэтапно познакомимся в следующих разделах.

```
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: // Байты: Transparency Blue Green Red
         return StringFormat("TBGR(0x%08X)", v);
   }
   
   return (string)v;
}
```

Для проверки работы класса *SymbolMonitor* создан простой скрипт *SymbolMonitor.mq5*. Он выводит в журнал все свойства рабочего символа графика.

```
#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>();
}
```

Например, если запустить скрипт на графике EURUSD можем получить следующие записи (приведено с сокращениями).

```
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
```

В частности, видно, что цены символа транслируются с 5 знаками (SYMBOL_DIGITS), символ существует (SYMBOL_EXIST), размер контракта составляет 100000.0 (SYMBOL_TRADE_CONTRACT_SIZE), и т.д. Вся информация соответствует спецификации.

6.1.9 Проверка состояния символа

Ранее мы рассмотрели несколько функций, связанных с состоянием символа. Напомним, что для проверки существования символа используется — *SymbolExist*, а для включения или исключения из списка *Обзора рынка* — *SymbolSelect*. Среди свойств символа есть несколько аналогичных по назначению флагов, использование которых имеет как плюсы, так и минусы по сравнению с вышеупомянутыми функциями.

В частности, свойство SYMBOL_SELECT позволяет узнать, выбран ли указанный символ в *Обзоре рынка*, в то время как функция *SymbolSelect* меняет это свойство.

Функция *SymbolExist*, в отличие от похожего свойства SYMBOL_EXIST, дополнительно заполняет выходную переменную признаком того, что символ является пользовательским. При запросе свойств потребовалось бы анализировать эти два атрибута по отдельности, так как признак пользовательского символа хранится в другом свойстве SYMBOL_CUSTOM. Однако в некоторых случаях в программе может быть нужно лишь одно свойство, и тогда возможность отдельного запроса становится плюсом.

Все флаги представляют собой логические значения, получаемые через функцию *SymbolInfoInteger*.

Идентификатор	Описание
SYMBOL_EXIST	Признак того, что символ с заданным именем существует
SYMBOL_SELECT	Признак того, что символ выбран в *Обзор рынка*
SYMBOL_VISIBLE	Признак того, что заданный символ отображается в *Обзоре рынка*

Особый интерес представляет SYMBOL_VISIBLE. Дело в том, что некоторые символы (как правило, это кросс-курсы, которые необходимы для расчёта маржинальных требований и

прибыли в валюте депозита) выбираются в *Обзор рынка* автоматически и при этом не отображаются в списке, видимом пользователю. Для отображения такие символы должны быть выбраны явно (пользователем или программно). Таким образом, именно свойство SYMBOL_VISIBLE позволяет определить, виден ли символ в окне: оно может оказаться равным *false* у некоторых элементов списка, полученного с помощью пары функций *SymbolsTotal* и *SymbolName* с параметром *selected* равным *true*.

Рассмотрим простой скрипт (*SymbolInvisible.mq5*), который ищет в терминале неявным образом выбранные символы, то есть такие, которые не отображаются в Обзоре рынка (SYMBOL_VISIBLE сброшен), но тем не менее для них SYMBOL_SELECT равно *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[];
   // цикл по всем имеющимся символам
   for(int i = 0; i < n; ++i)
   {
      const string s = SymbolName(i, false);
      if(SymbolInfoInteger(s, SYMBOL_SELECT))
      {
         selected++;
         if(!SymbolInfoInteger(s, SYMBOL_VISIBLE))
         {
            // собираем в массив выбранные, но невидимые символы
            PUSH(invisible, s);
         }
      }
   }
   PrintFormat("Symbols: total=%d, selected=%d, implicit=%d",
      n, selected, ArraySize(invisible));
   if(ArraySize(invisible))
   {
      ArrayPrint(invisible);
   }
}
```

Попробуйте откомпилировать и запустить скрипт на разных счетах. Ситуация, когда символ выбран неявным образом встречается не всегда. Например, если в *Обзоре рынка* выбраны тикеры российских голубых фишек, которые котируются в рублях, а торговый счет в валюте (например, доллары или евро, но не рубли), то символ "USDRUB" будет выбран автоматически. Разумеется, при этом предполагается, что он не был ранее добавлен в *Обзор рынка* явным образом. Тогда получим в журнале результат такого вида:

```
Symbols: total=50681, selected=49, implicit=1
"USDRUB"
```

6.1.10 Тип цены для построения графиков по символу

Бары на ценовых графиках MetaTrader 5 могут строиться по ценам *Bid* или *Last* — это указано в спецификации каждого инструмента. MQL-программа может узнать эту характеристику посредством вызова функции *SymbolInfoInteger* для свойства SYMBOL_CHART_MODE. Возвращаемое значение является элементом перечисления ENUM_SYMBOL_CHART_MODE.

Идентификатор	Описание
SYMBOL_CHART_MODE_BID	Бары строятся по ценам Bid
SYMBOL_CHART_MODE_LAST	Бары строятся по ценам Last

Режим с ценами *Last* используется для символов, торгуемых на биржах (в отличие от децентрализованного рынка Forex) — для них доступен стакан цен. Глубину стакана можно узнать по свойству SYMBOL_TICKS_BOOKDEPTH.

Свойство SYMBOL_CHART_MODE пригодится для корректировки сигналов индикаторов или стратегий, которые строятся, например, по цене графика *Last*, в то время как торговые приказы будут выполняться "по рынку", то есть по ценам *Ask* или *Bid*, в зависимости от направления.

Также тип цены бывает востребован при расчете баров пользовательского инструмента: если тот зависит от стандартных инструментов, может иметь смысл учет их настроек по типу цены. Когда пользователь сам вводит формулу синтетического инструмента в окне *Собственный символ* (вызывается по нажатию кнопки *Создать символ* в диалоге *Символы*), он имеет возможность выбирать типы цен согласно спецификациям соответствующих использованных стандартных символов. Однако когда алгоритм расчета формируется в MQL-программе, именно она отвечает за правильный выбор типа цены.

Для начала соберем статистику по использованию цен *Bid* и *Last* для построения графиков на конкретном счете. Этим займется скрипт *SymbolStatsByPriceType.mq5*.

```
const bool MarketWatchOnly = false;

void OnStart()
{
   const int n = SymbolsTotal(MarketWatchOnly);
   int k = 0;
   // проходим по всем доступным символам
   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);
}
```

Попробуйте его на разных счетах (на некоторых может не быть биржевых символов). Вот как может выглядеть результат:

```
Symbols in total: 52304
Symbols using price types: Bid=229, Last=52075
```

Более практичным примером послужит индикатор *SymbolBidAskChart.mq5*, предназначенный для отрисовки диаграммы в виде баров, сформированных по цене заданного типа. Это позволит сравнить свечи графика, использующего для своего построения цены из свойства SYMBOL_CHART_MODE, с барами на альтернативном типе цен. Например, можно увидеть бары по цене *Bid* на графике инструмента по цене *Last*. Или получить бары по цене *Ask*, чего стандартные графики терминала не поддерживают.

В качестве основы нового индикатора возьмем уже готовый индикатор *IndDeltaVolume.mq5*, представленный в разделе Ожидание данных и управление видимостью. В том индикаторе мы закачивали для определенного количества баров *BarCount* тиковую историю и считали по ним дельту объемов, то есть раздельно объемы для покупок и продаж. В новом индикаторе нам будет достаточно заменить расчетный алгоритм на поиск цен *Open*, *High*, *Low*, *Close* по тикам внутри каждого бара.

Настройки индикатора включают 4 буфера и 1 диаграмму в виде баров (DRAW_BARS), выводимую в основном окне.

```
#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;"
```

Отображение в виде баров выбрано, чтобы облегчить их восприятие при наложении на свечи основного графика (то есть, чтобы были видны оба варианта каждого бара).

Во входные параметры добавлен *ChartMode*, чтобы пользователь мог выбрать один из трех типов цены (обратите внимание, что *Ask* является нашим расширением по сравнению со стандартным набором элементов в 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;
```

Бывший класс *CalcDeltaVolume* сменил название на *CalcCustomBars*, но остался почти без изменений. Среди отличий упомянем новый набор из 4-х буферов и поле *chartMode*, которое инициализируется в конструкторе из входной переменной *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) ...
   {
      // регистрируем массивы как индикаторные буфера
      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);
   }
   ...
```

В зависимости от режима *chartMode* вспомогательный метод *price* возвращает из каждого тика конкретный тип цены.

```
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; // ошибка
   }
   ...
```

С помощью метода *price* легко реализовать модификацию основного расчетного метода *calc*, где заполняются буфера для бара под номером *i* на основе массива тиков *ticks* для этого бара.

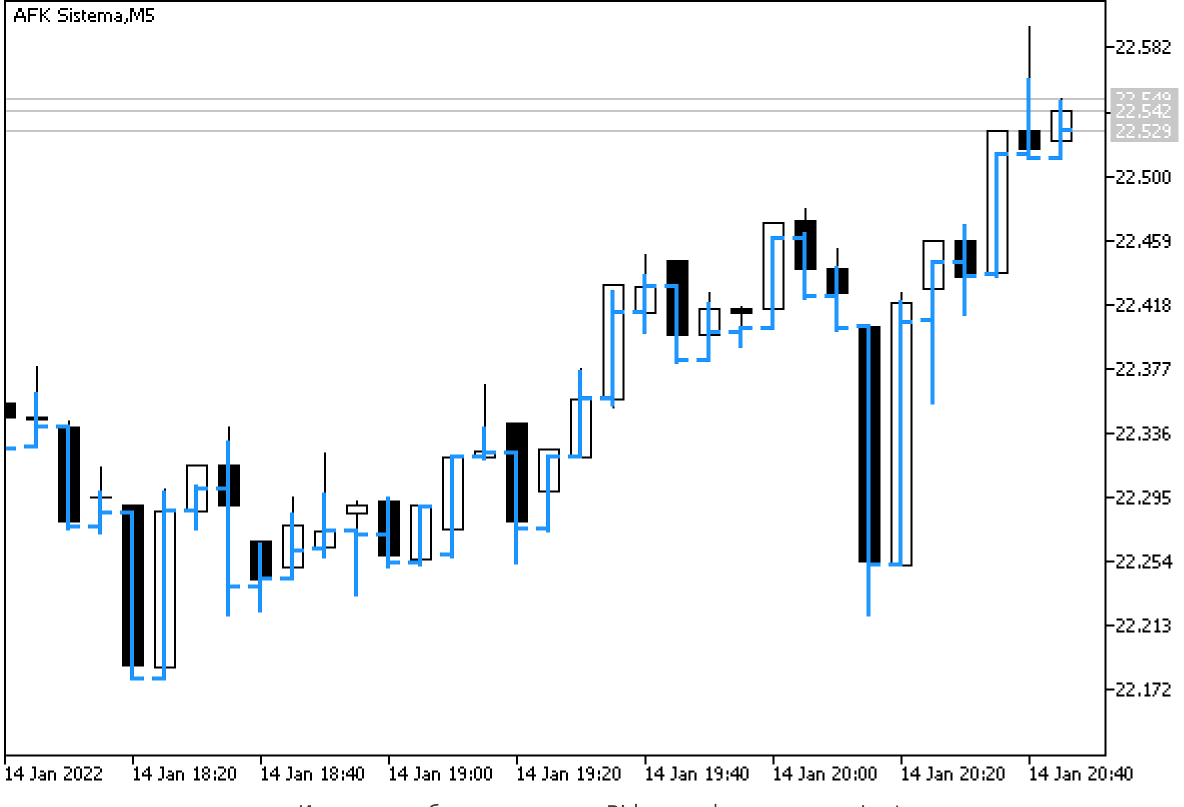
```
   void calc(const int i, const MqlTick &ticks[], const int skip = 0)
   {
      const int n = ArraySize(ticks);
      for(int j = skip; j < n; ++j)
      {
         const double p = price(ticks[j]);
         if(open[i] == EMPTY_VALUE)
         {
            open[i] = p;
         }
         
         if(p > high[i] || high[i] == EMPTY_VALUE)
         {
            high[i] = p;
         }
         
         if(p < low[i])
         {
            low[i] = p;
         }
         
         close[i] = p;
      }
   }
```

Остальные фрагменты исходного кода и принципы их работы соответствуют описанию *IndDeltaVolume.mq5*.

В обработчике *OnInit* мы дополнительно выводим текущий тип цены графика и выдаем предупреждение, если пользователь решил построить индикатор по типу цены *Last* для инструмента, где *Last* отсутствует.

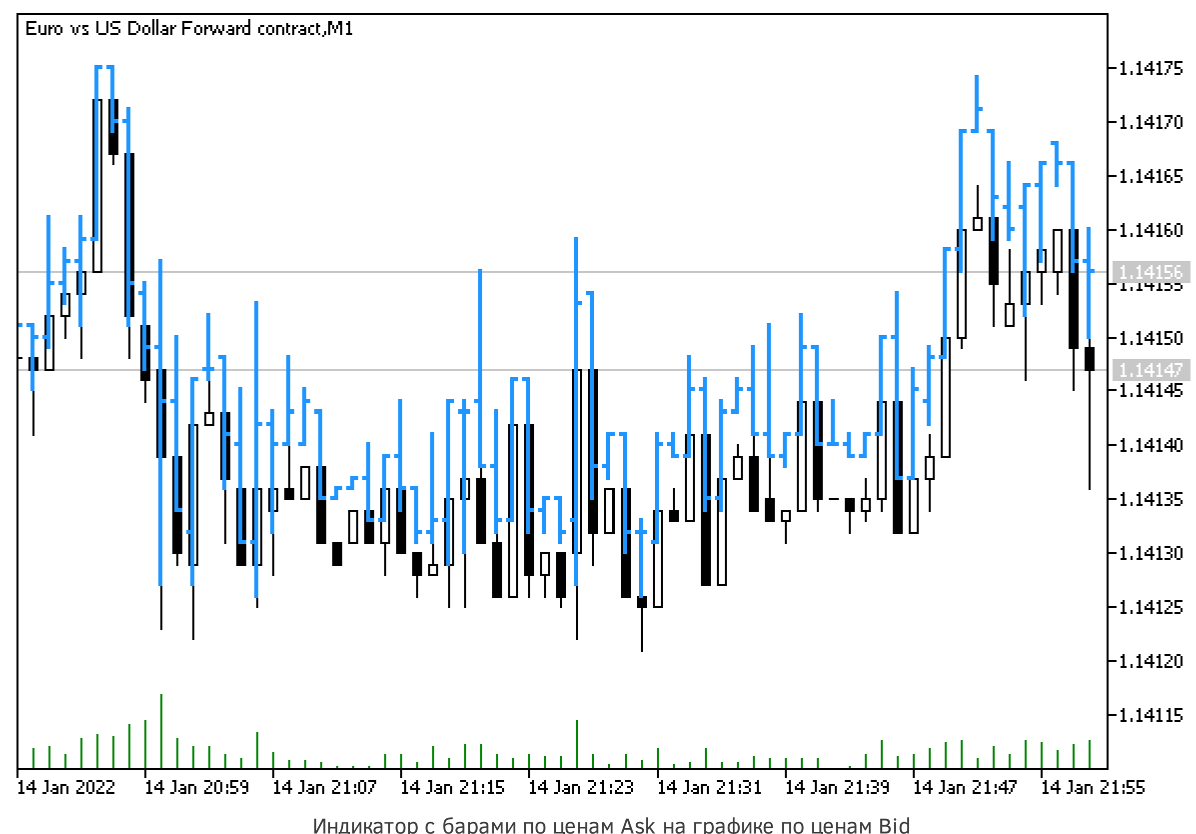
```
int OnInit()
{
   ...
   ENUM_SYMBOL_CHART_MODE mode =
      (ENUM_SYMBOL_CHART_MODE)SymbolInfoInteger(_Symbol, SYMBOL_CHART_MODE);
   Print("Chart mode: ", EnumToString(mode));

   if(mode == SYMBOL_CHART_MODE_BID
      && ChartMode == _SYMBOL_CHART_MODE_LAST)
   {
      Alert("Last price is not available for ", _Symbol);
   }

   return INIT_SUCCEEDED;
}
```

Ниже показана копия экрана для инструмента с режимом построения графика по цене *Last*, на который наложен индикатор с типом цены *Bid*.



Индикатор с барами по ценам Bid на графике по ценам Last

Также интересно посмотреть на бары по цене *Ask*, наложенные на обычный график по ценам *Bid*.



Индикатор с барами по ценам Ask на графике по ценам Bid

В часы низкой ликвидности, когда спред расширяется, можно заметить существенное отличие *Bid*- и *Ask*-графиков.

6.1.11 Базовая, котировочная и маржинальная валюты инструмента

Как известно, одними из самых важных свойств каждого финансового инструмента являются его рабочие валюты:

- базовая валюта, в которой выражается покупаемый или продаваемый актив (для инструментов Forex);
- валюта расчета прибыли (котирования);
- валюта расчета залога;

MQL-программа может получить названия этих валют с помощью функции *SymbolInfoString* и трех свойств из следующей таблицы.

Идентификатор	Описание
SYMBOL_CURRENCY_BASE	Базовая валюта
SYMBOL_CURRENCY_PROFIT	Валюта прибыли
SYMBOL_CURRENCY_MARGIN	Валюта, в которой вычисляются залоговые средства

С помощью них удобно анализировать инструменты Forex, в имена которых многие брокеры добавляют различные префиксы и суффиксы, а также биржевые инструменты. В частности,

алгоритм сможет найти символ для получения кросс-курса двух заданных валют или подобрать корзину индексов с заданной общей валютой котирования.

Поскольку поиск инструментов согласно неким требованиям является частой задачей, создадим класс *SymbolFilter* (*SymbolFilter.mqh*) для построения списка подходящих символов и их избранных свойств. В дальнейшем используем этот класс не только для анализа валют, но и других характеристик.

Сначала рассмотрим упрощенную версию, а потом дополним её удобным функционалом.

В разработке воспользуемся уже готовыми вспомогательными средствами: ассоциативным массивом-картой (*MapArray.mqh*) для хранения пар ключ-значение выбранных типов и монитором свойств символа (*SymbolMonitor.mqh*).

```
#include <MQL5Book/MapArray.mqh>
#include <MQL5Book/SymbolMonitor.mqh>
```

Для упрощения инструкций по накоплению результатов работы в массивах применим усовершенствованную версию маркоса PUSH, уже встречавшегося ранее в примерах, а также его вариант EXPAND для многомерных массивов (простое присваивание в этом случае невозможно).

```
#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)
```

Объект класса *SymbolFilter* должен иметь хранилище значений свойств, по которым программист желает фильтровать символы. Поэтому опишем в классе 3 массива *MapArray* для целочисленных, вещественных и строковых свойств.

```
class SymbolFilter
{
   MapArray<ENUM_SYMBOL_INFO_INTEGER,long> longs;
   MapArray<ENUM_SYMBOL_INFO_DOUBLE,double> doubles;
   MapArray<ENUM_SYMBOL_INFO_STRING,string> strings;
   ...
```

Установка требуемых свойств фильтра производится с помощью перегруженных методов *let*.

```
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;
   }
   ...
```

Обратите внимание, что методы возвращают указатель на сам фильтр, что позволяет записывать условия в виде цепочки: например, если ранее в коде был описан объекта *f* типа *SymbolFilter*, то для наложения двух условий на тип цены и название валюты прибыли можно написать:

```
f.let(SYMBOL_CHART_MODE, SYMBOL_CHART_MODE_LAST).let(SYMBOL_CURRENCY_PROFIT, "USD");
```

Формирование массива символов, удовлетворяющих условиям, выполняется объектом-фильтром в нескольких вариантах метода *select*, самый простой из которых представлен ниже (другие варианты рассмотрим позднее).

Параметр *watch* определяет контекст поиска символов: среди выбранных в *Обзор рынка* (*true*) или всех доступных (*false*). Выходной массив *symbols* будет заполнен названиями походящих символов. Структура кода внутри метода нам уже знакома: здесь организован цикл по символам, для каждого из которых создается объект-монитор *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);
         }
      }
   }
```

Именно с помощью монитора мы можем унифицированно получить значение любого свойства. Проверка соответствия свойств текущего символа с сохраненным набором условий в массивах

longs, *doubles*, *strings* делегируется вспомогательному методу *match*. Все запрошенные свойства должны совпасть, чтобы имя символа было сохранено в выходном массиве *symbols*.

В простейшем случае реализация метода *match* такова (впоследствии он будет изменен).

```
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;
   }
```

Если хоть одно из значений в массиве *data* не совпадает с соответствующим свойством символа, метод возвращает *false*. Если все свойства совпали (или условий для свойств данного типа нет), метод возвращает *true*.

Сравнение двух значений выполняется методом *equal*. Учитывая тот факт, что среди свойств могут быть свойства типа *double*, реализация не так проста, как можно было бы подумать.

```
template<typename V>
static bool equal(const V v1, const V v2)
{
   return v1 == v2 || eps(v1, v2);
}
```

Для типа *double* выражение *v1 == v2* может не сработать для близких чисел, в связи с чем следует учитывать точность вещественного типа DBL_EPSILON. Это делается в отдельном методе *eps*, перегруженном раздельно для типа *double* и всех остальных типов за счет шаблона.

```
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;
}
```

При равенстве значений любых типов кроме *double* шаблонный метод *eps* просто не будет вызван, а во всех иных случаях (в том числе и при различии значений) он возвращает *false*, как и требовалось (таким образом, работать будет только условие *v1 == v2*).

Вышеописанный вариант фильтра позволяет проверять свойства только на равенство значениям. Однако на практике часто требуется анализировать условия на неравенство, а также на

больше/меньше. В связи с этим в классе *SymbolFilter* определено перечисление *IS* с основными операциями сравнения (при желании его можно дополнить).

```
class SymbolFilter
{
   ...
   enum IS
   {
      EQUAL,
      GREATER,
      NOT_EQUAL,
      LESS
   };
   ...
```

Для каждого свойства из перечислений ENUM_SYMBOL_INFO_INTEGER, ENUM_SYMBOL_INFO_DOUBLE, ENUM_SYMBOL_INFO_STRING требуется сохранить не только искомое значение свойства (вспоминаем про ассоциативные массивы *longs*, *doubles*, *strings*), но и способ сравнения из нового перечисления *IS*.

Поскольку элементы стандартных перечислений имеют непересекающиеся значения (здесь есть одно исключение, связанное с объемами, но оно не критично), имеет смысл зарезервировать под способ сравнения один общий массив-карту *conditions*. При этом встает вопрос, какой тип выбрать для ключа карты, чтобы технически "объединить" разные перечисления. Для этого потребовалось описать фиктивное перечисление ENUM_ANY — оно лишь обозначает некий тип обобщенного перечисления. Напомним, что все перечисления имеют внутреннее представление, эквивалентное целому *int*, и потому могут приводиться одно к другому.

```
   enum ENUM_ANY
   {
   };

   MapArray<ENUM_ANY,IS> conditions;
   MapArray<ENUM_ANY,long> longs;
   MapArray<ENUM_ANY,double> doubles;
   MapArray<ENUM_ANY,string> strings;
   ...
```

Теперь мы можем дополнить все методы *let*, устанавливающие искомое значение свойства, входным параметром *cmp*, который определяет способ сравнения. По умолчанию он задает проверку на равенство (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;
   }
```

Здесь представлен вариант для целочисленных свойств. Остальные две перегрузки изменяются аналогично.

С учетом новой информации о разных способах сравнения и одновременного избавления от разных типов ключей в массивах-картах мы модифицируем метод *match*. В нем для каждого заданного свойства, по ключу в массиве-карте *data* извлекается условие из массива *conditions* и с помощью оператора *switch* выполняются соответствующие проверки.

```
template<typename V>
bool match(const SymbolMonitor &m, const MapArray<ENUM_ANY,V> &data) const
{
   // фиктивная переменная для выбора перегрузки метода m.get ниже
   static const V type = (V)NULL;
   // цикл по условиям, наложенным на свойства символа
   for(int i = 0; i < data.getSize(); ++i)
   {
      const ENUM_ANY key = data.getKey(i);
      // выбор способа сравнения в условии
      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;
}
```

Новый шаблонный метод *greater* реализован тривиально.

```
template<typename V>
static bool greater(const V v1, const V v2)
{
   return v1 > v2;
}
```

Теперь вызов метода *match* можно записать более кратко, так как единственный оставшийся тип шаблона *V* автоматически определяется по передаваемому аргументу *data* (а это один из массивов *longs*, *doubles* или *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);
         }
      }
   }
```

Это еще не окончательная версия класса *SymbolFilter*, но мы уже можем проверить его в действии.

Создадим скрипт *SymbolFilterCurrency.mq5*, способный фильтровать символы по свойствам базовой валюты и валюты прибыли — в данном случае "USD". Параметр *MarketWatchOnly* по умолчанию задает поиск только по *Обзору рынка*.

```
#include <MQL5Book/SymbolFilter.mqh>

input bool MarketWatchOnly = true;

void OnStart()
{
   SymbolFilter f;   // объект фильтра
   string symbols[]; // массив для результатов
   ...
```

Допустим, что мы хотим найти инструменты Forex, которые имеют прямые котировки, то есть в их названиях "USD" фигурирует в начале. Чтобы не зависеть от особенностей формирования названий у конкретного брокера, воспользуемся свойством SYMBOL_CURRENCY_BASE. Именно оно содержит первую валюту.

Запишем условие, чтобы базовая валюта символа равнялась "USD", и применим фильтр.

```
   f.let(SYMBOL_CURRENCY_BASE, "USD")
   .select(MarketWatchOnly, symbols);
   Print("===== Base is USD =====");
   ArrayPrint(symbols);
   ...
```

Полученный массив выводится в журнал.

```
===== Base is USD =====
"USDCHF" "USDJPY" "USDCNH" "USDRUB" "USDCAD" "USDSEK" "SP500m" "Brent"
```

Как видно, в массив попали не только Forex-символы, у которых "USD" идет в начале тикера, но также индекс S&P500 и товар (нефть). Два последних символа котируются в долларах, но и базовая валюта у них такая же. В то же время у Forex-символов валюта котирования (она же

валюта прибыли) идет второй и отличается от "USD". Это позволяет дополнить фильтр таким образом, чтобы не-Forex-символы перестали ему соответствовать.

Очистим массив, добавим в фильтр условие, чтобы валюта прибыли не равнялась "USD", и снова запросим подходящие символы (прежнее условие сохранилось в объекте *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);
}
```

На этот раз в журнале выведены действительно только искомые символы.

```
===== Base is USD and Profit is not USD =====
"USDCHF" "USDJPY" "USDCNH" "USDRUB" "USDCAD" "USDSEK"
```

6.1.12 Точность представления и шаг изменения цен

Ранее мы уже встречали два взаимосвязанных свойства рабочего символа графика: минимальный шаг изменения его цены (*Point*) и точность представления цен как количество десятичных знаков после запятой (*Digits*). Они же доступны среди Предопределенных переменных. Для получения аналогичных свойств произвольного символа следует запросить свойства SYMBOL_POINT и SYMBOL_DIGITS, соответственно. Со свойством SYMBOL_POINT тесно связан минимальный размер изменения цены (известный для MQL-программы как свойство SYMBOL_TRADE_TICK_SIZE) и его стоимость (SYMBOL_TRADE_TICK_VALUE), как правило, в валюте торгового счета (но некоторые символы могут быть настроены на учет в базовой валюте — при необходимости уточните этот момент у брокера). Ниже в таблице приведена вся группа этих свойств.

Идентификатор	Описание
SYMBOL_DIGITS	Количество знаков после запятой
SYMBOL_POINT	Значение одного пункта в валюте котировки
SYMBOL_TRADE_TICK_VALUE	Значение SYMBOL_TRADE_TICK_VALUE_PROFIT
SYMBOL_TRADE_TICK_VALUE_PROFIT	Текущая стоимость тика для прибыльной позиции
SYMBOL_TRADE_TICK_VALUE_LOSS	Текущая стоимость тика для убыточной позиции
SYMBOL_TRADE_TICK_SIZE	Минимальное изменение цены в валюте котировки

Все свойства кроме SYMBOL_DIGITS являются вещественными числами и запрашиваются функцией *SymbolInfoDouble*. Свойство SYMBOL_DIGITS доступно через *SymbolInfoInteger*. Но для проверки работы с этими свойствами мы воспользуемся уже готовыми классами *SymbolFilter* и задействованным в нем *SymbolMonitor* — они автоматически вызовут нужную функцию для любого свойства.

Попутно усовершенствуем класс *SymbolFilter*: добавим новую перегрузку метода *select*, который мог бы заполнять не только массив с названиями подходящих символов, но и еще один массив со значениями их конкретного свойства.

В более общем случае нас могут интересовать по каждому символу сразу несколько свойств, поэтому желательно использовать для выходного массива не один из встроенных типов данных, а специальный композитный тип с разными полями.

В программировании такие типы называются кортежами (tuple) и в некоторой степени эквиваленты структурам MQL5.

```
template<typename T1,typename T2,typename T3> // можем описать до 64-х полей
struct Tuple3                                 // MQL5 позволяет 64 параметра шаблона
{
   T1 _1;
   T2 _2;
   T3 _3;
};
```

Однако структуры требуют предварительного описания со всеми полями, в то время как нам не известно заранее количество и перечень запрашиваемых свойств символа. Поэтому в целях упрощения кода представим наш "кортеж", как вектор во втором измерении динамического массива — массива принимающего результаты запроса.

```
T array[][S];
```

В качестве типа данных *T* мы можем использовать любой из встроенных типов и перечислений, используемых для свойств. Размер *S* должен совпадать с количеством запрашиваемых свойств.

Правда, такое упрощение ограничивает нас в одном запросе значениями одинаковых типов, то есть только целочисленными, только вещественными или только строковыми. Однако условия фильтрации могут включать любые свойства. Подход с кортежами мы реализуем чуть позже, на примере фильров других торговых сущностей: ордеров, сделок и позиций.

Итак, новая версия метода *SymbolFilter::select* принимает в качестве входного параметра ссылку на массив *property* с идентификаторами свойств, которые следует прочитать у прошедших фильтрацию символов. Сами названия символов и значения этих свойств будут записаны в выходные массивы *symbols* и *data*.

```
template<typename E,typename V>
bool select(const bool watch, const E &property[], string &symbols[],
   V &data[][], const bool sort = false) const
{
   // размер массива запрашиваемых свойств должен совпадать с выходным кортежем
   const int q = ArrayRange(data, 1);
   if(ArraySize(property) != q) return false;

   const int n = SymbolsTotal(watch);
   // перебираем символы
   for(int i = 0; i < n; ++i)
   {
      const string s = SymbolName(i, watch);
      // доступ к свойствам символа обеспечивает монитор
      SymbolMonitor m(s);
      // проверяем все условия фильтра
      if(match(m, longs)
      && match(m, doubles)
      && match(m, strings))
      {
         // свойства подходящего символа записываем в массивы
         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;
}
```

Дополнительно новый метод умеет сортировать выходной массив по первому измерению (первому запрошенному свойству): этот функционал оставлен для самостоятельного изучения по исходным кодам. Для включения сортировки установите параметр *sort* в *true*. Массивы с именами символов и данными сортируются согласованно.

Чтобы обойтись без кортежей в вызывающем коде, когда требуется запросить только одно свойство у отфильтрованных символов, в *SymbolFilter* реализован следующий вариант *select*: внутри него определены промежуточные массивы свойств (*properties*) и значений (*tuples*) с размером 1 во второй размерности, которые используются для вызова вышеприведенной "полной" версии *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;
}
```

С помощью усовершенствованного фильтра попробуем построить список символов, отсортированный по стоимости тика SYMBOL_TRADE_TICK_VALUE (см. файл *SymbolFilterTickValue.mq5*). В предположении, что валюта депозита USD, мы должны получить стоимость равную 1.0 для Forex-инструментов, котирование которых производится в USD (вида XXXUSD). Для других активов увидим нетривиальные значения.

```
#include <MQL5Book/SymbolFilter.mqh>

input bool MarketWatchOnly = true;

void OnStart()
{
   SymbolFilter f;       // объект фильтра
   string symbols[];     // массив с названиями символов
   double tickValues[]; // массив для результатов

   // применяем фильтр без условий, заполняем и сортируем массив
   f.select(MarketWatchOnly, SYMBOL_TRADE_TICK_VALUE, symbols, tickValues, true);

   PrintFormat("===== Tick values of the symbols (%d) =====",
      ArraySize(tickValues));
   ArrayPrint(symbols);
   ArrayPrint(tickValues, 5);
}
```

Вот результат запуска скрипта.

```
===== 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 Разрешенные объемы торговых операций

В следующих главах, посвященных программированию торговых советников, нам потребуется контролировать множество характеристик символов, от которых зависит успех отправки торговых приказов. В частности, это относится к той части спецификации символа, где оговариваются разрешенные объемы операций. Соответствующие свойства доступны и в MQL5. Все они имеют тип *double* и запрашиваются функцией *SymbolInfoDouble*.

Идентификатор	Описание
SYMBOL_VOLUME_MIN	Минимальный объем сделки в лотах
SYMBOL_VOLUME_MAX	Максимальный объем сделки в лотах
SYMBOL_VOLUME_STEP	Минимальный шаг изменения объема сделки в лотах
SYMBOL_VOLUME_LIMIT	Максимально допустимый совокупный объем открытой позиции и отложенных ордеров в одном направлении (покупка или продажа)
SYMBOL_TRADE_CONTRACT_SIZE	Размер торгового контракта = 1 лот

Попытки купить или продать финансовый инструмент объемом меньше минимального, больше максимального или не кратным шагу приведут к ошибке. В главе про торговые API мы реализуем код для унификации необходимых проверок и нормализации объемов перед вызовом торговых функций MQL5 API.

Кроме всего прочего MQL-программе следует проверять и SYMBOL_VOLUME_LIMIT. Например, при ограничении в 5 лотов можно иметь открытую позицию на покупку объемом 5 лотов и выставить отложенный ордер *Sell Limit* объемом 5 лотов. Но при этом нельзя выставить отложенный ордер *Buy Limit* (поскольку совокупный объем в одном направлении превысит ограничение) или выставить *Sell Limit* объемом более 5 лотов.

В качестве вводного примера рассмотрим скрипт *SymbolFilterVolumes.mq5*, который выводит в журнал значения вышеперечисленных свойств для выбранных символов. Во входные параметры добавим переменную *MinimalContractSize*, чтобы можно было фильтровать символы по свойству SYMBOL_TRADE_CONTRACT_SIZE: выводим только те, у которых размер контракта больше указанного (по умолчанию, 0, то есть все символы удовлетворяют условию).

```
#include <MQL5Book/SymbolFilter.mqh>

input bool MarketWatchOnly = true;
input double MinimalContractSize = 0;
```

В начале *OnStart* определим объект фильтра, выходные массивы для получения списков имен и значений свойств в виде векторов *double* на 4 поля. Перечень интересующих нас 4-х свойств указываем в массиве *volumeIds*.

```
void OnStart()
{
   SymbolFilter f;                      // объект фильтра
   string symbols[];                    // приемный массив с именами
   double volumeLimits[][4];            // приемный массив с векторами данных
   
   // запрашиваемые свойства символов
   ENUM_SYMBOL_INFO_DOUBLE volumeIds[] =
   {
      SYMBOL_VOLUME_MIN,
      SYMBOL_VOLUME_STEP,
      SYMBOL_VOLUME_MAX,
      SYMBOL_VOLUME_LIMIT
   };
   ...
```

Далее применяем фильтр по размеру контракта (должен быть больше указанного) и получаем поля спецификации, связанные с объемами, для подошедших символов.

```
   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);
   }
}
```

Для настроек по умолчанию скрипт может показать примерно следующие результаты (с сокращениями).

```
===== 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
```

Некоторые символы могут не иметь ограничений по SYMBOL_VOLUME_LIMIT (значение равно 0). Вы можете сравнить результаты со спецификациями символов: они должны совпасть.

6.1.14 Разрешения на торговлю

Продолжая тему правильной подготовки торговых приказов, которую мы начали в предыдущем разделе, обратимся к следующей паре свойств, играющих весьма важную роль при разработке экспертов.

Идентификатор	Описание
SYMBOL_TRADE_MODE	Разрешения для разных торговых режимов по символу (см. ENUM_SYMBOL_TRADE_MODE далее)
SYMBOL_ORDER_MODE	Флаги разрешенных типов приказов, битовая маска (см. далее)

Оба свойства имеют целочисленный тип и доступны через функцию *SymbolInfoInteger*.

Свойство SYMBOL_TRADE_MODE мы уже задействовали в скрипте *SymbolPermissions.mq5*. Его значением является один из элементов перечисления ENUM_SYMBOL_TRADE_MODE.

Идентификатор	Значение	Описание
SYMBOL_TRADE_MODE_DISABLED	0	Торговля по символу запрещена
SYMBOL_TRADE_MODE_LONGONLY	1	Разрешены только покупки
SYMBOL_TRADE_MODE_SHORTONLY	2	Разрешены только продажи
SYMBOL_TRADE_MODE_CLOSEONLY	3	Разрешены только операции закрытия позиций
SYMBOL_TRADE_MODE_FULL	4	Нет ограничений на торговые операции

Напомним, что в классе *Permissions* есть метод *isTradeOnSymbolEnabled*, который проверяет несколько аспектов, влияющих на доступность торговли по символу, и один из них — свойство SYMBOL_TRADE_MODE. По умолчанию мы считаем, что нас интересует полный доступ к торговле, то есть продажи и покупки: SYMBOL_TRADE_MODE_FULL. В зависимости от торговой стратегии MQL-программа может рассматривать достаточными, например, разрешения только на покупку, только на продажу или только на закрытие операций.

```
static bool isTradeOnSymbolEnabled(string symbol, const datetime now = 0,
   const ENUM_SYMBOL_TRADE_MODE mode = SYMBOL_TRADE_MODE_FULL)
{
   // проверка сессий
   bool found = now == 0;
   ...
   // проверка режима торговли для символа
   return found && (SymbolInfoInteger(symbol, SYMBOL_TRADE_MODE) == mode);
}
```

В дополнение к режиму торгов нам в будущем потребуется анализировать и разрешения на приказы разных типов: они обозначаются отдельными битами в свойстве SYMBOL_ORDER_MODE и могут быть произвольным образом скомбинированы логическим ИЛИ ('|'). Например, значение 127 (0x7F) соответствует всем взведенным битам, то есть доступность всех типов приказов.

Идентификатор	Значение	Описание
SYMBOL_ORDER_MARKET	1	Разрешены рыночные ордера (Buy и Sell)
SYMBOL_ORDER_LIMIT	2	Разрешены лимитные ордера (Buy Limit и Sell Limit)
SYMBOL_ORDER_STOP	4	Разрешены стоп-ордера (Buy Stop и Sell Stop)
SYMBOL_ORDER_STOP_LIMIT	8	Разрешены стоп-лимит ордера (Buy Stop Limit и Sell Stop Limit)
SYMBOL_ORDER_SL	16	Разрешена установка Stop Loss
SYMBOL_ORDER_TP	32	Разрешена установка Take Profit
SYMBOL_ORDER_CLOSEBY	64	Разрешение на закрытие позиции с помощью встречной — то есть открытой в противоположном направлении по тому же инструменту.

Свойство SYMBOL_ORDER_CLOSEBY задается только для счетов с хеджинговой системой учета (ACCOUNT_MARGIN_MODE_RETAIL_HEDGING, см. Тип счета).

В тестовом скрипте *SymbolFilterTradeMode.mq5* запросим пару описываемых свойств для видимых символов *Обзора рынка*. Вывод битов и их сочетаний в виде чисел не очень информативен, поэтому воспользуемся тем, что в классе *SymbolMonitor* у нас есть удобный метод *stringify* для печати элементов перечислений и битовых масок всех свойств.

```
void OnStart()
{
   SymbolFilter f;                      // объект фильтра
   string symbols[];                    // массив под имена
   long permissions[][2];               // массив для данных (значений свойств)

   // перечень запрашиваемых свойств символов
   ENUM_SYMBOL_INFO_INTEGER modes[] =
   {
      SYMBOL_TRADE_MODE,
      SYMBOL_ORDER_MODE
   };

   // применим фильтр, получим массивы с результатами
   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)
      {
         // вывод описаний битов и чисел "как есть"
         PrintFormat("  %s (%d)",
            SymbolMonitor::stringify(permissions[i][j], modes[j]),
            permissions[i][j]);
      }
   }
}
```

Ниже с сокращениями приводится фрагмент журнала, получившегося в результате запуска скрипта.

```
===== 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)
```

Обратите внимание, что по последнему символу "SP500m" торговля полностью запрещена (его котировки поставляются только как "индикативные"). При этом его набор флагов по типам приказов отличен от 0, но не играет роли.

В зависимости от событий на рынке брокер может менять свойства символа по своему усмотрению, например, оставить на какое-то время только возможность закрывать позиции, поэтому корректный торговый робот обязан контролировать данные свойства перед каждой операцией.

6.1.15 Торговые условия и режимы исполнения приказов по символу

В данном разделе мы еще больше погрузимся в аспекты автоматизации торговли, зависящие от настроек финансовых инструментов. Пока мы изучим только сами свойства, а их использование на практике начнем в следующих главах. Предполагается, что читатель уже знаком с базовой терминологией, такой как рыночный и отложенный ордер, сделка, позиция.

При отправке торгового запроса на исполнение необходимо учитывать, что на финансовых рынках нет гарантии того, что в конкретный момент по данному финансовому инструменту доступен весь запрашиваемый объем по желаемой цене. Поэтому проведение торговых операций в реальном времени регулируется с помощью режимов исполнения по цене и объему. Режимы, или по-другому политики исполнения, определяют правила для случаев, когда изменилась цена или запрашиваемый объем нельзя выполнить полностью в текущий момент.

В MQL5 API данные режимы доступны для каждого символа как следующие свойства, получаемые через функцию *SymbolInfoInteger*.

Идентификатор	Описание
SYMBOL_TRADE_EXEMODE	Режим заключения сделок по цене
SYMBOL_FILLING_MODE	Флаги разрешенных режимов заливки ордера по объему (битовая маска, см. далее)

Значение свойства SYMBOL_TRADE_EXEMODE является элементом перечисления ENUM_SYMBOL_TRADE_EXECUTION.

Идентификатор	Описание
SYMBOL_TRADE_EXECUTION_REQUEST	Торговля по предварительно запрошенной цене
SYMBOL_TRADE_EXECUTION_INSTANT	Немедленное исполнение (торговля по потоковым ценам)
SYMBOL_TRADE_EXECUTION_MARKET	Исполнение ордеров по рынку
SYMBOL_TRADE_EXECUTION_EXCHANGE	Биржевое исполнение

Все или большинство этих режимов должно быть известно пользователям терминала по выпадающему списку *Тип* в диалоге *Новый ордер* (F9). Напомним вкратце, что они означают, а за подробностями следует обратиться к документации по терминалу.

- Исполнение по запросу (SYMBOL_TRADE_EXECUTION_REQUEST) — исполнение рыночного ордера по цене, предварительно полученной от брокера. Перед отправкой рыночного ордера у брокера запрашивается актуальная цена, после получения которой выполнение ордера по ней можно либо подтвердить, либо отклонить.
- Немедленное исполнение (SYMBOL_TRADE_EXECUTION_INSTANT) — исполнение рыночного ордера по текущей цене. При отправке торгового запроса на исполнение терминал автоматически подставляет в ордер текущие цены. Если брокер принимает цену, то ордер будет исполнен. Если брокер не принимает запрошенную цену, то происходит "реквотирование" (Requote) — брокер возвращает цены, по которым может быть исполнен данный ордер.
- Исполнение по рынку (SYMBOL_TRADE_EXECUTION_MARKET) — цену исполнения подставляет в приказ брокер без дополнительного согласования с трейдером. Отправка рыночного ордера в таком режиме подразумевает досрочное согласие с ценой, по которой он будет выполнен.
- Биржевое исполнение (SYMBOL_TRADE_EXECUTION_EXCHANGE) — торговые операции совершаются по ценам текущих рыночных предложений.

Что касается битов в SYMBOL_FILLING_MODE, которые могут комбинироваться оператором логического ИЛИ ('|'), то их наличие или отсутствие обозначает следующие действия.

Идентификатор	Значение	Политика заполнения
SYMBOL_FILLING_FOK	1	Все или Ничего (Fill Or Kill) — ордер должен быть исполнен исключительно в указанном объеме или отменен
SYMBOL_FILLING_IOC	2	Немедленно и хотя бы Частично (Immediate or Cancel) — сделка по максимально доступному на рынке объему в пределах указанного в ордере или отмена
(Идентификатор отсутствует)	(любое, в т.ч. 0)	Вернуть (Return) — в случае частичного исполнения рыночный или лимитный ордер с остаточным объемом не снимается, а продолжает действовать

Возможность использования режимов FOK и IOC определяется торговым сервером.

Если разрешен режим SYMBOL_FILLING_FOK, то MQL-программа сможет для отправки приказа с помощью функции *OrderSend* использовать в структуре *MqlTradeRequest* созвучный режиму способ заполнения ордера — ORDER_FILLING_FOK. Если при этом на рынке не окажется достаточного объема финансового инструмента, то ордер не будет исполнен. Следует учитывать, что необходимый объем может быть составлен из нескольких предложений, доступных в данный момент на рынке, в результате чего получится несколько сделок.

Если разрешен режим SYMBOL_FILLING_IOC, MQL-программе становится доступен одноименный способ заливки приказов ORDER_FILLING_IOC (он также указывается в специальном поле "заливки" (*type_filling*) в структуре *MqlTradeRequest* перед отправкой приказа функцией *OrderSend*). При использовании этого режима в случае невозможности полного исполнения ордер будет исполнен на доступный объем, а оставшийся объем ордера будет отменен.

Последняя политика без идентификатора является режимом по умолчанию и доступна вне зависимости от других режимов (именно поэтому она соответствует нулю или любому другому значению). Иными словами, даже если мы получим для свойства SYMBOL_FILLING_MODE значение 1 (SYMBOL_FILLING_FOK), 2 (SYMBOL_FILLING_IOC) или 3 (SYMBOL_FILLING_FOK | SYMBOL_FILLING_IOC), режим "вернуть" будет подразумеваться. Для использования этой политики следует при формировании ордера (заполнении структуры *MqlTradeRequest*) указывать тип заливки ORDER_FILLING_RETURN.

> Среди всех режимов SYMBOL_TRADE_EXEMODE есть одна особенность, касающаяся исполнения по рынку (SYMBOL_TRADE_EXECUTION_MARKET): Return-ордера всегда запрещены в режиме исполнения по рынку.

Поскольку ORDER_FILLING_FOK соответствует константе 0, отсутствие явного указания типа заливки в торговом запросе будет подразумевать именно этот режим.

Все эти нюансы мы рассмотрим на практике при разработке экспертов, а пока проверим чтение свойств в простом скрипте *SymbolFilterExecMode.mq5*.

```
#include <MQL5Book/SymbolFilter.mqh>

void OnStart()
{
   SymbolFilter f;                      // объект фильтра
   string symbols[];                    // массив названий символов
   long permissions[][2];               // массив с векторами значений свойств

   // свойства для чтения
   ENUM_SYMBOL_INFO_INTEGER modes[] =
   {
      SYMBOL_TRADE_EXEMODE,
      SYMBOL_FILLING_MODE
   };
   // применяем фильтр - заполняем массивы
   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)
      {
         // выводим свойства в виде описаний и чисел
         PrintFormat("  %s (%d)",
            SymbolMonitor::stringify(permissions[i][j], modes[j]),
            permissions[i][j]);
      }
   }
}
```

Ниже приведен фрагмент журнала с результатами работы скрипта. Здесь практически все символы имеют режим немедленного исполнения по ценам (SYMBOL_TRADE_EXECUTION_INSTANT) за исключением последнего SP500m (SYMBOL_TRADE_EXECUTION_MARKET). Заполнение объемов встречается всех типов — как отдельные SYMBOL_FILLING_FOK, SYMBOL_FILLING_IOC, так и их комбинация. Только для BTCUSD указан SYMBOL_FILLING_RETURN, то есть было получено значение равное 0 (отсутствуют биты FOK и 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)
```

Напомним, что подчеркивания в идентификаторах режимов заливки появляются из-за того, что нам пришлось для визуализации констант (битов в маске SYMBOL_FILLING_MODE) определить собственное перечисление SYMBOL_FILLING (*SymbolMonitor.mqh*) с элементами, равными константам. Это было сделано, поскольку MQL5 не имеет подобного встроенного перечисления, но при этом мы не можем называть элементы своего перечисления точно так же, как встроенные константы — это вызвало бы конфликт имен.

6.1.16 Маржинальные требования

Наиболее востребованная для трейдера информация о финансовом инструменте — это размер денежных средств, необходимых для открытия позиции. Не зная того, сколько денег необходимо для покупки или продажи заданного количества лотов, в эксперте нельзя реализовать систему управления капиталом и контролировать состояние счета.

Поскольку MetaTrader 5 используется для торговли различными инструментами (валюты, товары, акции, облигации, опционы, фьючерсы) принципы расчета залога существенно различаются. В документации приведены подробности, в частности, для Forex и фьючерсов, а также биржи.

Несколько свойств MQL5 API позволяют определить разновидность рынка и способ расчета маржи для конкретного инструмента.

Забегая вперед, скажем, что для заданного сочетания таких параметров как тип торговой операции, инструмент, объем и цена, MQL5 позволяет вычислить маржу с помощью функции *OrderCalcMargin* — это наиболее простой способ, но он имеет существенное ограничение: функция не учитывает текущие открытые позиции и отложенные ордера. Тем самым, в частности, игнорируются возможные поправки на перекрывающиеся объемы, когда на счете разрешены встречные позиции.

Таким образом, чтобы получить детализацию текущих занятых под залог средств счета в разбивке по открытым позициям и ордерам, от MQL-программы может потребоваться анализ нижеприведенных свойств и вычисления по формулам. Кроме того, функция *OrderCalcMargin* запрещена для использования в индикаторах. Заранее оценить свободную маржу после совершения предполагаемой сделки можно с помощью *OrderCheck*.

Идентификатор	Описание
SYMBOL_TRADE_CALC_MODE	Способ вычисления залога и прибыли (см. ENUM_SYMBOL_CALC_MODE)
SYMBOL_MARGIN_HEDGED_USE_LEG	Логический флаг включения (true) или отключения (false) режима расчета хеджированной маржи по наибольшей из перекрытых позиций (покупки и продажи)
SYMBOL_MARGIN_INITIAL	Начальная маржа для биржевого инструмента
SYMBOL_MARGIN_MAINTENANCE	Поддерживающая маржа для биржевого инструмента
SYMBOL_MARGIN_HEDGED	Размер контракта или маржи для одного лота перекрытых позиций (разнонаправленные позиции по одному символу)

Первые два свойства входят в перечисление ENUM_SYMBOL_INFO_INTEGER, а последние три — в ENUM_SYMBOL_INFO_DOUBLE, и их можно прочитать, соответственно, функциями *SymbolInfoInteger* и *SymbolInfoDouble*.

Конкретные формулы расчета маржи зависят от свойства SYMBOL_TRADE_CALC_MODE и приведены в таблице ниже. Более полную информацию можно найти в документации MQL5.

Обратите внимание, что начальная и поддерживающая маржа не используются для инструментов Forex и для них эти свойства всегда равны 0.

Начальная маржа обозначает размер необходимых залоговых средств в маржинальной валюте для открытия позиции объемом в один лот. Она используется при проверке достаточности средств клиента перед входом в рынок. Для получения окончательного размера взимаемой маржи в зависимости от типа и направления ордера следует проверить коэффициенты маржи с помощью функции *SymbolInfoMarginRate* — таким образом брокер может задать индивидуальное "плечо" или дисконт для каждого инструмента.

Поддерживающая маржа указывает минимальное значение средств в маржинальной валюте инструмента для поддержания открытой позиции в один лот. Она используется при проверке достаточности средств клиента при изменении состояния счета (торговых условий). Если уровень средств упадет ниже суммы поддерживающей маржи всех позиций, брокер начнет их принудительно закрывать.

Если свойство с поддерживающей маржой равно 0, то используется начальная маржа. Как и в случае начальной маржи, для получения окончательного размера взимаемой маржи в зависимости от типа и направления следует проверить коэффициенты маржи с помощью функции *SymbolInfoMarginRate*.

Перекрытые позиции, то есть разнонаправленные позиции по одному и тому же символу, могут существовать только на торговом счету с хеджированием. Очевидно, что и расчет хеджированной маржи вместе со свойствами SYMBOL_MARGIN_HEDGED_USE_LEG, SYMBOL_MARGIN_HEDGED имеют смысл только на таких счетах. Хеджированная маржа применяется для перекрытого объема

Брокер может выбрать для каждого инструмента один из двух существующих способов расчета маржи для перекрытых позиций:

- Базовый расчет применяется, когда режим расчета по наибольшей стороне отключен, то есть свойство SYMBOL_MARGIN_HEDGED_USE_LEG равно *false*. В этом случае маржа складывается из 3 компонентов: залог за неперекрытый объем существующей позиции, залог за перекрытый объем (если есть встречные позиции и свойство SYMBOL_MARGIN_HEDGED ненулевое), залог для отложенных ордеров. Если при этом для инструмента задана первоначальная маржа (свойство SYMBOL_MARGIN_INITIAL ненулевое), то хеджированная маржа указывается как абсолютное значение (в деньгах). Если первоначальная маржа не задана (равна 0), то в SYMBOL_MARGIN_HEDGED указывается размер контракта, который будет использован при расчете маржи по формуле, соответствующей типу торгового инструмента (SYMBOL_TRADE_CALC_MODE).
- Расчет по наибольшей позиции применяется, когда свойство SYMBOL_MARGIN_HEDGED_USE_LEG равно *true*. При этом значение SYMBOL_MARGIN_HEDGED не учитывается. Вместо этого вычисляется объем всех коротких и всех длинных позиций по инструменту, для каждой стороны рассчитывается средневзвешенная цена открытия. Далее по формулам, соответствующим типу инструмента (SYMBOL_TRADE_CALC_MODE), рассчитывается маржа для короткой и для длинной стороны. В качестве итогового значения используется наибольшее.

В следующей таблице приведены элементы ENUM_SYMBOL_CALC_MODE и соответствующие им способы расчета маржи. Это же свойство (SYMBOL_TRADE_CALC_MODE) отвечает и за расчет прибыли/убытка позиции, но мы рассмотрим этот аспект позднее, в главе про торговые функции MQL5.

Идентификатор	Формула
SYMBOL_CALC_MODE_FOREX *Forex*	Lots * ContractSize * MarginRate / Leverage
SYMBOL_CALC_MODE_FOREX_NO_LEVERAGE *Forex без "плеча"*	Lots * ContractSize * MarginRate
SYMBOL_CALC_MODE_CFD *CFD*	Lots * ContractSize * MarketPrice * MarginRate
SYMBOL_CALC_MODE_CFDLEVERAGE *CFD с "плечом"*	Lots * ContractSize * MarketPrice * MarginRate / Leverage
SYMBOL_CALC_MODE_CFDINDEX	Lots * ContractSize * MarketPrice * TickPrice / TickSize * MarginRate

Идентификатор	Формула
CFD на индексы	
SYMBOL_CALC_MODE_EXCH_STOCKS *Ценные бумаги на бирже*	Lots * ContractSize * LastPrice * MarginRate
SYMBOL_CALC_MODE_EXCH_STOCKS_MOEX *Ценные бумаги на MOEX*	Lots * ContractSize * LastPrice * MarginRate
SYMBOL_CALC_MODE_FUTURES *Фьючерсы*	Lots * InitialMargin * MarginRate
SYMBOL_CALC_MODE_EXCH_FUTURES *Фьючерсы на бирже*	Lots * InitialMargin * MarginRate *или* Lots * MaintenanceMargin * MarginRate
SYMBOL_CALC_MODE_EXCH_FUTURES_FORTS *Фьючерсы на FORTS*	Lots * InitialMargin * MarginRate *или* Lots * MaintenanceMargin * MarginRate
SYMBOL_CALC_MODE_EXCH_BONDS *Облигации на бирже*	Lots * ContractSize * FaceValue * OpenPrice / 100
SYMBOL_CALC_MODE_EXCH_BONDS_MOEX *Облигации на MOEX*	Lots * ContractSize * FaceValue * OpenPrice / 100
SYMBOL_CALC_MODE_SERV_COLLATERAL	Неторгуемый актив (маржа неприменима)

В формулах использованы следующие обозначения:

- Lots — объем позиции или ордера в лотах (долях контракта);
- ContractSize — размер контракта (одного лота, SYMBOL_TRADE_CONTRACT_SIZE);
- Leverage — "плечо" торгового счета (ACCOUNT_LEVERAGE);
- InitialMargin — начальная маржа (SYMBOL_MARGIN_INITIAL);
- MaintenanceMargin — поддерживающая маржа (SYMBOL_MARGIN_MAINTENANCE);
- TickPrice — стоимость тика (SYMBOL_TRADE_TICK_VALUE);
- TickSize — размер тика (SYMBOL_TRADE_TICK_SIZE);
- MarketPrice — последняя известная цена *Bid/Ask* в зависимости от типа сделки;
- LastPrice — последняя известная цена *Last*;
- OpenPrice — средневзвешенная цена позиции или открытия ордера;
- FaceValue — номинальная цена облигации;
- MarginRate — коэффициент маржи согласно функции *SymbolInfoMarginRate*, может также иметь 2 разных значения: для начальной и поддерживающей маржи.

Альтернативная реализация вычислений по формулам для большинства типов символов приведена в файле *MarginProfitMeter.mqh* (см. раздел Оценка прибыли торговой операции). Её можно применить и в индикаторах.

Сделаем пару замечаний по некоторым режимам.

В вышеприведенной таблице лишь в трех формулах для фьючерсов применена начальная маржа (SYMBOL_MARGIN_INITIAL). Однако если это свойство имеет ненулевое значение в спецификации любого другого символа, то именно оно определяет маржу.

Некоторые биржы могут налагать собственную специфику на корректировку маржи, как, например, система дисконтов для FORTS (SYMBOL_CALC_MODE_EXCH_FUTURES_FORTS). Подробности — в документации MQL5 и у вашего брокера.

При режиме SYMBOL_CALC_MODE_SERV_COLLATERAL стоимость инструмента учитывается в Активах (Assets), которые суммируются с собственными средствами (Equity). Тем самым открытые позиции по такому инструменту увеличивают размер свободных средств (Free Margin) и служат дополнительным обеспечением под открытые позиции по торгуемым инструментам. Рыночная стоимость открытой позиции рассчитывается на основании объема, размера контракта, текущей цены рынка и коэффициента ликвидности: *Lots * ContractSize * MarketPrice * LiqudityRate* (последнее значение можно получить как свойство SYMBOL_TRADE_LIQUIDITY_RATE).

В качестве примера работы со свойствами, связанными с маржой, рассмотрим скрипт *SymbolFilterMarginStats.mq5*. Его задачей будет подсчет статистики по способам расчета маржи в перечне избранных символов, а также опциональный вывод в журнал этих свойств для каждого символа. Отбор символов для анализа будем осуществлять с помощью уже известного класса фильтра *SymbolFilter* и условий для него, поставляемых из входных переменных.

```
#include <MQL5Book/SymbolFilter.mqh>

input bool UseMarketWatch = false;
input bool ShowPerSymbolDetails = false;
input bool ExcludeZeroInitMargin = false;
input bool ExcludeZeroMainMargin = false;
input bool ExcludeZeroHedgeMargin = false;
```

По умолчанию информация запрашивается для всех доступных символов. Для ограничения контекста только обзором рынка следует установить *UseMarketWatch* в *true*.

Параметр *ShowPerSymbolDetails* позволяет включить вывод подробной информации о каждом символе (по умолчанию параметр равен *false*, и выводится только статистика).

Последние 3 параметра предназначены для фильтрации символов по условиям нулевых значений маржи (начальной, поддерживающей и хеджирующей, соответственно).

Для сбора и удобного отображения в журнале полного набора свойств по каждому символу (когда включен режим *ShowPerSymbolDetails*) в коде определена структура *MarginSettings*.

```
struct MarginSettings
{
   string name;
   ENUM_SYMBOL_CALC_MODE calcMode;
   bool hedgeLeg;
   double initial;
   double maintenance;
   double hedged;
};
```

Поскольку часть свойств является целочисленной (SYMBOL_TRADE_CALC_MODE, SYMBOL_MARGIN_HEDGED_USE_LEG), а часть — вещественной (SYMBOL_MARGIN_INITIAL, SYMBOL_MARGIN_MAINTENANCE, SYMBOL_MARGIN_HEDGED), запрашивать их объектом-фильтром придется по отдельности.

Теперь обратимся непосредственно к рабочему коду в *OnStart*. Здесь как обычно определим объект фильтра (*f*), выходные массивы для имен символов (*symbols*) и значений запрашиваемых свойств (*flags*, *values*). Помимо них добавим массив структур *MarginSettings*.

```
void OnStart()
{
   SymbolFilter f;                // объект фильтра
   string symbols[];              // массив для имен
   long flags[][2];               // массив для целочисленных векторов
   double values[][3];            // массив для вещественных векторов
   MarginSettings margins[];      // составной выходной массив
   ...
```

Для подсчета статистики введен массив-карта *stats* с ключом типа ENUM_SYMBOL_CALC_MODE и целым *int*-значением — количеством, сколько раз встретился каждый способ. Также все случаи нулевой маржи и включенного режима расчета по большей "стороне" должны фиксироваться в соответствующих переменных-счетчиках.

```
   MapArray<ENUM_SYMBOL_CALC_MODE,int> stats; // счетчики для каждого способа/режима
   int hedgeLeg = 0;                          // и прочие счетчики
   int zeroInit = 0;                          // ...
   int zeroMaintenance = 0;
   int zeroHedged = 0;
   ...
```

Далее укажем интересующие нас свойства, связанные с маржой, которые будут считываться из настроек символов. Сперва целочисленные в массиве *ints* и затем вещественные в массиве *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
};
...
```

В зависимости от входных параметров настроим условия фильтрации.

```
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
...
```

Теперь все готово для отбора символов по условиям и получения в массивы их свойств. Делаем это дважды, раздельно для целочисленных и вещественных свойств.

```
f.select(UseMarketWatch, ints, symbols, flags);
const int n = ArraySize(symbols);
ArrayResize(symbols, 0, n);
f.select(UseMarketWatch, doubles, symbols, values);
...
```

Массив с символами приходится обнулять после первого применения фильтра, чтобы имена не задваивались. Несмотря на 2 раздельных запроса, порядок элементов во всех выходных массивах (*ints* и *doubles*) одинаковый, поскольку условия фильтрации не меняются.

Если пользователем включен подробный лог, выделяем память под массив структур *margins*.

```
if(ShowPerSymbolDetails) ArrayResize(margins, n);
```

Наконец, вычисляем статистику, перебирая все элементы полученных массивов, и опционально заполняем массив структур.

```
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];
   }
}
...
```

Остается лишь вывести статистику в журнал.

```
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();
...
```

Поскольку элементы перечисления ENUM_SYMBOL_CALC_MODE выводятся как целые числа (что не очень информативно), мы также отображаем "легенду", где для каждого значения указано название (из *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)));
}
...
```

Если требуется подробная информация по отобранным символам, выводим массив структур *margins*.

```
   if(ShowPerSymbolDetails)
   {
      Print("Settings per symbol:");
      ArrayPrint(margins);
   }
}
```

Запустим скрипт пару раз с разными настройками. Для начала — с настройками по умолчанию.

```
===== 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
```

Для второго прогона установим *ShowPerSymbolDetails* и *ExcludeZeroInitMargin* в *true*. Тем самым запрашивается подробная информация о всех символах, в которых задано ненулевое значения начальной маржи.

```
===== 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 Правила истечения сроков отложенных ордеров

При работе с отложенными ордерами (включая уровни *Stop Loss* и *Take Profit*) MQL-программе следует проверять пару свойств, задающих правила их истечения. Оба свойства доступны как элементы перечисления ENUM_SYMBOL_INFO_INTEGER для вызова функции *SymbolInfoInteger*.

Идентификатор	Описание
SYMBOL_EXPIRATION_MODE	Флаги разрешенных режимов истечения ордера (битовая маска)
SYMBOL_ORDER_GTC_MODE	Срок действия определен одним из элементов перечисления ENUM_SYMBOL_ORDER_GTC_MODE

Свойство SYMBOL_ORDER_GTC_MODE принимается во внимание, только если SYMBOL_EXPIRATION_MODE содержит SYMBOL_EXPIRATION_GTC (см. далее). GTC является аббревиатурой "Good Till Canceled".

Для каждого финансового инструмента в свойстве SYMBOL_EXPIRATION_MODE может быть указано несколько режимов срока действия (истечения) отложенных ордеров. Каждому режиму сопоставлен флаг (бит).

Идентификатор (Значение)	Описание
SYMBOL_EXPIRATION_GTC (1)	Ордер действителен согласно свойству ENUM_SYMBOL_ORDER_GTC_MODE
SYMBOL_EXPIRATION_DAY (2)	Ордер действителен до конца текущего дня
SYMBOL_EXPIRATION_SPECIFIED (4)	Дата и время истечения указывается в ордере
SYMBOL_EXPIRATION_SPECIFIED_DAY (8)	Дата истечения указывается в ордере

Флаги могут комбинироваться операцией логического ИЛИ ('|'), например, SYMBOL_EXPIRATION_GTC | SYMBOL_EXPIRATION_SPECIFIED, что эквивалентно 1 | 4, то есть числу 5. Для проверки разрешения конкретного режима для инструмента выполните операцию логического И ('&') над полученным результатом функции и битом желаемого режима: ненулевое значение означает доступность режима.

В случае SYMBOL_EXPIRATION_SPECIFIED_DAY ордер действует до 23:59:59 указанного дня. Если это время не попадает на торговую сессию, истечение наступит в ближайшее следующее торговое время.

Перечисление ENUM_SYMBOL_ORDER_GTC_MODE содержит следующие элементы.

Идентификатор	Описание
SYMBOL_ORDERS_GTC	Отложенные ордера и уровни Stop Loss/Take Profit действительны неограниченное время вплоть до явной отмены
SYMBOL_ORDERS_DAILY	Ордера действуют только внутри одного торгового дня: по его завершении удаляются все отложенные ордера, а также уровни Stop Loss и Take Profit
SYMBOL_ORDERS_DAILY_EXCLUDING_STOPS	При смене торгового дня удаляются только отложенные ордера, уровни Stop Loss и Take Profit сохраняются

В зависимости от установленных битов в свойстве SYMBOL_EXPIRATION_MODE, при подготовке ордера к отправке MQL-программа может выбрать один из режимов, соответствующих этим битам. Технически это делается путем заполнения поля type_time в специальной структуре *MqlTradeRequest* перед вызовом функции *OrderSend*. Значение поля должно быть элементом перечисления ENUM_ORDER_TYPE_TIME (см. Сроки действия отложенных ордеров): как мы увидим в дальнейшем, оно перекликается с вышеприведенным набором флагов, то есть каждый флаг задает соответствующий режим в ордере — ORDER_TIME_GTC, ORDER_TIME_DAY, ORDER_TIME_SPECIFIED, ORDER_TIME_SPECIFIED_DAY. Само время или день истечения должен быть указан в другом поле той же структуры.

Скрипт *SymbolFilterExpiration.mq5* позволяет узнать статистику применения каждого из флагов в имеющихся символах (в обзоре рынка или вцелом, в зависимости от входного параметра *UseMarketWatch*). Второй параметр *ShowPerSymbolDetails*, будучи взведенным в *true*, вызовет вывод в журнал всех флагов для каждого символа — будьте внимательны: если при этом режим *UseMarketWatch* равен *false*, сгенерируется очень большое число записей в журнале.

```
#property script_show_inputs

#include <MQL5Book/SymbolFilter.mqh>

input bool UseMarketWatch = false;
input bool ShowPerSymbolDetails = false;
```

В функции *OnStart* помимо объекта фильтра и приемных массивов для названий символов и значений свойств описываем карты *MapArray* для подсчета статистики - раздельно для каждого из свойств SYMBOL_EXPIRATION_MODE и SYMBOL_ORDER_GTC_MODE.

```
void OnStart()
{
   SymbolFilter f;                 // объект фильтра
   string symbols[];               // приемный массив для названий символов
   long flags[][2];                // приемный массив для значений свойств

   MapArray<SYMBOL_EXPIRATION,int> stats;         // счетчики режимов
   MapArray<ENUM_SYMBOL_ORDER_GTC_MODE,int> gtc; // счетчики GTC

   ENUM_SYMBOL_INFO_INTEGER ints[] =
   {
      SYMBOL_EXPIRATION_MODE,
      SYMBOL_ORDER_GTC_MODE
   };
   ...
```

Далее применяем фильтр и посчитываем статистику.

```
   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)
         {
            // свойства в виде описаний и чисел
            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]);
         }
      }
   }
   ...
```

Наконец, выводим полученные числа в журнал.

```
   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)));
   }
}
```

Запустим скрипт два раза. Первый раз — с настройками по умолчанию — можем получить примерно следующую картину.

```
===== 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
```

Здесь видно, что практически все флаги разрешены для большинства символов, а для режима SYMBOL_EXPIRATION_GTC используется единственный вариант SYMBOL_ORDERS_GTC.

Второй раз запустим скрипт, установив *UseMarketWatch* и *ShowPerSymbolDetails* в *true* (предполагается, что в *Обзор рынка* выбрано ограниченное число символов).

```
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
```

Из 15-ти выбранных символов только у 6-ти установлен флаг SYMBOL_EXPIRATION_SPECIFIED_DAY. Подробности о флагах по каждому символу можно найти выше.

6.1.18 Спреды и отступы приказов от текущей цены

Для многих торговых стратегий, в особенности тех, что основываются на краткосрочных операциях, важна информация о спреде и дистанции от текущей цены, допускающей установку или модификацию ордеров. Все эти свойства являются частью перечисления ENUM_SYMBOL_INFO_INTEGER и доступны через функцию *SymbolInfoInteger*.

Идентификатор	Описание
SYMBOL_SPREAD	Размер спреда (в пунктах)
SYMBOL_SPREAD_FLOAT	Логический признак плавающего спреда
SYMBOL_TRADE_STOPS_LEVEL	Минимальный разрешенный отступ (в пунктах) от текущей цены для установки уровней Stop Loss, Take Profit и отложенных ордеров
SYMBOL_TRADE_FREEZE_LEVEL	Дистанция от текущей цены (в пунктах) для заморозки ордеров и позиций

В вышеприведенной таблице под текущей ценой понимается цена *Ask* или *Bid*, в зависимости от сути выполняемой операции.

Так защитные уровни *Stop Loss* и *Take Profit* предписывают закрытие позиции, которое выполняется операцией противоположного направления к открытию. Поэтому для покупок, открываемых по цене *Ask*, защитные уровни указывают *Bid*, а для продаж, открываемых по *Bid*, защитные уровни указывают *Ask*. При установке отложенных ордеров, тип цены открытия выбирается стандартным образом: покупка (*Buy Stop*, *Buy Limit*, *Buy Stop Limit*) — по *Ask*, продажа (*Sell Stop*, *Sell Limit*, *Sell Stop Limit*) — по *Bid*. Именно с учетом таких типов цен в разрезе упомянутых торговых операций происходит отсчет дистанции в пунктах для свойств SYMBOL_TRADE_STOPS_LEVEL и SYMBOL_TRADE_FREEZE_LEVEL.

Свойство SYMBOL_TRADE_STOPS_LEVEL, если оно ненулевое, запрещает такую модификацию уровней *Stop Loss*, *Take Profit* у открытой позиции, чтобы любой из них оказался от текущей цены на более близком расстоянии, чем указанное. Аналогично, нельзя перенести цену открытия отложенного ордера ближе, чем SYMBOL_TRADE_STOPS_LEVEL пунктов от текущей цены.

Свойство SYMBOL_TRADE_FREEZE_LEVEL, если оно ненулевое, ограничивает любые торговые операции по отложенному ордеру или открытой позиции в заданной окрестности текущей цены. Для отложенного ордера заморозка наступает, когда заданная цена открытия оказывается на дистанции меньше SYMBOL_TRADE_FREEZE_LEVEL пунктов от текущей цены (опять же тип текущей цены — *Ask* или *Bid* — зависит от покупки или продажи, соответственно). Для позиции под заморозку попадают уровни *Stop Loss* и *Take Profit*, оказавшиеся рядом с текущей ценой, а потому для них отсчет ведется для "обратных" типов цен.

Если свойство SYMBOL_SPREAD_FLOAT равно *true*, свойство SYMBOL_SPREAD не является частью спецификации символа, а содержит актуальный спред, динамически меняющийся при каждом вызове согласно рыночным обстоятельствам. Его также можно найти как разницу между ценами *Ask* и *Bid* в структуре *MqlTick*, вызвав *SymbolInfoTick*.

Проанализировать указанные свойства позволит скрипт *SymbolFilterSpread.mq5*. В нем определено пользовательское перечисление ENUM_SYMBOL_INFO_INTEGER_PART, в которое включены лишь интересующие нас в данном контексте свойства из 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
};
```

С помощью нового перечисления определен входной параметр *Property*, который задает, какое именно свойство из 4-х будет анализироваться. Параметры *UseMarketWatch* и *ShowPerSymbolDetails* уже известным образом управляют процессом, как и в предыдущих тестовых скриптах.

```
input bool UseMarketWatch = true;
input ENUM_SYMBOL_INFO_INTEGER_PART Property = SPREAD_FIXED;
input bool ShowPerSymbolDetails = true;
```

Для удобного отображения информации по каждому символу (имя и значение свойства в каждой строке) с помощью функции *ArrayPrint* определена вспомогательная структура *SymbolDistance* (используется, только когда *ShowPerSymbolDetails* равно *true*).

```
struct SymbolDistance
{
   string name;
   int value;
};
```

В обработчике *OnStart* описываем необходимые объекты и массивы.

```
void OnStart()
{
   SymbolFilter f;                // объект фильтра
   string symbols[];              // приемный массив для имен
   long values[];                 // приемный массив для значений
   SymbolDistance distances[];    // массив для печати
   MapArray<long,int> stats;      // счетчики конкретных значений выбранного свойства
   ...
```

Затем применяем фильтр и заполняем приемные массивы значениями указанного свойства *Property*, с сортировкой.

```
   f.select(UseMarketWatch, (ENUM_SYMBOL_INFO_INTEGER)Property, symbols, values, true
   const int n = ArraySize(symbols);
   if(ShowPerSymbolDetails) ArrayResize(distances, n);
   ...
```

В цикле считаем статистику и заполняем структуры *SymbolDistance*, если необходимо.

```
      for(int i = 0; i < n; ++i)
      {
         stats.inc(values[i]);
         if(ShowPerSymbolDetails)
         {
            distances[i].name = symbols[i];
            distances[i].value = (int)values[i];
         }
      }
      ...
```

Наконец выводим результаты в журнал.

```
      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);
      }
   }
```

Вот что можно получить при запуске скрипта с настройками по умолчанию, что соответствует анализу спредов.

```
===== 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
```

Чтобы понять, являются ли спреды текущими (меняющимися динамически) или фиксированными, запустим скрипт с другими настройками: Property = SPREAD_FLOAT, ShowPerSymbolDetails = false.

```
===== Distances for Market Watch symbols =====
Total symbols: 13
Stats per SYMBOL_SPREAD_FLOAT:
    [key] [value]
[0]     1      13
```

Согласно этим данным все символы в обзоре рынка имеют плавающий спред (значение 1 в ключе *key* — это *true* в SYMBOL_SPREAD_FLOAT). Поэтому если снова и снова запускать скрипт с настройками по умолчанию, мы будем получать новые значения (при открытом рынке).

6.1.19 Получение величины свопов

Для реализации среднесрочных и долгосрочных стратегий становятся важны величины свопов, которые могут оказать существенное, как правило, негативное, влияние на финансовый результат. Однако некоторые читатели наверняка являются поклонниками стратегии Carry Trade, изначально построенной на извлечении прибыли из положительных свопов. В MQL5 имеется несколько свойств символов, предоставляющих доступ к строкам спецификации, которые связаны со свопами.

Идентификатор	Описание
SYMBOL_SWAP_MODE	Модель расчета свопа ENUM_SYMBOL_SWAP_MODE
SYMBOL_SWAP_ROLLOVER3DAYS	День недели для начисления тройного свопа ENUM_DAY_OF_WEEK
SYMBOL_SWAP_LONG	Значение свопа для длинной позиции
SYMBOL_SWAP_SHORT	Значение свопа для короткой позиции

Перечисление ENUM_SYMBOL_SWAP_MODE содержит элементы, задающие варианты единиц измерения и принципы начисления свопов. Также как и SYMBOL_SWAP_ROLLOVER3DAYS они относятся к целочисленным свойствам ENUM_SYMBOL_INFO_INTEGER.

Непосредственно значения свопов указаны в свойствах SYMBOL_SWAP_LONG и SYMBOL_SWAP_SHORT в составе ENUM_SYMBOL_INFO_DOUBLE, то есть типа *double*.

Ниже приведены элементы ENUM_SYMBOL_SWAP_MODE.

Идентификатор	Описание
SYMBOL_SWAP_MODE_DISABLED	Нет свопов
SYMBOL_SWAP_MODE_POINTS	Пункты
SYMBOL_SWAP_MODE_CURRENCY_SYMBOL	Базовая валюта символа
SYMBOL_SWAP_MODE_CURRENCY_MARGIN	Маржинальная валюта символа
SYMBOL_SWAP_MODE_CURRENCY_DEPOSIT	Валюта депозита
SYMBOL_SWAP_MODE_INTEREST_CURRENT	Годовые проценты от цены инструмента на момент расчета свопа
SYMBOL_SWAP_MODE_INTEREST_OPEN	Годовые проценты от цены открытия позиции по символу
SYMBOL_SWAP_MODE_REOPEN_CURRENT	Пункты (с переоткрытием позиции по цене закрытия)
SYMBOL_SWAP_MODE_REOPEN_BID	Пункты (с переоткрытием позиции по цене Bid нового дня). (в параметрах SYMBOL_SWAP_LONG и SYMBOL_SWAP_SHORT)

Для вариантов SYMBOL_SWAP_MODE_INTEREST_CURRENT и SYMBOL_SWAP_MODE_INTEREST_OPEN подразумевается "банковский" учет дней в году — 360.

Для вариантов SYMBOL_SWAP_MODE_REOPEN_CURRENT и SYMBOL_SWAP_MODE_REOPEN_BID позиция принудительно закрывается в конце торгового дня, а далее их поведение отличается.

В случае SYMBOL_SWAP_MODE_REOPEN_CURRENT позиция переоткрывается на следующий день по цене вчерашнего закрытия +/- указанное количество пунктов. В случае SYMBOL_SWAP_MODE_REOPEN_BID позиция переоткрывается на следующий день по текущей

цене Bid +/- указанное количество пунктов. В обоих случаях количество пунктов находится в параметрах SYMBOL_SWAP_LONG и SYMBOL_SWAP_SHORT.

Проверим работу свойств с помощью скрипта *SymbolFilterSwap.mq5*. Во входных параметрах предусмотрим выбор контекста анализа: *Обзор рынка* или все символы в зависимости от *UseMarketWatch*. Когда параметр *ShowPerSymbolDetails* равен *false*, будем подсчитывать статистику, сколько раз в символах используется тот или иной режим из ENUM_SYMBOL_SWAP_MODE. Когда параметр *ShowPerSymbolDetails* равен *true*, будем выводить массив всех символов с режимом, заданным в *Mode*, причем отсортируем массив по убыванию значений в полях SYMBOL_SWAP_LONG и SYMBOL_SWAP_SHORT.

```
input bool UseMarketWatch = true;
input bool ShowPerSymbolDetails = false;
input ENUM_SYMBOL_SWAP_MODE Mode = SYMBOL_SWAP_MODE_POINTS;
```

Для элементов объединенного массива свопов опишем структуру *SymbolSwap* с именем символа и величиной свопа. Направление свопа будем обозначать префиксом в поле name: "+" для свопов длинных позиций, "-" для свопов коротких позиций.

```
struct SymbolSwap
{
   string name;
   double value;
};
```

По традиции опишем объект фильтра в начале *OnStart*. Однако дальнейший код существенно различается в зависимости от значения переменной *ShowPerSymbolDetails*.

```
void OnStart()
{
   SymbolFilter f;                // объект фильтра
   PrintFormat("===== Swap modes for %s symbols =====",
      (UseMarketWatch ? "Market Watch" : "all available"));

   if(ShowPerSymbolDetails)
   {
      // сводная таблица свопов выбранного режима Mode
      ...
   }
   else
   {
      // подсчет статистики режимов
      ...
   }
}
```

Сперва представим вторую ветвь. Здесь мы заполняем с помощью фильтра массивы с именами символов (*symbols*) и режимами свопов (*values*), которые берутся из свойства SYMBOL_SWAP_MODE. Полученные значения аккумулируются в карте-массиве *MapArray<ENUM_SYMBOL_SWAP_MODE,int> stats*.

```
// подсчет статистики режимов
string symbols[];
long values[];
MapArray<ENUM_SYMBOL_SWAP_MODE,int> stats; // счетчики для каждого режима
// применяем фильтр и собираем значения режимов
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]);
}
...
```

Далее выводим собранную статистику.

```
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)));
}
```

Для случая построения таблицы со значениями свопов алгоритм следующий. Свопы для длинных и коротких позиций запрашиваются раздельно, поэтому мы определяем парные массивы для имен и значений. Воедино они будут сведены в массиве структур *swaps*.

```
// сводная таблица свопов выбранного режима Mode
string buyers[], sellers[];    // массивы для имен
double longs[], shorts[];      // массивы для значений свопов
SymbolSwap swaps[];            // общий массив для распечатки
```

Установим в фильтре условие на выбранный режим свопов. Это необходимо, чтобы можно было сравнивать и сортировать элементы массива.

```
f.let(SYMBOL_SWAP_MODE, Mode);
```

Затем применяем фильтр дважды для разных свойств (SYMBOL_SWAP_LONG, SYMBOL_SWAP_SHORT) и заполняем их значениями разные массивы (*longs*, *shorts*). Внутри каждого вызова массивы сортируются по возрастанию.

```
f.select(UseMarketWatch, SYMBOL_SWAP_LONG, buyers, longs, true);
f.select(UseMarketWatch, SYMBOL_SWAP_SHORT, sellers, shorts, true);
```

В принципе, размеры массивов должны быть одинаковы, так как условие фильтра одно и то же, но для наглядности выделим под каждый размер свою переменную. Поскольку каждый символ попадет в результирующую таблицу дважды — для длинной и короткой стороны — распределяем под массив *swaps* двойной размер.

```
      const int l = ArraySize(longs);
      const int s = ArraySize(shorts);
      const int n = ArrayResize(swaps, l + s); // должно быть l == s
      PrintFormat("Total symbols with %s: %d", EnumToString(Mode), l);
```

Далее объединяем два массива *longs* и *shorts*, обходя их в обратном порядке, так как нам требуется сортировка от положительных к отрицательным значениям.

```
      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);
      }
```

Интересно запустить скрипт несколько раз с разными настройками. Например, по умолчанию мы можем получить следующие результаты.

```
===== 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
```

Из данной статистики видно 10 символов имеет режим свопов SYMBOL_SWAP_MODE_POINTS, для двух — свопы отключены SYMBOL_SWAP_MODE_DISABLED, а для одного — в базовой валюте SYMBOL_SWAP_MODE_CURRENCY_SYMBOL.

Выясним, что за символы имеют SYMBOL_SWAP_MODE_POINTS и узнаем их свопы. Для этого установим *ShowPerSymbolDetails* в *true* (параметр *Mode* уже имеет значение 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
```

Вы можете сравнить величины со спецификациями символов.

Наконец, поменяем Mode на SYMBOL_SWAP_MODE_CURRENCY_SYMBOL — должны получить в нашем случае один символ, но разнесенный на две строки: с плюсом и минусом в названии.

```
===== 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
```

Как видно из таблицы, оба свопа имеют отрицательные значения.

6.1.20 Текущая рыночная информация (тик)

В разделе Получение последнего тика по символу мы уже познакомились с функцией *SymbolInfoTick*, которая предоставляет полную информацию о последнем тике (событии изменении цены) в виде структуры *MqlTick*. При необходимости MQL-программа может запросить значения цен и объемов, соответствующих полям этой структуры, по отдельности. Все они обозначаются свойствами разных типов, входящих в состав перечислений ENUM_SYMBOL_INFO_INTEGER и ENUM_SYMBOL_INFO_DOUBLE.

Идентификатор	Описание	Тип свойства
SYMBOL_TIME	Время последней котировки	datetime
SYMBOL_BID	Цена Bid — лучшее предложение на продажу	double
SYMBOL_ASK	Цена Ask — лучшее предложение на покупку	double
SYMBOL_LAST	Last — цена совершения последней сделки	double
SYMBOL_VOLUME	Объем в последней сделке	long
SYMBOL_TIME_MSC	Время последней котировки в миллисекундах с 1970.01.01	long
SYMBOL_VOLUME_REAL	Объем в последней сделке с повышенной точностью	double

Следует отметить, что код двух свойств, связанных с объемом — SYMBOL_VOLUME и SYMBOL_VOLUME_REAL — одинаковый в обоих перечислениях. Это единственный случай, когда идентификаторы элементов разных перечислений пересекаются. Дело в том, что они возвращают по сути одно и то же свойство тика, но с разной точностью представления.

В отличие от структуры, среди свойств нет аналога полю *uint flags*, которое сообщает, какие именно изменения на рынке повлекли генерацию тика. Данное поле имеет смысл только внутри структуры.

Попробуем запросить свойства тика по отдельности и сравнить с результатом вызова *SymbolInfoTick*. При быстром рынке существует вероятность, что результаты будут отличаться. Новый тик (или даже несколько тиков) может прийти между вызовами функций.

```
void OnStart()
{
   PRTF(TimeToString(SymbolInfoInteger(_Symbol, SYMBOL_TIME), TIME_DATE | TIME_SECONE
   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);
}
```

Нетрудно убедиться, что в конкретном случае информация совпала.

```
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 Описательные свойства символов

Платформа предоставляет для MQL-программ группу текстовых свойств, описывающих важные качественные характеристики. Например, при разработке индикаторов или торговых стратегий на основе корзины финансовых инструментов может потребоваться отобрать символы по стране происхождения, сектору экономики или названию базового актива (если инструмент производный).

Идентификатор	Описание
SYMBOL_BASIS	Имя базового актива для производного инструмента
SYMBOL_CATEGORY	Название категории, к которой принадлежит финансовый инструмент
SYMBOL_COUNTRY	Страна, к которой отнесен финансовый инструмент
SYMBOL_SECTOR_NAME	Сектор экономики, к которому относится финансовый инструмент
SYMBOL_INDUSTRY_NAME	Отрасль экономики или вид промышленности, к которой относится финансовый инструмент
SYMBOL_BANK	Источник текущей котировки
SYMBOL_DESCRIPTION	Строковое описание символа
SYMBOL_EXCHANGE	Название биржи или торговой площадки, на которой торгуется символ
SYMBOL_ISIN	Уникальный 12-разрядный буквенно-цифровой код в системе международных идентификационных кодов ценных бумаг — ISIN (International Securities Identification Number)
SYMBOL_PAGE	Адрес интернет страницы с информацией по символу
SYMBOL_PATH	Путь в дереве символов

Другой случай, когда программа может применить анализ этих свойств, возникает при поиске курса пересчета из одной валюты в другую. Мы уже знаем, как найти инструмент с необходимым сочетанием базовой и котировочной валюты, но сложность в том, что таких инструментов может

найтись несколько. Тогда и может помочь чтение свойств вроде SYMBOL_SECTOR_NAME (нужно искать "Currency" или синоним — уточните в спецификации вашего брокера) или SYMBOL_PATH.

Путь SYMBOL_PATH содержит всю иерархию папок в каталоге символов, внутри которых содержится конкретный символ: имена папок разделены обратной косой чертой ('\\') по аналогии с файловой системой. Последний элемент пути — это непосредственно название символа.

Для некоторых строковых свойств имеются аналоги среди целочисленных свойств. В частности, вместо SYMBOL_SECTOR_NAME вы можете использовать свойство SYMBOL_SECTOR, которое возвращает элемент перечисления ENUM_SYMBOL_SECTOR со всеми поддерживаемыми секторами. По аналогии для SYMBOL_INDUSTRY_NAME имеется похожее свойство SYMBOL_INDUSTRY с типом перечисления ENUM_SYMBOL_INDUSTRY.

При необходимости MQL-программа может выяснить даже цвет фона, используемого при отображении символа в Обзоре рынка, — достаточно прочитать свойство SYMBOL_BACKGROUND_COLOR. Это позволит тем программам, которые создают на графике собственный интерфейс с помощью графических объектов (диалоговые окна, списки и пр.), сделать его унифицировано с родными элементами управления терминала.

Рассмотрим пример скрипта *SymbolFilterDescription.mq5*, который выводит для символов *Обзора рынка* 4 предопределенных текстовых свойства. Первое из них — SYMBOL_DESCRIPTION (не путать с названием самого символа), и именно по нему будет выполняться сортировка результирующего списка. Остальные три для ознакомления: SYMBOL_SECTOR_NAME, SYMBOL_COUNTRY, SYMBOL_PATH. Все значения заполняются специфическим образом у каждого брокера (могут быть разночтения для одного и того же тикера).

Кроме того, мы оставили за кадром, что в нашем классе *SymbolFilter* на самом деле реализована особая перегрузка метода *equal* для сравнения строк: она поддерживает поиск на вхождение подстроки с шаблоном, в котором символом подстановки '*' обозначается 0 или несколько произвольных символов. Например, "*ian*" найдет все символы, в которых встречается подстрока "ian" (в любом месте), а "*Index" — только строки, оканчивающиеся на "Index".

Такая возможность напоминает поиск по подстроке в самом диалоге *Символы*, доступном пользователям. Правда там не нужно указывать подстановочный символ, потому что всегда ищется подстрока. В алгоритме, с которым можно ознакомиться в исходных кодах (*SymbolFilter.mqh*), мы оставили возможность искать либо полное соответствие (символов '*' нет), либо подстроку (есть хотя бы одна звездочка).

Сравнение делается с учетом регистра. При необходимости легко адаптировать код для сравнения без различения строчных и прописных букв.

Учитывая новую возможность, определим входную переменную для искомой строки в описании символов. Если переменная пуста, будут выводиться все символы *Обзора рынка*.

```
input string SearchPattern = "";
```

Далее все привычно.

```
void OnStart()
{
   SymbolFilter f;                      // объект фильтра
   string symbols[];                    // массив имен
   string text[][4];                    // массив векторов с данными

   // свойств для чтения
   ENUM_SYMBOL_INFO_STRING fields[] =
   {
      SYMBOL_DESCRIPTION,
      SYMBOL_SECTOR_NAME,
      SYMBOL_COUNTRY,
      SYMBOL_PATH
   };

   if(SearchPattern != "")
   {
      f.let(SYMBOL_DESCRIPTION, SearchPattern);
   }

   // применяем фильтр и получаем отсортированные по описанию массивы
   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);
   }
}
```

Вот возможный вариант списка (с сокращениями).

```
===== 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"
...
```

Если введем строку поиска "*ian*" во входную переменную *SearchPattern*, получим следующий результат.

```
===== 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 Глубина стакана цен

MetaTrader 5 позволяет получать о биржевых инструментах не только информацию о ценах и объемах сделок, упакованную в тики, но и "стакан цен" (или "глубину рынка"), то есть распределение объемов в выставленных заявках на покупку и продажу на нескольких ближайших уровнях вокруг текущей цены. Одно из целочисленных свойств символа SYMBOL_TICKS_BOOKDEPTH содержит максимальное количество уровней, показываемых в стакане. Это количество разрешено для каждой из сторон, то есть общий размер стакана может быть в два раза больше (причем здесь не учитываются ценовые уровни с нулевыми объемами, которые не транслируются).

В зависимости от обстановки на рынке, актуальный размер транслируемого стакана может стать и меньше, чем указано в данном свойстве. Для небиржевых инструментов данное свойство, как правило, равно 0, хотя некоторые брокеры могут транслировать стакан и для Forex-символов, ограниченный лишь заявками своих клиентов.

Сам стакан и уведомления о его обновлении должны быть запрошены заинтересованной MQL-программой с помощью специального API, которое мы рассмотрим в следующей главе.

> Следует отметить, что в силу архитектурных особенностей платформы данное свойство не связано напрямую с трансляцией стакана, то есть это всего лишь поле спецификации, заполняемой брокером. Иными словами, ненулевое значение свойства не означает, что стакан обязательно будет поступать в терминал при открытом рынке. Это зависит от прочих настроек сервера и наличия на нем активного подключения к поставщику данных.

Попробуем получить статистику по глубине стаканов по всем или избранным символам с помощью скрипта *SymbolFilterBookDepth.mq5*.

```
input bool UseMarketWatch = false;
input int ShowSymbolsWithDepth = -1;
```

Параметр *ShowSymbolsWithDepth*, равный по умолчанию -1, предписывает собрать статистику по разным настройкам стакана среди всех символов. Если установить параметр в иное значение, программа попытается найти все символы с указанной глубиной стакана.

```
void OnStart()
{
   SymbolFilter f;                   // объект фильтра
   string symbols[];                 // массив для имен символов
   long depths[];                    // массив значений свойств
   MapArray<long,int> stats;         // счетчики встречаемости каждой глубины
   
   if(ShowSymbolsWithDepth > -1)
   {
      f.let(SYMBOL_TICKS_BOOKDEPTH, ShowSymbolsWithDepth);
   }
   
   // применяем фильтр и заполняем массивы
   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);
   ...
```

Если задана конкретная глубина, просто выводим массив символов (они все удовлетворяют условию фильтра), и завершаем работу.

```
   if(ShowSymbolsWithDepth > -1)
   {
      ArrayPrint(symbols);
      return;
   }
   ...
```

В противном случае ведем подсчет статистики и выводим её.

```
      for(int i = 0; i < n; ++i)
      {
         stats.inc(depths[i]);
      }

      Print("Stats per depth:");
      stats.print();
      Print("Legend: key=depth, value=count");
   }
```

При настройках по умолчанию можем получить следующую картину.

```
===== 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
```

Если установить *ShowSymbolsWithDepth* в одно из обнаруженных значений, например, 32, получим перечень символов с такой глубиной стакана.

```
===== 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 Свойства пользовательских символов

Во введении в данную главу мы уже упоминали пользовательские символы — символы, для которых котировки создаются непосредственно в терминале по команде пользователя или программным способом.

В виде пользовательского символа легко сформировать, например, синтетический инструмент на основе формулы, включающей другие символы Обзора рынка — это доступно пользователю непосредственно в интерфейсе терминала.

MQL-программа может реализовывать в MQL5 более сложные сценарии, вроде склейки разных инструментов за разные периоды, генерации рядов по заданному случайному распределению или получение данных (котировки, бары, тики) из внешних источников.

Чтобы в алгоритмах можно было отличить стандартный символ от пользовательского, в MQL5 имеется свойство SYMBOL_CUSTOM — это логический признак того, что символ является пользовательским.

Если для символа задана формула, она доступна через строковое свойство SYMBOL_FORMULA. В формулах, как известно, можно использовать имена других символов, а также математические функции и операторы. Вот несколько примеров:

- Синтетический символ: "@ESU19"/EURCAD
- Календарный спред: "Si-9.13"-"Si-6.13"
- Индекс евро: 34.38805726 * pow(EURUSD,0.3155) * pow(EURGBP,0.3056) * pow(EURJPY,0.1891) * pow(EURCHF,0.1113) * pow(EURSEK,0.0785)

Указание формулы удобно для пользователя, но, как правило, не используется из MQL-программ, поскольку они могут рассчитывать формулы непосредственно в коде, с нестандартными функциями и с большим контролем, в частности, на каждом тике, а не по таймеру 1 раз в 100мс.

Проверим работу со свойствами в скрипте *SymbolFilterCustom.mq5*: он выводит в журнал все пользовательские символы и их формулы (если есть).

```
input bool UseMarketWatch = false;

void OnStart()
{
   SymbolFilter f;                 // объект фильтра
   string symbols[];               // массив для имен символов
   string formulae[];              // массив для формул

   // применяем фильтр и заполняем массивы
   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]);
   }
}
```

Ниже показан результат с единственным найденным пользовательским символом.

```
===== All available custom symbols =====
Total symbols: 1
synthEURUSD SP500m/UK100
```

6.1.24 Специфические свойства (биржа, срочный рынок, облигации)

В этом заключительном разделе главы мы сделаем краткий обзор других свойств символов, которые выходят за рамки книги, но могут пригодиться для реализации продвинутых торговых стратегий. Подробную информацию об этих свойствах можно получить в документации MQL5.

Как известно, MetaTrader 5 позволяет торговать инструментами срочного рынка — опционами и фьючерсами, а также облигациями. Это находит отражение и в программном интерфейсе. MQL5 API предоставляет множество специфических свойств символов, относящихся к упомянутым категориям инструментов.

В частности, для опционов это — период обращения (даты начала SYMBOL_START_TIME и окончания SYMBOL_EXPIRATION_TIME торгов), цена исполнения (SYMBOL_OPTION_STRIKE), право покупки или продажи (SYMBOL_OPTION_RIGHT, Call/Put), европейский или американский тип (SYMBOL_OPTION_MODE) в зависимости от возможности досрочного погашения, изменение цен закрытия день ко дню (SYMBOL_PRICE_CHANGE) и волатильность (SYMBOL_PRICE_VOLATILITY), а также оценочные коэффициенты "греки", характеризующие динамику поведения цен.

Для облигаций особый интерес представляют накопленный купонный доход (SYMBOL_TRADE_ACCRUED_INTEREST), номинальная стоимость (SYMBOL_TRADE_FACE_VALUE), коэффициент ликвидности (SYMBOL_TRADE_LIQUIDITY_RATE).

Для фьючерсов — открытый интерес (SYMBOL_SESSION_INTEREST) и общие объемы ордеров в разбивке по покупкам (SYMBOL_SESSION_BUY_ORDERS_VOLUME) и продажам (SYMBOL_SESSION_SELL_ORDERS_VOLUME), цена клиринга на закрытии торговой сессии (SYMBOL_SESSION_PRICE_SETTLEMENT).

Помимо текущих рыночных данных, составляющих тик, MQL5 позволяет узнать их диапазон за день: максимальные и минимальные значения по каждому из полей тика. Например, SYMBOL_BIDHIGH — это максимальный *Bid* за день, а SYMBOL_BIDLOW — минимальный. Обратите внимание, что свойства SYMBOL_VOLUMEHIGH, SYMBOL_VOLUMELOW (типа *long*) фактически дублируют, но только с меньшей точностью, объемы в SYMBOL_VOLUMEHIGH_REAL и SYMBOL_VOLUMELOW_REAL (*double*).

Информация по ценам *Last* и объемам доступна, как правило, только у биржевых символов.

Следует иметь в виду, что заполнение свойств зависит от настроек сервера брокером.

6.2 Стакан цен

Помимо актуальных рыночных данных о ценах нескольких типов (*Ask/Bid/Last*) и объемах последних сделок, поступающих в терминал в виде тиков, MetaTrader 5 поддерживает трансляцию стакана цен — массива записей об объемах выставленных заявок на покупку и продажу в окрестности текущей рыночной цены. Объемы агрегируются на нескольких уровнях выше и ниже текущей цены, с минимальным шагом цены согласно спецификации символа. Максимальный размер стакана (количество ценовых уровней) установлен, как мы видели, в свойстве символа SYMBOL_TICKS_BOOKDEPTH.

Активные пользователи терминала, несомненно, знакомы с тем, как стакан выглядит в интерфейсе и по каким принципам работает, но все желающие могут узнать подробности в документации.

Стакан содержит расширенную рыночную информацию, которую принято называть "глубиной рынка". Её знание позволяет создавать более изощренные торговые системы.

Действительно, информация о тике является лишь малым срезом стакана. В некотором упрощенном смысле, тик — это стакан глубиной 2 уровня: одна ближайшая цена на покупку *Ask* (доступное предложение), и одна ближайшая цена на продажу *Bid* (доступный спрос). Причем в тике мы не видим объемы заявок по этим ценам.

Изменения стакана могут происходить гораздо чаще, чем тики, поскольку затрагивают не только реакцию на заключенные сделки, но и изменения объемов отложенных лимитных ордеров в глубине рынка.

Обычно поставщиками данных для стакана и котировок (тиков, сделок) являются разные инстанции, и события тиков (*OnTick* в экспертах, *OnCalculate* в индикаторах) не совпадают с событиями стакана. Оба потока поступают асинхронно и параллельно, но в конечном счете попадают в очередь событий MQL-программы.

Важно отметить, что стакан доступен, как правило, для биржевых инструментов, но бывают исключения как в одну, так и в другую сторону:

- стакан может отсутствовать по тем или иным причинам у биржевого инструмента;
- стакан может предоставляться брокером для внебиржевого инструмента на основе собранной им информации о приказах своих клиентов;

В MQL5 информация о стакане доступна для экспертов и индикаторов. С помощью специальных функций (*MarketBookAdd*, *MarketBookRelease*) программы могут включать или отключать в платформе свою подписку на получение уведомлений об изменении стакана. Для получения самих уведомлений программа должна определить в своем коде функцию-обработчик события *OnBookEvent*. После получения уведомления прочитать данные стакана позволяет функция *MarketBookGet*.

> Терминал поддерживает историю котировок и тиков, но не стакана цен. В частности, пользователь или MQL-программа могут скачать историю на требуемую ретроспективу (при наличии у брокера) и тестировать на ней эксперты и индикаторы.
>
> В отличие от этого, стакан цен транслируется только онлайн и не доступен в тестере. Брокер не имеет архива стаканов на сервере. Для эмуляции поведения стакана в тестере необходимо самостоятельно собирать историю стакана онлайн и затем каким-либо образом считывать из MQL-программы, запущенной в тестере. Вы можете найти готовые продукты в MQL5 Маркете.

6.2.1 Управление подпиской на события о стакане цен

Получение информации о содержимом стакана цен производится в терминале по принципу подписки: MQL-программа должна сообщить о своем желании получать события о стакане или, наоборот, прекратить свою подписку с помощью вызова соответствующих функций *MarketBookAdd* и *MarketBookRelease*.

Функция *MarketBookAdd* производит подписку на получение извещений об изменении стакана цен по указанному инструменту. Таким образом, можно подписаться на стаканы множества инструментов, а не только рабочего инструмента текущего графика.

bool MarketBookAdd(const string symbol)

Обычно, эта функция вызывается из *OnInit* или в конструкторе класса долгоживущего объекта. Уведомления об изменении стакана поступают в программу в виде событий *OnBookEvent*, поэтому для их обработки в программе должна присутствовать одноименная функция-обработчик.

Если указанный символ не был выбран в Обзоре рынка перед вызовом функции, он будет добавлен в окно автоматически.

Функция *MarketBookRelease* отменяет подписку на извещения об изменении указанного стакана.

bool MarketBookRelease(const string symbol)

Обычно эта функция должна вызываться из *OnDeinit* или деструктора класса долгоживущего объекта.

Обе функции возвращают значение *true* в случае успеха, а иначе — *false*.

Для всех приложений, запущенных на одном графике, ведутся отдельные счетчики подписок в разрезе символов. Иными словами, на графике может быть несколько подписок на разные символы, и для каждого из них поддерживается свой собственный счетчик.

Подписка или отписка одиночным вызовом любой из функций изменяет счетчик подписок только для конкретного символа, на конкретном графике, где работает программа. Это означает, что на двух соседних графиках могут быть подписки на события *OnBookEvent* на один и тот же символ, но с разными значениями счетчиков подписок.

Начальное значение счетчика подписок равно нулю. При каждом вызове *MarketBookAdd* счетчик подписок для указанного символа на данном графике увеличивается на 1 (символ графика и символ в *MarketBookAdd* не обязаны совпадать). При вызове *MarketBookRelease* счетчик подписок на указанный символ в пределах графика уменьшается на 1.

Трансляция событий *OnBookEvent* по любому символу в пределах графика продолжается до тех пор, пока счетчик подписок по данному символу больше нуля. Поэтому важно, чтобы каждая MQL-программа, которая содержит вызовы *MarketBookAdd*, при завершении своей работы правильно отписывалась от получения событий по каждому использованному символу с помощью *MarketBookRelease*. Для этого достаточно, чтобы количество вызовов *MarketBookAdd* и *MarketBookRelease* совпадало. MQL5 не позволяет узнать значение счетчика.

В роли первого примера выступит простой безбуферный индикатор *MarketBookAddRelease.mq5*, который включает подписку на стакан в момент запуска и отключает при своей выгрузке. Во входном параметре *WorkSymbol* можно указать символ для подписки. Если его оставить пустым (по умолчанию), подписка будет инициирована для рабочего символа текущего графика.

```
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 dc
{
   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]);
   }
}
```

В качестве дополнительной возможности разрешено задать несколько инструментов через запятую. В этом случае будет запрошена подписка на все.

При запуске индикатора в журнал выводится признак успеха подписки или код ошибки. Аналогичным образом в обработчике *OnDeinit* делается попытка отменить подписку.

С настройками по умолчанию, на графике с символом, по которому стакан доступен, получим следующие записи в логе.

```
MarketBookAdd(symbols[i])=true / ok
MarketBookRelease(symbols[i])=true / ok
```

Если нанести индикатор на график с символом без стакана, увидим коды ошибок.

```
MarketBookAdd(symbols[i])=false / BOOKS_CANNOT_ADD(4901)
MarketBookAdd(XPDUSD) failed
MarketBookRelease(symbols[i])=false / BOOKS_CANNOT_DELETE(4902)
MarketBookRelease(XPDUSD) failed
```

Вы можете поэкспериментировать, указывая во входном параметре *WorkSymbol* существующие или отсутствующие символы. Случай с подпиской на стаканы нескольких символов мы рассмотрим в следующем разделе.

6.2.2 Получение событий об изменении стакана цен

Событие *OnBookEvent* генерируется терминалом при изменении состояния стакана цен и обрабатывается функцией *OnBookEvent*, определенной в исходном коде. Для того чтобы терминал стал посылать MQL-программе уведомления *OnBookEvent* по конкретному символу, следует предварительно подписаться на их получение с помощью функции *MarketBookAdd*.

Для того чтобы отписаться от получения события *OnBookEvent* по символу, необходимо вызвать функцию *MarketBookRelease*.

Событие *OnBookEvent* является широковещательным — это означает, что достаточно одной MQL-программе на графике подписаться на события *OnBookEvent*, и все остальные программы на том же графике тоже начнут получать их, при наличии в коде обработчика *OnBookEvent*. Поэтому необходимо анализировать имя символа, которое передается в обработчик в качестве параметра.

Прототип обработчика *OnBookEvent* следующий.

```
void OnBookEvent(const string &symbol)
```

События *OnBookEvent* ставятся в очередь, даже если в данный момент еще не закончена обработка предыдущего события *OnBookEvent*.

Важно, что события *OnBookEvent* являются лишь уведомлениями и не несут в себе состояния стакана заявок. Для получения данных стакана следует вызвать функцию *MarketBookGet*.

При этом следует учитывать, что вызов *MarketBookGet*, даже если он производится непосредственно из обработчика *OnBookEvent*, получит текущее актуальное состояние стакана на момент вызова *MarketBookGet*, которое не обязательно совпадает с тем состоянием стакана, которое инициировало отправку события *OnBookEvent*. Такое может случиться при поступлении в терминал последовательности очень быстрых изменений стакана.

В связи с этим, для получения максимально полной хронологии изменений стакана необходимо писать реализацию *OnBookEvent* с приоритетом на оптимизацию по скорости выполнения.

Вместе с тем, гарантированного способа получения всех уникальных состояний стакана в MQL5 не существует.

> Если ваша программа начала успешно получать уведомления, а затем при открытом рынке они пропали (а тики продолжают приходить), это может свидетельствовать о проблемах в подписке. В частности, неверно спроектированная другая MQL-программа могла отменить подписку большее количество раз, чем требовалось. В таких случаях рекомендуется повторить подписку с помощью нового вызова *MarketBookAdd* после предопределенного таймаута (например, несколько десятков секунд или минуту).

Пример безбуферного индикатора *MarketBookEvent.mq5* позволяет отслеживать поступление событий *OnBookEvent* и выводит имя символа и текущее время (миллисекундный системный счетчик) в комментарий. Для наглядности мы используем функцию многострочного комментария из файла *Comments.mqh* из раздела Вывод сообщений в окно графика.

Интересно, что если оставить входной параметр *WorkSymbol* пустым (значение по умолчанию), сам индикатор не инициирует подписку на стакан, но сможет перехватывать сообщения, заказанные другими MQL-программами на том же графике. Мы это проверим.

```
#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));
   }
}
```

Давайте запустим на графике *MarketBookEvent* с настройками по умолчанию (без собственной подписки), а затем добавим индикатор *MarketBookAddRelease* из предыдущего раздела, причем укажем для него список из нескольких символов с доступными стаканами (в примере ниже это — "XAUUSD,BTCUSD,USDCNH"). На каком именно графике запускать индикаторы — не важно: это может быть и совсем другой символ, вроде EURUSD.

Сразу после запуска *MarketBookEvent* график будет пуст (без комментариев), потому что подписок еще нет. После того как стартует *MarketBookAddRelease* (в журнале должно появиться три строки со статусом успешной подписки *true*), в комментариях начнут попеременно появляться названия символов, по мере обновления их стаканов (сами стаканы мы пока не научились читать, об этом речь пойдет в следующем разделе).

Вот как это выглядит на экране.

Уведомления об изменении стаканов "чужих" символов

Если теперь удалить индикатор *MarketBookAddRelease*, он отменит свои подписки, и комментарий перестанет обновляться. Последующее удаление *MarketBookEvent* очистит комментарий.

Обратите внимание, что между запросом на отмену подписки и моментом, когда события стакана реально перестают обновлять комментарий, проходит некоторое время (секунда-две).

Вы можете запустить на графике только индикатор *MarketBookEvent* с каким-либо символом в его параметре *WorkSymbol*, чтобы убедиться в работе уведомлений внутри одной программы. *MarketBookAddRelease* был ранее использован лишь для того, чтобы продемонстрировать широковещательную природу уведомлений — иными словами, включение подписки на изменения стакана в одной программе действительно влияют на получение уведомлений в другой.

6.2.3 Чтение данных текущего стакана цен

После успешного выполнена функция *MarketBookAdd*, MQL-программа может запрашивать состояния стакана с помощью функции *MarketBookGet* по приходу событий OnBookEvent. Функция *MarketBookGet* заполняет передаваемый по ссылке массив структур *MqlBookInfo* записями стакана цен указанного символа.

```
bool MarketBookGet(string symbol, MqlBookInfo &book[])
```

Под приемный массив можно заранее распределить память для достаточного количества записей. Если динамический массив имеет нулевой или недостаточный размер, терминал сам выделит под него память.

Функция возвращает признак успеха (*true*) или ошибки (*false*).

Обычно *MarketBookGet* используется непосредственно в коде обработчика *OnBookEvent* или в функциях, вызываемых из него.

Отдельная запись о ценовом уровне стакана хранится в структуре *MqlBookInfo*.

```
struct MqlBookInfo
{
   ENUM_BOOK_TYPE type;          // тип заявок
   double         price;         // цена
   long           volume;        // объем
   double         volume_real;   // объем с повышенной точностью
};
```

Перечисление ENUM_BOOK_TYPE содержит следующие элементы.

Идентификатор	Описание
BOOK_TYPE_SELL	Заявка на продажу
BOOK_TYPE_BUY	Заявка на покупку
BOOK_TYPE_SELL_MARKET	Заявка на продажу по рыночной цене
BOOK_TYPE_BUY_MARKET	Заявка на покупку по рыночной цене

Массив стакана отсортирован таким образом, что в верхней его половине расположены заявки на продажу, а в нижней — на покупку. Как правило, это приводит к соблюдению последовательности элементов от больших цен к малым. Иными словами, под 0-м индексом идет самая высокая цена, в последней записи — самая низкая, а между ними цены постепенно уменьшаются. При этом минимальный шаг цен между уровнями составляет SYMBOL_TRADE_TICK_SIZE, однако уровни с нулевыми объемами не транслируются, то есть соседние элементы могут отстоять и на большую величину.

В пользовательском интерфейсе терминала, в окне стакана имеется опция для включения/отключения *Расширенного режима*, в котором уровни с нулевыми объемами начинают отображаются, но по умолчанию, в стандартном режиме, такие уровни скрыты (пропускаются в таблице).

На практике содержимое стакана может иногда противоречить озвученным правилам. В частности, может оказаться, что некоторые заявки на покупку или продажу попадают в противоположную половину стакана (вероятно, кто-то поставил покупку по невыгодно высокой цене или продажу по невыгодно низкой, однако не исключены и технические ошибки при агрегировании данных у поставщика). В результате, из-за соблюдения приоритета "сверху все заявки на продажу, снизу все заявки на покупку", последовательность цен в стакане окажется нарушенной (см. пример ниже). Кроме того, могут обнаружиться повторяющиеся значения цен (уровней) как в одной половине стакана, так и в противоположных.

В принципе, совпадение цен на покупку и продажу в середине стакана — корректно. Оно означает нулевой спред. Однако, к сожалению, задваивание уровней случается и на большей глубине стакана.

Когда мы говорим "половина" стакана, это не следует понимать буквально. В зависимости от ликвидности количество уровней предложения и спроса может не совпадать. В общем случае, стакан несимметричен.

MQL-программа должна проверять корректность стакана (в частности, порядок сортировки цен) и быть готовой обработать потенциально возможные отклонения.

К менее серьезным нештатным ситуациям (которые, тем не менее, следует учесть в алгоритме) можно отнести:

- Последовательные одинаковые стаканы, т.е. без изменений
- Пустой стакан
- Стакан с одним уровнем

Ниже показан фрагмент реального стакана цен, полученного от брокера. Буквами 'S' и 'B' помечены, соответственно, цены заявок на продажу и покупку.

Обратите внимание, что уровни покупок и продаж фактически перекрываются: визуально это не сильно заметно, потому что все записи 'S' в стакане специально вынесены вверх (начало приемного массива), а записи 'B' — вниз (конец массива). Однако приглядитесь: цены для покупок в элементах 20 и 21 равны 143.23 и 138.86, соответственно, а это больше всех предложений на продажу. И в то же время, цены для продаж в элементах 18 и 19 равны 134.62 и 133.55, а это ниже всех предложений на покупку.

```
...
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    // Дубликат
17 S 138.11 200
18 S 134.62 420    // Низкая
19 S 133.55 10627  // Низкая

20 B 143.23 9564   // Высокая
21 B 138.86 533    // Высокая
22 B 138.39 739    // Дубликат
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
...
```

Кроме того, цена 138.39 встречается и в верхней половине под номером 16, и в нижней под номером 22.

Ошибки в стакане наиболее вероятны в экстремальных условиях: при сильной волатильности или недостатке ликвидности.

Проверим получение стакана с помощью индикатора *MarketBookDisplay.mq5*. Он будет подписываться на события стакана для заданного символа в параметре *WorkSymbol* (если там оставить пустую строку, подразумевается рабочий символ текущего графика).

```
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));
}
```

Для обработки событий в коде определен обработчик *OnBookEvent*, в котором вызывается *MarketBookGet*, и все элементы полученного массива *MqlBookInfo* выводятся в многострочный комментарий.

```
void OnBookEvent(const string &symbol)
{
   if(symbol == _WorkSymbol) // берем только стаканы запрошенного символа
   {
      MqlBookInfo mbi[];
      if(MarketBookGet(symbol, mbi)) // получаем текущий стакан
      {
         ...
         int half = ArraySize(mbi) / 2; // оценка середины стакана
         bool correct = true;
         // собираем информацию об уровнях и объемах в одну строку (с переносами)
         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) // ищем середину стакана как смену типа заявок
            {
               if(mbi[i - 1].type == BOOK_TYPE_SELL
                  && mbi[i].type == BOOK_TYPE_BUY)
               {
                  half = i; // это середина, т.к. произошла смена типов
               }

               if(mbi[i - 1].price <= mbi[i].price)
               {
                  correct = false; // обратный порядок = проблема в данных
               }
            }
         }
         Comment(s + (!correct ? "\nINCORRECT BOOK" : ""));
         ...
      }
   }
}
```

Поскольку стакан меняется довольно быстро, следить за комментарием не очень удобно. Поэтому добавим в индикатор пару буферов, в которые в виде гистограмм будем выводить содержимое двух половин стакана: раздельно продажи и покупки. Нулевой бар будет соответствовать центральным уровням, формирующим спред. С увеличением номеров баров происходит увеличение "глубины рынка", то есть там отображаются все более и более отдаленные ценовые уровни: в верхней гистограмме это означает более низкие цены с заявками на покупку, а в нижней — более высокие с заявками на продажу.

```
#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[];
```

Предусмотрим возможность визуализировать стакан в стандартном и расширенном режиме (то есть пропускать или показывать уровни с нулевыми объемами), а также отображать сами объемы в долях лотов или единицах. Обе опции имеют аналоги во встроенном окне стакана.

```
input bool AdvancedMode = false;
input bool ShowVolumeInLots = false;
```

Настройку буферов и получение некоторых свойств символа (которые нам потребуются в дальнейшем) выполним в *OnInit*.

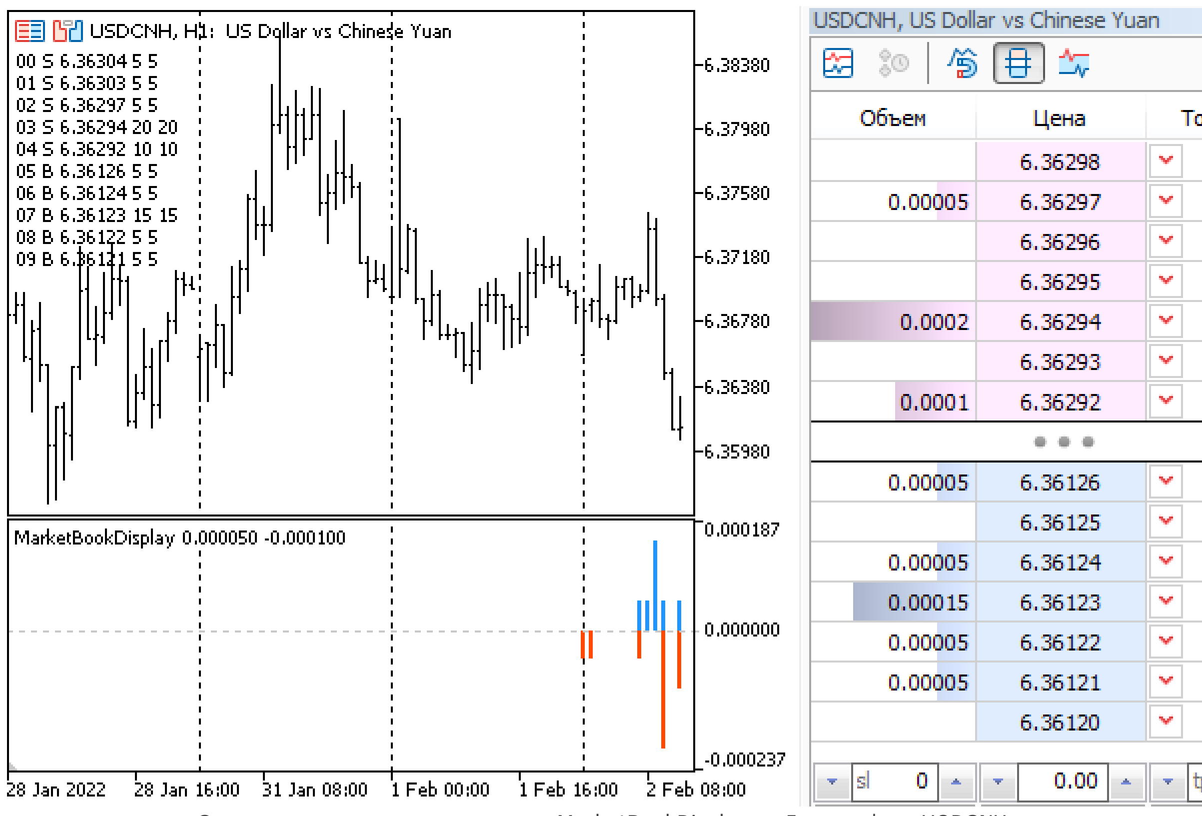
```
int depth, digits;
double tick, contract;

void OnInit()
{
   ...
   // настройка индикаторных буферов
   SetIndexBuffer(0, buys);
   SetIndexBuffer(1, sells);
   ArraySetAsSeries(buys, true);
   ArraySetAsSeries(sells, true);
   // получение необходимых свойств символа
   depth = (int)PRTF(SymbolInfoInteger(_WorkSymbol, SYMBOL_TICKS_BOOKDEPTH));
   tick = SymbolInfoDouble(_WorkSymbol, SYMBOL_TRADE_TICK_SIZE);
   contract = SymbolInfoDouble(_WorkSymbol, SYMBOL_TRADE_CONTRACT_SIZE);
}
```

В обработчик *OnBookEvent* добавим заполнение буферов.

```
#define VOL(V) (ShowVolumeInLots ? V / contract : V)

void OnBookEvent(const string &symbol)
{
   if(symbol == _WorkSymbol) // берем только стаканы запрошенного символа
   {
      MqlBookInfo mbi[];
      if(MarketBookGet(symbol, mbi)) // получаем текущий стакан
      {
         // чистим буфера на глубину с 10-кратным запасом от максимальной глубины,
         // потому что в расширенном режиме может появиться много пустых элементов
         for(int i = 0; i <= depth * 10; ++i)
         {
            buys[i] = EMPTY_VALUE;
            sells[i] = EMPTY_VALUE;
         }
         ... // далее формируем и выводим комментарий как прежде
         if(!correct) return;

         // заполняем буфера данными
         if(AdvancedMode) // включен показ пропусков
         {
            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 // стандартный режим: показываем только значащие элементы
         {
            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);
               }
            }
         }
      }
```

```
    }
  }
```

Следующее изображение демонстрирует работу индикатора с настройками *AdvancedMode=true*, *ShowVolumeInLots=true*.



Содержимое стакана в индикаторе MarketBookDisplay.mq5 на графике USDCNH

Покупки выводятся как положительные величины (синяя гистограмма вверху), продажи — как отрицательные (красная внизу). Для наглядности справа размещено стандартное окно стакана с такими же настройками (в расширенном режиме, объемы в лотах), так что можно убедиться в совпадении значений.

Следует отметить, что индикатор может не успевать перерисовываться достаточно оперативно, чтобы сохранять синхронизацию со встроенным стаканом. Это не значит, что MQL-программа не получила вовремя событие, а лишь побочный эффект асинхронной отрисовки графиков. В рабочих алгоритмах для стакана, как правило, производится аналитическая обработка и выставление приказов, а не визуализация.

В данном случае обновление графика неявным образом запрашивается в момент вызова функции *Comment*.

6.2.4 Использование данных стакана в прикладных алгоритмах

Стакан цен считается весьма полезной технологией для разработки продвинутых торговых систем. В частности, анализ распределения объемов стакана на уровнях, близких к рынку, позволяет заранее узнать среднюю цену исполнения ордера конкретного объема: достаточно просуммировать объемы уровней (противоположного направления), которые обеспечат его заливку. На тонком рынке, при недостаточности объемов, алгоритм может воздержаться от открытия сделки, чтобы избежать существенного проскальзывания цены.

На основе данных стакана можно конструировать и другие стратегии. Например, бывает важно знать ценовые уровни, на которых расположены крупные объемы.

MarketBookVolumeAlert.mq5

В следующем тестовом индикаторе *MarketBookVolumeAlert.mq5* реализуем простой алгоритм для отслеживания объемов или их изменений, превышающих заданную величину.

```
#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;
```

В индикаторе нет диаграмм. Контролируемый символ вводится в параметре *WorkSymbol* (если оставить его пустым, подразумевается рабочий символ графика). Минимальный порог отслеживаемых объектов, то есть чувствительность алгоритма, указывается в параметре *VolumeLimit*. В зависимости от другого параметра *CountVolumeInLots*, объемы анализируются и выводятся пользователю в нотации лотов (*true*) или единиц (*false*) — это также влияет на то, как должно вводиться значение *VolumeLimit*. Перевод из единиц в доли лотов обеспечивает макрос VOL: используемый в нем размер контракта *contract* инициализируем в *OnInit* (см. ниже).

```
#define VOL(V) (CountVolumeInLots ? V / contract : V)
```

При обнаружении крупных объемов выше порога программа выведет сообщение о соответствующем уровне в комментарий. Для сохранения ближайшей истории предупреждений используем уже известный нам класс многострочных комментариев (*Comments.mqh*).

```
#define N_LINES 25                // количество строк в буфере комментариев
#include <MQL5Book/Comments.mqh>
```

В обработчике *OnInit* подготовим необходимые настройки и подпишемся на стакан.

```
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));
}
```

Свойства SYMBOL_SESSION_BUY_ORDERS_VOLUME и SYMBOL_SESSION_SELL_ORDERS_VOLUME, если они заполнены вашим брокером для выбранного символа, позволят сориентироваться с тем, какой порог имеет смысл выбирать. По умолчанию *VolumeLimit* равен 0, из-за чего абсолютно все изменения стакана будут генерировать предупреждения. Чтобы отсеять малозначительные флуктуации, рекомендуется установить *VolumeLimit* в значение, которое превышает средний размер объемов на всех уровнях (посмотрите заранее во встроенном стакане или в индикаторе *MarketBookDisplay.mq5*).

Привычным образом реализуем финализацию.

```
void OnDeinit(const int)
{
   MarketBookRelease(_WorkSymbol);
   Comment("");
}
```

Основную работу выполняет обработчик *OnBookEvent*. В нем описан статический массив *MqlBookInfo mbp* для хранения предыдущего варианта стакана (с прошлого вызова функции).

```
void OnBookEvent(const string &symbol)
{
   if(symbol != _WorkSymbol) return; // обрабатываем только запрошенный символ

   static MqlBookInfo mbp[];      // предыдущая таблица/книга
   MqlBookInfo mbi[];
   if(MarketBookGet(symbol, mbi)) // читаем текущую книгу
   {
      if(ArraySize(mbp) == 0) // первый раз просто сохраняем, т.к. не с чем сравниват
      {
         ArrayCopy(mbp, mbi);
         return;
      }
      ...
```

При наличии старого и нового стакана производим сравнение объемов на их уровнях друг с другом во вложенных циклах по *i* и *j*. Напомним, что увеличение индекса означает уменьшение цены.

```
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);
         }
         // продолжаем цикл увеличивая ++j к меньшим ценам
      }
      else // mbp[j].price < mbi[i].price
      {
         break;
      }
   }
   if(!found) // уникальная (новая) цена
   {
      if(VOL(mbi[i].volume_real) >= VolumeLimit)
      {
         NotifyVolumeChange("Added", mbi[i].price, 0.0, VOL(mbi[i].volume_real)
      }
   }
}
...
```

Здесь акцент делается не на типе уровня, а только на величине объема, но при желании легко добавить в уведомления обозначение покупок или продаж, в зависимости от поля *type* того уровня, где произошло важное изменение.

В завершение сохраняем новую копию *mbi* в статическом массиве *mbp* для сравнения относительно него на следующем вызове функции.

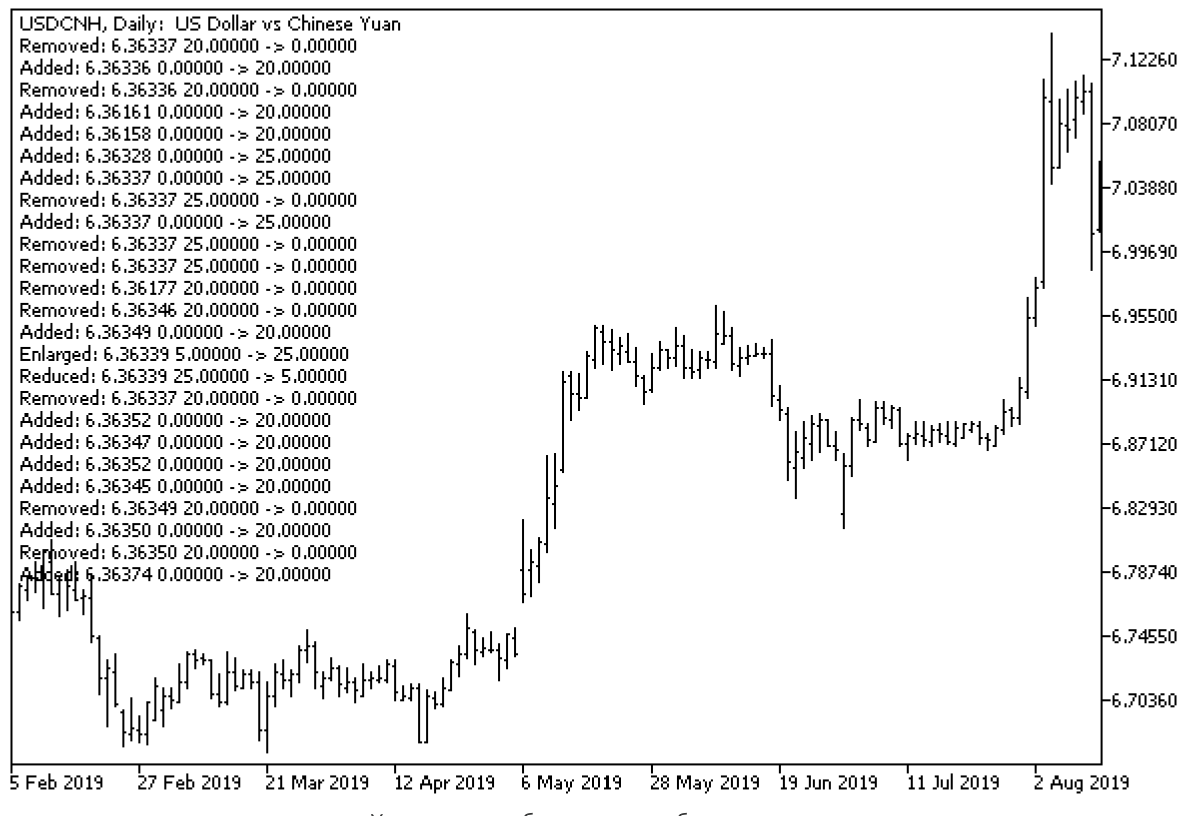
```
      if(ArrayCopy(mbp, mbi) <= 0)
      {
         Print("ArrayCopy failed:", _LastError);
      }
      if(ArrayResize(mbp, ArraySize(mbi)) <= 0) // ужимаем если нужно
      {
         Print("ArrayResize failed:", _LastError);
      }
   }
}
```

ArrayCopy не ужимает динамический приемный массив автоматически, если он оказался больше, чем массив-источник, поэтому мы явным образом устанавливаем его точный размер с помощью *ArrayResize*.

Вспомогательная функция *NotifyVolumeChange* просто добавляет информацию о найденном изменении в комментарий.

```
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);
}
```

На следующем изображении показан результат работы индикатора для настроек *CountVolumeInLots=false*, *VolumeLimit=20*.



Уведомления об изменениях объема в стакане

MarketBookQuasiTicks.mq5

В качестве второго примера возможного применения стакана обратимся к проблеме получения мультивалютных тиков. Мы уже касались её в разделе Генерация пользовательских событий, где было предоставлено одно из возможных решений и индикатор *EventTickSpy.mq5*. Теперь, после знакомства с API стакана цен, мы может реализовать альтернативный вариант.

Создадим индикатор *MarketBookQuasiTicks.mq5*, который будет подписываться на стаканы заданного списка инструментов и находить в них цены лучшего предложения и спроса, то есть пары цен вокруг спреда, а это — ни что иное как цены *Ask* и *Bid*.

Разумеется, данная информация не является полным эквивалентом стандартных тиков (напомним, что потоки сделок/тиков и стакана могут поступать от совершенно разных поставщиков), но обеспечивает адекватное и оперативное представление о рынке.

Новые значения цен по символам будем выводить в многострочный комментарий.

Перечень рабочих символов задается во входном параметре *SymbolList*, как список, через запятую. Включение и отключение подписки на стаканы производится в обработчиках *OnInit* и *OnDeinit*.

```
#define N_LINES 25                // количество строк в буфере комментариев
#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("");
}
```

Анализ каждого нового стакана осуществляется в *OnBookEvent*.

```
void OnBookEvent(const string &symbol)
{
   MqlBookInfo mbi[];
   if(MarketBookGet(symbol, mbi)) // получаем текущий стакан
   {
      int half = ArraySize(mbi) / 2; // оценка середины стакана
      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; // уточняем середину стакана
            }

            if(mbi[i - 1].price <= mbi[i].price)
            {
               correct = false;
            }
         }
      }

      if(correct) // извлекаем лучшие цены Bid/Ask из корректного стакана
      {
         // mbi[half - 1].price // Ask
         // mbi[half].price     // Bid
         OnSymbolTick(symbol, mbi[half].price);
      }
   }
}
```

Найденные рыночные цены *Ask/Bid* передаются во вспомогательную функцию *OnSymbolTick* для вывода в комментарий.

```
void OnSymbolTick(const string &symbol, const double price)
{
   const string message = StringFormat("%s %s",
      symbol, DoubleToString(price, (int)SymbolInfoInteger(symbol, SYMBOL_DIGITS)));
   ChronoComment(message);
}
```

При желании вы можете убедиться, что наши синтезированные тики мало отличаются от стандартных тиков.

Вот как информация о поступающих квази-тиках выглядит на графике.

Мультисимвольные квази-тики, полученные на основе событий стаканов

Вместе с тем, следует еще раз отметить, что события стакана доступны на платформе только онлайн, но не в тестере. Если торговая система будет построена исключительно на квази-тиках из стакана, для её тестирования потребуется применение сторонних решений, обеспечивающих сбор и воспроизведение стаканов в тестере.

6.3 Информация о торговом счете

В этой главе мы изучим последний важный аспект торгового окружения MQL-программ и, в частности, экспертов, разработкой которых мы вплотную займемся в нескольких следующих главах. Речь пойдет о торговом счете.

Наличие действующего счета и активного подключения к нему являются необходимым условием функционирования большинства MQL-программ. До сих пор мы не заостряли на этом внимание, но получение котировок, тиков и, в принципе, возможность открыть работоспособный график подразумевает успешное подключение к торговому счету.

А в контексте экспертов счет дополнительно отражает финансовое состояние клиента, аккумулирует историю торгов и определяет специфические режимы, разрешенные для торговли.

MQL5 API позволяет получать свойств счета, начиная с его номера и заканчивая текущей прибылью. Все они доступны в терминале только на чтение и устанавливаются брокером на сервере.

В каждый момент времени терминал может быть подключен только к одному счету. Именно с ним и работают все MQL-программы. Как мы уже отмечали в разделе Особенности запуска и остановки программ разных типов, смена счета инициирует перезагрузку индикаторов и экспертов, прикрепленных к графикам. При этом в обработчике *OnDeinit* программа может узнать причину деинициализации, которая при смене счета будет равна REASON_ACCOUNT.

6.3.1 Обзор функций получения свойств счета

Полный набор свойств счета логически поделен на 3 группы в зависимости от их типа. Строковые свойства сведены в перечисление ENUM_ACCOUNT_INFO_STRING и запрашиваются функцией *AccountInfoString*. Свойства вещественного типа объединены в перечислении ENUM_ACCOUNT_INFO_DOUBLE, и для них работает функция *AccountInfoDouble*. Перечисление ENUM_ACCOUNT_INFO_INTEGER, используемое в функции *AccountInfoInteger*, содержит идентификаторы целочисленных и логических свойств (флагов), а также нескольких прикладных ENUM_ACCOUNT_INFO-перечислений.

```
double AccountInfoDouble(ENUM_ACCOUNT_INFO_DOUBLE property)
long AccountInfoInteger(ENUM_ACCOUNT_INFO_INTEGER property)
string AccountInfoString(ENUM_ACCOUNT_INFO_STRING property)
```

Для упрощения чтения свойств создан класс *AccountMonitor* (*AccountMonitor.mqh*). В нем за счет перегрузки методов *get* обеспечен автоматический вызов нужной функции API в зависимости от элемента конкретного перечисления, переданного в параметре.

```
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);
   }
   ...
```

Кроме того реализовано несколько перегрузок метода *stringify*, формирующих понятное для пользователя строковое представление значений свойств (оно, в частности, пригодится для прикладных перечислений, которые в противном случае отображались бы в виде малоинформативных чисел). Особенности каждого свойства мы рассмотрим в следующих разделах.

```
static string boolean(const long v)
{
   return v ? "true" : "false";
}

template<typename E>
static string enumstr(const long v)
{
   return EnumToString((E)v);
}

// "расшифровываем" свойства согласно подтипу внутри целочисленных значений
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);
}
...
```

Наконец, для получения исчерпывающей информации о счете предназначен шаблонный метод *list2log*.

```
    // список имен и значений всех свойств типа перечисления E
    template<typename E>
    void list2log()
    {
       E e = (E)0; // подавляем предупреждение '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));
       }
    }
};
```

Мы проверим новый класс в действии в следующем разделе.

6.3.2 Идентификация счета, клиента, сервера и брокера

Пожалуй, самыми главными свойствами счета являются его номер и идентификационные данные: название сервера и компании брокера, а также имя клиента. Все эти свойства, кроме номера, являются строковыми.

Идентификатор	Описание
ACCOUNT_LOGIN	Номер счета (long)
ACCOUNT_NAME	Имя клиента
ACCOUNT_SERVER	Имя торгового сервера
ACCOUNT_COMPANY	Имя компании, обслуживающей счет

Воспользуемся классом *AccountMonitor* из предыдущего раздела, чтобы вывести в журнал эти, а также многие другие свойства, о которых речь пойдет чуть позже. Создадим соответствующий объект и вызовем его свойства в скрипте *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>();
}
```

Вот пример возможного результата работы скрипта.

```
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
```

Обратите внимание на свойства данного раздела (ACCOUNT_LOGIN, ACCOUNT_NAME, ACCOUNT_COMPANY, ACCOUNT_SERVER). В данном случае скрипт выполнялся на счету демонстрационного сервера "MetaQuotes-Demo". Очевидно, что это должен быть демо-счет, и об этом говорит не только название сервера, но и еще одно свойство ACCOUNT_TRADE_MODE, которому посвящен следующий раздел.

Идентификаторы счета обычно используют для привязки MQL-программ к конкретному торговому окружению. Пример такого алгоритма был представлен в разделе Сервисы.

6.3.3 Вид счета: реальный, демо или конкурсный

MetaTrader 5 поддерживает несколько видов счетов, которые могут быть открыты для клиента. Узнать вид текущего счета позволяет свойство ACCOUNT_TRADE_MODE, входящее в состав ENUM_ACCOUNT_INFO_INTEGER. Возможные значения данного свойства описаны в перечислении ENUM_ACCOUNT_TRADE_MODE.

Идентификатор	Описание
ACCOUNT_TRADE_MODE_DEMO	Демонстрационный торговый счет
ACCOUNT_TRADE_MODE_CONTEST	Конкурсный торговый счет
ACCOUNT_TRADE_MODE_REAL	Реальный торговый счет

Данное свойство удобно использовать для сборки демонстрационных (бесплатных) версий MQL-программ. Полнофункциональная, платная версия может требовать привязки к номеру счета, причем счет должен быть реальным.

Как мы видели в примере запуска скрипта *AccountInfo.mq5* в предыдущем разделе, счет на сервере "MetaQuotes-Demo" действительного относится к виду ACCOUNT_TRADE_MODE_DEMO.

6.3.4 Валюта счета

Баланс, прибыль, залог, комиссии и прочие финансовые показатели всегда в итоге пересчитываются в валюту счета, даже если для некоторых торговых операций спецификации предусматривают расчеты в других валютах, например, залог в маржинальной валюте пары Forex.

MQL5 API предоставляет два свойства, описывающих валюту счета, — непосредственно её наименование и точность представления сумм, то есть размер минимальной единицы измерения (такой как центы).

Идентификатор	Описание
ACCOUNT_CURRENCY	Валюта депозита (строка)
ACCOUNT_CURRENCY_DIGITS	Количество знаков после запятой для валюты счета, необходимых для точного отображения торговых результатов (целое число)

Например, для демо-счета, использованного для тестирования скрипта *AccountInfo* в разделе об Идентификации счета, свойство ACCOUNT_CURRENCY равнялось "USD", а точность ACCOUNT_CURRENCY_DIGITS — 2 цифрам после десятичной точки. Мы использовали свойство ACCOUNT_CURRENCY_DIGITS в классе *AccountMonitor* в методе *stringify* для значений типа *double* (в характеристиках счета они все связаны с деньгами).

6.3.5 Тип счета: неттинг или хедж

MetaTrader 5 поддерживает счета нескольких типов, в частности, неттинг и хедж. При неттинге разрешено иметь лишь одну позицию по каждому символу. При хеджинге можно открыть несколько позиций по символу, в том числе и разнонаправленных. Приказы, сделки и позиции будут подробно рассмотрены в следующих главах.

MQL-программа определяет тип счета путем запроса свойства ACCOUNT_MARGIN_MODE с помощью функции *AccountInfoInteger*. Как можно понять из названия свойства, оно описывает не

только тип счета, но и режим расчета маржи. Его возможные значения указаны в перечислении ENUM_ACCOUNT_MARGIN_MODE.

Идентификатор	Описание
ACCOUNT_MARGIN_MODE_RETAIL_NETTING	Внебиржевой рынок с учетом позиций в режиме "неттинг". Расчет маржи осуществляется на основе свойства инструмента SYMBOL_TRADE_CALC_MODE.
ACCOUNT_MARGIN_MODE_EXCHANGE	Биржевой рынок с учетом позиций в режиме "неттинг". Расчет маржи осуществляется на основе правил биржи с возможностью дисконтов, указанных брокером в настройках инструментов.
ACCOUNT_MARGIN_MODE_RETAIL_HEDGING	Внебиржевой рынок с независимым учетом позиций в режиме "хеджинг". Расчет маржи осуществляется на основе свойства инструмента SYMBOL_TRADE_CALC_MODE и с учетом размера захеджированной маржи SYMBOL_MARGIN_HEDGED.

Например, запуск скрипта *AccountInfo* в разделе об Идентификации счета показал, что счет имеет тип ACCOUNT_MARGIN_MODE_RETAIL_HEDGING.

6.3.6 Ограничения и разрешения для операций по счету

Среди свойств счета имеется несколько, представляющих собой ограничения на торговые операции, включая и полный их запрет. Все эти свойства относятся к перечислению ENUM_ACCOUNT_INFO_INTEGER и являются логическими флагами, за исключением ACCOUNT_LIMIT_ORDERS.

Идентификатор	Описание
ACCOUNT_TRADE_ALLOWED	Разрешение торговли на текущем счете
ACCOUNT_TRADE_EXPERT	Разрешение алгоритмической торговли экспертами и скриптами
ACCOUNT_LIMIT_ORDERS	Максимально допустимое количество действующих отложенных ордеров
ACCOUNT_FIFO_CLOSE	Признак того, что позиции нужно закрывать только по правилу FIFO

Поскольку наша книга посвящена программированию на MQL5, что включает и алготрейдинг, следует отметить, что запрет ACCOUNT_TRADE_EXPERT является столь же критическим, как и общий запрет на торговлю, когда ACCOUNT_TRADE_ALLOWED равно *false*. Брокер имеет возможность запретить торговлю через эксперты и скрипты, но при этом оставить торговлю вручную.

Свойство ACCOUNT_TRADE_ALLOWED, как правило, равно *false*, если подключение к счету выполнено по инвестиционному паролю.

Если значение свойства ACCOUNT_FIFO_CLOSE равно *true*, позиции по каждому символу разрешается закрывать только в том порядке, в котором они были открыты — сначала самую старую, затем более новую и т.д вплоть до последней. При попытке закрыть позиции в ином порядке будет получена ошибка. Для счетов без хеджингового учета позиций, то есть если свойство ACCOUNT_MARGIN_MODE не равно ACCOUNT_MARGIN_MODE_RETAIL_HEDGING, свойство ACCOUNT_FIFO_CLOSE всегда равно *false*.

В разделах Разрешения и Расписания торговых и котировочных сессий мы уже начали разрабатывать класс для выявления доступных MQL-программе торговых операций. Сейчас мы можем его дополнить проверками разрешений счета и привести к окончательной версии (*Permissions.mqh*).

Напомним, что уровни ограничений сведены в перечисление TRADE_RESTRICTIONS, которое после добавления двух новых элементов, связанных со свойствами счета, принимает такой вид.

```
class Permissions
{
   enum TRADE_RESTRICTIONS
   {
      NO_RESTRICTIONS = 0,
      TERMINAL_RESTRICTION = 1, // запрет пользователем для всех программ
      PROGRAM_RESTRICTION = 2,  // запрет пользователем для конкретной программы
      SYMBOL_RESTRICTION = 4,   // символ не торгуется согласно спецификации
      SESSION_RESTRICTION = 8,  // рынок закрыт согласно расписанию сессий
      ACCOUNT_RESTRICTION = 16, // пароль инвестора или ограничение брокера
      EXPERTS_RESTRICTION = 32, // брокер запретил алготрейдинг
   };
   ...
```

В ходе проверки MQL-программа может обнаружить несколько запретов по разным причинам, в связи с чем элементы кодируются отдельными битами, и окончательный результат может являться их суперпозицией.

Последние два ограничения как раз соответствуют новым свойствам и взводятся в методе *getTradeRestrictionsOnAccount*. Общая битовая маска обнаруженных ограничений (если они есть) формируется в переменной *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());
   }
   ...
```

Если вызывающий код не интересуется причиной блокировки, а требуется лишь определить возможность выполнения торговых операций, удобно пользоваться методом *isTradeOnAccountEnabled*, возвращающим логический признак (*true/false*).

По аналогичному принципу реорганизованы проверки свойств символа и терминала. Например, метод *getTradeRestrictionsOnSymbol* содержит исходный код, уже знакомый по предыдущей версии класса (с проверкой времени торговых сессий и режима торговли по символу), но возвращает маску флагов — если хоть один бит взведен, он описывает источник ограничения.

```
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;
   // проверка торговых сессий символа и установка found в true,
   // если время now попало внутрь одной из сессий
   ...
   
   // в дополнение к сессиям проверяем режим торговли
   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));
}
...
```

Наконец, общая проверка по всем потенциальным "инстанциям", включая (помимо предыдущих уровней) настройки терминала и программы, осуществляется в методах *getTradeRestrictions* и *isTradeEnabled*.

```
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));
}
```

Комплексную проверку разрешений торговли с новым классом демонстрирует скрипт *AccountPermissions.mq5*.

```
#include <MQL5Book/Permissions.mqh>

void OnStart()
{
   PrintFormat("Run on %s", _Symbol);
   if(!Permissions::isTradeEnabled()) // проверка для текущего символа, по умолчанию
   {
      Print("Trade is disabled for following reasons:");
      Print(Permissions::explainLastRestrictionBitMask());
   }
   else
   {
      Print("Trade is enabled");
   }
}
```

В случае обнаружения ограничений их битовую маску можно вывести в понятном строковом представлении с помощью метода *explainLastRestrictionBitMask*.

Вот несколько результатов запуска скрипта. В первых двух случаях в глобальных настройках терминала была запрещена торговля (свойства TERMINAL_TRADE_ALLOWED, и как следствие MQL_TRADE_ALLOWED равнялись *false*, что соответствует битам TERMINAL_RESTRICTION и PROGRAM_RESTRICTION).

На символе "USDRUB" в часы, когда рынок закрыт, получим дополнительно SESSION_RESTRICTION:

```
Trade is disabled for USDRUB following reasons:
TERMINAL_RESTRICTION PROGRAM_RESTRICTION SESSION_RESTRICTION
```

Для символа "SP500m", для которого торговля запрещена в принципе, появляется флаг SYMBOL_RESTRICTION.

```
Trade is disabled for SP500m following reasons:
TERMINAL_RESTRICTION PROGRAM_RESTRICTION SYMBOL_RESTRICTION SESSION_RESTRICTION
```

Наконец, разрешив торговлю в терминале, но зайдя на счет с инвесторским паролем, увидим запрет ACCOUNT_RESTRICTION на любом символе.

```
Run on XAUUSD
Trade is disabled for following reasons:
ACCOUNT_RESTRICTION
```

Заблаговременная проверка разрешений в MQL-программе позволит избежать серийных неудачных попыток отправки торговых приказов.

6.3.7 Маржинальные настройки счета

Для торговых роботов важно контролировать сумму заблокированных залоговых средств и доступную сумму для обеспечения новых сделок. В частности, при нехватке свободных средств программа не сможет совершить сделку. При поддержании открытых убыточных позиций сначала возникает запрос на пополнение счета (Margin Call), а в случае его невыполнения — происходит их принудительное закрытие брокером (Stop Out). Все связанные с этим свойства счета включены в перечисление ENUM_ACCOUNT_INFO_DOUBLE.

Идентификатор	Описание
ACCOUNT_MARGIN	Текущий размер зарезервированных залоговых средств на счете в валюте депозита
ACCOUNT_MARGIN_FREE	Текущий размер свободных средств на счете в валюте депозита, доступных для открытия позиции
ACCOUNT_MARGIN_LEVEL	Текущий уровень залоговых средств на счете в процентах (средства/маржа*100)
ACCOUNT_MARGIN_SO_CALL	Минимальный уровень залоговых средств, при котором требуется пополнение счета (Margin Call)
ACCOUNT_MARGIN_SO_SO	Минимальный уровень залоговых средств, при котором происходит принудительное закрытие самой убыточной позиции (Stop Out)
ACCOUNT_MARGIN_INITIAL	Размер средств, зарезервированных на счёте, для обеспечения гарантийной суммы по всем отложенным ордерам
ACCOUNT_MARGIN_MAINTENANCE	Размер средств, зарезервированных на счёте, для обеспечения минимальной суммы по всем открытым позициям

Пара свойств — ACCOUNT_MARGIN_SO_CALL и ACCOUNT_MARGIN_SO_SO — выражается в процентах или валюте депозита в зависимости от установленного режима ACCOUNT_MARGIN_SO_MODE (см. далее). Это свойство со способом измерения пороговых значений маржи для запроса на пополнение (Margin Call) или блокировку счета (Stop Out) входит в перечисление ENUM_ACCOUNT_INFO_INTEGER. Кроме того, там же указывается и общее кредитное плечо (используемое для расчета маржи инструментов некоторых типов).

Идентификатор	Описание
ACCOUNT_LEVERAGE	Размер предоставленного плеча
ACCOUNT_MARGIN_SO_MODE	Режим задания минимально допустимого уровня залоговых средств из перечисления ENUM_ACCOUNT_STOPOUT_MODE

А вот и элементы перечисления ENUM_ACCOUNT_STOPOUT_MODE.

Идентификатор	Описание
ACCOUNT_STOPOUT_MODE_PERCENT	Уровень задается в процентах
ACCOUNT_STOPOUT_MODE_MONEY	Уровень задается в валюте счета

Например, для варианта ACCOUNT_STOPOUT_MODE_PERCENT заданный процент (Margin Call или Stop Out) следует проверять относительно отношения собственных средств (эквити) к значению свойства ACCOUNT_MARGIN:

```
AccountInfoDouble(ACCOUNT_EQUITY) / AccountInfoDouble(ACCOUNT_MARGIN) * 100
    > AccountInfoDouble(ACCOUNT_MARGIN_SO_CALL)
```

Про свойство ACCOUNT_EQUITY и прочие финансовые показатели счета — подробнее в следующем разделе.

Правда, текущий уровень маржи в процентах уже предоставляется готовым в свойстве ACCOUNT_MARGIN_LEVEL. Это легко проверить с помощью скрипта *AccountInfo.mq5*, выводящего в журнал все свойства счета, включая и вышеперечисленные.

В разделе об Идентификации счета мы уже запускали этот скрипт. В тот момент была открыта одна позиция (1 лот USDRUB, составляющий 100000 USD), и финансовые показатели были следующими:

```
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
```

По марже 1000.00 USD легко проверить, что кредитное плечо счета ACCOUNT_LEVERAGE действительно равно 100 (согласно формуле расчета маржи для Forex и коэффициенту маржи, равному 1.0). Сумма маржи не требует пересчета по текущему курсу в валюту счета, поскольку она совпадает с базовой валютой инструмента.

Чтобы получить 992.12 в ACCOUNT_MARGIN_LEVEL достаточно разделить 9921.24 на 1000.00 и умножить на 100%.

Затем была открыта еще одна позиции 1 лот, а котировки пошли в неблагоприятную сторону, в результате чего ситуация изменилась:

```
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
```

Мы видим убыток в графе ACCOUNT_PROFIT и соответствующее уменьшение собственных средств ACCOUNT_EQUITY. Залог ACCOUNT_MARGIN увеличился пропорционально с 1000 до 2000, свободная маржа и уровень маржи уменьшились (но еще далеки от предельных 50% и 30%). Опять же уровень 425.70 получается как результат выражения 8513.93 / 2000.00 * 100.

Более практичным является расчет по данной формуле будущего уровня маржи перед предполагаемым открытием новой позиции. В этом случае необходимо увеличить размер существующего залога на дополнительную величину залога *X*. Кроме того, если сделка входа в рынок предполагает моментальное списание комиссии *C*, то её следовало бы, строго говоря, также учесть (правда, обычно она имеет размер существенно меньше залога и ею можно пренебречь, плюс ко всему API не предоставляет способа узнать комиссию заранее, до

совершения сделки: оценить её можно только по комиссиям уже совершенных сделок в истории торгов).

```
(AccountInfoDouble(ACCOUNT_EQUITY) - C) / (AccountInfoDouble(ACCOUNT_MARGIN) + X) * 1
   > AccountInfoDouble(ACCOUNT_MARGIN_SO_CALL)
```

Впоследствии мы научимся получать величину *X* с помощью функции *OrderCalcMargin*, но дополнительно к ней могут потребоваться корректировки согласно правилам, озвученным в разделе Маржинальные требования, в частности, с учетом возможного перекрытия позиций, дисконта и корректировки маржи.

Для варианта задания предела маржи в деньгах (ACCOUNT_STOPOUT_MODE_MONEY) проверка достаточности средств должна быть другой.

```
AccountInfoDouble(ACCOUNT_EQUITY) > AccountInfoDouble(ACCOUNT_MARGIN_SO_CALL)
```

Здесь комиссия опущена. Следует обратить внимание, что залог *X* под готовящуюся к открытию новую позицию теперь никак не влияет на оценку будущей маржи.

Однако, в любом случае желательно не нагружать депозит настолько, чтобы неравенства выполнялись едва-едва. Значения ACCOUNT_MARGIN_SO_CALL и ACCOUNT_MARGIN_SO_SO довольно близки, и хотя маржа на уровне ACCOUNT_MARGIN_SO_CALL — это всего лишь предупреждение трейдеру, от него легко попасть в ситуацию принудительного закрытия позиций. Именно поэтому в формулах использовано свойство ACCOUNT_MARGIN_SO_CALL.

6.3.8 Текущие финансовые показатели счета

MQL5 API позволяет контролировать несколько свойств счета с его основными финансовыми показателями. Все они включены в перечисление ENUM_ACCOUNT_INFO_DOUBLE.

Идентификатор	Описание
ACCOUNT_BALANCE	Баланс счета в валюте депозита
ACCOUNT_PROFIT	Размер текущей прибыли на счете в валюте депозита
ACCOUNT_EQUITY	Значение собственных средств на счете в валюте депозита
ACCOUNT_CREDIT	Размер предоставленного брокером кредита в валюте депозита
ACCOUNT_ASSETS	Текущий размер активов на счёте
ACCOUNT_LIABILITIES	Текущий размер обязательств на счёте
ACCOUNT_COMMISSION_BLOCKED	Текущая сумма заблокированных комиссий по счёту

В предыдущих разделах мы видели примеры значений этих свойств при запуске скрипта *AccountInfo.mq5* при разных условиях. Попробуйте сравнить у себя эти свойства для разных счетов.

В процессе торговли нас будут в первую очередь интересовать первые три свойства: баланс, прибыль (или убыток, если значение отрицательное) и собственные средства, в которых совокупно учитываются баланс, кредит, прибыль и накладные расходы (свопы и комиссии).

Комиссии могут учитываться по-разному, в зависимости от настроек брокера. Если комиссии сразу вычитаются из баланса счета в момент совершения сделок и отражаются в свойствах сделок, то свойство счета ACCOUNT_COMMISSION_BLOCKED будет равно 0. Однако если расчет комиссии откладывается на конец периода (например, день или месяц), заблокированная под комиссию сумма появится в этом свойстве — когда окончательная величина комиссии будет определена и списана с баланса в конце периода, свойство обнулится.

Свойства ACCOUNT_ASSETS и ACCOUNT_LIABILITIES заполняются, как правило, только при торговле на бирже и отражают текущую стоимость длинных и коротких позиций по ценным бумагам.

6.4 Создание экспертов

В этой главе мы начинаем изучение торгового API MQL5, используемого для реализации экспертов (советников). Данный тип программ является, пожалуй, наиболее сложным и требовательным в плане безошибочности кодирования и широты перечня задействованных технологий. В частности, нам потребуются многие навыки, полученные в процессе знакомства с материалами предыдущих глав, начиная от ООП и заканчивая прикладными аспектами работы с графическими объектами, индикаторами, символами и настройками программной среды.

В зависимости от выбранной торговой стратегии разработчику эксперта может потребоваться уделить особое внимание, например:

- скорости принятия решения и отправки приказов (для HFT — High Frequency Trading),
- подбору оптимальной корзины инструментов на основе их корреляций и волатильности (для кластерной торговли),
- динамическому расчету лотов и дистанции между ордерами (для мартингейла и сеточных стратегий),
- анализу новостей или внешних источников данных (об этом речь пойдет в 7-ой части книги).

Все подобные особенности должны будут оптимальным образом накладываться разработчиком на описываемые торговые механизмы, предоставляемые MQL5 API.

Далее мы подробно рассмотрим встроенные функции для управления торговой деятельностью, модель событий экспертов, специфические структуры данных, напомним основные принципы взаимодействия терминала и сервера, а также базовые понятия для алготрейдинга в MetaTrader 5: ордер, сделка и позиция.

Вместе с тем, из-за многогранности материала многие важные нюансы разработки экспертов, такие как тестирование и оптимизация, выделены в следующую главу.

Напомним, что мы уже описывали Конструирование MQL-программ различных типов, включая эксперты, а также Особенности запуска и остановки программ. Несмотря на то, что эксперт запускается на конкретном графике, для которого определен рабочий символ, нет никаких препятствий для того, чтобы централизованно управлять торговлей произвольным набором финансовых инструментов. Такие эксперты традиционно называются мультивалютными, хотя на самом деле в их портфеле могут быть CFD, акции, товары и тикеры прочих рынков.

В экспертах, также как и в индикаторах, работают Опорные события *OnInit* и *OnDeinit* — они не являются обязательными, но, как правило, присутствуют в коде для подготовки и штатного завершения работы программы: мы ими пользовались и еще будем пользоваться в примерах. В

отдельном разделе был приведен Обзор всех функций обработки событий: некоторые из них мы к данному моменту уже подробно изучили (например, события индикаторов *OnCalculate* или таймера *OnTimer*), а события, характерные только для экспертов (*OnTick*, *OnTrade*, *OnTradeTransaction*), будут описаны в этой главе.

В качестве торговых сигналов эксперты могут использовать самый широкий спектр исходных данных: котировки, тики, стакан цен, торговую историю счета или показания индикаторов — в последнем случае принципы создания экземпляров индикаторов и чтение значений из их буферов ничем не отличаются от тех, что были рассмотрены в главе Использование готовых индикаторов из MQL-программ. В примерах экспертов в следующих разделах мы продемонстрируем большинство из этих приемов.

Следует отметить, что торговые функции могут использоваться не только в экспертах, но и в скриптах. Мы приведем примеры для обоих вариантов.

6.4.1 Главное событие экспертов: OnTick

Событие *OnTick* генерируется терминалом для экспертов при появлении нового тика с ценой рабочего символа текущего графика, на котором запущен эксперт. Для обработки этого события в коде эксперта должна быть определена функция *OnTick* со следующим прототипом.

```
void OnTick(void)
```

Как видно, функция не имеет параметров. При необходимости само значение новой цены и прочих характеристик тика следует запрашивать с помощью вызова *SymbolInfoTick*.

С точки зрения реакции на событие нового тика данный обработчик является аналогом *OnCalculate* в индикаторах. Однако *OnCalculate* можно определять только в индикаторах, а *OnTick* — только в экспертах (если более точно, функция *OnTick* в коде индикатора, скрипта или сервиса будет просто проигнорирована).

Вместе с тем, эксперт не обязан обязательно содержать обработчик *OnTick*. Помимо этого события эксперты могут обрабатывать события *OnTimer*, *OnBookEvent*, *OnChartEvent* и выполнять все необходимые торговые операции из них.

Все события в экспертах обрабатываются одно за другим в порядке поступления, поскольку эксперты, как и все другие MQL-программы, являются однопоточными. Если в очереди уже есть событие *OnTick* или такое событие обрабатывается, то новые события *OnTick* в очередь не ставятся.

Событие *OnTick* генерируется независимо от того, запрещена или разрешена автоматическая торговля (кнопка *Алготрейдинг* в интерфейсе терминала). Запрет автоматической торговли означает только запрет на отправку торговых запросов из эксперта, но работа эксперта при этом не прекращается.

Следует напомнить, что события о тике генерируются только для одного символа — символа текущего графика. Если эксперт является мультивалютным, получение тиков с других символов следует организовать каким-либо альтернативным способом, например, с помощью индикатора-шпиона *EventTickSpy.mq5* или подписки на события стакана, как в *MarketBookQuasiTicks.mq5*.

В качестве простого примера рассмотрим эксперт *ExpertEvents.mq5*. В нем определены обработчики всех событий, которые обычно используются для запуска торговых алгоритмов. Некоторые прочие события (*OnTrade*, *OnTradeTransaction*, а также события тестера) мы изучим позднее.

Во всех обработчиках вызывается вспомогательная функция *Display*, которая выводит текущее время (метку миллисекундного системного счетчика) и название обработчика в многострочный комментарий.

```
#define N_LINES 25
#include <MQL5Book/Comments.mqh>

void Display(const string message)
{
   ChronoComment((string)GetTickCount() + ": " + message);
}
```

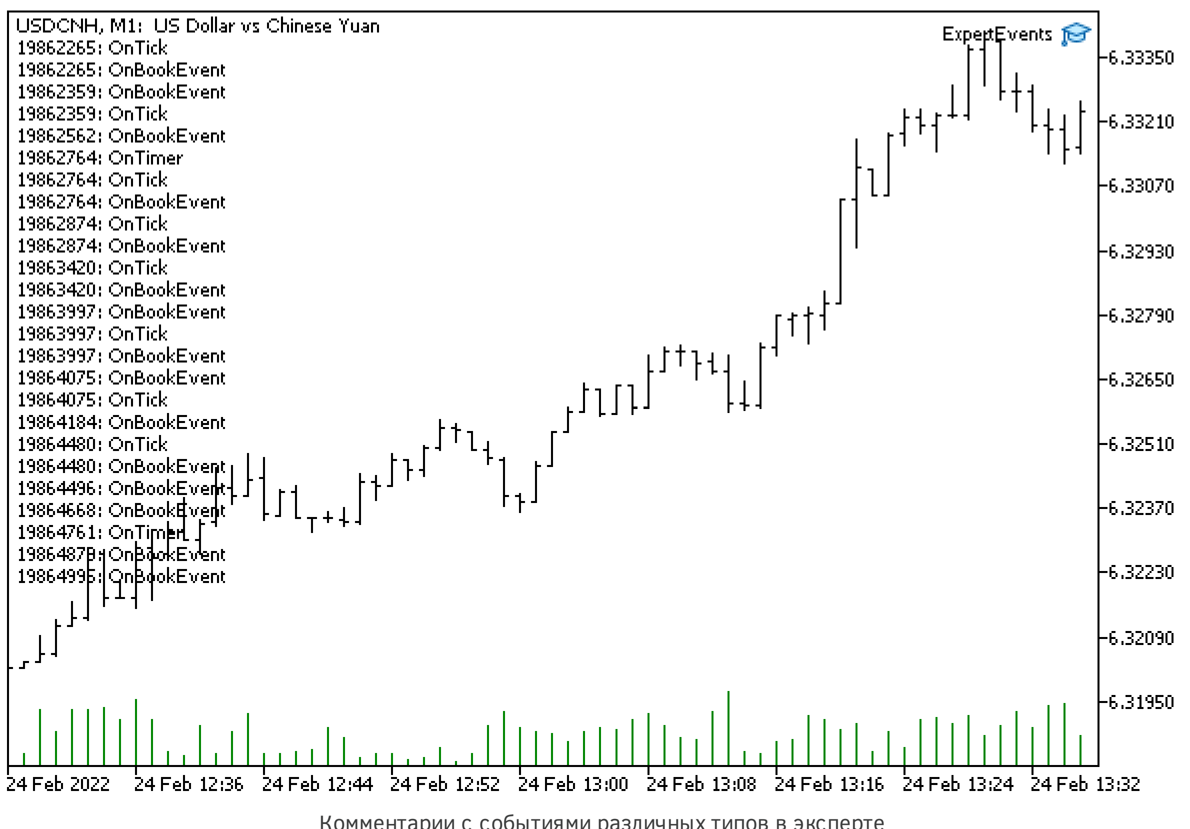
Событие *OnTick* будет вызываться автоматически по приходу новых тиков. Для событий таймера и стакана необходимо активировать соответствующие обработчики с помощью вызовов *EventSetTimer* и *MarketBookAdd* в *OnInit*.

```
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) // реагируем только на стаканы "своего" символа
   {
      Display(__FUNCTION__);
   }
}
```

Событие изменений графика также доступно: его можно использовать для торговли по разметке на основе графических объектов, по нажатию кнопок или горячих клавиш, а также по приходу пользовательских событий от других программ, например, индикаторов вроде *EventTickSpy.mq5*.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strir
{
   Display(__FUNCTION__);
}

void OnDeinit(const int)
{
   Print(__FUNCTION__);
   MarketBookRelease(_Symbol);
   Comment("");
}
```

На следующем скриншоте показан результат работы эксперта на графике.



Комментарии с событиями различных типов в эксперте

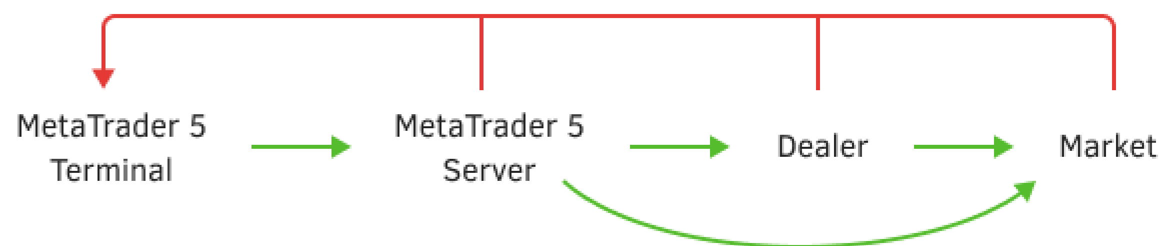
Обратите внимание, что событие *OnBookEvent* (если оно транслируется для символа) поступает чаще, чем *OnTick*.

6.4.2 Основные принципы и понятия: ордер, сделка, позиция

Прежде чем приступить к изучению разработки экспертов на MQL5, напомним общую архитектуру платформы и основные понятия, с помощью которых формализуется торговая деятельность.

MetaTrader 5 является клиентским терминалом, подключенным к многоуровневой серверной части, распределенной между компьютерами брокера, дилера или биржи. После того как пользователь заполняет приказ на выполнение торговой операции, тот подвергается нескольким этапам пересылки и проверки, после чего регистрируется или отклоняется у дилера или на бирже. Зарегистрированный на рынке ордер может затем исполниться или не исполниться в

зависимости от обстоятельств, например, наличия ликвидности, скорости изменения цены, приостановки торгов по символу или технических проблем.



Общая схема обработки торгового запроса

Здесь зелеными стрелками обозначено успешное выполнение торговой операции по мере её продвижения из терминала в рынок, а красными — потенциальный отказ от исполнения.

Аналогичные инстанции проходят и приказы, формируемые MQL-программами. При неблагоприятном исходе MQL5 API позволит нам узнать причину отказа через код ошибки.

Весь этот процесс выражается (и документируется в отчетах) в трех основополагающих терминах: ордер (он же приказ), сделка, позиция.

Ордер — это распоряжение трейдера для брокерской компании купить или продать финансовый инструмент. MetaTrader 5 поддерживает несколько типов ордеров, но в упрощенном виде их можно условно поделить на рыночные, отложенные и специальные защитные уровни *Take Profit* и *Stop Loss*.

В результате успешного исполнения ордера в торговой системе происходит сделка. В частности, сделка может быть заключена по текущей цене в случае рыночного ордера или при срабатывании отложенного ордера, когда цена достигает предопределенной в нем величины. Иными словами сделка — это факт покупки или продажи того или иного финансового инструмента.

Следует учитывать, что в некоторых условиях результатом исполнения ордера может быть сразу несколько сделок. Например, если в стакане цен нет предложения "товара" в достаточном количестве, то приказ о его покупке может выполняться за счет разных встречных заявок, в том числе и по слегка отличной цене.

Купленный или проданный согласно сделке финансовый инструмент формирует, соответственно, длинную или короткую позицию, которая отражается в активах/пассивах торгового счета. В результате последующего изменения цены инструмента позиции на счете образуется плавающая прибыль или убыток, которые можно зафиксировать, ликвидировав позицию обратными торговыми операциями (ордерами и сделками). В зависимости от типа торгового счета (неттинг или хедж) сделки по одному и тому же инструменту модифицируют единственную нетто-позицию или создают/удаляют независимые позиции.

Более подробную информацию можно получить в руководстве пользователя терминала.

Все ордера, сделки и позиции попадают в торговую историю счета.

Далее мы рассмотрим программное API, включающее функции для отправки торговых приказов, получения текущего состояния "портфеля" на счету, проверки маржинальной нагрузки и потенциальных прибылей/убытков, а также анализа торговой истории.

6.4.3 Типы торговых операций

Торговля в MQL5 осуществляется посредством отправки приказов с помощью функции *OrderSend* — её мы изучим в одном из следующих разделов, потому что для её описания требуется сначала познакомиться с несколькими понятиями.

Самым первым новым понятием будет тип торговой операции. Каждый торговый приказ содержит указание на тип запрашиваемой торговой операции, и позволяет выполнить такие действия, как открытие и закрытие позиций, а также установку, модификацию и удаление отложенных ордеров. Все типы торговых операций описаны в перечислении ENUM_TRADE_REQUEST_ACTIONS.

Идентификатор	Описание
TRADE_ACTION_DEAL	Установить торговый ордер на немедленное совершение сделки с указанными параметрами (поставить рыночный ордер)
TRADE_ACTION_PENDING	Установить торговый ордер на совершение сделки при указанных условиях (отложенный ордер)
TRADE_ACTION_SLTP	Изменить значения *Stop Loss* и *Take Profit* у открытой позиции
TRADE_ACTION_MODIFY	Изменить параметры ранее установленного ордера
TRADE_ACTION_REMOVE	Удалить ранее выставленный отложенный ордер
TRADE_ACTION_CLOSE_BY	Закрыть позицию встречной

При запросе TRADE_ACTION_DEAL и TRADE_ACTION_PENDING программе потребуется указать конкретный тип ордера. Это еще одно важное понятие, которое имеет свое отображение в MQL5 API, и мы рассмотрим его в следующем разделе.

6.4.4 Типы ордеров

Как известно, MetaTrader 5 поддерживает несколько типов ордеров: 2 рыночных — на покупку и продажу по текущей цене, — а также 6 отложенных с предопределенными уровнями активации выше и ниже рынка. Все эти типы доступны в MQL5 API и описаны элементами перечисления ENUM_ORDER_TYPE. Каким именно образом в программе можно создать ордер конкретного типа, мы рассмотрим позднее, а пока познакомимся с перечислением.

Идентификатор	Описание
ORDER_TYPE_BUY	Рыночный ордер на покупку
ORDER_TYPE_SELL	Рыночный ордер на продажу
ORDER_TYPE_BUY_LIMIT	Отложенный ордер *Buy Limit*
ORDER_TYPE_SELL_LIMIT	Отложенный ордер *Sell Limit*
ORDER_TYPE_BUY_STOP	Отложенный ордер *Buy Stop*
ORDER_TYPE_SELL_STOP	Отложенный ордер *Sell Stop*
ORDER_TYPE_BUY_STOP_LIMIT	Отложенный ордер *Buy Limit* будет выставлен по достижении ценой заданного верхнего уровня
ORDER_TYPE_SELL_STOP_LIMIT	Отложенный ордер *Sell Limit* будет выставлен по достижении ценой заданного нижнего уровня
ORDER_TYPE_CLOSE_BY	Ордер на закрытие одной позиции другой встречной позицией

Последний элемент соответствует действию по закрытию встречных позиций: такая возможность существует только на счетах с хеджингом и для финансовых инструментов, в свойствах которых разрешены подобные операции (SYMBOL_ORDER_CLOSEBY).

Общие принципы активации отложенных ордеров может напомнить следующая картинка. На ней серым цветом показаны предполагаемые будущие движения цены, но в текущий момент времени, разумеется, не известно, какой прогноз окажется верным.

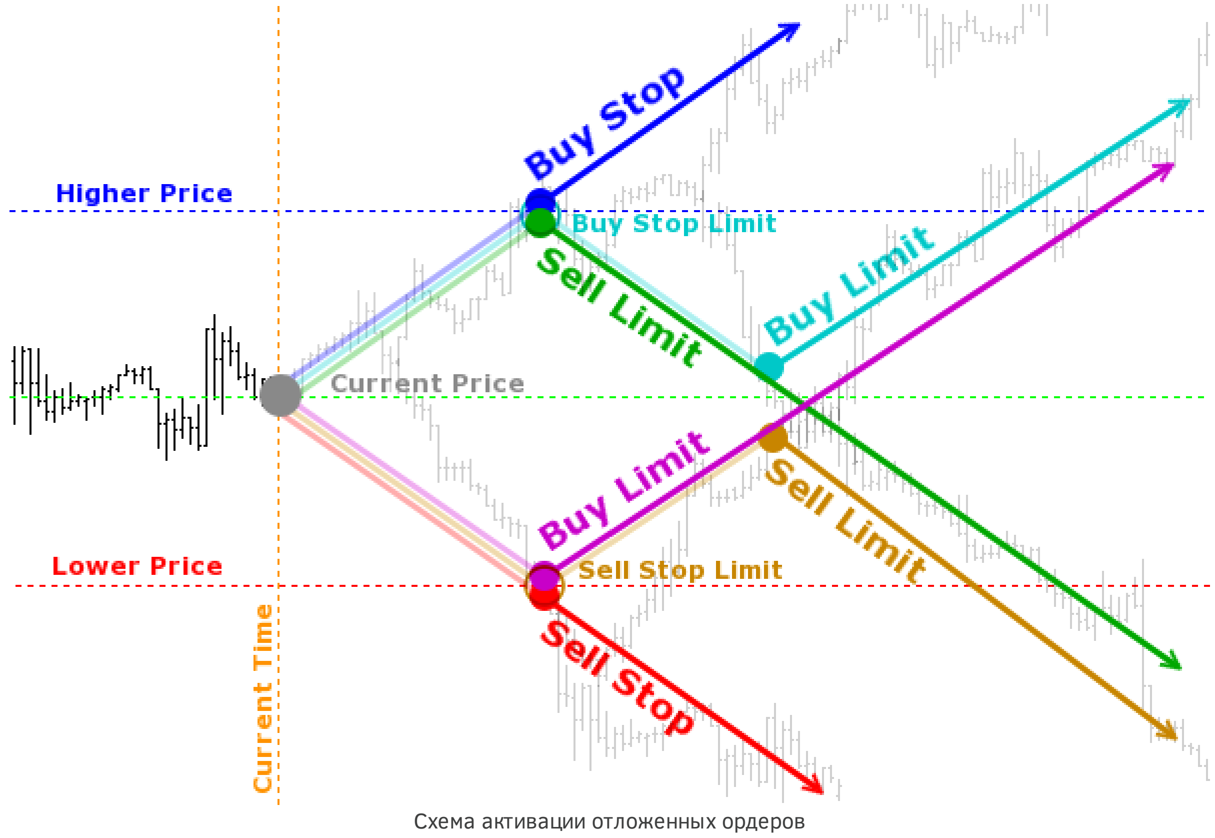


Схема активации отложенных ордеров

Отложенные ордера *Buy Stop* и *Sell Stop* работают по принципу пробоя уровня: для *Buy Stop* этот уровень должен располагаться выше текущей цены, а для *Sell Stop* — ниже. Иными словами, на заданном уровне предполагается совершение покупки или продажи в расчете на дальнейшую торговлю по тренду.

Buy Limit и *Sell Limit* реализуют стратегию отбоя от уровня, причем в этом случае цена активации покупки находится ниже текущей цены, а цена продажи — выше. Это подразумевает смену тренда или колебания в коридоре. На вышеприведенной схеме одни и те же верхний (Higher Price) и нижний (Lower Price) уровни активации отложенных ордеров используются для иллюстрации как пробоя, так и отскока.

Отложенные ордера можно ставить на текущую цену — они при этом будут с большой вероятностью сразу же исполнены. Кроме того, такой прием с лимитными ордерами гарантирует цену сделки, не хуже запрошенной, в отличие от рыночного ордера.

Ордера типов *Buy Stop Limit* и *Sell Stop Limit* в результате своей активации не входят в рынок, а устанавливают, соответственно, отложенные ордера *Buy Limit* или *Sell Limit* на неких дополнительных уровнях, заданных в исходном приказе.

Для биржевых инструментов лимитные ордера (*Buy Limit*, *Sell Limit*), как правило, напрямую выводятся в стакан и видны остальным участникам рынка.

В отличие от этого, *Stop* и *Stop Limit* ордера (*Buy Stop*, *Sell Stop*, *Buy Stop Limit* и *Sell Stop Limit*) не выводятся во внешнюю торговую систему напрямую. До достижения стоп-цены данные типы ордеров обрабатываются внутри платформы MetaTrader 5. При достижении стоп-цены, указанной в Buy Stop или Sell Stop ордере, выполняется соответствующая рыночная операция. При достижении стоп-цены, указанной в *Buy Stop Limit* или *Sell Stop Limit* ордере, выставляется соответствующая лимитная заявка.

В режиме биржевого исполнения цена, указываемая при выставлении лимитных ордеров, не проверяется. Ее можно указать выше текущей цены *Ask* (для ордеров на покупку) и ниже цены *Bid* (для ордеров на продажу). При выставлении ордера с такой ценой он практически сразу срабатывает и превращается в рыночный.

Следует учесть, что не все типы ордеров могут быть разрешены для конкретного финансового инструмента: свойство SYMBOL_ORDER_MODE описывает флаги разрешенных типов ордеров.

6.4.5 Режимы исполнения ордеров по цене и объемам

При отправке торговых запросов нам нужно будет указать в алгоритме особым образом цену и объем покупки/продажи. При этом необходимо учитывать, что на финансовых рынках нет гарантии, что в данный момент по финансовому инструменту доступен весь запрашиваемый объем по желаемой цене. Поэтому проведение торговых операций регулируется с помощью режимов (или политик) исполнения по цене и объему. Они определяют правила для случаев, когда в процессе отправки запроса изменилась цена или его не удается удовлетворить в полном объеме.

В главе про символы, в разделе Торговые условия и режимы исполнения приказов мы уже встречались с настройками исполнения ордеров по цене (SYMBOL_TRADE_EXEMODE) и заливки ордеров по объему (SYMBOL_FILLING_MODE), которые устанавливает брокер. В соответствие с тем, какие режимы SYMBOL_FILLING_MODE доступны, MQL-программа должна выбрать режим заливки нового формируемого ордера в специальной структуре *MqlTradeRequest* (скоро мы увидим это на практике).

Варианты сведены в перечисление ENUM_ORDER_TYPE_FILLING: их идентификаторы перекликаются с идентификаторами SYMBOL_FILLING_MODE.

Политика исполнения (Значения)	Описание
ORDER_FILLING_FOK (0)	Все или Ничего (Fill or Kill)
ORDER_FILLING_IOC (1)	Немедленно и хотя бы Частично (Immediate or Cancel)
ORDER_FILLING_RETURN (2)	Вернуть (Return)

При политике ORDER_FILLING_FOK ордер может быть исполнен исключительно в указанном объеме. Если на рынке в данный момент не присутствует достаточного объема финансового инструмента, то ордер не будет исполнен. Необходимый объем может быть составлен из нескольких предложений, доступных в данный момент на рынке. Возможность использования FOK-ордеров определяется наличием разрешения SYMBOL_FILLING_FOK.

При политике ORDER_FILLING_IOC трейдер соглашается совершить сделку по максимально доступному на рынке объему в пределах указанного в ордере. В случае невозможности полного покрытия ордер будет исполнен на доступный объем, а недостающий объем будет отменен. Возможность использования IOC-ордеров определяется наличием разрешения SYMBOL_FILLING_IOC.

При политике ORDER_FILLING_RETURN в случае частичного исполнения ордер с остаточным объемом не снимается, а продолжает действовать. Это — режим по умолчанию, доступный всегда. Однако есть одно исключение: Return-ордера не разрешены в режиме исполнения по рынку (SYMBOL_TRADE_EXECUTION_MARKET в свойстве символа SYMBOL_TRADE_EXEMODE).

Таким образом, перед отправкой рыночного (не отложенного) ордера MQL-программе следует правильно выставить одну из политик ORDER_TYPE_FILLING на основе свойства SYMBOL_FILLING_MODE соответствующего финансового инструмента: напомним, что в этом свойстве содержится комбинация битовых флагов разрешенных режимов.

Для отложенных ордеров независимо от режима исполнения SYMBOL_TRADE_EXEMODE необходимо использовать политику ORDER_FILLING_RETURN, так как такие ордера будут заполняться объемом позднее и по тем правилам, которые брокер установит на тот момент.

> В отличие от политики заливки объема ордера, режим его исполнения по цене нельзя выбирать — он предопределен брокером для каждого символа, и это влияет на то, какие поля структуры *MqlTradeRequest* потребуется заполнять перед отправкой торгового запроса.

Применение политик заливки в зависимости от режимов исполнения можно представить в виде таблицы ('+' — разрешено, '-' — запрещено, '±' — зависит от настроек символа):

Политика заполнения Режим исполнения	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	±	±	+
Отложенные	-	-	+

В режимах исполнения SYMBOL_TRADE_EXECUTION_INSTANT и SYMBOL_TRADE_EXECUTION_REQUEST разрешены все политики заливки объемов.

6.4.6 Сроки действия отложенных ордеров

Для отложенных ордеров важной характеристикой является режим их истечения. В MQL5 API срок действия ордера можно задать в поле *type_time* специальной структуры *MqlTradeRequest* при отправке торгового запроса функцией *OrderSend*. Допустимые значения описаны в перечислении ENUM_ORDER_TYPE_TIME.

Идентификатор (Значение)	Описание
ORDER_TIME_GTC (0)	Ордер будет находиться в очереди до тех пор, пока не будет снят
ORDER_TIME_DAY (1)	Ордер будет действовать только в течение текущего торгового дня
ORDER_TIME_SPECIFIED (2)	Ордер будет действовать до даты истечения
ORDER_TIME_SPECIFIED_DAY (3)	Ордер будет действовать до 23:59:59 указанного дня (если это время не попадает в торговую сессию, истечение наступит в ближайшее следующее торговое время)

Следует отметить, что для каждого финансового инструмента существует два свойства SYMBOL_EXPIRATION_MODE и SYMBOL_ORDER_GTC_MODE, которые определяют Правила истечения сроков отложенных ордеров по этому инструменту. При формировании ордера MQL-программа может выбрать один из разрешенных режимов. Пример мы рассмотрим после изучения функции *OrderSend*.

6.4.7 Расчет залога для будущего ордера: OrderCalcMargin

Прежде чем отправлять торговый приказ на сервер, MQL-программа может вычислить размер залога, который потребуется для планируемой торговой операции, с помощью функции

OrderCalcMargin. Рекомендуется всегда это делать во избежание чрезмерной нагрузки на депозит.

bool OrderCalcMargin(ENUM_ORDER_TYPE action, const string symbol,
  double volume, double price, double &margin)

Функция вычисляет размер маржи, необходимой для указанного типа ордера *action* и финансового инструмента *symbol* в объеме *volume* лотов. При этом учитываются настройки текущего счета, но не учитываются имеющиеся отложенные ордера и открытые позиции. Перечисление ENUM_ORDER_TYPE было представлено в разделе Типы ордеров.

Значение маржи (в валюте счета) записывается в передаваемый по ссылке параметр *margin*.

Следует особо подчеркнуть, что это — оценка маржи для отдельно взятой новой позиции или ордера, а не общая величина залога, каким он станет после исполнения. Более того, оценка делается, как если бы на текущем счете не было других отложенных ордеров и открытых позиций. В реальности значение маржи зависит от многих факторов, включая другие ордера и позиции, и может меняться при изменении рыночного окружения (например, кредитного плеча).

Функция возвращает признак успеха (*true*) или ошибки (*false*). Код ошибки можно привычным образом получить из переменной *_LastError*.

> Функцию *OrderCalcMargin* можно использовать только в экспертах и скриптах. Для расчета маржи в индикаторах нужно реализовать альтернативный способ, например, запускать вспомогательный эксперт в объекте-графике с передачей ему параметров и получением результата через механизм событий или самостоятельно описать в MQL5 вычисления по формулам согласно типам инструментов. В следующем разделе мы приведем пример такой реализации, вместе с оценкой потенциальной прибыли/убытка.

Мы могли бы написать простой скрипт, который вызывает *OrderCalcMargin* для символов из *Обзора рынка*, и сравнить значения маржи для них. Вместо этого слегка усложним задачу и рассмотрим заголовочный файл *LotMarginExposure.mqh*, который позволяет оценить загрузку депозита и уровень маржи после открытия позиции с предопределенным уровнем риска. Чуть позже мы познакомимся с функцией *OrderCheck*, способной предоставить аналогичную информацию. Однако наш алгоритм дополнительно сможет решать обратную задачу — выбирать размер лота по заданным уровням загрузки или риска.

Применение новых возможностей продемонстрировано в неторгующем эксперте *LotMarginExposureTable.mq5*.

> В принципе, тот факт, что MQL-программа реализована как эксперт не означает, что в ней обязательно должны выполняться торговые операции. Очень часто, как и в нашем случае, в виде эксперта создаются различные утилиты. Их преимущество по сравнению со скриптами в том, что они остаются на графике и могут бесконечно долго выполнять свои функции в ответ на те или иные события.

В новом эксперте мы задействуем навыки создания интерактивного графического интерфейса с помощью объектов. Проще говоря, для заданного списка символов эксперт выведет на график таблицу с несколькими колонками маржинальных показателей, причем по каждой из колонок таблицу можно сортировать. Перечень колонок мы приведем чуть позже.

Поскольку анализ лотов, маржи, загрузки депозита является общей востребованной задачей, выделим реализацию в отдельный заголовочный файл *LotMarginExposure.mqh*.

Все функции файла объединены в пространство имен во избежание конфликтов и для наглядности (указание контекста перед вызовом внутренней функции дает знать о происхождении и размещении этой функции).

```
namespace LEMLR
{
   ...
};
```

Аббревиатура *LEMLR* означает "Lot, Exposure, Margin Level, Risk".

Основные расчеты выполняются в функции *Estimate*. Учитывая прототип встроенной функции *OrderCalcMargin*, легко предположить, что в параметрах функции *Estimate* потребуется передать название символа, тип ордера, объем и цену. Но это не все, что нам понадобится.

```
bool Estimate(const ENUM_ORDER_TYPE type, const string symbol, const double lot,
   const double price,...)
```

Мы предполагаем оценить несколько показателей торговой операции, которые взаимосвязаны и могут рассчитываться в разных направлениях, в зависимости от того, что пользователь ввел как исходные данные, а что хочет рассчитать. Например, с помощью вышеперечисленных параметров легко узнать новый уровень маржи и загрузку счета. Напомним, что их формулы похожи с точностью "до наоборот":

```
Ml = money / margin * 100
Ex = margin / money * 100
```

Здесь переменной *margin* обозначена сумма маржи, для получения которой достаточно вызвать *OrderCalcMargin*.

Однако трейдеры часто предпочитают исходить из предопределенного уровня загрузки или маржи, и рассчитывать для них объем. Более того, существует не менее популярный подход с расчетом лота на основе риска. Под риском понимается размер потенциального убытка от торговли при неблагоприятном движении цены, в результате чего будет уменьшаться содержимое другой переменной из вышеприведенных формул — *money*.

Для вычисления убытка важно знать волатильность финансового инструмента на торговом периоде (длительности стратегии) или дистанцию предполагаемого пользователем стоп-лосса.

Поэтому перечень параметров функции *Estimate* расширяется.

```
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,...)
```

В параметре *exposure* укажем желаемую загрузку депозита в процентах, а в параметре *riskLevel* — часть депозита (тоже в процентах), которой мы готовы рискнуть. Для расчетов на основе риска можно передать размер стоп-лосса в пунктах в параметре *riskPoints*. Когда он равен 0, в дело вступает параметр *riskPeriod*: в нем указывается период, для которого алгоритм автоматически рассчитает диапазон котировок символа в пунктах. Наконец, в параметре *money* можно задать произвольный размер свободных средств для оценки лота. Некоторые трейдеры условно делят депозит между несколькими роботами. Когда *money* равно 0, функция заполнит эту переменную свойством *AccountInfoDouble(ACCOUNT_MARGIN_FREE)*.

Осталось решить, как возвращать результаты работы функции. Поскольку она способна оценить много торговых показателей и несколько вариантов объема, имеет смысл определить структуру *SymbolLotExposureRisk*.

```
struct SymbolLotExposureRisk
{
   double lot;                      // запрошенный объем (или минимальный)
   int atrPointsNormalized;         // диапазон цен нормализованный по размеру тик
   double atrValue;                 // диапазон как сумма прибыли/убытка для 1 лот
   double lotFromExposureRaw;       // ненормализованный (м.б. меньше минимальногс
   double lotFromExposure;          // нормализованный лот из загрузки депозита
   double lotFromRiskOfStopLossRaw; // ненормализованный (м.б. меньше минимальногс
   double lotFromRiskOfStopLoss;    // нормализованный лот из риска
   double exposureFromLot;          // загрузка исходя из объема 'lot'
   double marginLevelFromLot;       // уровень маржи из объема 'lot'
   int lotDigits;                   // количество цифр в нормализованных лотах
};
```

Поле *lot* в структуре содержит переданный в функцию *Exposure* лот, если он не равен 0. Если переданный лот — нулевой, вместо него подставляется свойство символа SYMBOL_VOLUME_MIN.

Под расчетные значения объемов исходя из загрузки депозита и риска выделено по два поля: с суффиксом *Raw* (*lotFromExposureRaw*, *lotFromRiskOfStopLossRaw*) и без него (*lotFromExposure*, *lotFromRiskOfStopLoss*). *Raw*-поля содержат "чистый арифметический" результат, который может не соответствовать спецификации символа. В полях без суффикса лоты нормализованы с учетом минимума, максимума и шага. Такое дублирование полезно, в частности, для тех случаев, когда расчет выдает значения меньше минимального лота (например, *lotFromExposureRaw* равно 0.023721 при минимуме 0.1, из-за чего *lotFromExposure* сводится к нулю): тогда по содержимому *Raw*-поля можно оценить, сколько средств добавить или насколько повысить риск, чтобы добраться до минимального лота.

Опишем последний выходной параметр функции *Estimate* как ссылку на данную структуру. В теле функции постепенно заполним все её поля. Прежде всего, получим размер маржи для одного лота с помощью вызова *OrderCalcMargin* и сохраним в локальную переменную *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;
   }
   ...
```

Если входная цена не указана, т.е. *price* равно 0, вспомогательная функция *GetCurrentPrice* возвращает подходящую цену на основе типа ордера: для покупок будет взято свойство символа SYMBOL_ASK, и для продаж — SYMBOL_BID. Эта и другие вспомогательные функции здесь опущены, с их содержимым можно ознакомиться в прилагаемых исходных кодах.

Если расчет маржи завершился ошибкой или получено нулевое значение, функция *Estimate* вернет *false*.

Следует иметь в виду, что нулевая маржа может быть нормой или ошибкой, в зависимости от инструмента и типа ордера. Так для биржевых тикеров отложенные ордера облагаются залогом, а для внебиржевых — нет (то есть залог 0 корректен). Этот нюанс следует учитывать в вызывающем коде: он должен запрашивать маржу только по таким сочетаниям символов и типов операций, для которых она имеет смысл и предполагается ненулевой.

Имея залог для одного лота, мы можем рассчитать количество лотов для обеспечения заданной загрузки депозита.

```
   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);
   ...
```

Вспомогательная функция *NormalizeLot* приведена ниже.

Для того чтобы получить лот в зависимости от риска и волатильности потребуется чуть больше вычислений.

```
   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)));
   // округление по размеру тика
   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);
   ...
```

Здесь мы находим стоимость одного пункта инструмента и диапазон его изменений за указанный период, после чего уже рассчитываем лот.

Наконец, получим загрузку счета и уровень маржи для заданного лота.

```
   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;
}
```

В случае успешного расчета функция вернет *true*.

А вот и функция *NormalizeLot* с сокращениями (все проверки на 0 для простоты опущены). Подробности про соответствующие свойства можно найти в разделе Разрешенные объемы торговых операций.

```
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;
}
```

Приведенная выше реализация *Estimate* не учитывает поправки на перекрывающиеся позиции — как правило, они приводят к уменьшению залога, поэтому текущая оценка загрузки счета и уровня маржи может быть более пессимистичной, чем получится в реальности, но это обеспечивает дополнительную защиту. Желающие могут добавить код для анализа состава уже замороженных средств счета (их общая сумма содержится в свойстве счета ACCOUNT_MARGIN) в

разбивке по позициям и ордерам: тогда станет возможным учесть потенциальный взаимозачет маржи с новым ордером (например, учтётся только наибольшая позиция из встречных или применится пониженная ставка хеджированной маржи, см. подробности в разделе Маржинальные требования).

Пришло время применить оценку маржи и лотов на практике в эксперте *LotMarginExposureTable.mq5*. С учетом того, что *Raw*-поля мы будем показывать только в тех случаях, когда нормализация лотов привела к их обнулению, общее количество колонок в результирующей таблице показателей равно 8.

```
#include <MQL5Book/LotMarginExposure.mqh>
#define TBL_COLUMNS 8
```

Во входных параметрах предусмотрим задание типа ордера, перечня анализируемых символов (список с разделителем-запятой), размер доступных средств, а также лот, целевую загрузку депозита, уровень маржи и риска.

```
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;
```

Для отложенных типов ордеров необходимо обязательно выбирать биржевые символы, поскольку для иных символов будет получена нулевая маржа, что спровоцирует ошибку в функции *Estimate*. Если список символов оставлен пустым, эксперт обработает только символ текущего графика. Нулевые по умолчанию значения в параметрах *Money* и *Lot* означают, соответственно, текущий размер свободных средств на счете и минимальный лот по каждому символу.

Значение 0 в параметре *RiskPoints* предписывает получение диапазона цен за *RiskPeriod* (по умолчанию неделя).

Входной параметр *UpdateFrequency* задает периодичность повторения пересчета в секундах. Если оставить его равным нулю, пересчет выполняется на каждом новом баре.

```
input int UpdateFrequency = 0; // UpdateFrequency (sec, 0 - once per bar)
```

В глобальном контексте описаны: массив символов (позднее заполняемый парсингом входного параметра *WorkList*) и временная метка последнего успешного расчета.

```
string symbols[];
datetime lastTime;
```

При запуске включаем секундный таймер.

```
void OnInit()
{
   Comment("Starting...");
   lastTime = 0;
   EventSetTimer(1);
}
```

В обработчике таймера обеспечиваем первый вызов основного расчета в *OnTick*, если *OnTick* еще не был вызван по приходу тика — такое может быть, например, на выходных или на спокойном рынке. Также *OnTimer* является точкой входа для повторных расчетов с заданной частотой.

```
void OnTimer()
{
   if(lastTime == 0) // расчет первый раз (если не успел сработать OnTick)
   {
      OnTick();
      Comment("Started");
   }
   else if(lastTime != -1)
   {
      if(UpdateFrequency <= 0) // если частоты нет, работаем по новым барам в OnTick
      {
         EventKillTimer();      // и таймер больше не нужен
      }
      else if(TimeCurrent() - lastTime >= UpdateFrequency)
      {
         lastTime = LONG_MAX; // предотвращаем повторный вход в эту ветвь if
         OnTick();
         if(lastTime != -1)   // отработали без ошибки
         {
            lastTime = TimeCurrent(); // обновляем метку времени
         }
      }
      Comment("");
   }
}
```

В обработчике *OnTick* прежде всего проверяем входные параметры и преобразуем список символов в массив строк. В случае обнаружения проблем в *lastTime* записывается признак ошибки: значение -1, и обработка последующих тиков прерывается в самом начале.

```
void OnTick()
{
   if(lastTime == -1) return; // уже была ошибка, выходим

   if(UpdateFrequency <= 0)   // если частота обновления не задана
   {
      if(lastTime == iTime(NULL, 0, 0)) return; // ждем нового бара
   }
   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;
   }
   ...
```

В частности, ошибкой считается, если входные значения *Exposure* и *RiskLevel* не лежат в диапазоне от 0 до 100, присущем процентам. В случае нормальных входных данных обновляем временную метку, описываем структуру *LEMLR::SymbolLotExposureRisk* для приема расчетных показателей из функции *LEMLR::Estimate* (по одному символу), а также двумерный массив *LME* (от "Lot Margin Exposure") для сбора показателей по всем символам.

```
   lastTime = UpdateFrequency > 0 ? TimeCurrent() : iTime(NULL, 0, 0);

   LEMLR::SymbolLotExposureRisk r = {};

   double LME[][13];
   ArrayResize(LME, ns);
   ArrayInitialize(LME, 0);
   ...
```

В цикле по символам вызываем функцию *LEMLR::Estimate* и заполняем массив *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]);
}
...
```

В качестве индексов массива использованы элементы специального перечисления LME_FIELDS, которое предоставляет одновременно названия и номера для показателей из структуры.

```
enum LME_FIELDS // 10 полей + 3 дополнительных свойства символа
{
   eLot,
   eAtrPointsNormalized,
   eAtrValue,
   eLotFromExposureRaw,
   eLotFromExposure,
   eLotFromRiskOfStopLossRaw,
   eLotFromRiskOfStopLoss,
   eExposureFromLot,
   eMarginLevelFromLot,
   eLotDig,
   eMinLot,
   eContract,
   eSymbol
};
```

Свойства SYMBOL_VOLUME_MIN и SYMBOL_TRADE_CONTRACT_SIZE добавляются для справки. Название символа "упаковывается" в приблизительное значение типа *double* с помощью функции *pack2double*, чтобы впоследствии реализовать унифицированную сортировку по любому из полей, включая и названия.

```
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;
}
```

На этом этапе мы могли бы уже запустить эксперт и распечатать результаты в журнале, примерно так.

```
ArrayPrint(LME);
```

Но смотреть все время в журнал неудобно. Кроме того, единое форматирование величин из разных колонок, и тем более представление "упакованных" строк в *double* никак нельзя назвать дружественным. Поэтому был разработан класс табло (*Tableau.mqh*) для отображения произвольной таблицы на графике. Помимо того что при подготовке таблицы мы можем сами управлять форматом каждого поля (в перспективе — подсвечивать разным цветом), этот класс позволяет интерактивно сортировать таблицу по любой колонке: первый щелчок мышью сортирует в одном направлении, второй — в обратном, третий — отменяет сортировку.

Здесь мы не будем описывать класс подробно. При необходимости можно ознакомиться с его исходным кодом. Важно лишь отметить, что интерфейс строится на базе графических объектов. Фактически ячейки таблицы формируются объектами типа OBJ_LABEL, и все их свойства уже знакомы читателю. Однако кое-какие технические приемы, использованные в исходном коде табло — в частности, работа с графическими ресурсами и измерение отображаемого текста, будут представлены позднее, в седьмой части.

Конструктор класса *Tableau* принимает несколько параметров:

- prefix — префикс для имен создаваемых графических объектов;
- rows — количество строк;
- cols — количество колонок;
- height — высота строки в пикселях (-1 означает удвоенный размер шрифта);
- width — ширина ячейки в пикселях;
- c — угол графика для привязки объектов;
- g — зазор в пикселях между ячейками;
- f — размер шрифта;
- font — название шрифта для обычных ячеек;
- bold — название жирного шрифта для заголовков;
- bgc — цвет фона;
- bgt — прозрачность фона.

```
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)
      ...
};
```

Большинство этих параметров пользователь может задать во входных переменных эксперта *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
```

Количество колонок таблицы у нас определено заранее, количество строк равно количеству символов, плюс верхняя строка с заголовками.

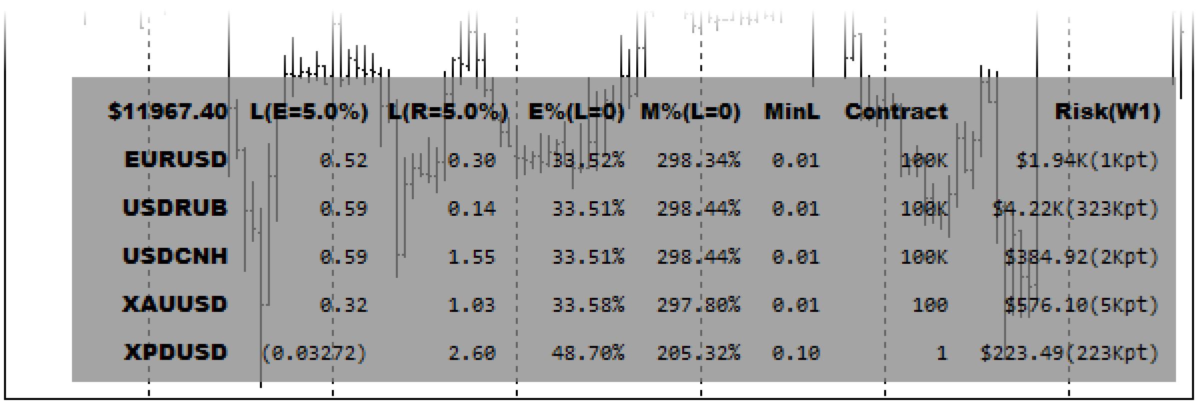
Важно отметить, что шрифты для таблицы следует выбирать без пропорционального начертания букв, поэтому в переменной *MotoTypeFontsHint* приводится подсказка с набором стандартных моноширинных шрифтов Windows.

Заполнение созданных графических объектов данными производит метод *fill* класса *Tableau*.

```
bool fill(const string &data[], const string &hint[]) const;
```

В него наш эксперт передает массив строк *data*, которые получены из массива *LME* путем серии преобразований через *StringFormat*, а также массив *hint* со всплывающими подсказками для заголовков.

На следующем изображении представлен фрагмент графика с работающим экспертом при настройках по умолчанию, но со списком символов "EURUSD,USDRUB,USDCNH,XAUUSD,XPDUSD".



Уровни загрузки депозита и маржи при минимальном лоте по каждому символу

В левой колонке выводятся имена символов. В качестве заголовка первой колонки отображается сумма средств (в данном случае, свободных на счете в текущий момент, т.к. во входном параметре *Money* оставлено значение 0). При наведении курсора мыши на название колонки можно увидеть всплывающую подсказку с пояснением.

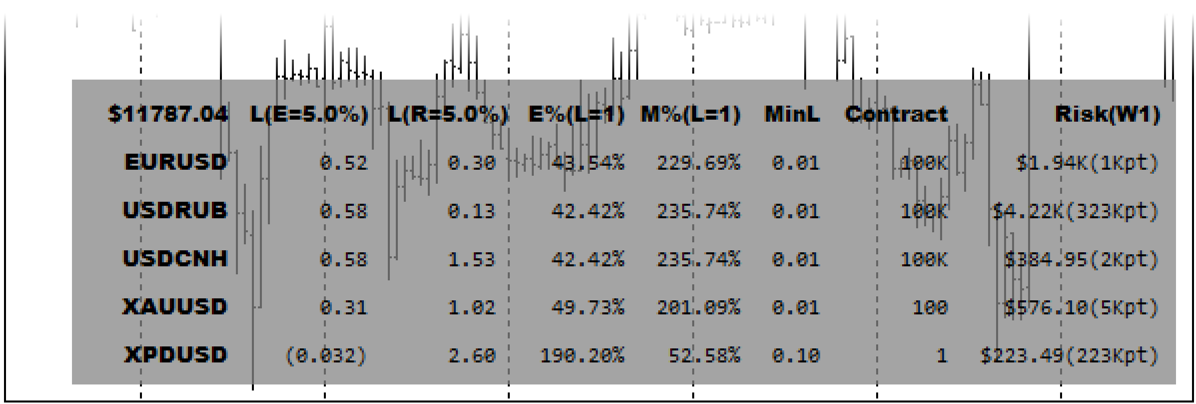
В последующих колонках:

- L(E) — лот, вычисленный для уровня загрузки E депозита 5% после сделки;
- L(R) — лот, вычисленный при риске R на 5% депозита после неудачной торговли (диапазон в пунктах и сумма риска — в последней колонке);
- E% — загрузка депозита после входа минимальным лотом;
- M% — уровень маржи после входа минимальным лотом;
- MinL — минимальный лот для каждого символа;
- Contract — размер контракта (1 лот) для каждого символа;
- Risk — прибыль/убыток в деньгах при торговле 1 лотом и этот же диапазон в пунктах.

В колонках E% и M% в данном случае использованы минимальные лоты, поскольку входной параметре *Lot* равен 0 (по умолчанию).

Видно, что при загрузке депозита 5% торговля возможна для всех выбранных символов, кроме "XPDUSD". Для последнего получился объем 0.03272, что меньше минимального лота 0.1, в связи с чем результат заключен в скобки. Если разрешим загрузку 20% (введем 20 в параметр *Exposure*), получим для "XPDUSD" минимальный лот 0.1.

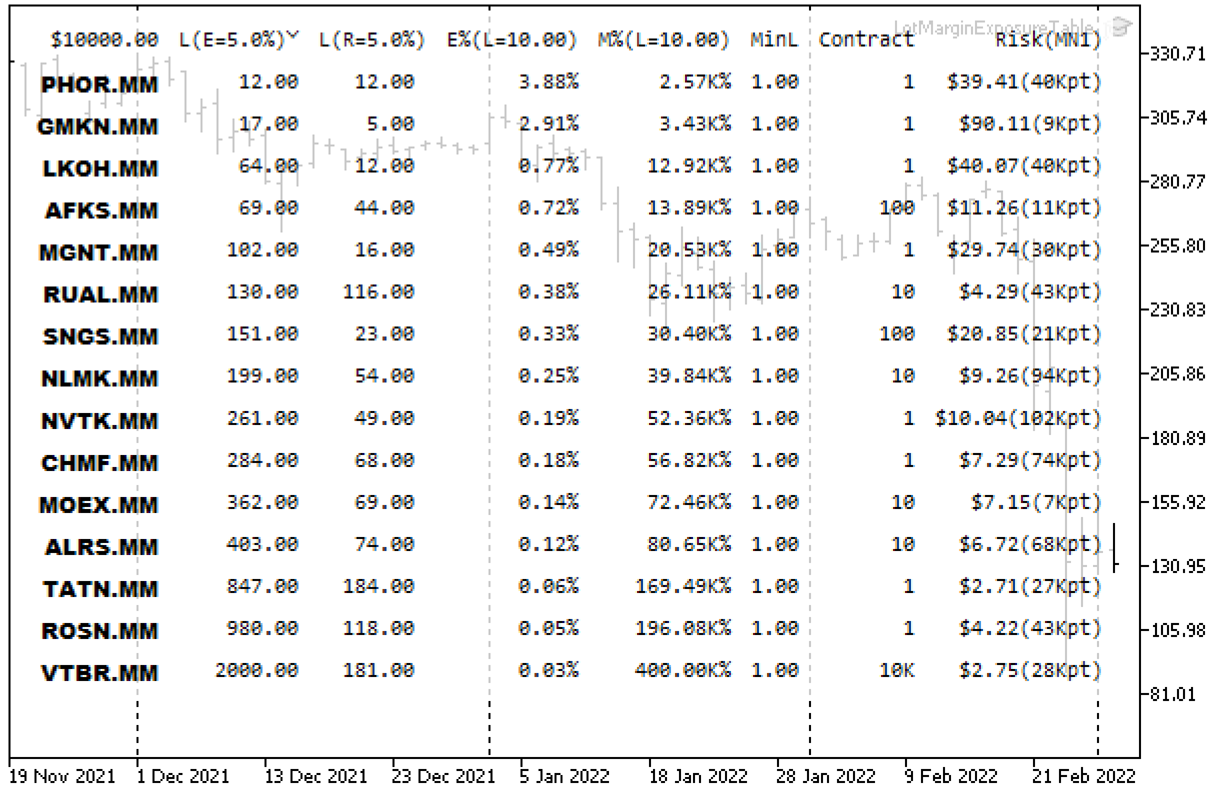
Если введем в параметр *Lot* значение 1, увидим в таблице обновленные значения в колонках E% и M% (загрузка увеличится, уровень маржи упадет).



$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)

Уровни загрузки депозита и маржи при единичном лоте по каждому символу

На последнем скриншоте, иллюстрирующем работу эксперта, приведен большой набор "голубых фишек" российской биржи MOEX с сортировкой по объему, рассчитанному для 5%-загрузки депозита (2-я колонка). Среди нестандартных настроек можно отметить, что *Lot=10*, а период для вычисления ценового диапазона и риска равен MN1. Фон сделан полупрозрачным белым, привязка — к левому верхнему углу графика.



$10000.00	L(E=5.0%)	L(R=5.0%)	E%(L=10.00)	M%(L=10.00)	MinL	Contract	Risk(MN1)
PHOR.MM	12.00	12.00	3.88%	2.57K%	1.00	1	$39.41(40Kpt)
GMKN.MM	17.00	5.00	2.91%	3.43K%	1.00	1	$90.11(9Kpt)
LKOH.MM	64.00	12.00	0.77%	12.92K%	1.00	1	$40.07(40Kpt)
AFKS.MM	69.00	44.00	0.72%	13.89K%	1.00	100	$11.26(11Kpt)
MGNT.MM	102.00	16.00	0.49%	20.53K%	1.00	1	$29.74(30Kpt)
RUAL.MM	130.00	116.00	0.38%	26.11K%	1.00	10	$4.29(43Kpt)
SNGS.MM	151.00	23.00	0.33%	30.40K%	1.00	100	$20.85(21Kpt)
NLMK.MM	199.00	54.00	0.25%	39.84K%	1.00	10	$9.26(94Kpt)
NVTK.MM	261.00	49.00	0.19%	52.36K%	1.00	1	$10.04(102Kpt)
CHMF.MM	284.00	68.00	0.18%	56.82K%	1.00	1	$7.29(74Kpt)
MOEX.MM	362.00	69.00	0.14%	72.46K%	1.00	10	$7.15(7Kpt)
ALRS.MM	403.00	74.00	0.12%	80.65K%	1.00	10	$6.72(68Kpt)
TATN.MM	847.00	184.00	0.06%	169.49K%	1.00	1	$2.71(27Kpt)
ROSN.MM	980.00	118.00	0.05%	196.08K%	1.00	1	$4.22(43Kpt)
VTBR.MM	2000.00	181.00	0.03%	400.00K%	1.00	10K	$2.75(28Kpt)

Лоты, загрузка депозита и уровень маржи для инструментов MOEX

6.4.8 Оценка прибыли торговой операции: OrderCalcProfit

Одна из функций MQL5 API — *OrderCalcProfit* — позволяет заранее оценить финансовый результат торговой операции при выполнении предполагаемых условий. Например, с помощью неё можно узнать сумму прибыли при достижении уровня *Take Profit* и сумму убытка при срабатывании *Stop Loss*.

```
bool OrderCalcProfit(ENUM_ORDER_TYPE action, const string symbol, double volume,
  double openPrice, double closePrice, double &profit)
```

Функция вычисляет размер прибыли или убытка в валюте счета для текущего рыночного окружения и на основании переданных параметров.

В параметре *action* указывается тип ордера, причем разрешено использовать только рыночные ордера ORDER_TYPE_BUY или ORDER_TYPE_SELL из перечисления ENUM_ORDER_TYPE. Название финансового инструмента и его объем передаются в параметрах *symbol* и *volume*. Цены входа и выхода из рынка задаются параметрами *openPrice* и *closePrice*, соответственно. Последним параметром по ссылке передается переменная *profit*, в которую будет записано значение прибыли.

Функция возвращает признак успеха (*true*) или ошибки (*false*).

Формула расчета финансового результата, применяемая внутри *OrderCalcProfit*, зависит от типа инструмента.

Идентификатор	Формула
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

В формулах использованы следующие обозначения:

- Lots — объем позиции в лотах (долях контракта);
- ContractSize — размер контракта (одного лота, SYMBOL_TRADE_CONTRACT_SIZE);
- TickPrice — стоимость тика (SYMBOL_TRADE_TICK_VALUE);
- TickSize — размер тика (SYMBOL_TRADE_TICK_SIZE);
- MarketPrice — последняя известная цена *Bid/Ask* в зависимости от типа сделки;
- OpenPrice — цена открытия позиции;
- ClosePrice — цена закрытия позиции;
- FaceValue — номинальная цена облигации (SYMBOL_TRADE_FACE_VALUE);
- LiqudityRate — коэффициент ликвидности (SYMBOL_TRADE_LIQUIDITY_RATE);
- AccruedInterest — накопленный купонный доход (SYMBOL_TRADE_ACCRUED_INTEREST).

Функцию *OrderCalcProfit* можно применять только в экспертах и скриптах. Для расчета потенциальной прибыли/убытка в индикаторах нужно реализовать альтернативный способ, например, самостоятельные вычисления по формулам.

Чтобы обойти ограничение на применение функций *OrderCalcProfit* и *OrderCalcMargin* в индикаторах, был разработан набор функций, выполняющих расчеты по формулам из этого раздела, а также раздела Маржинальные требования. Функции находятся в заголовочном файле *MarginProfitMeter.mqh*, внутри общего пространства имен *MPM* (от "Margin Profit Meter").

В частности, для расчета финансового результата важно иметь стоимость одного пункта конкретного инструмента. В вышеприведенных формулах она опосредованно участвует в разности цен открытия и закрытия *(ClosePrice - OpenPrice)*.

Вычислить стоимость одного ценового пункта позволяет функция *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);
      ...
```

В начале функции мы запрашиваем все свойства символа, необходимые для расчета. Затем в зависимости от типа инструмента получаем прибыль/убыток в валюте прибыли этого инструмента. Обратите внимание, здесь отсутствуют облигации, формулы которых учитывают номинальную цену и купонный доход.

```
      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))
      }
      ...
```

Наконец, переводим сумму в валюту счета, если она отличается.

```
      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;
   }
   ...
};
```

Для конвертации сумм используется вспомогательная функция *Convert*. Она, в свою очередь, зависит от функции *FindExchangeRate*, производящей поиск среди всех доступных символов такого, который содержит курс из валюты *current* в валюту *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;
}
```

Функция *FindExchangeRate* просматривает символы в *Обзоре рынка* и возвращает название первого подходящего Forex-символа, если их несколько, в параметре *result*. Если котировка соответствует прямому порядку валют "*current/account*", функция вернет +1, а если обратному — "*account/current*" — -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;
}
```

С полным кодом функций можно ознакомиться в прилагаемом файле *MarginProfitMeter.mqh*.

Проверим работу функции *OrderCalcProfit* и группы функций *MPM* с помощью тестового скрипта *ProfitMeter.mq5*: вычислим оценку прибыли/убытка для виртуальных трейдов по всем символам Обзора рынка, причем сделаем это двумя способами — встроенным и собственным.

Во входных параметрах скрипта можно выбрать типа операции *Action* (покупка или продажа), размер лота *Lot* и длительность удержания позиции в барах *Duration*. Финансовый результат рассчитывается для котировок *Duration* последних баров текущего таймфрейма.

```
#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)
```

В теле скрипта подключаем заголовочные файлы и выводим "шапку" с параметрами.

```
#include <MQL5Book/MarginProfitMeter.mqh>
#include <MQL5Book/Periods.mqh>

void OnStart()
{
   // гарантируем, что операцией будет только покупка или продажа
   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));
   ...
```

Затем в цикле по символам выполняем расчеты двумя способами и печатаем результаты для сравнения.

```
   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 приемных переменных

      // стандартный способ
      if(!OrderCalcProfit(type, symbol, Lot, enter, exit, profit1))
      {
         PrintFormat("OrderCalcProfit(%s) failed: %d", symbol, _LastError);
         continue;
      }

      // собственный способ
      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;

      // выводим в лог для сравнения
      PrintFormat("%s: %f %f", symbol, profit1, profit2);
   }
}
```

Попробуйте запустить скрипт для разных счетов и наборов инструментов.

```
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
```

В идеале числа в каждой строке должны совпасть.

6.4.9 Структура торгового запроса MqlTradeRequest

Торговые функции MQL5 API, в частности *OrderCheck* и *OrderSend*, оперируют несколькими встроенными структурами. Поэтому нам придется рассмотреть эти структуры до того, как переходить к самим функциям.

Начнем мы со структуры *MqlTradeRequest*, которая содержит все поля, необходимые для заключения торговых сделок.

```
struct MqlTradeRequest
{
   ENUM_TRADE_REQUEST_ACTIONS action;       // Тип выполняемого действия
   ulong                      magic;        // Уникальный номер эксперта
   ulong                      order;        // Тикет ордера
   string                     symbol;       // Имя торгового инструмента
   double                     volume;       // Запрашиваемый объем сделки в лотах
   double                     price;        // Цена
   double                     stoplimit;    // Уровень StopLimit ордера
   double                     sl;           // Уровень Stop Loss ордера
   double                     tp;           // Уровень Take Profit ордера
   ulong                      deviation;    // Максимальное отклонение от заданной це
   ENUM_ORDER_TYPE            type;         // Тип ордера
   ENUM_ORDER_TYPE_FILLING    type_filling; // Тип ордера по исполнению
   ENUM_ORDER_TYPE_TIME       type_time;    // Тип ордера по времени действия
   datetime                   expiration;   // Срок истечения ордера
   string                     comment;      // Комментарий к ордеру
   ulong                      position;     // Тикет позиции
   ulong                      position_by;  // Тикет встречной позиции
};
```

Не следует пугаться большого количества полей: структура спроектирована для обслуживания абсолютно всех возможных видов торговых запросов, однако в каждом конкретном случае обычно используется всего несколько полей.

Перед заполнением полей рекомендуется обнулить структуру либо с помощью явной инициализации в её определении, либо вызовом функции *ZeroMemory*.

```
MqlTradeRequest request = {};
...
ZeroMemory(request);
```

Это позволит избежать потенциальных ошибок и побочных эффектов от передачи в функции API случайных значений в тех поля, которые не были явно присвоены.

В следующей таблице приводится краткое описание полей. Про их заполнение будет сказано в описаниях торговых операций.

Поле	Описание
action	Тип торговой операции из ENUM_TRADE_REQUEST_ACTIONS
magic	Идентификатор эксперта (опционально)
order	Тикет отложенного ордера, для которого запрашивается модификация
symbol	Имя торгового инструмента
volume	Запрашиваемый объем сделки в лотах
price	Цена, при достижении которой ордер должен быть исполнен
stoplimit	Цена, где будет выставлен лимитный ордер при активации ордеров ORDER_TYPE_BUY_STOP_LIMIT и ORDER_TYPE_SELL_STOP_LIMIT
sl	Цена, по которой сработает *Stop Loss* ордер при движении цены в неблагоприятном направлении
tp	Цена, по которой сработает *Take Profit* ордер при движении цены в благоприятном направлении
deviation	Максимально приемлемое отклонение от запрашиваемой цены, в пунктах
type	Тип ордера из ENUM_ORDER_TYPE
type_filling	Тип ордера по заполнению объемом из ENUM_ORDER_TYPE_FILLING
type_time	Тип истечения отложенного ордера из ENUM_ORDER_TYPE_TIME
expiration	Срок истечения отложенного ордера
comment	Комментарий к ордеру
position	Тикет позиции
position_by	Тикет встречной позиции для операции TRADE_ACTION_CLOSE_BY

Для отправки приказов на совершение торговых операций необходимо заполнять различный набор полей, в зависимости от сути операции. Некоторые поля являются обязательными, а

некоторые — опциональными (могут быть опущены при заполнении). Далее мы подробно рассмотрим требования к полям в контексте конкретных действий.

Оформленную структуру *MqlTradeRequest* программа может проверить на корректность с помощью функции *OrderCheck* или отправить на сервер с помощью функции *OrderSend*, и в случае успеха будет выполнена требуемая операция.

Поле *action* является единственным, необходимым для всех торговых действий.

Уникальный номер в поле *magic* имеет смысл указывать только в запросах на покупку/продажу по рынку или при создании нового отложенного ордера. Это приводит к последующей маркировке совершенных сделок и позиций этим числом, что позволяет организовать аналитическую обработку торговых действий. При модификации ценовых уровней позиции или отложенных ордеров, а также их удалении это поле не имеет эффекта.

При ручном выполнении торговых операций из интерфейса MetaTrader 5 magic-идентификатор нельзя установить, и потому он равен нулю. Это дает популярный, но не совсем надежный способ отличить ручную и автоматическую торговлю при анализе истории. На самом деле, эксперты также могут использовать нулевой идентификатор. Поэтому выяснять, кто и как выполнял конкретные торговые действия, следует с помощью соответствующих свойств ордеров (ORDER_REASON), сделок (DEAL_REASON) и позиций (POSITION_REASON).

Каждый эксперт может выставлять свой собственный уникальный идентификатор или даже использовать несколько для разных целей (в разбивке по торговым стратегиям, сигналам и т.д.). *Magic*-номер позиции соответствует *magic*-номеру последней сделки, участвовавшей в формировании позиции.

Название инструмента в поле *symbol* важно только для операций открытия или наращивания позиции, а также при выставлении отложенных ордеров. В случаях модификации и закрытии ордеров и позиций оно будет проигнорировано, но здесь есть небольшое исключение. Поскольку на неттинг-счетах по каждому символу может существовать только одна позиция, то поле *symbol* можно использовать для идентификации позиции в запросе на изменение её защитных ценовых уровней (*Stop Loss* и *Take Profit*).

Похожим образом используется и поле *volume*: оно нужно в приказах на немедленную покупку/продажу или при создании отложенных ордеров. Следует учитывать, что реальное значение объема в сделке будет зависеть от режима исполнения и может отличаться от запрошенного.

Поле *price* также имеет некоторые ограничения: при отправке рыночных ордеров (TRADE_ACTION_DEAL в поле *action*) по инструментам с режимом исполнения SYMBOL_TRADE_EXECUTION_MARKET или SYMBOL_TRADE_EXECUTION_EXCHANGE это поле игнорируется.

Поле *stoplimit* имеет смысл только при установке стоп-лимитных ордеров, то есть когда в поле *type* содержится ORDER_TYPE_BUY_STOP_LIMIT или ORDER_TYPE_SELL_STOP_LIMIT. Здесь задается цена, по которой будет выставлен отложенный лимитный ордер при достижении ценой значения *price* (этот факт отслеживается сервером MetaTrader 5, и до этого момента отложенный ордер в торговую систему не выводится).

При установке отложенных ордеров их правила истечения задаются в паре полей *type_time* и *expiration*. Последнее содержит значение типа *datetime*, которое принимается во внимание, только если *type_time* равно ORDER_TIME_SPECIFIED или ORDER_TIME_SPECIFIED_DAY.

Наконец, пара последних полей связана с идентификацией позиций в запросах. Каждой новой позиции, созданной на основе приказов (вручную или программно), системой присваивается тикет — уникальное число. Как правило, оно соответствует тикету ордера, в результате которого позиция открыта, но может меняться в результате служебных операций на сервере, например, начисления свопов переоткрытием позиции.

О получении свойств позиций, сделок и ордеров мы поговорим в отдельных разделах. Здесь же для нас пока важно, что поле *position* следует заполнять при изменении и закрытии позиции в целях её однозначной идентификации. В принципе, на неттинговых счетах для этого достаточно указания инструмента позиции в поле *symbol*, но для унификации алгоритмов лучше оставить поле *position* в работе.

Поле *position_by* используется для операции закрытия встречных позиций (TRADE_ACTION_CLOSE_BY). В нем следует указывать позицию, открытую по тому же инструменту, но в противоположном направлении по отношению к *position* (это возможно только на хеджинговом счете).

Поле *deviation* влияет на исполнение рыночных ордеров только в режимах Instant Execution и Request Execution.

Примеры заполнения структуры для торговых операций каждого типа будут приведены в соответствующих разделах.

6.4.10 Структура проверки запроса MqlTradeCheckResult

Прежде чем отправить торговому серверу запрос на торговую операцию, рекомендуется проверить, что он заполнен без формальных ошибок. Проверка осуществляется функцией *OrderCheck*, которой передается сам проверяемый запрос в структуре *MqlTradeRequest* и приемная переменная типа структуры *MqlTradeCheckResult*, куда и будет записан результат проверки.

Помимо корректности запроса структура позволит оценить состояние счета после выполнения торговой операции, в частности, баланс, средства и маржу.

```
struct MqlTradeCheckResult
{
   uint   retcode;      // Код ответа
   double balance;      // Баланс после совершения сделки
   double equity;       // Эквити после совершения сделки
   double profit;       // Плавающая прибыль
   double margin;       // Маржинальные требования
   double margin_free;  // Свободная маржа
   double margin_level; // Уровень маржи
   string comment;      // Комментарий к коду ответа (описание ошибки)
};
```

В следующей таблице приводится описание полей.

Поле	Описание
retcode	Предполагаемый код возврата
balance	Значение баланса, которое будет после выполнения торговой операции
equity	Значение собственных средств, которое будет после выполнения торговой операции
profit	Значение плавающей прибыли, которое будет после выполнения торговой операции
margin	Общий размер заблокированной маржи после торговой операции
margin_free	Размер свободных собственных средств, которые останутся после выполнения торговой операции
margin_level	Уровень маржи, который установится после выполнения торговой операции
comment	Комментарий к коду ответа, описание ошибки

В структуре, заполненной вызовом *OrderCheck*, поле *retcode* будет содержать код результата из числа тех, что платформа поддерживает для обработки реальных торговых запросов и проставляет в аналогичное поле *retcode* структуры *MqlTradeResult* после вызова торговых функций *OrderSend* и *OrderSendAsync*.

Константы с кодами возврата представлены в документации MQL5. Для их более наглядного вывода в журнал при отладке экспертов было определено прикладное перечисление TRADE_RETCODE в файле *TradeRetcode.mqh*. В нем все элементы имеют идентификаторы, совпадающие со встроенными константами, но без общего префикса "TRADE_RETCODE_". Например,

```
enum TRADE_RETCODE
{
   OK_0           = 0,      // нет стандартной константы
   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))
```

Таким образом, использование *TRCSTR(r.retcode)*, где *r* — структура, предоставит минимальное описание числового кода.

Пример применения макроса и анализа структуры мы рассмотрим в следующем разделе про функцию *OrderCheck*.

6.4.11 Проверка корректности запроса: OrderCheck

Для выполнения любой торговой операции MQL-программа должна предварительно заполнить необходимыми данными структуру *MqlTradeRequest*. Перед отправкой на сервер с помощью торговых функций её имеет смысл проверить на формальную корректность и оценить последствия выполнения запроса, в частности, размер залога, который потребуется, и оставшихся свободных средств. Данную проверку выполняет функция *OrderCheck*.

bool OrderCheck(const MqlTradeRequest &request, MqlTradeCheckResult &result)

В случае нехватки средств или ошибочно заполненных параметров функция возвращает *false*. Кроме того, функция реагирует отказом и при отключенном разрешении торговать — как в терминале в целом, так и для конкретной программы. Конкретный код ошибки смотрите в поле *retcode* структуры *result*.

Успешная проверка структуры *request* и торгового окружения завершается со статусом *true*, однако это не гарантирует, что запрашиваемая операция непременно выполнится успешно, если её повторить с помощью функций *OrderSend* или *OrderSendAsync*. Между вызовами могут измениться торговые условия или у брокера на сервере применены настройки для конкретной внешней торговой системы, которые невозможно учесть в алгоритме формальной проверки, которую делает *OrderCheck*.

Для получения описания предполагаемого финансового результата следует анализировать поля структуры *result*.

В отличие от функции *OrderCalcMargin*, которая рассчитывает оценку требуемого залога только под одну предполагаемую позицию или ордер, *OrderCheck* учитывает, хоть и в упрощенном режиме, общее состояние торгового счета. Так, она заполняет поле *margin* в структуре *MqlTradeCheckResult* и прочие связанные поля (*margin_free*, *margin_level*) совокупными показателями, которые сложатся после выполнение приказа. Например, если уже открыта позиция по какому-либо инструменту на момент вызова *OrderCheck* и проверяемый запрос наращивает позицию, поле *margin* отразит сумму залога, включая прежние маржинальные обязательства. Если же новый приказ содержит операцию в противоположном направлении, размер маржи не увеличится (в реальности он должен уменьшится, потому что на неттинговом счете позиция таким образом может вообще закрыться, а на счете с хеджингом — применится хеджирующая маржа для встречных позиций, однако столь точный расчет функция не выполняет).

Прежде всего, *OrderCheck* пригодится программистам на начальном этапе знакомства с торговым API, чтобы экспериментировать с запросами, не отправляя их на сервер.

Протестируем работу функции *OrderCheck* с помощью простого неторгующего эксперта *CustomOrderCheck.mq5*. Мы сделали его именно экспертом, а не скриптом, для удобства пользования — так он будет оставаться на графике после прикрепления с текущими настройками, и их легко редактировать, меняя отдельные входные параметры. Если бы это был скрипт, нам пришлось бы каждый раз начинать заново установку нескольких полей со значений по умолчанию.

Чтобы запустить проверку, установим в *OnInit* таймер.

```
void OnInit()
{
   // инициируем отложенное выполнение
   EventSetTimer(1);
}
```

А в обработчике таймера будет реализован основной алгоритм, причем в самом начале мы отменяем таймер, поскольку нам нужно, чтобы код выполнился один раз, а затем ждал, пока пользователь поменяет параметры.

```
void OnTimer()
{
   // выполняем код однократно и ждем новых установок пользователя
   EventKillTimer();
   ...
}
```

Входные параметры эксперта полностью повторяют набор полей структуры торгового запроса.

```
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;
```

Многие из них не влияют на проверку и финансовые показатели, но оставлены, чтобы можно было в этом убедиться.

По умолчанию состояние переменных соответствует запросу на открытие позиции минимальным лотом текущего инструмента. В частности, параметр *Type* в отсутствие явной инициализации получит значение 0, которое равно элементу ORDER_TYPE_BUY структуры ENUM_ORDER_TYPE. В параметре *Action* мы задали явную инициализацию, потому что 0 не соответствует ни одному элементу перечисления ENUM_TRADE_REQUEST_ACTIONS (первый элемент TRADE_ACTION_DEAL равен 1).

```
void OnTimer()
{
   ...
   // инициализируем структуры нулями
   MqlTradeRequest request = {};
   MqlTradeCheckResult result = {};

   // значения по умолчанию
   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;
   ...
```

Заполняем структуру. В реальных роботах обычно требуется присвоить только несколько полей, но поскольку этот тест носит универсальный характер, мы должны обеспечить передачу любых параметров, которые введет пользователь.

```
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;
...
```

Обратите внимание, что здесь мы пока не делаем нормализацию цен и лотов, хотя в реальной программе она требуется. Тем самым данный тест дает возможность ввести "неровные" значения и убедиться, что они приводят к ошибке. В последующих примерах нормализация уже будет включена.

Затем вызываем *OrderCheck* и выводим в журнал структуры *request* и *result*. Из последней нас интересует, на самом деле, только поле *retcode*, поэтому оно дополнительно печатается с "расшифровкой" в виде текста, макросом TRCSTR (*TradeRetcode.mqh*). Вы также можете анализировать строковое поле *comment*, но его формат может меняться, так что оно в большей степени подойдёт для отображения пользователю.

```
ResetLastError();
PRTF(OrderCheck(request, result));
StructPrint(request, ARRAYPRINT_HEADER);
Print(TRCSTR(result.retcode));
StructPrint(result, ARRAYPRINT_HEADER, 2);
...
```

Вывод структур обеспечивает вспомогательная функция *StructPrint*, которая основана на *ArrayPrint*. Из-за этого мы пока получим "сырое" отображение данных — в частности, элементы перечислений представлены числами "как есть". Позднее мы разработаем функцию для более дружественного (понятного для пользователя) вывода структуры *MqlTradeRequest* (см. *TradeUtils.mqh*).

Чтобы облегчить анализ результатов, в начале функции *OnTimer* выведем текущее состояние счета, а в конце для сравнения — вычислим для заданной торговой операции маржу с помощью функции *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));
   ...
   // заполнение структуры MqlTradeRequest
   // вызов OrderCheck и печать результатов
   ...
   double margin = 0;
   ResetLastError();
   PRTF(OrderCalcMargin(Type, symbol, volume, price, margin));
   PRTF(margin);
}
```

Ниже представлен пример вывода в журнал на "XAUUSD" с настройками по умолчанию.

```
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
 » [type_filling] [type_time]        [expiration] [comment] [position] [position_by]
 »             0           0 1970.01.01 00:00:00 ""                 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"
OrderCalcMargin(Type,symbol,volume,price,margin)=true / ok
margin=19.0 / ok
```

А вот пример с оценкой предполагаемого увеличения маржи на счете, где уже есть открытая позиция, и мы её собираемся удвоить.

```
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
 » [type_filling] [type_time]        [expiration] [comment] [position] [position_by]
 »              0           0 1970.01.01 00:00:00 ""                  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"
OrderCalcMargin(Type,symbol,volume,price,margin)=true / ok
margin=79.04000000000001 / ok
```

Попробуйте менять любые параметры запроса и посмотрите, является ли запрос успешным. Некорректные сочетания параметров будут вызывать коды ошибок из стандартного перечня, но поскольку неверных вариантов гораздо больше, чем зарезервированных (самых распространенных ошибок), функция часто может возвращать обобщенный код TRADE_RETCODE_INVALID (10013). В связи с этим рекомендуется реализовать собственные проверки структуры с большей степенью диагностики.

При отправке реальных запросов на сервер этот же код TRADE_RETCODE_INVALID используется при различных непредвиденных обстоятельствах, например, при попытке повторно изменить ордер, операция изменения которого уже запущена (но еще не завершилась) во внешней торговой системе.

6.4.12 Результат отправки запроса: структура MqlTradeResult

В ответ на торговый запрос, выполняемый функциями *OrderSend* или *OrderSendAsync*, которые мы рассмотрим в следующем разделе, сервер возвращает результаты обработки запроса. Для этой цели используется специальная предопределенная структура *MqlTradeResult*.

```
struct MqlTradeResult
{
   uint      retcode;          // Код результата операции
   ulong     deal;             // Тикет сделки, если она совершена
   ulong     order;            // Тикет ордера, если он выставлен
   double    volume;           // Объем сделки, подтверждённый брокером
   double    price;            // Цена в сделке, подтверждённая брокером
   double    bid;              // Текущая рыночная цена спроса
   double    ask;              // Текущая рыночная цена предложения
   string    comment;          // Комментарий брокера к операции
   uint      request_id;       // Идентификатор запроса, устанавливается терминалом пр
   uint      retcode_external; // Код ответа внешней торговой системы
};
```

В следующей таблице приводится описание её полей.

Поле	Описание
retcode	Код возврата торгового сервера
deal	Тикет сделки, если она совершена (при торговой операции TRADE_ACTION_DEAL)
order	Тикет ордера, если он выставлен (при торговой операции TRADE_ACTION_PENDING)
volume	Объем сделки, подтверждённый брокером (зависит от режимов исполнения ордера)
price	Цена в сделке, подтверждённая брокером (зависит от поля *deviation* в торговом запросе, режима исполнения и торговой операции)
bid	Текущая рыночная цена спроса
ask	Текущая рыночная цена предложения
comment	Комментарий брокера к операции (по умолчанию заполняется расшифровкой кода возврата торгового сервера)
request_id	Идентификатор запроса, проставляемый терминалом при отсылке на торговый сервер
retcode_external	Код ошибки, которую вернула внешняя торговая система

Как мы увидим далее, при проведении торговых операций переменная типа *MqlTradeResult* передается вторым параметром по ссылке в функции *OrderSend* или *OrderSendAsync*, и в ней возвращается результат.

При отправке торгового запроса на сервер терминал устанавливает идентификатор *request_id* в уникальное значение. Это нужно для анализа последующих торговых событий, что наиболее востребовано, если применяется асинхронная функция *OrderSendAsync*. Этот идентификатор позволяет связать отправленный запрос с результатом его обработки, передаваемым в обработчик события *OnTradeTransaction*.

Наличие и виды ошибок в поле *retcode_external* зависят от брокера и внешней торговой системы, в которую выводятся торговые операции.

Анализировать результаты запросов требуется по-разному, в зависимости от сути торговых операций и способа их отправки. Мы займемся этим в последующих разделах, посвященных конкретным действиям: совершению покупок и продаж по рынку, установке и удалению отложенных ордеров, модификации и закрытию позиций.

6.4.13 Отправка торгового запроса: OrderSend и OrderSendAsync

Для выполнения торговых операций MQL5 API предоставляет две функции: *OrderSend* и *OrderSendAsync*. Они, также как и *OrderCheck*, выполняют в терминале формальную проверку параметров запроса, переданных в виде структуры *MqlTradeRequest*, а затем, в случае успеха, отправляют запрос на сервер.

Различие между двумя функциями заключается в следующем. *OrderSend* дожидается постановки приказа в очередь на обработку на сервере и получает оттуда значащие данные в полях структуры *MqlTradeResult*, передаваемой вторым параметром функции. *OrderSendAsync* сразу же возвращает управление вызывающему коду, не заботясь о том, что ответит сервер. При этом из всех полей структуры *MqlTradeResult*, помимо *retcode*, важной информацией заполняется только *request_id*. Используя этот идентификатор запроса, MQL-программа может получать последующую информацию о ходе обработки этого запроса в событии *OnTradeTransaction*. Альтернативный подход — периодически анализировать списки ордеров, сделок и позиций, причем это можно делать и в цикле, задавшись некоторым таймаутом на случай проблем со связью.

Важно отметить, что, несмотря на суффикс "Async" в названии второй функции, первая функция без этого суффикса также не является в полном смысле синхронной. Дело в том, что результат обработки приказа сервером, в частности, совершение сделки (или, вероятно, нескольких сделок на основании одного приказа) и открытие позиции, в общем случае происходит асинхронно — во внешней торговой системе. Таким образом, функция *OrderSend* также требует отложенного сбора и анализа последствий выполнения запроса, который MQL-программ должна, при необходимости, реализовать сама. Мы рассмотрим пример действительно синхронной отправки запроса и приема всех его результатов позднее (см. *MqlTradeSync.mqh*).

bool OrderSend(const MqlTradeRequest &request, MqlTradeResult &result)

Функция возвращает *true* в случае успешной базовой проверки структуры *request* в терминале и нескольких дополнительных проверок на сервере. Однако это лишь свидетельствует о принятии ордера сервером и не гарантирует успешного выполнения торговой операции.

Торговый сервер может заполнить в возвращаемой структуре *result* значения полей *deal* или *order*, если эти данные будут ему известны в момент формирования ответа на вызов *OrderSend*. Однако в общем случае события исполнения сделок или выставления лимитных ордеров, соответствующих ордеру, могут произойти уже после того, как ответ будет отправлен MQL-программе, в терминал. Поэтому для любого типа торгового запроса при получении результата выполнения *OrderSend* необходимо проверять код возврата торгового сервера *retcode* и код ответа внешней торговой системы *retcode_external* (при необходимости), которые доступны в возвращаемой структуре *result*. На их основе следует принять решение об ожидании незавершенных действий на сервере или предпринять собственные действия.

Каждый принятый ордер хранится на торговом сервере в ожидании обработки, пока не наступит какое-либо из событий, влияющих на его жизненный цикл:

- исполнение при появлении встречного запроса,
- срабатывание при поступлении цены исполнения,
- истечение срока действия,
- отмена пользователем или MQL-программой,
- удаление брокером (например, при клиринге или нехватке средств, по *Stop Out*)

Прототип *OrderSendAsync* полностью повторяет *OrderSend*.

bool OrderSendAsync(const MqlTradeRequest &request, MqlTradeResult &result)

Функция предназначена для высокочастотной торговли, когда по условиям алгоритма недопустимо терять время на ожидание ответа от сервера. Скорость обработки запросов сервером и их выведение во внешнюю торговую систему не увеличивается от использования *OrderSendAsync*.

> Внимание! В тестере функция *OrderSendAsync* работает как *OrderSend*. Это затрудняет отладку отложенной обработки асинхронных запросов.

Функция возвращает *true* по факту успешной отсылки запроса на сервер MetaTrader 5, однако это не означает, что запрос дошел до сервера и был принят для обработки. При этом в приёмной структуре *result* код ответа содержит значение TRADE_RETCODE_PLACED (10008) — "ордер размещен".

Сервер при обработке полученного запроса отправит терминалу ответное сообщение об изменении текущего состояния позиций, ордеров и сделок, которое приводит к генерации события *OnTrade* в MQL-программе. Там она может проанализировать новое торговое окружение и историю счета — далее мы рассмотрим соответствующие примеры.

Также подробности исполнения торгового запроса на сервере можно отслеживать при помощи обработчика *OnTradeTransaction*. При этом следует учитывать, что в результате исполнения одного торгового запроса обработчик *OnTradeTransaction* будет вызван несколько раз. Например, при отсылке запроса на покупку по рынку, он принимается на обработку сервером, для счета создается соответствующий ордер на покупку, происходит исполнение ордера и заключение сделки, в результате чего он удаляется из списка открытых и добавляется в историю ордеров, далее сделка добавляется в историю и создается новая позиция. Для каждого из этих событий будет вызвана функция *OnTradeTransaction*.

Для начала рассмотрим простой пример эксперта *CustomOrderSend.mq5*. Он позволяет задать во входных параметрах все поля запроса, что аналогично *CustomOrderCheck.mq5*, но далее отличается тем, что отправляет запрос на сервер вместо простой проверки в терминале. Запускайте эксперт на демо-счете. После завершения экспериментов не забудьте удалить эксперт с графика или закрыть график, чтобы не отправлять тестовый запрос при каждом следующем запуске терминала.

В новом примере есть и несколько других усовершенствований. Прежде всего добавлен входной параметр *Async*.

```
input bool Async = false;
```

Эта опция предназначена для выбора того, какой функцией запрос будет отсылаться на сервер. По умолчанию параметр равен *false*, и используется функция *OrderSend*. Если задать его равным *true*, будет вызываться *OrderSendAsync*.

Кроме того, с этого примера мы начнем описывать и пополнять специальный набор функций в заголовочном файле *TradeUtils.mqh*, который пригодится для упрощения кодирования роботов. Все функции помещены в пространство имен TU (от "Trade Utilities"), и первыми из них представим функции для удобного вывода в журнал структур *MqlTradeRequest* и *MqlTradeResult*.

```
namespace TU
{
   string StringOf(const MqlTradeRequest &r)
   {
      SymbolMetrics p(r.symbol);

      // главный блок: действие, тип, символ
      string text = EnumToString(r.action);
      if(r.symbol != NULL) text += ", " + r.symbol;
      text += ", " + EnumToString(r.type);
      // блок объемов
      if(r.volume != 0) text += ", V=" + p.StringOf(r.volume, p.lotDigits);
      text += ", " + EnumToString(r.type_filling);
      // блок всех цен
      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;
      // блок истечения отложенных ордеров
      if(IsPendingType(r.type)) text += ", " + EnumToString(r.type_time);
      if(r.expiration != 0) text += ", " + TimeToString(r.expiration);
      // блок модификации
      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;
      // вспомогательные данные
      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;
   }
   ...
};
```

Суть функций — предоставить в кратком, но удобном виде все значащие (непустые) поля: они выводятся в одну строку с уникальным обозначением каждого.

Как можно заметить, в функции для *MqlTradeRequest* используется класс *SymbolMetrics*. Он упрощает нормализацию нескольких цен или объемов по одному и тому же инструменту. Напомним, что нормализация цен и объемов — обязательное условие подготовки корректного торгового запроса.

```
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);
   }
};
```

Непосредственно нормализация величин поручена вспомогательным функциям *NormalizePrice* и *NormalizeLot* (принцип работы последней идентичен тому, что мы видели в файле *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;
}
```

С подключенным файлом *TradeUtils.mqh* пример *CustomOrderSend.mq5* приобретает следующий вид (опущенные фрагменты кода '...' остались без изменений с *CustomOrderCheck.mq5*).

```
void OnTimer()
{
   ...
   MqlTradeRequest request = {};
   MqlTradeCheckResult result = {};

   TU::SymbolMetrics sm(symbol);

   // заполняем структуру запроса
   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;

   // отправляем запрос и выводим результат
   ResetLastError();
   if(Async)
   {
      PRTF(OrderSendAsync(request, result));
   }
   else
   {
      PRTF(OrderSend(request, result));
   }
   Print(TU::StringOf(request));
   Print(TU::StringOf(result));
}
```

Из-за того, что цены и объем теперь нормализуются, вы можете попробовать вводить в соответствующие входные параметры "неровные" значения — они часто получаются в программах в ходе вычислений, а наш код их преобразует согласно спецификации инструмента.

С настройками по умолчанию эксперт создает запрос на покупку минимального лота текущего инструмента по рынку, причем делает это функцией *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
```

Как правило, при разрешенных торгах эта операция должна завершиться успешно (статус DONE, комментарий "Request executed"). В структуре *result* мы сразу получили номер сделки *D*.

Если открыть настройки эксперта и заменить значение параметра *Async* на *true*, мы отправим аналогичный запрос, но уже функцией *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
```

В этом случае статус равен PLACED, и номер сделки на момент возврата функции не известен. Мы знаем только уникальный идентификатор запроса *Req=2*. Чтобы получить номер сделки и позиции, необходимо перехватить сообщение TRADE_TRANSACTION_REQUEST с таким же идентификатором запроса в обработчике *OnTradeTransaction*, куда в качестве параметра поступит заполненная структура *MqlTradeResult*.

С точки зрения пользователя оба запроса должны выполниться одинаково быстро.

Сравнить скорость работы этих двух функций непосредственно в коде MQL-программы можно будет с помощью другого примера эксперта (см. раздел о синхронных и асинхронных запросах), который мы рассмотрим после изучения модели торговых событий.

> Следует отметить, что торговые события посылаются в обработчик *OnTradeTransaction* (при его наличии в коде), независимо от того, какая функция используется для отправки запросов — *OrderSend* или *OrderSendAsync*. Просто в случае применения *OrderSend* некоторая или вся информация о выполнении приказа сразу доступна в приемной структуре *MqlTradeResult*. Однако в общем случае результат распределен по времени и по объемам, например, при "заливке" одного ордера в несколько сделок. Тогда полную информацию можно получить из торговых событий или анализируя историю сделок и ордеров.

Если попробовать отправить заведомо некорректный запрос, например, изменить тип ордера на отложенный ORDER_TYPE_BUY_STOP, получим сообщение об ошибке, потому что для таких ордеров следует использовать действие TRADE_ACTION_PENDING, а кроме того они должны располагаться на удалении от текущей цены (у нас же по умолчанию подставляется рыночная). Перед этим тестом важно не забыть вернуть режим запросов на синхронный (*Async = false*), чтобы сразу увидеть ошибку в структуре *MqlTradeResult* по завершению вызова *OrderSend* — в противном случае *OrderSendAsync* вернула бы *true*, но ордер всё равно не был бы установлен, причем информацию об этом программа могла бы получить только в *OnTradeTransaction*, которого у нас пока нет.

```
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
```

В данном случае ошибка сообщает о неверной цене "Requote".

Примеры использования функций для выполнения конкретных торговых действий будут представлены в следующих разделах.

6.4.14 Совершение покупки или продажи

В этом разделе мы, наконец, приступаем к изучению прикладных вариантов применения функций MQL5 для конкретных торговых задач. Все они сводятся к тому, чтобы заполнить особым образом структуру *MqlTradeRequest* и вызвать функцию *OrderSend или OrderSendAsync*.

Первым действием, с которым мы познакомимся, является покупка или продажа финансового инструмента по текущей рыночной цене. Процедура выполнения этого действия включает:

- создание рыночного ордера на основе отправленной заявки,
- совершение сделки (одной или нескольких) по ордеру,
- результатом должна стать открытая позиция.

Как мы видели в разделе о типах торговых операций, "моментальной" покупке/продаже соответствует элемент TRADE_ACTION_DEAL в перечислении ENUM_TRADE_REQUEST_ACTIONS. Поэтому при заполнении структуры *MqlTradeRequest* в поле *action* следует записывать TRADE_ACTION_DEAL.

Направление торговли задается с помощью поля *type*, где должен быть один из типов ордеров: ORDER_TYPE_BUY или ORDER_TYPE_SELL.

Разумеется, для выполнения покупки или продажи требуется указать имя символа в поле *symbol* и его желаемый объем в поле *volume*.

Поле *type_filling* нужно заполнить одной из политик заливки из перечисления ENUM_ORDER_TYPE_FILLING, которая выбирается на основе свойства символа SYMBOL_FILLING_MODE с разрешенными политиками.

Опционально программа может заполнить поля с защитными ценовыми уровнями (*sl* и *tp*), комментарий (*comment*) и идентификатор эксперта (*magic*).

Содержимое других полей устанавливается по-разному в зависимости от режима исполнения по цене для выбранного символа. В некоторых режимах некоторые поля не имеют эффекта. Например, в режимах по запросу цен (Request Execution) и немедленного исполнения (Instant Execution) поле с ценой *price* должно быть обязательно заполнено подходящей ценой (последний известный *Ask* для покупки, *Bid* для продажи), а поле *deviation* может содержать максимальное допустимое отклонение цены от заданной для успешного заключения сделки. В прочих режимах исполнения — биржевом (Exchange Execution) и рыночном (Market Execution) — эти поля игнорируются. В целях упрощения исходного кода вы можете заполнять цену и проскальзывание единообразно во всех режимах, но в двух последних вариантах цена все равно будет выбрана и подставлена торговым сервером по правилам режимов.

Остальные, не упомянутые здесь поля структуры *MqlTradeRequest* для данных торговых операций не используются.

В следующей таблице обобщены правила заполнения полей в разбивке по режимам исполнения. Обязательные поля помечены звездочкой, опциональные — плюсом.

Поле	Request	Instant	Exchange	Market
action	*	*	*	*
symbol	*	*	*	*
volume	*	*	*	*
type	*	*	*	*
type_filling	*	*	*	*
price	*	*		
sl	+	+	+	+
tp	+	+	+	+
deviation	+	+		
magic	+	+	+	+
comment	+	+	+	+

При некоторых настройках сервера может быть запрещено заполнять поля с защитными уровнями *sl* и *tp* в момент открытия позиции. Как правило, это характерно для режимов биржевого исполнения или исполнения по рынку, но в MQL5 API не предусмотрено свойств для выяснения этого обстоятельства заранее. В таких случаях *Stop Loss* и *Take Profit* следует устанавливать путем модификации уже открытой позиции. Кстати говоря, такой способ может быть рекомендован и для всех режимов исполнения, поскольку только он позволяет точно отложить защитные уровни от реальной цены открытия позиции. С другой стороны, создание и настройка позиции в два хода может привести к ситуации, когда позиция открыта, а второй запрос на установку защитных уровней не удалось выполнить по тем или иным причинам.

Вне зависимости от направления торговли (покупка/продажа) *Stop Loss* ордер всегда выставляется как стоп-ордер (ORDER_TYPE_BUY_STOP или ORDER_TYPE_SELL_STOP), а *Take Profit* ордер — как лимитный (ORDER_TYPE_BUY_LIMIT или ORDER_TYPE_SELL_LIMIT). Причем, стоп-ордера всегда контролируются сервером MetaTrader 5 и только по достижении ценой указанного уровня, отправляются во внешнюю торговую систему. В отличие от этого, лимитные ордера могут выводиться напрямую во внешнюю торговую систему. В частности, это, как правило, так для биржевых инструментов.

В целях упрощения кодирования торговых операций — не только покупки и продажи, но и всех других — мы начнем с данного раздела разрабатывать классы, а точнее структуры, обеспечивающие автоматическое и корректное заполнение полей для торговых запросов, а также по-настоящему синхронное ожидание результата. Последнее особенно важно, учитывая, что функции *OrderSend* и *OrderSendAsync* возвращают управление в вызывающий код до того, как

торговое действие будет полностью завершено. В частности, при покупке и продаже по рынку алгоритму, как правило, требуется "знать" не номер тикета созданного на сервере ордера, а открыта ли позиция или нет. В зависимости от этого её можно, например, модифицировать путем установки *Stop Loss* и *Take Profit*, если она открыта, или повторять попытки её открыть, если ордер был отклонен.

Чуть позднее мы познакомимся с торговыми событиями *OnTrade* и *OnTradeTransaction*, которые "сообщают" программе об изменении состояния счета, включая статусы ордеров, сделок и позиций. Однако разнесение алгоритма на два фрагмента — отдельно формирование приказов по неким сигналам или правилам, и отдельно анализ ситуации в обработчиках событий — делает код менее понятным и легким для сопровождения.

В принципе, асинхронная парадигма программирования не уступает синхронной ни в быстродействии, ни в легкости кодирования. Однако способы её реализации могут быть разными, например, на основе прямых указателей на функции обратного вызова (базовый прием в Java, JavaScript и множестве других языков) или событий (как в MQL5), что предопределяет некоторые особенности, о которых речь пойдет в разделе про *OnTradeTransaction*. Асинхронный режим позволяет ускорить отправку запросов за счет отложенного контроля за их исполнением. Но этот контроль все равно потребуется рано или поздно осуществить в том же потоке, поэтому средняя производительность схем одинакова.

Все новые структуры расположим в файле *MqlTradeSync.mqh*. Чтобы не "изобретать велосипед", возьмем в качестве отправной точки встроенные структуры MQL5 и опишем свои структуры дочерними. Например, для получения результатов запросов определим *MqlTradeResultSync*, производную от *MqlTradeResult*. Сюда будем добавлять полезные поля и методы, в частности, поле *position* для хранения тикета открытой позиции в результате рыночной покупки или продажи.

```
struct MqlTradeResultSync: public MqlTradeResult
{
   ulong position;
   ...
};
```

Вторым важным усовершенствованием будет конструктор, обнуляющий все поля (это избавляет нас от необходимости указывать явную инициализацию при описании переменных типа структуры).

```
   MqlTradeResultSync()
   {
      ZeroMemory(this);
   }
```

Далее представим универсальный механизм синхронизации, то есть ожидания результатов запроса (для каждого типа запроса будут собственные правила проверки готовности).

Он основывается на специальном типе функции обратного вызова *condition*. Функция такого типа должна принимать параметр-структуру *MqlTradeResultSync* и возвращать *true* в случае успеха: результат операции получен.

```
typedef bool (*condition)(MqlTradeResultSync &ref);
```

Подобные функции предназначены для передачи в метод *wait*, реализующий циклическую проверку готовности результата по заданному условию (см. параметр типа *condition* ниже) в течении предопределенного таймаута в миллисекундах.

```
bool wait(condition p, const ulong msc = 1000)
{
   const ulong start = GetTickCount64();
   bool success;
   while(!(success = p(this)) && GetTickCount64() - start < msc);
   return success;
}
```

Сразу поясним, что таймаут — это максимальное время ожидания: даже если оно будет выбрано очень большим, цикл закончится сразу же, как только будет получен результат, может быть мгновенно. Разумеется, осмысленный таймаут должен длиться в пределах нескольких секунд.

Приведем пример метода, который соответствует прототипу *condition* и будет использоваться для синхронного ожидания появления ордера на сервере (не важно — с каким статусом: анализ статусов — это задача вызывающего кода).

```
static bool orderExist(MqlTradeResultSync &ref)
{
   return OrderSelect(ref.order) || HistoryOrderSelect(ref.order);
}
```

Здесь применены две встроенных функции MQL5 API *OrderSelect* и *HistoryOrderSelect*: они производят поиск и логическое выделение ордера по его тикету во внутреннем торговом окружении терминала. Это, во-первых, подтверждает наличие ордера (если одна из функций вернула *true*), а во-вторых, позволяет читать его свойства с помощью других функций, что для нас пока не важно. Мы рассмотрим все эти функции в отдельных разделах. Две функции записаны в связке, потому что рыночный ордер может исполниться настолько быстро, что его активная фаза (попадающая в *OrderSelect*) сразу перетечет в историю (*HistoryOrderSelect*).

Обратите внимание, что метод объявлен статическим. Это связано с тем, что MQL5 не поддерживает (по крайней мере, пока) указатели на методы объектов. Если бы это было так, мы могли бы объявить метод нестатическим, а сам прототип указателя на функции обратного вызова *condition* сделать без параметра-ссылки на *MqlTradeResultSync* (поскольку все поля присутствуют внутри самого объекта *this*).

Для запуска механизма ожидания (после вызова *OrderSend*) будет достаточно написать так:

```
if(wait(orderExist))
{
   // есть ордер
}
else
{
   // таймаут
}
```

Разумеется, этот фрагмент должен выполняться после того, как мы получим от сервера результат со статусами TRADE_RETCODE_DONE или TRADE_RETCODE_DONE_PARTIAL, и поле *order* в структуре *MqlTradeResultSync* гарантированно содержит тикет ордера. Правда, в силу

распределенности системы ордер с сервера может не сразу отобразиться в окружении терминала. Поэтому и нужно ожидание.

Пока функция *orderExist* возвращает *false* в метод *wait*, цикл ожидания внутри выполняется вплоть до истечения таймаута. При нормальных обстоятельствах мы практически мгновенно обнаружим ордер в окружении терминала, и цикл закончится с признаком успеха (*true*).

По похожему принципу, но несколько более сложно организована функция проверки наличия открытой позиции *positionExist* (приводится с упрощениями). Поскольку предыдущая функция *orderExist* уже выполнит проверку ордера, его тикет, содержащийся в поле *ref.order* структуры, будет подтвержден как рабочий.

```
static bool positionExist(MqlTradeResultSync &ref)
{
   ulong posid, ticket;
   if(HistoryOrderGetInteger(ref.order, ORDER_POSITION_ID, posid))
   {
      // в большинстве случаев идентификатор позиции равен тикету,
      // но не всегда: в полном коде реализовано получение тикета по идентификатор
      // для чего нет встроенных средств MQL5
      ticket = posid;

      if(HistorySelectByPosition(posid))
      {
         ref.position = ticket;
         ...
         return true;
      }
   }
   return false;
}
```

С помощью встроенных функций *HistoryOrderGetInteger* и *HistorySelectByPosition* мы получаем идентификатор и тикет позиции на основании ордера.

Позднее мы увидим использование *orderExist* и *positionExist* при верификации запроса на покупку/продажу, а сейчас обратимся к другой структуре: *MqlTradeRequestSync*. Она также унаследована от встроенной и содержит дополнительные поля, в частности структуру с результатом (чтобы не описывать её в вызывающем коде) и время таймаута для синхронных запросов.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   MqlTradeResultSync result;
   ulong timeout;
   ...
```

Поскольку унаследованные поля новой структуры являются по-прежнему публичными, MQL-программа может присвоить им значения явно, также как это делалось со стандартной структурой *MqlTradeRequest*. Методы, которые мы добавим для выполнения торговых операций, учтут, проверят и при необходимости, исправят эти значения на корректные.

В конструкторе обнулим все поля и установим символ в значение по умолчанию, если параметр опущен.

```
MqlTradeRequestSync(const string s = NULL, const ulong t = 1000): timeout(t)
{
   ZeroMemory(this);
   symbol = s == NULL ? _Symbol : s;
}
```

В принципе, из-за того, что все поля структуры публичны, их можно технически присваивать напрямую, но это не рекомендуется для тех полей, которые требуют проверки и для которых мы реализуем методы установки: они будут вызываться перед выполнением торговых операций. Первый из таких методов — *setSymbol*.

Он не только заполняет поле *symbol*, убедившись в существовании переданного тикера, но и инициирует последующую настройку режима работы с объемами.

```
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;
}
```

Таким образом, изменение символа с помощью *setSymbol* автоматически подберет правильный режим заливки через вложенный вызов *setFilling*.

Метод *setFilling* обеспечивает автоматическое заполнение способа заливки объемов на основе свойств символа SYMBOL_FILLING_MODE и SYMBOL_TRADE_EXEMODE (см. раздел Торговые условия и режимы исполнения приказов).

```
private:
   void setFilling()
   {
      const int filling = (int)SymbolInfoInteger(symbol, SYMBOL_FILLING_MODE);
      const bool market = SymbolInfoInteger(symbol, SYMBOL_TRADE_EXEMODE)
         == SYMBOL_TRADE_EXECUTION_MARKET;

      // поле может быть уже заполнено
      // и совпадение битов означает допустимый режим
      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;
      }
   }
```

Данный метод неявным образом (без ошибок и сообщений) корректирует поле *type_filling*, если эксперт установил его неверно. Если ваш алгоритм требует гарантированного специфического способа заливки, без которого торговля невозможна, сделайте подходящие правки для прерывания процесса.

> Для разрабатываемого набора структур подразумевается, что кроме поля *type_filling*, напрямую можно устанавливать только опциональные поля без специфических требований к их содержимому, такие как *magic* или *comment*.

Далее многие методы приводятся с сокращениями в целях упрощения изложения. В них присутствуют фрагменты для типов операций, которые мы рассмотрим позднее, а также разветвленная проверка на ошибки.

Для операции покупки и продажи нам необходимы поля *price* и *volume* — оба значения следует нормализовать и проверить на допустимый диапазон. Это поручено методу *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);
}
```

Если цена операции не задана (*p == 0*), программа автоматически возьмет актуальную цену правильного типа, в зависимости от направления, которое легко прочитать в поле *type*.

Хотя уровни *Stop Loss* и *Take Profit* не являются обязательными, их также следует нормализовать при наличии, поэтому они добавлены в параметры этого метода.

Аббревиатура TU уже известна нам — она обозначает пространство имен в файле *TradeUtilits.mqh* с массой полезных функций, в том числе для нормализации цен и объемов.

Обработка полей *sl* и *tp* делегирована отдельному методу *setSLTP*, потому что она понадобится не только в операциях покупки и продажи, но и модификации существующей позиции.

```
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; // удаляем 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; // удаляем TP
   }
   return true;
}
```

Помимо нормализации и присваивания величин полям *sl* и *tp* данный метод проверяет взаимное правильное расположение уровней относительно цены *price*. Для этой цели в пространстве TU описан класс *TradeDirection*.

Его конструкторы позволяют указать анализируемое направление торговли: покупки или продажи, в контексте чего легко выявить прибыльное или убыточное взаимное расположение двух цен. Благодаря классу анализ выполняется унифицированным образом, и проверки в коде сокращаются в 2 раза, так как нет необходимости раздельно обрабатывать покупки и продажи. В частности метод *worse* имеет два ценовых параметра *p1*, *p2* и возвращает *true*, если цена *p1* расположена хуже, то есть убыточно, по отношению к цене *p2*. Аналогичный метод *better* представляет обратную логику: он вернет *true*, если цена *p1* лучше цены *p2*. Например, для продажи лучшая цена находится ниже, поскольку *Take Profit* ниже текущей цены.

```
TU::TradeDirection dir(ORDER_TYPE_SELL);
Print(dir.better(100, 200)); // true
```

Сейчас неверное размещение уровня приводит в функции *setSLTP* к выводу предупреждения в журнал и прерыванию процесса проверки, без попытки исправить значения, поскольку подходящий принцип реагирования может быть своим в каждой программе. Например, из двух переданных уровней *stop* и *take* может быть неправильным только один, и тогда, вероятно, имеет смысл применить второй (правильный).

Вы можете изменить поведение, например, путем пропуска присваивания неверных значений (тогда защитные уровни просто не будут изменены) или добавить в структуру поле с признаком ошибки (для такой структуры попытка отправить запрос должна пресекаться, чтобы не нагружать сервер заведомо невыполнимыми запросами). Отправка некорректного запроса закончится с кодом ошибки *retcode*, равным TRADE_RETCODE_INVALID_STOPS.

Также метод *setSLTP* проверяет, чтобы защитные уровни не располагались ближе к текущей цене, чем количество пунктов в свойстве SYMBOL_TRADE_STOPS_LEVEL символа (если это свойство задано, т.е. больше 0), и модификация позиции не запрашивалась, когда она оказывается внутри области заморозки SYMBOL_TRADE_FREEZE_LEVEL (опять же, если она задана). Здесь эти нюансы не показаны: с ними можно ознакомиться в исходном коде.

Теперь у нас все готово для реализации группы торговых методов. Например, для покупки и продажи с наиболее полным набором полей определим методы *buy* и *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);
   }
```

Как уже было сказано, для установки опциональных полей вроде *deviation*, *comment*, *magic* следует делать прямое присваивание перед вызовом *buy/sell*. Это тем более удобно, что *deviation* и *magic* в большинстве случаев задаются единожды и используются в последующих запросах.

Методы возвращают тикет ордера, но далее мы покажем в действии механизм "синхронного" получения тикета позиции, причем это будет тикет созданной или модифицированной позиции (если делалась "доливка" или частичное закрытие).

Методы *buy* и *sell* отличаются только значением поля *type*, а все остальное в них одинаково. Поэтому общая часть оформлена в виде отдельного метода *_market*. Именно здесь мы устанавливаем *action* в TRADE_ACTION_DEAL, вызываем *setSymbol* и *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;
      ...
```

Далее мы могли бы просто вызвать *OrderSend*, но учитывая возможность реквот (обновлений цены на сервере за время отправки приказа), заключим вызов в цикл. За счет этого метод сможет несколько раз повторить попытку, но не более чем предустановленное число раз MAX_REQUOTES (макрос выбран равным 10 в коде).

```
      int count = 0;
      do
      {
         ZeroMemory(result);
         if(OrderSend(this, result)) return result.order;
         // автоматический подбор цены означает автоматическую обработку реквот
         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;
   }
```

Поскольку финансовый инструмент по умолчанию задается в конструкторе структуры, мы можем предоставить пару упрощенных перегрузок методов *buy/sell* без параметра *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);
   }
```

Таким образом, в минимальной конфигурации программе достаточно будет вызвать *request.buy(1.0)*, чтобы совершить покупку одним лотом.

Теперь вернемся к проблеме получения окончательного результата запроса, что для случая операции TRADE_ACTION_DEAL означает тикет позиции. В структуре *MqlTradeRequestSync* эту задачу решает метод *completed*: для каждого типа операции он должен "попросить" вложенную структуру *MqlTradeResultSync* дождаться своего заполнения в соответствии с типом операции.

```
bool completed()
{
   if(action == TRADE_ACTION_DEAL)
   {
      const bool success = result.opened(timeout);
      if(success) position = result.position;
      return success;
   }
   ...
   return false;
}
```

Открытие позиции контролирует метод *opened*. Внутри мы найдем пару вызовов описанного выше метода *wait*: первый — для *orderExist*, второй — для *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;
   }
```

Разумеется, ждать появления ордера и позиции имеет смысл только при статусе в *retcode*, сигнализирующем успех. Другие статусы относятся к ошибкам или отмене операции, либо к специфическим промежуточным кодам (TRADE_RETCODE_PLACED, TRADE_RETCODE_TIMEOUT), которые не сопровождаются полезной информацией в прочих полях. В обоих случаях это препятствует дальнейшей обработке в рамках данного "синхронного" фреймворка.

Важно отметить, что мы используем *OrderSync* и потому рассчитываем на обязательное присутствие тикета ордера в полученной с сервера структуре.

В некоторых случаях система присылает не только тикет ордера, но сразу и тикет сделки. Тогда из сделки можно узнать позицию быстрее. Но даже при наличии информации о сделке, в торговом окружении терминала может временно не быть информации о новой позиции. Именно поэтому следует подождать её с помощью *wait(positionExist)*.

Подведем промежуточный итог. Созданные структуры позволяют написать следующий код для покупки 1 лота текущего символа:

```
MqlTradeRequestSync request;
if(request.buy(1.0) && request.completed())
{
   Print("OK Position: P=", request.result.position);
}
```

Внутрь блока условного оператора мы попадаем только с гарантированно открытой позицией и нам известен её тикет. Если бы мы использовали методы *buy/sell* сами по себе, то получили бы на их выходе тикет ордера и должны были бы сами проверять исполнение. В случае ошибки мы не попадем внутрь блока *if*, а код сервера будет содержаться в *request.result.retcode*.

Когда мы реализуем в следующих разделах методы для других торговых операций, они могут выполняться в аналогичном "блокирующем" режиме, например, для модификации стоп-уровней:

```
if(request.adjust(SL, TP) && request.completed())
{
   Print("OK Adjust")
}
```

Разумеется, вы не обязаны вызывать *completed*, если не хотите проверять результат операции в блокирующем режиме. Вместо этого можно придерживаться асинхронной парадигмы и анализировать окружение в обработчиках торговых событий. Но и в этом случае структура *MqlTradeRequestAsync* может быть полезна проверкой и нормализацией параметров операции.

Напишем тестовый эксперт *MarketOrderSend.mq5*, где все это собрано воедино. Входные параметры обеспечат ввод значений для основных и некоторых опциональных полей торгового запроса.

```
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;
```

Перечисление ENUM_ORDER_TYPE_MARKET является подмножеством стандартного ENUM_ORDER_TYPE и введено для того, чтобы ограничить доступные типы операций только двумя: покупкой и продажей по рынку.

Действие будет запускаться один раз по таймеру, по такому же принципу как в предыдущих примерах.

```
void OnInit()
{
   // планируем отложенный запуск
   EventSetTimer(1);
}
```

В обработчике таймера отключаем таймер, чтобы запрос выполнился только один раз. Для следующего запуска нужно будет поменять параметры эксперта.

```
void OnTimer()
{
   EventKillTimer();
   ...
```

Опишем переменную типа *MqlTradeRequestSync* и подготовим значения для основных полей.

```
   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);
   ...
```

Опциональные поля заполним напрямую.

```
   request.magic = Magic;
   request.deviation = Deviation;
   request.comment = Comment;
   ...
```

Среди опциональных полей можно выбрать режим заливки (*type_filling*). По умолчанию, *MqlTradeRequestSync* автоматически записывает в это поле первый из разрешенных режимов ENUM_ORDER_TYPE_FILLING. Напомним, в структуре для этого есть специальный метод *setFilling*.

Далее вызываем метод *buy* или *sell* с параметрами, и если он возвращает тикет ордера, ждем появления открытой позиции.

```
   ResetLastError();
   const ulong order = (wantToBuy ?
      request.buy(volume, Price) :
      request.sell(volume, Price));
   if(order != 0)
   {
      Print("OK Order: #=", order);
      if(request.completed()) // ждем открытой позиции
      {
         Print("OK Position: P=", request.result.position);
      }
   }
   Print(TU::StringOf(request));
   Print(TU::StringOf(request.result));
}
```

В конце функции для справки в журнал выводятся структуры запроса и результата.

Если запустить эксперт с параметрами по умолчанию (покупка минимальным лотом текущего символа), можем получить следующий результат на "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
```

Обратите внимание на предупреждение о временном отсутствии позиции — оно будет возникать всегда из-за распределенной обработки запросов (сами предупреждения можно отключить, убрав макрос SHOW_WARNINGS в коде эксперта, но ситуация останется). Но благодаря разработанным новым структурам, прикладной код не отвлекается на эти внутренние сложности и написан в виде последовательности простых шагов, где каждый следующий "уверен" в успехе предыдущих.

На неттинговом счете легко добиться интересного эффекта переворота позиции путем последующей продажи удвоенным минимальным лотом (в данном случае — 0.02).

```
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
```

Важно отметить, что после переворота тикет позиции перестает быть равным идентификатору позиции: идентификатор остался от первого ордера, а тикет — от второго. Мы намеренно обошли стороной задачу выяснения тикета позиции по её идентификатору в целях упрощения изложения. В большинстве случаев тикет и идентификатор совпадают, но для точного контроля используйте функцию *TU::PositionSelectById*. Желающие могут изучить прилагаемый исходный код.

Идентификаторы постоянны пока существует позиция (то есть пока не закроется в ноль по объему) и полезны для анализа истории счета. Тикеты описывают позиции, пока они открыты (не существует понятия тикета позиции на истории) и используются в некоторых типах запросов, в частности, для модификации защитных уровней или закрытия встречной позицией. Но здесь есть нюансы, связанные с заливкой по частям. Более подробно о свойствах позиций мы поговорим в отдельном разделе.

При совершении покупки или продажи наши методы *buy/sell* позволяют сразу задать уровни *Stop Loss* и/или *Take Profit*. Для этого достаточно передать их дополнительными параметрами, полученными из входных переменных или рассчитанными по каким-либо формулам. Например,

```
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));
   ...
```

Все методы новых структур обеспечивают автоматическую нормализацию передаваемых параметров, поэтому нет необходимости самим применять *NormalizeDouble* или что-то еще.

Выше уже отмечалось, что некоторые настройки серверов могут запрещать установку защитных уровней одновременно с открытием позиции. В этом случае следует выполнить установку полей *sl* и *tp* отдельным запросом. Точно такой запрос используется и в тех случаях, когда требуется модифицировать уже установленные уровни, в частности, для реализации "трейлинг стопа" или "трейлинг профита".

В следующем разделе мы дополним текущий пример отложенной установкой *sl* и *tp* вторым запросом после успешного открытия позиции.

6.4.15 Модификация уровней Stop Loss и/или Take Profit позиции

У открытой позиции MQL-программа может менять защитные ценовые уровни *Stop Loss* и *Take Profit*. Для этой цели предназначен элемент TRADE_ACTION_SLTP в перечислении ENUM_TRADE_REQUEST_ACTIONS, то есть при заполнении структуры *MqlTradeRequest* в поле *action* следует записывать TRADE_ACTION_SLTP.

Это единственное обязательное поле. Необходимость заполнения других полей обуславливается режимом работы счета ENUM_ACCOUNT_MARGIN_MODE. Так на счетах с неттингом обязательно заполнять поле *symbol*, но можно опустить тикет позиции. На счетах с хеджированием, наоборот, обязательно указывать тикет позиции *position*, но можно опустить символ. Это объясняется особенностями идентификации позиции на счетах разных типов. Напомним, при неттинге по каждому символу может существовать только одна позиция.

В целях унификации кода рекомендуется заполнять оба поля при наличии информации.

Непосредственно защитные ценовые уровни устанавливаются в полях *sl* и *tp*. Допускается задать только *sl* или только *tp*. Для удаления защитных уровней следует присвоить им нулевые значения.

В следующей таблице в обобщенном виде представлены требования по заполнению полей в зависимости от режимов счета. Обязательные поля помечены звездочкой, опциональные — плюсом.

Поле	Неттинг	Хедж
action	*	*
symbol	*	+
position	+	*
sl	+	+
tp	+	+

В структуре *MqlTradeRequestSync* для выполнения операции модификации защитных уровней введем несколько перегрузок метода *adjust*.

```
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);
   ...
};
```

Как мы видели выше, в зависимости от окружения, модификацию можно делать только по тикету или только по символу позиции — эти варианты учтены в первых двух прототипах.

Кроме того, поскольку структура может уже быть использована для предыдущих запросов, в ней могут быть заполнены поля *position* и *symbol*. Тогда можно вызывать метод с последним прототипом.

Мы пока не показываем реализацию этих трех методов, потому что ясно, что она должна иметь общую основную часть с отправкой запроса. Эта часть оформлена, как вспомогательный закрытый метод *_adjust* с полным набором параметров. Здесь его код приводится с некоторыми сокращениями, не влияющими на логику работы.

```
  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);
     }
```

Суть проста — заполняем все поля структуры по вышеприведенным правилам, вызывая ранее описанные методы *setSymbol* и *setSLTP*, а затем отправляем запрос на сервер. Результатом является статус успеха (*true*) или ошибки (*false*).

Исходные параметры для запроса каждый из перегруженных методов *adjust* подготавливает по своему. Вот, например, как это сделано при наличии тикета позиции.

```
  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);
     }
```

Здесь с помощью встроенной функции *PositionSelectByTicket* делается проверка наличия позиции и её выделение в торговом окружении терминала, что необходимо для последующего чтения её свойств, в данном случае — символа (*PositionGetString(POSITION_SYMBOL)*). Затем происходит вызов универсального варианта *adjust*.

При модификации позиции по имени символа (что доступно только на неттинг-счете) можно использовать другой вариант *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);
     }
```

Здесь выбор позиции происходит с помощью встроенной функции *PositionSelect*, а из её свойств получается номер тикета (*PositionGetInteger(POSITION_TICKET)*).

Все эти функции будут рассмотрены подробно в соответствующих разделах о работе с позициями и свойствах позиций.

Вариант метода *adjust* с самым минималистским набором параметров — только уровнями *stop* и *take* — выглядит следующим образом.

```
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);
}
```

Этот код обеспечивает корректное заполнение полей *position* и *symbol* в различных режимах или досрочный выход с сообщением об ошибке в журнал. В конце вызывается приватная версия *_adjust*, отправляющая запрос через *OrderSend*.

Так же как и в случае методов *buy/sell*, представленный набор методов *adjust* работает "асинхронно" в том смысле, что по их завершении известен только статус отправки запроса, но нет подтверждения модификации уровней. Напомним, что в связке с биржей уровень *Take Profit* может выводиться на неё как лимитный ордер. Поэтому в структуре *MqlTradeResultSync* следует обеспечить "синхронное" ожидание, пока изменения не вступят в силу.

Общий механизм ожидания в виде метода *MqlTradeResultSync::wait* уже готов и был использован для ожидания открытия позиции. Метод *wait* получает в качестве первого параметра указатель на

другой метод с предопределенным прототипом *condition* — для опроса в цикле, пока не выполнится требуемое условие или не случится таймаут. Этот *condition*-совместимый метод должен, в данном случае, выполнять прикладную проверку стоп-уровней в позиции.

Добавим такой новый метод под именем *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;
   }
```

В первую очередь, разумеется, проверяем статус в поле *retcode*. Если он — один из штатных, продолжаем проверку самих уровней, передавая в *wait* вспомогательный метод *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;
   }
```

Данный код гарантирует, что позиция выбрана по тикету в торговом окружении терминала с помощью *PositionSelectByTicket* и читает свойства позиции POSITION_SL и POSITION_TP, которые надо сравнить с тем, что было в запросе. Проблема в том, что здесь мы не имеем доступа к объекту запроса и должны каким-то образом передать сюда пару значений для мест, помеченных '.?.'.

В принципе, поскольку структура *MqlTradeResultSync* проектируется нами, мы можем добавить в неё поля *sl* и *tp*, и заполнять их значениями из *MqlTradeRequestSync* перед отправкой запроса

(ядро не "знает" о наших добавленных полях и оставит их нетронутыми в процессе вызова *OrderSend*). Но для простоты мы воспользуемся тем, что уже имеется. Поля *bid* и *ask* в структуре *MqlTradeResultSync* используются только для сообщения цен реквот (статус TRADE_RETCODE_REQUOTE), что не относится к запросу TRADE_ACTION_SLTP, поэтому мы можем сохранить в них значение *sl* и *tp* из заполненной *MqlTradeRequestSync*.

Сделать это логично в методе *completed* структуры *MqlTradeRequestSync*, который запускает блокирующее ожидание результатов торговой операции с предопределенным таймаутом. До сих пор в его коде была только одна ветвь для действия TRADE_ACTION_DEAL. В продолжение добавим ветвь для 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)
      {
         // передаем исходные данные запроса для сравнения со свойствами позиции,
         // по-умолчанию их нет в структуре результата
         result.position = position;
         result.bid = sl; // поле bid свободно в этом типе результата, используем под
         result.ask = tp; // поле ask свободно в этом типе результата, используем под
         return result.adjusted(timeout);
      }
      return false;
   }
```

Как видно, после установки тикета позиции и ценовых уровней из запроса, мы вызываем метод *adjusted*, рассмотренный выше, в котором происходит проверка: *wait(checkSLTP)*. Теперь мы можем вернуться к вспомогательному методу *checkSLTP* в структуре *MqlTradeResultSync* и привести его к окончательному виду.

```
struct MqlTradeResultSync: public MqlTradeResult
{
   ...
   static bool checkSLTP(MqlTradeResultSync &ref)
   {
      if(PositionSelectByTicket(ref.position))
      {
         return TU::Equal(PositionGetDouble(POSITION_SL), ref.bid) // sl из запроса
            && TU::Equal(PositionGetDouble(POSITION_TP), ref.ask); // tp из запроса
      }
      else
      {
         Print("PositionSelectByTicket failed: P=" + (string)ref.position);
      }
      return false;
   }
```

На этом расширение функционала структур *MqlTradeRequestSync* и *MqlTradeResultSync* для операции модификации *Stop Loss* и *Take Profit* завершено.

С учетом этого продолжим пример эксперта *MarketOrderSend.mq5*, начатый в предыдущем разделе. Добавим в него входной параметр *Distance2SLTP*, позволяющий указать расстояние в пунктах до уровней *Stop Loss* и *Take Profit*.

```
input int Distance2SLTP = 0; // Distance to SL/TP in points (0 = no)
```

Когда он равен нулю, защитные уровни не будут ставиться.

В рабочем коде, после получения подтверждения об открытии позиции вычисляем значения уровней в переменных SL и TP, и выполняем синхронную модификацию: *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()) // ждем открытия позиции
      {
         Print("OK Position: P=", request.result.position);
         if(Distance2SLTP != 0)
         {
            // позиция "выбрана" в торговом окружении терминала внутри 'complete',
            // поэтому не требуется делать это явным образом по тикету
            // PositionSelectByTicket(request.result.position);
            
            // при выбранной позиции можно узнать её свойства, а нам нужна цена,
            // чтобы отступить от неё на заданное количество пунктов
            const double price = PositionGetDouble(POSITION_PRICE_OPEN);
            const double point = SymbolInfoDouble(symbol, SYMBOL_POINT);
            // отсчет уровней делаем с помощью вспомогательного класса TradeDirectior
            TU::TradeDirection dir((ENUM_ORDER_TYPE)Type);
            // SL всегда "хуже", а TP - "лучше" цены: код един для покупки и продажи
            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));
}
```

В первом вызове *completed* после успешной покупки или продажи тикет позиции сохраняется в поле *position* структуры запроса. Поэтому для модификации стопов достаточно только ценовых уровней, а символ и тикет позиции уже присутствуют в *request*.

Попробуем выполнить с помощью эксперта покупку с настройками по умолчанию, но установив *Distance2SLTP* в 500 пунктов.

```
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
```

Две последние строки соответствуют отладочному выводу в журнал содержимого структур *request* и *request.result*, инициированному в конце функции. В этих строках интересно, что поля

хранят в себе симбиоз значений из двух запросов: сначала была открыта позиция, а потом произведена её модификация. В частности, поля с объемом (0.01) и ценой (1.10889) в запросе остались после TRADE_ACTION_DEAL, но не помешали выполнению TRADE_ACTION_SLTP. В принципе, от этого легко избавиться, выполнив обнуление структуры между двумя запросами, однако мы предпочли оставить их как есть, потому что среди заполненных полей есть и полезные: поле *position* получило тикет, который нам нужен для запроса модификации. Если бы мы обнулили структуру, то нужно было бы вводить переменную для промежуточного хранения тикета.

В общем случае, конечно, желательно придерживаться строгой политики инициализации данных, но знание особенностей их использования в конкретных сценариях (таких как два или несколько взаимосвязанных запросов предопределенного типа) позволяет оптимизировать код.

Также не следует удивляться тому, что в структуре с результатом мы видим запрошенные уровни *sl* и *tp* в полях под цены *Bid* и *Ask*: записал их туда метод *MqlTradeRequestSync::completed* с целью сравнения с фактическими изменениями позиции. При выполнении запроса ядро системы заполнило в структуре *result* только *retcode* (DONE), *comment* ("Request executed") и *request_id* (26).

Далее мы рассмотрим другой пример модификации уровней, реализующий "трейлинг стоп".

6.4.16 Трейлинг стоп

Одна из самых распространенных задач, в которых используется возможность менять защитные ценовые уровни, заключается в последовательном сдвиге *Stop Loss* на более выгодную цену по мере сохранения благоприятного тренда. Это "трейлинг стоп". Реализуем его с помощью новых структур *MqlTradeRequestSync* и *MqlTradeResultSync* из предыдущих разделов.

Чтобы иметь возможность подключать механизм в любой эксперт, объявим его как класс *TrailingStop* (см. файл *TrailingStop.mqh*). В личных переменных класса будем хранить номер контролируемой позиции, её символ и размер пункта цены, а также требуемую дистанцию уровня стоплосса от текущей цены и шаг изменений уровня.

```
#include <MQL5Book/MqlTradeSync.mqh>

class TrailingStop
{
   const ulong ticket;  // тикет контролируемой позиции
   const string symbol; // символ позиции
   const double point;  // размер пункта цены символа
   const uint distance; // расстояние до стопа в пунктах
   const uint step;     // шаг движений (чувствительность) в пунктах
   ...
```

Дистанция нужна только для алгоритма стандартного сопровождения позиции, который предоставит базовый класс. Производные классы смогут двигать защитный уровень по другим принципам, таким как скользящая средняя, каналы, индикатор SAR и другие. После знакомства с базовым классом мы приведем пример производного класса со скользящей средней.

Под текущий уровень стоп-цены выделим переменную *level*. В переменной *ok* будем поддерживать актуальный статус позиции: *true* — позиция еще существует, *false* — возникла ошибка, позиция закрылась.

```
protected:
   double level;
   bool ok;
   virtual double detectLevel()
   {
      return DBL_MAX;
   }
```

Виртуальный метод *detectLevel* предназначен для переопределения в классах-наследниках, где стоп-цена должна вычисляться по произвольному алгоритму. В данной реализации возвращается специальное значение DBL_MAX, сигнализирующее работу по стандартному алгоритму (см. ниже).

В конструкторе заполним все поля значениями соответствующих параметров. Функция *PositionSelectByTicket* проверяет наличие позиции с заданным тикетом и выделяет её в программном окружении, так что последующий вызов *PositionGetString* возвращает её строковое свойство с именем символа.

```
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;
   }
```

Теперь рассмотрим главный публичный метод класса *trail*: MQL-программа должна будет вызывать его на каждом тике или по таймеру, чтобы сопровождать позицию. Метод возвращает *true*, пока позиция существует.

```
virtual bool trail()
{
   if(!PositionSelectByTicket(ticket))
   {
      ok = false;
      return false; // позиция закрылась
   }

   // выясним цены для расчетов: текущую котировку и стоп-уровень
   const double current = PositionGetDouble(POSITION_PRICE_CURRENT);
   const double sl = PositionGetDouble(POSITION_SL);
   ...
```

Здесь и далее используются функции чтения свойств позиции. Они будут подробно рассмотрены в отдельном разделе. В частности, нам требуется выяснить направление торговли — покупка и продажа, чтобы знать, в какую сторону следует откладывать стоп-уровень.

```
   // POSITION_TYPE_BUY  = 0 (false)
   // POSITION_TYPE_SELL = 1 (true)
   const bool sell = (bool)PositionGetInteger(POSITION_TYPE);
   TU::TradeDirection dir(sell);
   ...
```

Для расчетов и проверок далее используется вспомогательный класс *TU::TradeDirection* и его объект *dir*. Например, его метод *negative* позволяет вычислить цену, расположенную на заданном расстоянии от текущей цены в убыточном направлении, вне зависимости от типа операции. Это упрощает код, поскольку в противном случае пришлось бы делать "зеркальные" вычисления для покупок и продаж.

```
   level = detectLevel();
   // не можем тралить без уровня: удаление стоп-уровня должен делать вызывающий к
   if(level == 0) return true;
   // если есть значение по умолчанию, делаем стандартный отступ от текущей цены
   if(level == DBL_MAX) level = dir.negative(current, point * distance);
   level = TU::NormalizePrice(level, symbol);

   if(!dir.better(current, level))
   {
      return true; // нельзя ставить стоп-уровень с прибыльной стороны
   }
   ...
```

Другой метод класса *TU::TradeDirection* — *better* — проверяет, что полученный стоп-уровень расположен с правильной стороны от цены. Если бы не этот метод, нам опять потребовалось бы написать проверку дважды (для покупок и продаж).

Некорректное значение стоп-уровня у нас может появиться, поскольку метод *detectLevel* может быть переопределен в производных классах. При стандартном расчете эта проблема исключена, потому что уровень считает сам объект *dir*.

Наконец, когда уровень посчитан, необходимо применить его к позиции. Если у позиции еще нет стоп-лосса, подойдет любой корректный уровень. Если же стоп-лосс уже установлен, то новое значение должно быть лучше прежнего и отличается больше, чем на заданный шаг.

```
      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; // успех
   }
```

Непосредственно отправку запроса на модификацию позиции мы обернули в метод *move*, в котором используется уже знакомый метод *adjust* структуры *MqlTradeRequestSync* (см. раздел Модификация уровней Stop Loss и/или 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;
   }
};
```

Теперь все готово для подключения трейлинга в тестовый эксперт *TrailingStop.mq5*. Во входных параметрах можно указать направление торговли, расстояние до стоп-уровня в пунктах и шаг в пунктах. Параметр *TrailingDistance* равен по умолчанию 0, что означает автоматическое вычисление дневного размаха котировок и использование его половины в качестве дистанции.

```
#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;
```

При запуске эксперт выяснит, есть ли позиция на текущем символе с указанным *Magic*-номером, и создаст её в случае отсутствия.

Трейлинг будет осуществляться объектом класса *TrailingStop*, обернутым в умный указатель *AutoPtr*. Последний избавит нас от необходимости вручную удалять старый объект, когда ему на смену потребуется новый объект сопровождения для новой создаваемой позиции. При присваивании нового объекта умному указателю старый объект удаляется автоматически. Напомним, что разыменование умного указателя, то есть получение доступа к хранящемуся внутри рабочему объекту, производится с помощью перегруженного оператора [].

```
#include <MQL5Book/AutoPtr.mqh>

AutoPtr<TrailingStop> tr;
```

В обработчике *OnTick* проверим, есть ли объект, и если да, то существует ли позиция (признак возвращается из метода *trail*). Сразу после запуска программы объекта не будет, и указатель равен NULL. В этом случае следует либо создать новую позицию, либо найти уже открытую и создать для неё объект *TrailingStop*. Этим занимается функция *Setup*. При последующих вызовах *OnTick* объект начинает и продолжает сопровождение, не давая программе зайти внутрь блока *if*, пока позиция "жива".

```
void OnTick()
{
   if(tr[] == NULL || !tr[].trail())
   {
      // если трейлинга пока нет, создадим или найдем подходящую позицию
      Setup();
   }
}
```

А вот и функция *Setup*.

```
void Setup()
{
   int distance = 0;
   const double point = SymbolInfoDouble(_Symbol, SYMBOL_POINT);
   
   if(TrailingDistance == 0) // автоопределяем дневной размах цен
   {
      distance = (int)((iHigh(_Symbol, PERIOD_D1, 1) - iLow(_Symbol, PERIOD_D1, 1))
         / point / 2);
      Print("Autodetected daily distance (points): ", distance);
   }
   else
   {
      distance = TrailingDistance;
   }
   
   // обрабатываем только позицию текущего символа и нашего Magic
   if(GetMyPosition(_Symbol, Magic))
   {
      const ulong ticket = PositionGetInteger(POSITION_TICKET);
      Print("The next position found: ", ticket);
      tr = new TrailingStop(ticket, distance, TrailingStep);
   }
   else // нет нашей позиции
   {
      Print("No positions found, lets open it...");
      const ulong ticket = OpenPosition();
      if(ticket)
      {
         tr = new TrailingStop(ticket, distance, TrailingStep);
      }
   }
   
   if(tr[] != NULL)
   {
      // 1-й раз выполняем трейлинг сразу после создания или обнаружения позиции
      tr[].trail();
   }
}
```

Поиск подходящей открытой позиции делегирован функции *GetMyPosition*, а открытие новой позиции — функции *OpenPosition*. Обе представлены ниже. В любом случае мы получаем тикет позиции, и для неё создаем объект трейлинга.

```
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;
}
```

Из названий встроенных функций легко предположить их назначение и общий смысл алгоритма. В цикле по всем открытым позициям (*PositionsTotal*) мы последовательно выбираем каждую из них с помощью *PositionGetSymbol* и получаем её символ. Если символ совпал с запрошенным, читаем и сравниваем свойство позиции POSITION_MAGIC с переданным "магиком". Все функции для работы с позициями будут рассмотрены в отдельном разделе.

Функция вернет *true*, как только найдется первая подходящая позиция. При этом позиция останется выбранной в торговом окружении терминала, что дает возможность в остальном коде читать её другие свойства при необходимости.

Алгоритм открытия позиции нам уже знаком.

```
ulong OpenPosition()
{
   MqlTradeRequestSync request;

   // значения по умолчанию
   const bool wantToBuy = Type == MARKET_BUY;
   const double volume = SymbolInfoDouble(_Symbol, SYMBOL_VOLUME_MIN);
   // опциональные поля заполняем напрямую в структуре
   request.magic = Magic;
   request.deviation = Deviation;
   request.comment = Comment;
   ResetLastError();
   // выполняем выбранную торговую операцию и ждем её подтверждения
   if((bool)(wantToBuy ? request.buy(volume) : request.sell(volume))
      && request.completed())
   {
      Print("OK Order/Deal/Position");
   }

   return request.position; // ненулевое значение - признак успеха
}
```

Для наглядности посмотрим, как эта программа работает в тестере, в визуальном режиме.

После компиляции откроем панель тестера стратегий в терминале, на закладке *Обзор* выберем первый вариант: *Одиночный тест*.

На закладке *Настройки* выберем:

- в выпадающем списке *Советник*: MQL5Book\p6\TralingStop

- *Символ*: EURUSD
- *Таймфрейм*: H1
- *Интервал*: последний год, месяц или произвольный
- *Форвард*: нет
- *Задержки*: отключены
- *Моделирование*: на основе реальных или генерируемых тиков
- *Оптимизация*: отключена
- опция *Визуальный режим*: включена

Нажав кнопку *Старт*, увидим в отдельном окне тестера примерно такую картину:

Стандартный трейлинг-стоп в тестере

В журнале появятся записи следующего вида:

```
2022.01.10 00:02:00   Autodetected daily distance (points): 373
2022.01.10 00:02:00   No positions found, lets 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 1.13
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
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.
2022.01.10 10:06:38   Autodetected daily distance (points): 373
2022.01.10 10:06:38   No positions found, lets open it...
```

Обратите внимание, как алгоритм смещает уровень SL вверх при благоприятном движении цены, вплоть до момента, когда позиция закрывается по стоплоссу. Сразу после ликвидации позиции программа открывает новую.

Чтобы проверить возможность применения нестандартных механизмов сопровождения, реализуем пример алгоритма на скользящей средней. Для этого вернемся в файл *TrailingStop.mqh* и опишем производный класс *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];
   }
};
```

Его особенностью является создание в конструкторе экземпляра индикатора *iMA*: период, метод усреднения, тип цены передаются через параметры.

В переопределенном методе *detectLevel* мы считываем значение из буфера индикатора, причем по умолчанию это делается со смещением 1 бар, то есть бар — закрытый, и его показания не меняются по приходу тиков. Желающие могут брать значение с нулевого бара, но такие сигналы нестабильны для всех типов цен, кроме PRICE_OPEN.

Для использования нового класса в том же тестовом эксперте *TrailingStop.mq5* добавим еще один входной параметр *MATrailingPeriod* с периодом скользящей (прочие параметры индикатора оставим без изменений).

```
input int MATrailingPeriod = 0;    // Period for Trailing by MA (0 = disabled)
```

Значение 0 в данном параметре отключает трейлинг по скользящей средней. Если он включен, настройки дистанции в параметре *TrailingDistance* игнорируются.

В зависимости от его установки будем создавать либо стандартный объект трейлинга *TrailingStop*, либо производный с *iMA* — *TrailingStopByMA*.

```
      ...
      tr = MATrailingPeriod > 0 ?
         new TrailingStopByMA(ticket, MATrailingPeriod) :
         new TrailingStop(ticket, distance, TrailingStep);
      ...
```

Посмотрим, как обновленная программа ведет себя в тестере. В настройках эксперта поставьте ненулевой период для MA, например, 10.

Трейлинг-стоп по скользящей средней в тестере

Обратите внимание, что в те моменты, когда средняя подходит близко к цене, возникает эффект частого срабатывания стоплосса и закрытия позиции. Когда средняя располагается над котировками, защитный уровень вообще не ставится, потому что для покупки это некорректно. Это следствие того, что наш эксперт не имеет никакой стратегии и всегда открывает позиции одного типа, независимо от обстановки на рынке. Для продаж будет эпизодически возникать такая же парадоксальная ситуация, когда средняя идет под ценой — значит рынок растет, а робот "упрямо" встает в короткую позицию.

В рабочих стратегиях, как правило, направление позиции выбирается с учетом движения рынка, и скользящая средняя находится с правильной стороны относительно текущей цены — там, где разрешено ставить стоплосс.

6.4.17 Полное и частичное закрытие позиции

Технически закрытие позиции можно представить, как торговую операцию, обратную по направлению к той, что использовалась для открытия. Например, для выхода из покупки нужно осуществить продажу (ORDER_TYPE_SELL в поле *type*), а для выхода из продажи — покупку (ORDER_TYPE_BUY в поле *type*).

Тип торговой транзакции в поле *action* структуры *MqlTradeTransaction* остается прежним — TRADE_ACTION_DEAL.

На счете с хеджингом закрываемую позицию следует указать с помощью тикета в поле *position*. Для счетов с неттинговым учетом достаточно указать название символа в поле *symbol*, поскольку на них возможна только одна позиция по символу. Однако закрывать позиции по тикету можно и здесь.

В целях унификации кода имеет смысл заполнять оба поля *position* и *symbol* вне зависимости от типа счета.

Также обязательно задать объем в поле *volume*. Если он равен объему позиции, она будет закрыта полностью. Однако, указав меньшее значение, существует возможность закрыть лишь часть позиции.

В следующей таблице все поля структуры, которые обязательно задавать, помечены звездочкой, а опциональные — плюсом.

Поле	Неттинг	Хедж
action	*	*
symbol	*	+
position	+	*
type	*	*
type_filling	*	*
volume	*	*
price	*'	*'
deviation	±	±
magic	+	+
comment	+	+

Поле *price* помечено звездочкой с риской, потому что оно является обязательным только для символов с режимом исполнения по запросу (*Request*) и немедленным (*Instant*), а для биржевого (*Exchange*) и рыночного (*Market*) исполнения цена в структуре не учитывается.

По аналогичной причине поле *deviation* помечено знаком '±' — оно имеет эффект только для режимов *Instant* и *Request*.

Для упрощения программной реализации закрытия позиции вернемся к нашей расширенной структуре *MqlTradeRequestSync* в файле *MqlTradeSync.mqh*. Метод закрытия позиции по тикету имеет следующий код.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   double partial; // объем после частичного закрытия
   ...
   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;
      ...
```

Здесь мы первым делом проверяем существование позиции, вызвав функцию *PositionSelectByTicket*. Дополнительно этот вызов делает позицию выбранной в торговом окружении терминала, что позволяет читать её свойства последующими функциями. В частности, мы узнаем символ позиции из свойства POSITION_SYMBOL и "переворачиваем" её тип из POSITION_TYPE на обратный, чтобы получить нужный тип ордера.

Напомним, что типы позиций в перечислении ENUM_POSITION_TYPE - это POSITION_TYPE_BUY (значение 0) и POSITION_TYPE_SELL (значение 1). В перечислении типов ордеров ENUM_ORDER_TYPE точно такие же значения занимают рыночные операции: ORDER_TYPE_BUY и ORDER_TYPE_SELL. Именно поэтому мы можем приводить первое перечисление ко второму, а для получения обратного направления торговли достаточно переключить нулевой бит с помощью операции исключающего ИЛИ ('^'): из 0 получим 1, а из 1 — 0.

Обнуление поля *price* означает автоматический выбор правильной текущей цены (*Ask* или *Bid*) перед отправкой запроса: это делается чуть позднее, внутри вспомогательного метода *setVolumePrices*, который вызывается далее по ходу алгоритма, из метода *market*.

Вызов самого метода *_market* мы видим парой строк ниже. Метод *_market* формирует рыночный ордер на полный объем или часть, с учетом всех заполненных полей структуры.

```
      const double total = lot == 0 ? PositionGetDouble(POSITION_VOLUME) : lot;
      partial = PositionGetDouble(POSITION_VOLUME) - total;
      return _market(symbol, total);
   }
```

Данный фрагмент слегка упрощен по сравнению с актуальным исходным кодом — там предусмотрена обработка редкой, но возможной ситуации, когда объем позиции превышает максимальный разрешенный объем в одном ордере по символу (свойство SYMBOL_VOLUME_MAX). В таком случае позицию приходится закрывать по частям — несколькими приказами.

Также обратим внимание, что поскольку позиция может закрываться частично, нам пришлось добавить в структуру поле *partial*, куда кладется планируемый остаток объема после операции. Для полного закрытия это будет, разумеется, 0. Эта информация потребуется для дальнейшей проверки завершения операции.

Для счетов с неттингом имеется вариант метода *close*, идентифицирующий позицию по имени символа. Он сводится к выделению позиции по символу, получению её тикета и далее обращению к предыдущей версии *close*.

```
bool close(const string name, const double lot = 0)
{
   if(!PositionSelect(name)) return false;
   return close(PositionGetInteger(POSITION_TICKET), lot);
}
```

В структуре *MqlTradeRequestSync* у нас предусмотрен метод *completed*, обеспечивающий при необходимости синхронное ожидание завершения операции. Теперь нам требуется дополнить его для закрытия позиций, в ветви, где *action* равно TRADE_ACTION_DEAL. Различать открытие позиции и закрытие будем по нулевому значению в поле *position*: при открытии позиции тикета нет, при закрытии — он есть.

```
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);
      }
   }
```

Для проверки фактического закрытия позиции в структуру *MqlTradeResultSync* добавлен метод *closed*. Перед его вызовом мы записываем тикет позиции в поле *result.position*, чтобы структура результата могла отслеживать момент, когда соответствующий тикет пропадет из торгового окружения терминала, или когда объем сравняется с *result.partial* в случае частичного закрытия.

А вот и сам метод *closed*. Он построен по уже известному принципу: сначала проверка успешности кода возврата сервера, а затем ожидание с помощью метода *wait* некоторого условия.

```
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;
   }
```

В данном случае для проверки условия исчезновения позиции нам пришлось реализовать новую функцию *positionRemoved*.

```
   static bool positionRemoved(MqlTradeResultSync &ref)
   {
      if(ref.partial)
      {
         return PositionSelectByTicket(ref.position)
            && TU::Equal(PositionGetDouble(POSITION_VOLUME), ref.partial);
      }
      return !PositionSelectByTicket(ref.position);
   }
```

Операцию закрытия позиций протестируем на эксперте *TradeClose.mq5*, реализующем простую торговую стратегию: при двух последовательных барах в одном направлении входим в рынок, а как только очередной бар закроется в противоположном направлении предыдущему тренду — выходим из рынка. Повторяющиеся сигналы во время продолжительных трендов будут игнорироваться, то есть в рынке будет максимум одна позиция (минимальным лотом) или ни одной.

Никаких настраиваемых параметров у эксперта не будет: только величина проскальзывания (*Deviation*) и уникальный номер (*Magic*). Неявными параметрами являются таймфрейм и рабочий символ графика.

Для отслеживания наличия уже открытой позиции воспользуемся функцией *GetMyPosition* из предыдущего примера *TradeTrailing.mq5*: напомним, она осуществляет поиск среди позиций по символу и номеру эксперта, и возвращает логический признак *true*, если подходящая позиция найдена.

Также практически без изменений возьмем и функцию *OpenPosition*: она открывает позицию в соответствии с типом рыночного ордера, переданного в единственном параметре. Здесь этот параметр будет поступать из алгоритма определения тренда, а ранее (в *TrailingStop.mq5*) тип ордера задавался пользователем через входную переменную.

Новая функция, реализующая закрытие позиции, — это *ClosePosition*. Поскольку заголовочный файл *MqlTradeSync.mqh* взял на себя всю рутину, здесь нам остается, по большому счету, только

вызвать метод *request.close(ticket)* для переданного тикета позиции и дождаться завершения удаления посредством *request.completed()*.

В принципе, последнее можно не делать, если эксперт анализирует ситуацию на каждом тике. В этом случае потенциальная проблема с удалением позиции оперативно вскроется на следующем тике, и эксперт может повторить попытку удаления. Однако данный эксперт имеет торговую логику, основанную на барах, а потому не имеет смысла анализировать каждый тик. Далее мы реализуем специальный механизм для побаровой работы, и в связи с этим мы синхронно контролируем удаление, иначе позиция осталась бы "висеть" целый бар.

```
ulong LastErrorCode = 0;

ulong ClosePosition(const ulong ticket)
{
   MqlTradeRequestSync request; // пустая структура

   // опциональные поля заполняем напрямую в структуре
   request.magic = Magic;
   request.deviation = Deviation;

   ResetLastError();
   // выполняем закрытие и ждем подтверждения
   if(request.close(ticket) && request.completed())
   {
      Print("OK Close Order/Deal/Position");
   }
   else // в случае проблем выводим диагностику
   {
      Print(TU::StringOf(request));
      Print(TU::StringOf(request.result));
      LastErrorCode = request.result.retcode;
      return 0; // ошибка, код для анализа в LastErrorCode
   }

   return request.position; // ненулевое значение - успех
}
```

Можно задаться вопросом, почему бы функции *ClosePosition* не возвращать 0 в случае успешного удаления позиции, а иначе — непосредственно код ошибки. Этот, на первый взгляд, экономный подход, сделал бы поведение двух функций *OpenPosition* и *ClosePosition* различным: в вызывающем коде потребовалось бы вкладывать вызовы этих функций в противоположные по смыслу логические выражения, а это вносило бы путаницу. Кроме того, глобальная переменная *LastErrorCode* нам в любом случае понадобилась для добавления информации об ошибке внутри функции *OpenPosition*. Да и проверка в виде *if(условие)* более органично интерпретируется как успех, нежели *if(!условие)*.

Функция, формирующая торговые сигналы по вышеописанной стратегии, называется *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; // открыть длинную позицию
   }
   
   if(iClose(_Symbol, _Period, 1) < iClose(_Symbol, _Period, 2)
      && iClose(_Symbol, _Period, 2) < iClose(_Symbol, _Period, 3))
   {
      return ORDER_TYPE_SELL; // открыть короткую позицию
   }
   
   return (ENUM_ORDER_TYPE)-1; // закрыть
}
```

Функция возвращает значение типа ENUM_ORDER_TYPE, причем два стандартных элемента (ORDER_TYPE_BUY и ORDER_TYPE_SELL), как и следует ожидать, инициируют покупки и продажи, соответственно. А специальное значение -1 (отсутствующее в перечислении) будет использоваться как сигнал закрытия.

Для активации эксперта, исходя из торгового алгоритма, применим обработчик *OnTick*. Как мы помним, в принципе существуют другие опции, подходящие для других стратегий, например, таймер для торговли по новостям, события стакана для торговли с учетом объемов.

Сначала разберем функцию в упрощенном виде, без обработки потенциальных ошибок. В самом начале идет блок, обеспечивающий срабатывание дальнейшего алгоритма только при открытии нового бара.

```
void OnTick()
{
   static datetime lastBar = 0;
   if(iTime(_Symbol, _Period, 0) == lastBar) return;
   lastBar = iTime(_Symbol, _Period, 0);
   ...
```

Далее получаем текущий сигнал из функции *GetTradeDirection*.

```
   const ENUM_ORDER_TYPE type = GetTradeDirection();
```

В случае наличия позиции проверяем, получен ли сигнал на её закрытие и вызываем *ClosePosition* при необходимости. Если же позиции ещё нет, и есть сигнал войти в рынок, вызываем *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);
      }
   }
```

Для анализа ошибок потребуется заключить вызовы *OpenPosition* и *ClosePosition* в условные операторы, и предпринять некие действия для восстановления рабочего состояния программы. В простейшем случае достаточно повторить запрос на следующем тике, но делать это желательно ограниченное число раз. Поэтому заведем статические переменные со счетчиком и лимитом ошибок.

```
void OnTick()
{
   static int errors = 0;
   static const int maxtrials = 10; // не более чем 10 попыток на бар

   // ожидаем появления нового бара, если не было ошибок
   static datetime lastBar = 0;
   if(iTime(_Symbol, _Period, 0) == lastBar && errors == 0) return;
   lastBar = iTime(_Symbol, _Period, 0);
   ...
```

Важно, что механизм побаровой работы временно отключается, если появились ошибки, так как их желательно преодолеть как можно скорее.

Подсчет ошибок делаем в условных операторах вокруг *ClosePosition* и *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;
         }
      }
      // слишком много ошибок на бар
      if(errors >= maxtrials) errors = 0;
      // ошибка достаточно серьезная, чтобы сделать паузу
      if(IS_TANGIBLE(LastErrorCode)) errors = 0;
   }
```

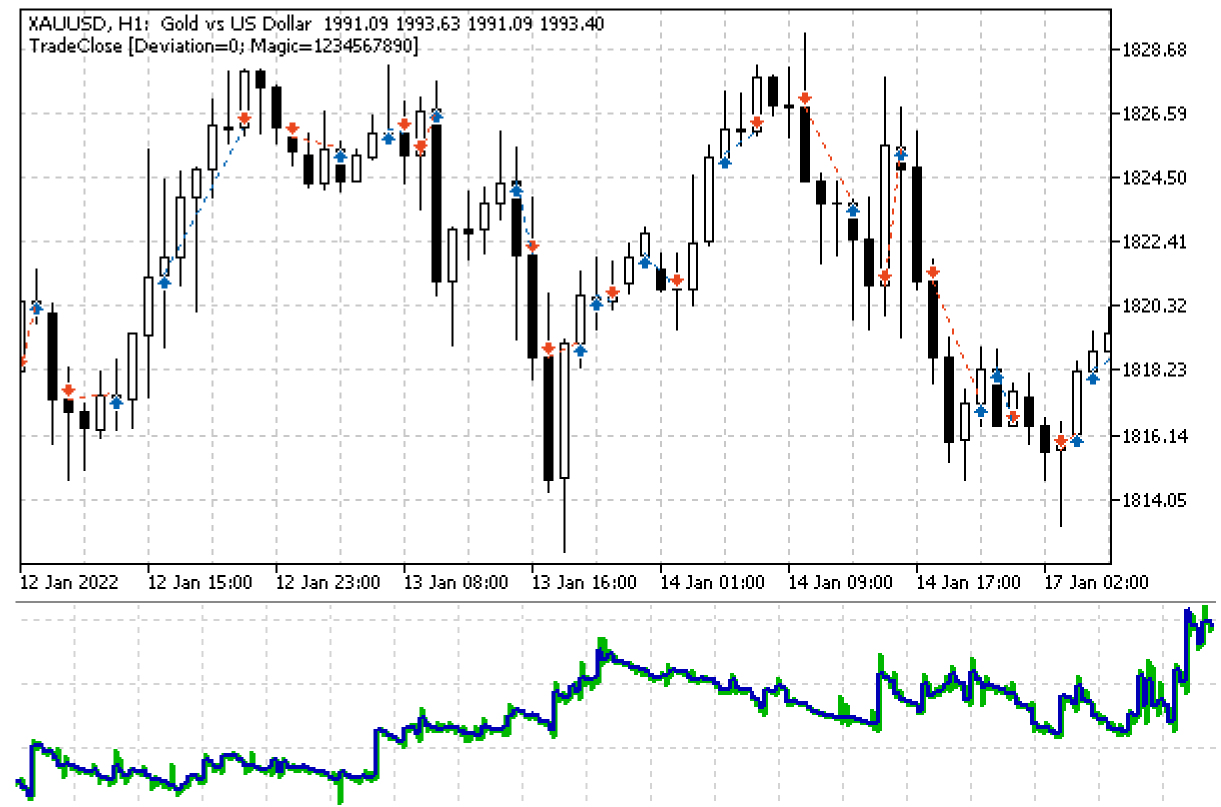
Установка переменной *errors* в 0 вновь включает механизм побаровой работы и прекращает попытки повторить запрос вплоть до следующего бара.

Макрос IS_TANGIBLE определен в *TradeRetcode.mqh* как:

```
   #define IS_TANGIBLE(T) ((T) >= TRADE_RETCODE_ERROR)
```

Ошибки с меньшими кодами являются операционными, то есть в некотором смысле нормальными. Большие коды требуют анализа и различных действий, в зависимости от причины проблемы: неверные параметры запроса, постоянные или временные запреты в торговом окружении, нехватка средства и так далее. Мы представим усовершенствованный классификатор ошибок в разделе Модификация отложенного ордера.

Запустим эксперт в тестере на XAUUSD,H1 с начала 2022 года, моделирование реальных тиков. На следующем коллаже показан фрагмент графика со сделками, а также кривая баланса.



Результаты тестирования TradeClose на XAUUSD,H1

По отчету и журналу легко убедиться, что связка нашей простой торговой логики и двух операций открытия и закрытия позиций работает исправно.

Помимо простого закрытия позиции платформа поддерживает возможность взаимного закрытия двух встречных позиций на счетах с хеджированием.

6.4.18 Полное и частичное закрытие встречных позиций (хедж)

На счетах с хеджированием разрешено одновременно открывать несколько позиций, причем в большинстве случаев эти позиции могут быть и противоположного направления. В некоторых юрисдикциях на хедж-счета накладывают ограничение: одновременно можно иметь только позиции одного направления — выяснить это можно, получив код ошибки TRADE_RETCODE_HEDGE_PROHIBITED при попытке выполнить встречную торговую операцию. Также зачастую данное ограничение коррелирует с настройкой свойства счета ACCOUNT_FIFO_CLOSE, равного *true*.

Когда две противоположных позиции открыты одновременно, платформа поддерживает механизм их одновременного взаимного закрытия с помощью операции TRADE_ACTION_CLOSE_BY. Для выполнения данного действия в структуре *MqlTradeTransaction* следует заполнить помимо поля *action* только два поля — *position* и *position_by* с тикетами закрываемых позиций.

Доступность этой возможности зависит от свойства SYMBOL_ORDER_MODE финансового инструмента: в битовой маске разрешенных флагов должен присутствовать SYMBOL_ORDER_CLOSEBY (64).

Данная операция не только упрощает закрытие (одна операция вместо двух), но и позволяет сэкономить один спред.

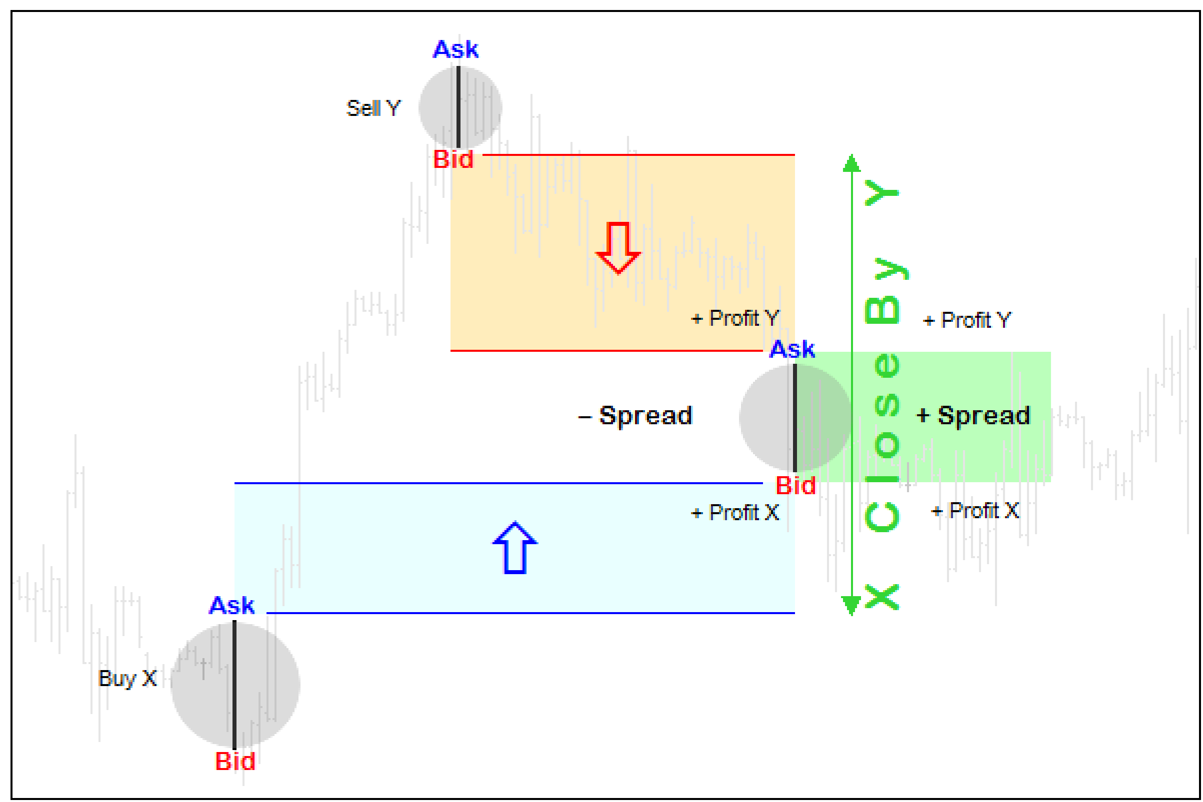
Как известно, любая новая позиция начинает торговаться с убытка величиной со спред. Например, при покупке какого-либо финансового инструмента сделка заключается по цене *Ask*, но для сделки выхода, то есть продажи, актуальной является цена *Bid*. Для короткий позиции — обратная ситуация: сразу после входа по цене *Bid* мы начинаем отслеживать цену *Ask* для потенциального выхода.

Если одновременно закрывать позиции штатным способом, их цены выхода окажутся на расстоянии текущего спреда друг от друга. Однако если воспользоваться операцией TRADE_ACTION_CLOSE_BY, то обе позиции закроются без учета текущих цен. Цена, по которой происходит взаимозачет позиций, равна цене открытия позиции *position_by* (в структуре запроса). Именно она фигурирует в ордере ORDER_TYPE_CLOSE_BY, генерируемом по запросу TRADE_ACTION_CLOSE_BY.

К сожалению, в отчетах в разрезе сделок и позиций цены закрытия и открытия противоположных позиций/сделок отображаются парами одинаковых значений, в зеркальном направлении, из-за чего складывается впечатление о двойной прибыли или убытке. На самом деле финансовый результат операции (разница между ценами с поправкой на лот) записывается только на сделку выхода первой позиции (поле *position* в структуре запроса). Результат второй сделки выхода всегда равен 0, невзирая на разницу цен.

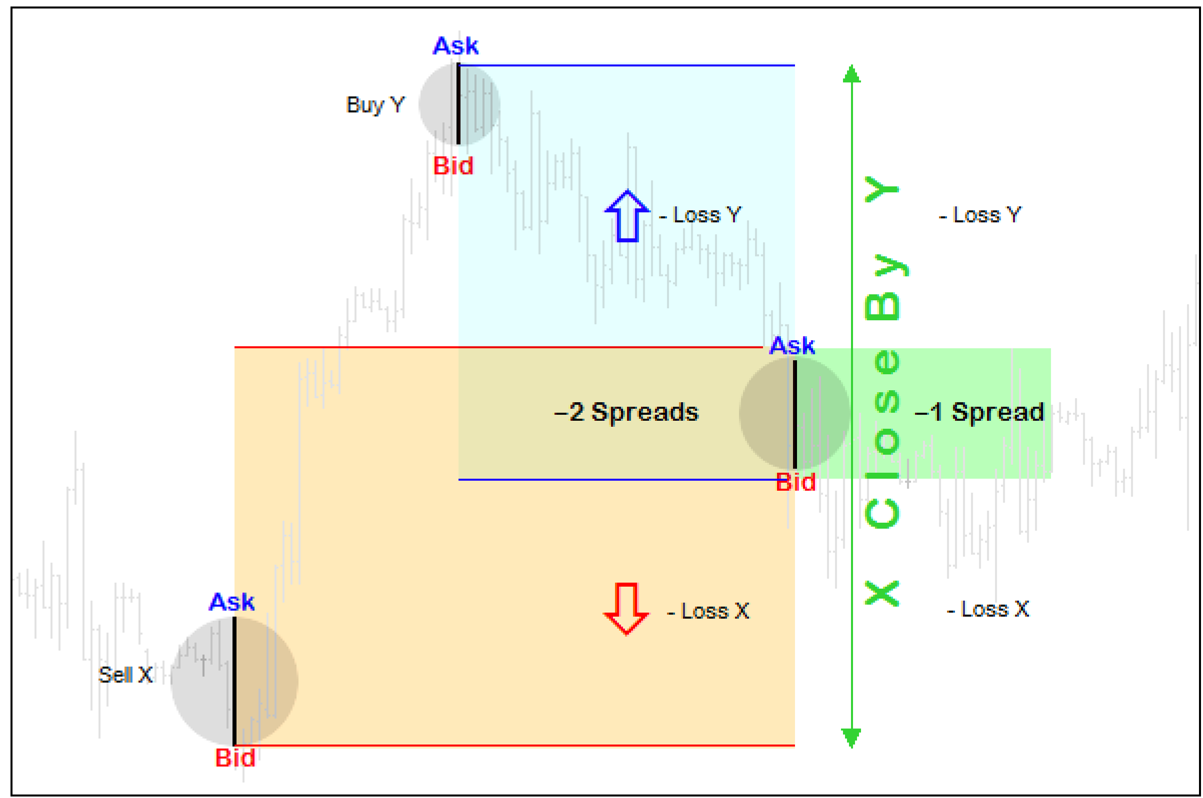
Еще одно последствие этой асимметричности заключается в том, что от перемены мест тикетов в полях *position* и *position_by*, в торговом отчете меняется статистика прибылей и убытков в разрезе длинных и коротких трейдов, например, прибыльных длинных трейдов может стать больше ровно настолько, насколько уменьшилось количество прибыльных коротких трейдов. Но на общий результат, это в принципе не должно влиять, если считать, что от порядка передачи тикетов задержка в исполнении приказа не зависит.

На следующей схеме приведено графическое пояснение процесса (размер спредов намеренно преувеличен).



Учет спреда при встречном закрытии прибыльных позиций

Здесь показан случай прибыльной пары позиций. Если бы позиции имели противоположные направления и находились в убытке, то при их раздельном закрытии спред учёлся бы дважды (в каждой). Встречное закрытие позволяет уменьшить убыток на один спред.



Учет спреда при встречном закрытии убыточных позиций

Встречные позиции не обязаны быть равного объема. Операция встречного закрытия сработает на минимальный из двух объемов.

В файле *MqlTradeSync.mqh* встречное закрытие реализовано с помощью метода *closeby* с двумя параметрами, принимающими тикеты позиций.

```
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)
      {
         // запоминаем, какая позиция должна исчезнуть
         if(volume1 < volume2)
            result.position = ticket1;
         else
            result.position = ticket2;
      }
      return OrderSend(this, result);
   }
```

Для контроля результата закрытия мы запоминаем в переменной *result.position* тикет меньшей позиции. В методе *completed* и структуре *MqlTradeResultSync* уже все готово для синхронного отслеживания закрытия позиции: этот же алгоритм работал и при обычном закрытии позиции.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   ...
   bool completed()
   {
      ...
      else if(action == TRADE_ACTION_CLOSE_BY)
      {
         return result.closed(timeout);
      }
      return false;
   }
```

Встречные позиции обычно используются как замена стоп-приказу или попытка взять прибыль на кратковременной коррекции, оставаясь в рынке и по основному тренду. Вариант использования псевдо-стоп-приказа позволяет отложить решение о реальном закрытии позиций на некоторое время, продолжая анализ движений рынка в надежде на обратный разворот в нужную сторону. Однако следует напомнить, что "локируемые" позиции требуют повышенных залоговых средств, а также подвержены начислению свопов. Именно поэтому сложно представить торговую стратегию, в чистом виде построенную на встречных позициях, которая могла бы послужить примером для данного раздела.

Разовьем идею побаровой стратегии "price action", изложенную в предыдущем примере. Назовем новый эксперт *TradeCloseBy.mq5*.

Оставим прежний сигнал на вход в рынок при обнаружении двух последовательных свечей, закрывшихся в одном направлении. За его формирование будет отвечать функция *GetTradeDirection* — без изменений. Однако разрешим повторные входы, если тренд продолжается. Общее максимально разрешенное количество позиций зададим во входной переменной *PositionLimit*, по умолчанию — 5.

Функция *GetMyPositions* претерпит некоторые изменения: у неё добавятся два параметра — ссылки на массивы, принимающие тикеты позиций — раздельно покупки и продажи.

```
#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;
}
```

Функция возвращает размер наименьшего массива из двух. Когда он больше нуля, у нас появляется возможность закрыть встречные позиции.

Если минимальный массив нулевого размера, функция вернет размер другого массива, но со знаком минус — просто, чтобы дать знать вызывающему коду, что все позиции в одном направлении.

Если позиций нет ни в одном направлении, функция вернет 0.

Открытие позиций останется в ведении функции *OpenPosition* — без изменений.

Закрытие будет осуществляться только в режиме двух встречных позиций в новой функции *CloseByPosition*. Иными словами, данный эксперт не способен закрывать позиции по одной, обычным способом. Разумеется, в реальном роботе такой принцип вряд встретится, но для примера встречного закрытия подойдет очень хорошо. Если нам потребуется закрыть одиночную позицию, достаточно открыть для неё встречную (в этот момент плавающая прибыль или убыток фиксируется) и вызвать *CloseByPosition* для двух.

```
bool CloseByPosition(const ulong ticket1, const ulong ticket2)
{
   MqlTradeRequestSync request;
   request.magic = Magic;

   ResetLastError();
   // отправляем запрос и ждем его завершения
   if(request.closeby(ticket1, ticket2))
   {
      Print("Positions collapse initiated");
      if(request.completed())
      {
         Print("OK CloseBy Order/Deal/Position");
         return true; // успех
      }
   }

   Print(TU::StringOf(request));
   Print(TU::StringOf(request.result));

   return false; // ошибка
}
```

Здесь как раз используется описанный выше метод *request.closeby* с заполнением полей *position*, *position_by* и вызовом *OrderSend*.

Торговая логика описана в обработчике *OnTick*, который как и раньше анализирует конфигурацию цен лишь в момент формирования нового бара и получает сигнал из функции *GetTradeDirection*.

```
void OnTick()
{
   static bool error = false;
   // ждем формирования нового бара, если не было ошибки
   static datetime lastBar = 0;
   if(iTime(_Symbol, _Period, 0) == lastBar && !error) return;
   lastBar = iTime(_Symbol, _Period, 0);

   const ENUM_ORDER_TYPE type = GetTradeDirection();
   ...
```

Далее заполняем массивы *ticketsLong* и *ticketsShort* тикетами позиций по рабочему символу и с заданным *Magic*-номером. Если функция *GetMyPositions* вернула значение больше нуля, оно означает количество сформировавшихся пар встречных позиций. Их можно закрыть в цикле с помощью функции *CloseByPosition*. Сочетание пар в данном случае выбирается случайно (в порядке следования позиций в окружении терминала), однако на практике может быть важно подбирать пары по объемам или таким образом, чтобы сперва закрывались наиболее прибыльные.

```
      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;
         }
      }
      ...
```

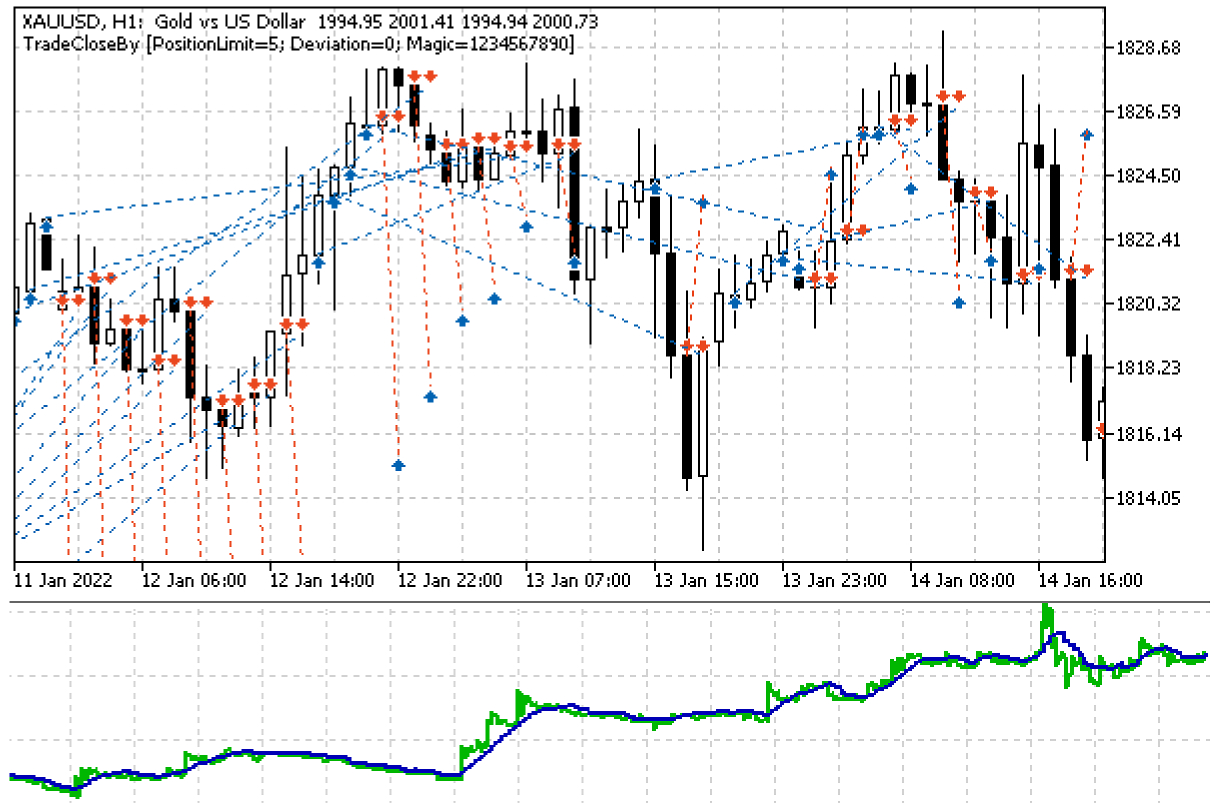
При любом другом значении *n* следует проверить, есть ли сигнал (возможно, повторный) на вход в рынок, и выполнить его, вызвав *OpenPosition*.

```
      else if(type == ORDER_TYPE_BUY || type == ORDER_TYPE_SELL)
      {
         error = !OpenPosition(type);
      }
      ...
```

Наконец, если открытые позиции все же есть, но они в одном направлении, мы проверяем, не достигло ли их количество лимита, и если да, то формируем встречную позицию, чтобы на следующем баре "схлопнуть" две из них (тем самым закрыть одну любую позицию из старых).

```
      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);
            }
         }
      }
   }
```

Запустим эксперт в тестере на XAUUSD,H1 с начала 2022 года, с настройками по умолчанию. Ниже представлен внешний вид графика с позициями в процессе работы программы, а также кривая баланса.

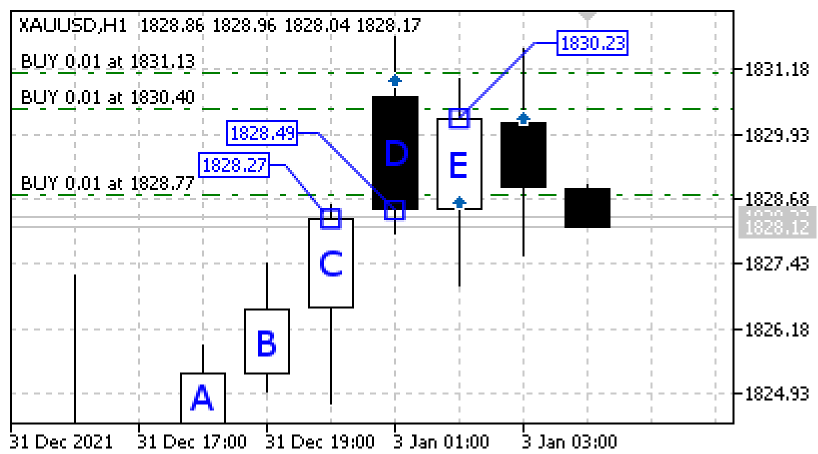


Результаты тестирования TradeCloseBy на XAUUSD,H1

В журнале нетрудно найти моменты, когда один тренд заканчивается (покупки с тикетами от #2 до #4), и начинают генерироваться сделки в противоположном направлении (продажа #5), после чего срабатывает встречное закрытие.

```
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
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
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
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 / 1826.2
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 18
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 XAUU
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
```

Интересным артефактом является сделка #3. Внимательный читатель заметит, что она открылась ниже предыдущей, вроде бы нарушая нашу стратегию. На самом деле ошибки здесь нет, и это следствие того, что условия сигналов прописаны максимально просто: только на основе цен закрытия баров. Поэтому разворотная медвежья свеча (D), открывшаяся с гепом вверх и закрывшаяся выше окончания предыдущей бычьей свечи (C), сгенерировала сигнал для покупки. Данная ситуация иллюстрируется следующим скриншотом.



Сделки на восходящем тренде по ценам закрытия

Все свечи в последовательности A, B, C, D, E закрываются выше предыдущей и стимулируют продолжение покупок. Для исключения подобных артефактов следует дополнительно проводить анализ направления самих баров.

Последний момент, на который следует обратить внимание в данном примере, это функция *OnInit*. Поскольку эксперт использует операцию TRADE_ACTION_CLOSE_BY, здесь выполняются проверки соответствующих настроек счета и рабочего символа.

```
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;
}
```

Если одно из свойств не поддерживает встречное закрытие, эксперт не сможет продолжить работу. При создании рабочих роботов эти проверки, как правило, осуществляются внутри торгового алгоритма и переключают программу в альтернативные режимы, в частности, на одиночное закрытие позиций и поддержание совокупной позиции в случае неттинга.

6.4.19 Установка отложенного ордера

В разделе Типы ордеров мы теоретически рассмотрели все поддерживаемые платформой варианты установки отложенных ордеров. С практической точки зрения ордера создаются с помощью функций *OrderSend/OrderSendAsync*, для которых предварительно заполняется структура запроса *MqlTradeRequest* по особым правилам. В частности, поле *action* должно содержать значение TRADE_ACTION_PENDING из перечисления ENUM_TRADE_REQUEST_ACTIONS. С учетом него, следующий перечень полей является обязательным:

- action
- symbol
- volume
- price
- type (значение по умолчанию 0 соответствует ORDER_TYPE_BUY)
- type_filling (значение по умолчанию 0 соответствует ORDER_FILLING_FOK)
- type_time (значение по умолчанию 0 соответствует ORDER_TIME_GTC)
- expiration (значение по умолчанию 0, не важно при ORDER_TIME_GTC)

Если нулевые умолчания отвечают поставленной задаче, некоторые из последних 4-х полей можно при заполнении пропустить.

Поле *stoplimit* является обязательным только для ордеров типов ORDER_TYPE_BUY_STOP_LIMIT и ORDER_TYPE_SELL_STOP_LIMIT.

Опциональными полями являются:

- sl
- tp
- magic
- comment

Нулевые значения в *sl* и *tp* обозначают отсутствие защитных уровней.

Дополним наши структуры в файле *MqlTradeSync.mqh* методами для проверки значений и заполнения полей. Принцип формирования всех типов ордеров одинаков, поэтому рассмотрим пару частных случаев установки лимитных ордеров на покупку и продажу. Остальные типы буду отличаться только значением поля type. Публичные методы с полным набором обязательных полей, а также защитных уровней, называются согласно типам: *buyLimit* и *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);
}
```

Поскольку в структуре присутствует поле *symbol*, при необходимости инициализируемое в конструкторе, существуют аналогичные методы без параметра *name*: они вызывают приведенные выше методы, передавая *symbol* первым параметром. Таким образом, чтобы создать ордер с минимальными усилиями, достаточно написать:

```
MqlTradeRequestSync request; // по умолчанию использует текущий символ графика
request.buyLimit(volume, price);
```

Общая часть кода по проверке переданных значений, их нормализации, сохранению в полях структуры и созданию отложенного ордера выведена во вспомогательный метод *_pending*. Он возвращает тикет ордера в случае успеха или 0 в случае проблем.

```
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;
}
```

Заполнение поля *action* и вызов методов *setSymbol* и *setVolumePrices* уже знакомы нам по предыдущим торговым операциям.

Многострочный оператор *if* гарантирует, что готовящаяся операция присутствует среди разрешенных операций по символу, прописанных в свойстве SYMBOL_ORDER_MODE. Целочисленное деление типа *type* пополам с последующим сдвигом 1 на полученное значение взводит правильный бит в маске разрешенных типов ордеров — это особенность сочетания констант в перечислении ENUM_ORDER_TYPE и свойства SYMBOL_ORDER_MODE. Например, ORDER_TYPE_BUY_STOP и ORDER_TYPE_SELL_STOP имеют значения 4 и 5, которые после деления на 2 дают 2 (с учетом отбрасывания дробной части). Операция *1 << 2* имеет результат 4, равный SYMBOL_ORDER_STOP.

Особенностью отложенных ордеров является обработка срока истечения. Этим занимается метод *setExpiration*. В нем следует убедиться, что заданный режим истечения ENUM_ORDER_TYPE_TIME *duration* разрешен для символа, а дата и время *until* правильно заполнены.

```
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;
}
```

Битовая маска разрешенных режимов доступна в свойстве SYMBOL_EXPIRATION_MODE. Сочетание битов в маске и констант ENUM_ORDER_TYPE_TIME таково, что нам достаточно вычислить выражение *1 << duration* и наложить его на маску: ненулевое значение служит признаком наличия режима.

Для режимов ORDER_TIME_SPECIFIED и ORDER_TIME_SPECIFIED_DAY поле *expiration* с конкретным значением *datetime* не может быть пустым. Причем указанные дата и время не могут быть в прошлом.

Поскольку метод *_pending*, представленный ранее, отправляет в конце запрос на сервер с помощью *OrderSend*, наша программа должна убедиться, что ордер с полученным тикетом действительно был создан (это особенно важно для лимитных ордеров, которые могут выводиться во внешнюю торговую систему). Поэтому в методе *completed*, который используется для "блокирующего" контроля результата, добавим ветвь для операции TRADE_ACTION_PENDING.

```
bool completed()
{
   // прежний код обработки
   // TRADE_ACTION_DEAL
   // TRADE_ACTION_SLTP
   // TRADE_ACTION_CLOSE_BY
   ...
   else if(action == TRADE_ACTION_PENDING)
   {
      return result.placed(timeout);
   }
   ...
   return false;
}
```

В структуре *MqlTradeResultSync* добавим метод *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;
}
```

Его главная задача — дождаться появления ордера с помощью ожидания в функции *orderExist*: она уже использовалась на первом этапе проверки открытия позиции.

Для тестирования нового функционала реализуем эксперт *PendingOrderSend.mq5*. Он позволяет выбрать с помощью входных переменных тип отложенного ордера и все его атрибуты, а затем выполнить запрос с подтверждением.

```
enum ENUM_ORDER_TYPE_PENDING
{                                                          // Строки интерфейса UI
   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;
```

Эксперт будет создавать новый ордер при каждом запуске или смене параметров. Автоматического удаления ордера пока не предусмотрено, потому что мы рассмотрим этот тип операции позднее. В связи с этим не забывайте удалять ордера вручную.

Однократная установка ордера выполняется, как и в некоторых предыдущих примерах, по таймеру (поэтому следует предварительно убедиться, что рынок открыт).

```
void OnTimer()
{
   // один раз выполняем и ждем изменений настроек пользователем
   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");
   }
}
```

Функция *PlaceOrder* принимает все настройки в качестве параметров, отправляет запрос и возвращает признак успеха (ненулевой тикет). Для ордеров всех поддерживаемых типов заранее предустановлены расстояния от текущей цены, рассчитываемые как часть дневного размаха котировок.

```
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[] = // индексируется типом ордера
   {
      0  ,   // ORDER_TYPE_BUY - не используется
      0  ,   // ORDER_TYPE_SELL - не используется
     -0.5,   // ORDER_TYPE_BUY_LIMIT - слегка под ценой
     +0.5,   // ORDER_TYPE_SELL_LIMIT - слегка над ценой
     +1.0,   // ORDER_TYPE_BUY_STOP - далеко над ценой
     -1.0,   // ORDER_TYPE_SELL_STOP - далеко под ценой
     +0.7,   // ORDER_TYPE_BUY_STOP_LIMIT - средне над ценой
     -0.7,   // ORDER_TYPE_SELL_STOP_LIMIT - средне под ценой
      0  ,   // ORDER_TYPE_CLOSE_BY - не используется
   };
   ...
```

Например, коэффициент -0.5 для ORDER_TYPE_BUY_LIMIT означает, что ордер будет поставлен ниже текущей цены на половину дневного размаха (отбой внутрь диапазона), а +1.0 для ORDER_TYPE_BUY_STOP — что ордер окажется на верхней границе диапазона (на пробой).

Сам дневной размах вычисляется следующим образом.

```
   const double range = iHigh(symbol, PERIOD_D1, 1) - iLow(symbol, PERIOD_D1, 1);
   Print("Autodetected daily range: ", (float)range);
   ...
```

Находим значения объема и пункта, которые потребуются ниже.

```
   const double volume = lot == 0 ? SymbolInfoDouble(symbol, SYMBOL_VOLUME_MIN) : lot
   const double point = SymbolInfoDouble(symbol, SYMBOL_POINT);
```

Ценовой уровень установки ордера рассчитываем в переменной *price* по приведенным коэффициентам от общего диапазона.

```
   const double price = TU::GetCurrentPrice(type, symbol) + range * coefficients[type
```

Поле *stoplimit* должно заполняться только для ордеров *_STOP_LIMIT. Значения для него хранит переменная *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;
```

При срабатывании ордеров этих двух типов новый отложенный ордер будет ставиться на цену, являющуюся сейчас текущей. Действительно, в данном сценарии цена движется от текущего значения до уровня *price*, где и происходит активация ордера, а потому "прежняя текущая" цена становится корректным уровнем для отскока, обозначенным лимитным ордером. Ниже мы проиллюстрируем эту ситуацию.

Защитные уровни определяются с привлечением объекта *TU::TradeDirection*, причем в случае стоп-лимитных ордеров отсчет ведется от *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);
```

Далее описывается структура, и заполняются опциональные поля.

```
   MqlTradeRequestSync request(symbol);

   request.magic = magic;
   request.comment = comment;
   // request.type_filling = SYMBOL_FILLING_FOK;
```

Здесь же можно выбрать режим заливки. По умолчанию, *MqlTradeRequestSync* автоматически выбирает первый из разрешенных режимов ENUM_ORDER_TYPE_FILLING.

В зависимости от выбранного пользователем типа ордера, вызываем тот или иной торговый метод.

```
   ResetLastError();
   // заполняем и проверяем нужные поля, отправляем запрос
   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, ur
      break;
   }
   ...
```

Если тикет получен, дожидаемся его появления в торговом окружении терминала.

```
      if(order != 0)
      {
         Print("OK order sent: #=", order);
         if(request.completed()) // ожидаем результат (подтверждение ордера)
         {
            Print("OK order placed");
         }
      }
      Print(TU::StringOf(request));
      Print(TU::StringOf(request.result));
      return order;
   }
```

Запустим эксперт на графике EURUSD с настройками по умолчанию и дополнительно выберем дистанцию до защитных уровней 1000 пунктов. В журнале увидим следующие записи (в предположении, что настройки по умолчанию совпадают с разрешениями для EURUSD на вашем счете).

```
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
```

На графике это выглядит так.

Отложенный ордер ORDER_TYPE_BUY_STOP

Удалим ордер вручную и поменяем тип ордера на ORDER_TYPE_BUY_STOP_LIMIT. В результате получим более сложную картину.

Отложенный ордер ORDER_TYPE_BUY_STOP_LIMIT

Цена, где расположена верхняя пара штрих-пунктирных линий, является ценой срабатывания ордера, в результате чего будет поставлен ORDER_TYPE_BUY_LIMIT ордер на уровне текущей цены, со значениями *Stop Loss* и *Take Profit*, помеченными красными линиями. Уровень *Take Profit* будущего ордера ORDER_TYPE_BUY_LIMIT практически совпадает с уровнем активации только что созданного предварительного ордера ORDER_TYPE_BUY_STOP_LIMIT.

В качестве дополнительного примера для самостоятельного изучения к книге прилагается эксперт *AllPendingsOrderSend.mq5*, который устанавливает сразу 6 отложенных ордеров: по одному каждого типа.

Отложенные ордера всех типов

В результате его запуска с настройками по умолчанию вы можете получить такие записи в журнале:

```
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 Модификация отложенного ордера

MetaTrader 5 позволяет модифицировать некоторые свойства отложенного ордера: цену активации, защитные уровни, срок истечения. Основные свойства, такие как тип ордера или объем, изменять нельзя. В таких случаях ордер следует удалить и поставить вместо него другой. Единственный случай, когда тип ордера может быть изменен самим сервером — это активация stoplimit-ордера, который превращается в соответствующий лимитный ордер.

Программная модификация ордеров выполняется операцией TRADE_ACTION_MODIFY: именно эту константу нужно записать в поле *action* структуры *MqlTradeRequest* перед отправкой на сервер функцией *OrderSend* или *OrderSendAsync*. Тикет модифицируемого ордера указывается в поле *order*. С учетом *action* и *order*, полный перечень обязательных полей для данной операции включает:

- action
- order
- price
- type_time (значение по умолчанию 0 соответствует ORDER_TIME_GTC)
- expiration (значение по умолчанию 0, не важно при ORDER_TIME_GTC)
- type_filling (значение по умолчанию 0 соответствует ORDER_FILLING_FOK)
- stoplimit (только для ордеров типов ORDER_TYPE_BUY_STOP_LIMIT и ORDER_TYPE_SELL_STOP_LIMIT)

Опциональные поля:

- sl
- tp

Если защитные уровни уже были установлены у ордера, их следует указывать, чтобы сохранить. Нулевые значения предписывают удаление *Stop Loss* и/или *Take Profit*.

В структуре *MqlTradeRequestSync* (*MqlTradeSync.mqh*) реализация модификации ордера находится в методе *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;

      // следующие поля нужны для проверок внутри подфункций
      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);
   }
```

Фактическое исполнение запроса, как обычно, производится в методе *completed*, в выделенной ветке оператора *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);
   }
   ...
}
```

Чтобы структура *MqlTradeResultSync* "знала" новые значения свойств редактируемого ордера и могла их сверить с результатом, мы записываем их в свободные поля (они не заполняются сервером в данном типе запроса). Далее в методе *modified* структура результата ожидает применения модификации.

```
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));
   }
```

Здесь мы видим, как свойства ордера считываются при помощи функции *OrderGetDouble* и сравниваются с установленными значениями. Все это происходит по уже привычной схеме: в цикле внутри функции wait, в пределах некоего таймаута *msc* (1000 миллисекунд по умолчанию).

В качестве примера разберем эксперт *PendingOrderModify.mq5*, наследующий от *PendingOrderSend.mq5* некоторые фрагменты кода: в частности, набор входных параметров и функцию *PlaceOrder* для создания нового ордера. Она используется при первом запуске, если ордера для заданного сочетания символа и *Magic*-числа еще нет, тем самым гарантируя, что эксперту есть что модифицировать.

Для поиска подходящего ордера потребовалась новая функция *GetMyOrder*. Она во многом похожа на функцию *GetMyPosition*, которая применялась в примере сопровождения позиции (*TrailingStop.mq5*) для поиска подходящей позиции. Назначение используемых внутри *GetMyOrder* встроенных функций MQL5 API должно быть в общих чертах ясно по их названиям, а техническое описание будет представлено в отдельных разделах.

```
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;
}
```

Входной параметр *Distance2SLTP* теперь отсутствует. Вместо него новый эксперт будет автоматически рассчитывать дневной размах цен и размещать защитные уровни на дистанции половины этого размаха. В начале каждого дня размах будет пересчитываться, также как и новые уровни в полях *sl* и *tp*, на основе чего сформируются запросы на модификацию ордера.

Те, отложенные ордера, которые сработают и превратятся в позиции, будут закрываться сами по достижению *Stop Loss* или *Take Profit*. В принципе, терминал способен сообщать MQL-программе об активации отложенный ордеров и закрытии позиций, если описать в ней обработчики торговых событий. Это позволило бы, например, не создавать новый ордер при наличии открытой позиции, но и текущая стратегия имеет право на существование, а событиями мы займемся позднее.

Основная логика эксперта "зашита" в обработчике *OnTick*.

```
void OnTick()
{
   static datetime lastDay = 0;
   static const uint DAYLONG = 60 * 60 * 24; // количество секунд в сутках
   // отбрасываем "дробную" часть, т.е. время
   if(TimeTradeServer() / DAYLONG * DAYLONG == lastDay) return;
   ...
```

В начале функции пара строк кода обеспечивает однократный запуск алгоритма в начале каждого дня. Для этого мы рассчитываем текущую дату без времени и сравниваем со значением переменной *lastDay*, куда должна записываться последняя дата успешной работы. Статус успеха или ошибки, разумеется, становится ясен уже в конце функции, поэтому мы вернемся к нему позднее.

Далее делается расчет ценового диапазона за предыдущий день.

```
   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);
   ...
```

В зависимости от того, найдется ли ордер или нет в функции *GetMyOrder*, мы либо создадим новый с помощью *PlaceOrder*, либо отредактируем имеющийся с помощью *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);
      }
      ...
```

Обе функции *PlaceOrder* и *ModifyOrder* работают на основе входных параметров эксперта и найденного ценового диапазона. Они возвращают статус запроса, который нужно будет неким образом проанализировать, чтобы решить, какое действие выполнить:

- Обновить переменную *lastDay*, если запрос успешный (ордер был обновлен и до начала следующего дня эксперт в "спячке");
- Оставить пока в *lastDay* "старый" день, чтобы повторить попытку на следующих тиках, если есть временные проблемы (например, торговая сессия еще не началась);
- Остановить эксперт, если обнаружены серьезные проблемы (например, выбранный тип ордера или направление торговли не разрешены на символе).

```
      ...
      if(/* какой-то анализ retcode */)
      {
         lastDay = TimeTradeServer() / DAYLONG * DAYLONG;
      }
   }
```

В разделе о полном и частичном закрытии позиции мы использовали упрощенный анализ с макросом IS_TANGIBLE, который давал ответ в категориях "да" и "нет": ошибка — не ошибка. Очевидно, что такой подход нужно усовершенствовать, и мы к этому вопросу скоро вернемся, а пока сосредоточимся на основном функционале эксперта.

Исходный код функции *PlaceOrder* практически не изменился по сравнению с предыдущим примером, а *ModifyOrder* приведена ниже.

Напомним, что расположение ордеров у нас определялось исходя из дневного диапазона, к которому применялась таблица коэффициентов. Сам принцип оставлен без изменений, однако поскольку у нас теперь две функции, работающие с ордерами — *PlaceOrder* и *ModifyOrder* — таблица коэффициентов *Coefficients* вынесена в глобальный контекст. Мы не будем её здесь повторять и сразу перейдем к функции *ModifyOrder*.

```
   uint ModifyOrder(const ulong ticket, const double range,
      ENUM_ORDER_TYPE_TIME expiration, datetime until)
   {
      // значения по умолчанию
      const string symbol = OrderGetString(ORDER_SYMBOL);
      const double point = SymbolInfoDouble(symbol, SYMBOL_POINT);
      ...
```

Ценовые уровни рассчитываются в зависимости от типа ордера и переданного диапазона range.

```
   const ENUM_ORDER_TYPE type = (ENUM_ORDER_TYPE)OrderGetInteger(ORDER_TYPE);
   const double price = TU::GetCurrentPrice(type, symbol) + range * Coefficients[type

   // origin заполняется только для ордеров *_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);
   ...
```

После вычисления всех значений создаем объект структуры *MqlTradeRequestSync* и выполняем запрос.

```
   MqlTradeRequestSync request(symbol);

   ResetLastError();
   // передаем данные для полей, отсылаем приказ и ждем результата
   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;
}
```

Для анализа *retcode*, который мы должны выполнить в вызывающем блоке внутри *OnTick*, был разработан новый механизм, дополнивший файл *TradeRetcode.mqh*. Все коды возврата сервера поделены на несколько групп "важности", описываемых элементами перечисления TRADE_RETCODE_SEVERITY.

```
enum TRADE_RETCODE_SEVERITY
{
   SEVERITY_UNDEFINED,   // что-то нестандартное - просто выведем в журнал
   SEVERITY_NORMAL,      // нормальное функционирование
   SEVERITY_RETRY,       // попробовать снова (вероятно, несколько раз), обновить окр
   SEVERITY_TRY_LATER,   // следует подождать и попробовать еще
   SEVERITY_REJECT,      // запрос отклонен, вероятно(!) можно попробовать еще раз
                         //
   SEVERITY_INVALID,     // требуется исправить запрос
   SEVERITY_LIMITS,      // требуется проверить ограничения и исправить запрос
   SEVERITY_PERMISSIONS, // требуется уведомить пользователя и изменить настройки про
   SEVERITY_ERROR,       // остановка, выводим информацию в журнал и пользователю
};
```

В упрощенном виде, первая половина соответствует устранимым ошибкам: обычно достаточно подождать некоторое время и выполнить запрос повторно. Вторая половина требует изменить содержимое запроса, проверить настройки счета или символа, разрешения для программы, а в худшем случае — остановить торговлю. Желающие могут провести условную разделительную черту не после SEVERITY_REJECT, как визуально выделено сейчас, а перед ним.

Деление всех кодов по группам выполняет функция *TradeCodeSeverity* (приводится с сокращениями).

```
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];
}
```

Благодаря этому функционалу обработчик *OnTick* может быть дополнен "умной" обработкой ошибок. В статической переменной *RetryFrequency* хранится периодичность, с которой программа будет пытаться повторить запрос в случае некритических ошибок. Последний раз, когда делалась такая попытка, запоминается в переменной *RetryRecordTime*.

```
void OnTick()
{
   ...
   const static int DEFAULT_RETRY_TIMEOUT = 1; // секунды
   static int RetryFrequency = DEFAULT_RETRY_TIMEOUT;
   static datetime RetryRecordTime = 0;
   if(TimeTradeServer() - RetryRecordTime < RetryFrequency) return;
   ...
```

После того, как из функции *PlaceOrder* или *ModifyOrder* получено значение *retcode*, мы узнаем его "критичность" и на её основе выбираем одну из трех альтернатив: остановка эксперта, ожидание в течение таймаута или штатная работа (помечаем успешную модификацию ордера текущим днем в *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;
}
```

В случае повторных проблем, которые отнесены к классу разрешимых, таймаут *RetryFrequency* постепенно увеличивается с каждой следующей ошибкой, но сбрасывается в 1 секунду при успешной обработке запроса.

Следует отметить, что методы прикладной структуры *MqlTradeRequestSync* проверяют большое количество сочетаний параметров на корректность и при обнаружении проблем прерывают процесс, не доводя до вызова *SendRequest*. Такое поведение заложено по умолчанию, но его можно отключить, определив пустой макрос RETURN(X) перед директивой *#include* с *MqlTradeSync.mqh*.

```
#define RETURN(X)
#include <MQL5Book/MqlTradeSync.mqh>
```

При таком макроопределении проверки станут только выводить предупреждения в журнал, но продолжат выполнение методов вплоть до вызова *SendRequest*.

В любом случае, после вызова того или иного метода структуры *MqlTradeResultSync* в поле *retcode* будет помещен код ошибки: либо сервером, либо проверяющими алгоритмами структуры *MqlTradeRequestSync* (здесь пришелся кстати тот факт, что экземпляр *MqlTradeResultSync* включен внутрь *MqlTradeRequestSync*). Запись кодов ошибок и использование макроса RETURN в методах *MqlTradeRequestSync* были для краткости опущены во фрагментах методов, приведенных в книге. Желающие могут ознакомиться с полным исходным кодом в файле *MqlTradeSync.mqh*.

Запустим в тестере, в визуальном режиме, эксперт *PendingOrderModify.mq5* на XAUUSD,H1 (в режиме всех тиков или реальных тиков). С настройками по умолчанию эксперт будет ставить ордера типа ORDER_TYPE_BUY_STOP минимальным лотом. Убедимся по журналу и торговой истории, что программа выставляет отложенные ордера и модифицирует их в начале каждого дня.

```
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 (1830.6
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, ORDE
  » @ 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
  » @ 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 execute
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
  » @ 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 execute
...
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
  » @ 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 execute
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
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 18
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 (1820.1
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, ORDE
  » @ 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,
```

В любой момент ордер может сработать, после чего позиция через какое-то время закрывается по стоплоссу или тейкпрофиту (как в приведенном выше фрагменте).

В некоторых случаях может возникнуть ситуация, когда позиция еще существует на начало следующего дня, и тогда новый ордер будет создан в дополнение к ней, как на приведенном ниже скриншоте.

Эксперт с торговой стратегией на отложенных ордерах в тестере

Обратите внимание, что из-за того, что мы запрашиваем котировки таймфрейма PERIOD_D1 для расчета дневного диапазона, визуальный тестер открывает соответствующий график, помимо текущего рабочего. Такой сервис работает не только в отношении таймфреймов, отличных от рабочего, но и других символов. Это пригодится, в частности, при разработке мультивалютных экспертов.

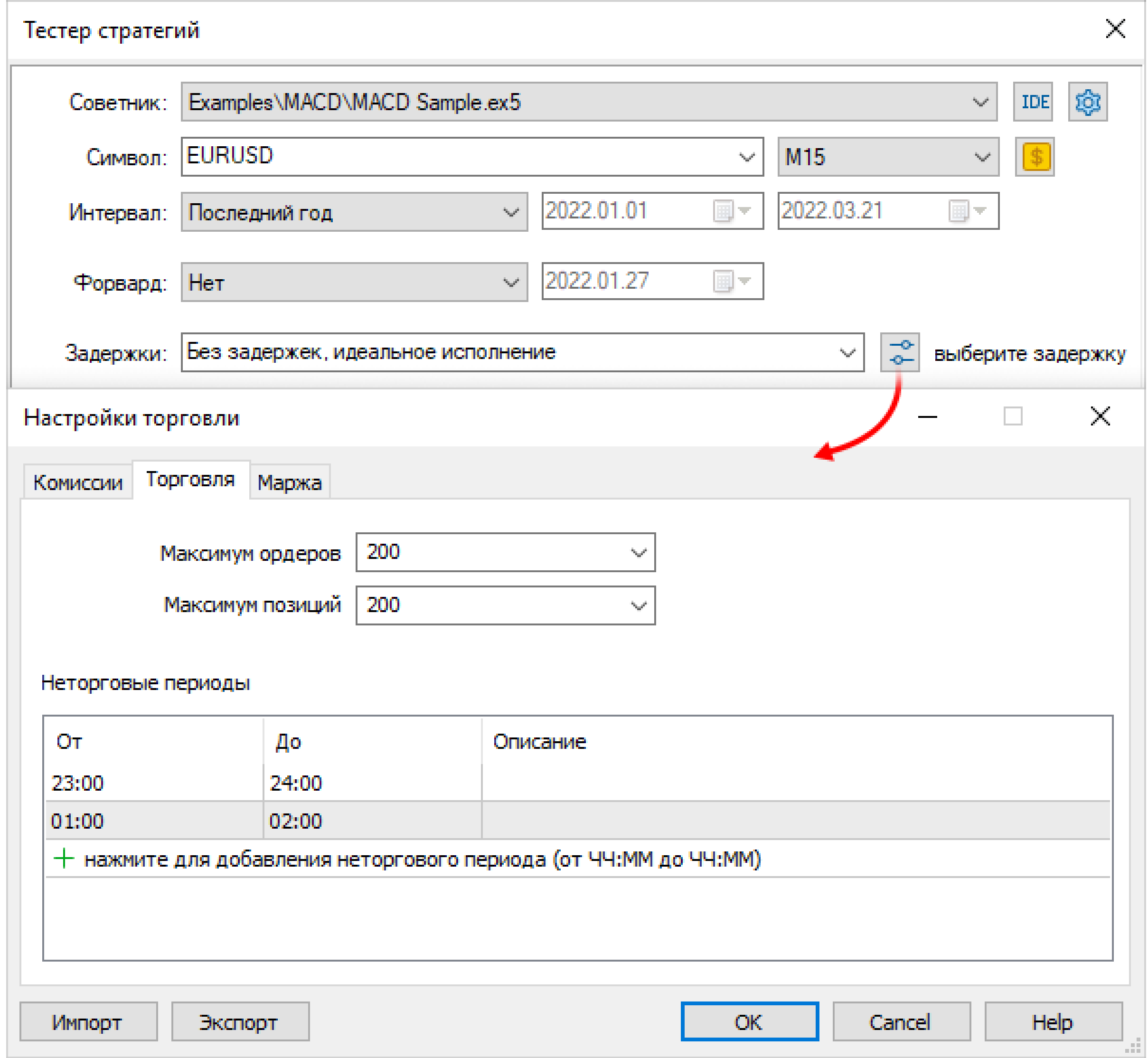
Чтобы проверить, как работает обработка ошибок, попробуйте запретить торговлю эксперту. В результате увидим в журнале:

```
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
```

Данная ошибка относится к критическим, и эксперт останавливает работу.

Для демонстрации одной из более легких ошибок можно было бы использовать обработчик *OnTimer*, а не *OnTick*. Тогда запуск этого же эксперта на символах, где торговые сессии занимают лишь часть суток, мог бы периодически генерировать последовательность некритических ошибок о закрытом рынке ("Market closed"). В этом случае эксперт будет продолжать попытки начать торговлю, постоянно увеличивая время ожидания.

Это, в частности, легко проверить и в тестере, который позволяет настраивать произвольные торговые сессии для любого символа. На вкладке *Настройки*, справа от выпадающего списка *Задержки* находится кнопка, по нажатию которой открывается диалог *Настройка торговли*. В нем следует включить опцию *Использовать свои настройки* и на закладке *Торговля* добавить хотя бы одну запись в таблицу *Неторговые периоды*.



Настройка неторговых периодов в тестере

Обратите внимание, что здесь задаются именно неторговые периоды, а не торговые сессии, то есть данная настройка действует с точностью до наоборот по сравнению со спецификацией символов.

Исключить многие потенциальные ошибки, связанные с ограничениями торговли, может предварительный анализ окружения с помощью класса вроде *Permissions*, представленного в разделе Ограничения и разрешения для операций по счету.

6.4.21 Удаление отложенного ордера

Удаление отложенного ордера выполняется на программном уровне с помощью операции TRADE_ACTION_REMOVE: эту константу следует присвоить полю *action* структуры *MqlTradeRequest* перед вызовом одной из разновидностей функции *OrderSend*. Единственным обязательным полем, помимо *action*, является *order*, куда следует записать тикет удаляемого ордера.

В связи с этим метод *remove* в прикладной структуре *MqlTradeRequestSync* из файла *MqlTradeSync.mqh* довольно простой.

```
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);
   }
```

Проверка факта удаления ордера делается по традиции в методе *completed*.

```
   bool completed()
   {
      ...
      else if(action == TRADE_ACTION_REMOVE)
      {
         result.order = order;
         return result.removed(timeout);
      }
      ...
   }
```

Ожидание фактического удаления ордера производится в методе *removed* структуры *MqlTradeResultSync*, также по привычной схеме.

```
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);
   }
```

Пример эксперта (*PendingOrderDelete.mq5*), демонстрирующего удаление ордера, построим практически целиком на *PendingOrderSend.mq5*. Это обусловлено тем, что так проще гарантировать наличие ордера перед удалением. Таким образом, сразу после запуска эксперт создаст новый ордер с заданными параметрами, а его удаление осуществляется в обработчике *OnDeinit*. Если изменить входные параметры советника, символ или таймфрейм графика, старый ордер также будет удален, а новый создан.

Для хранения тикета ордера заведена глобальная переменная *OwnOrder*, которая заполняется в результате вызова *PlaceOrder* (сама функция — без изменений).

```
ulong OwnOrder = 0;

void OnTimer()
{
   // однократно выполняем код для текущих параметров
   EventKillTimer();

   const string symbol = StringLen(Symbol) == 0 ? _Symbol : Symbol;
   OwnOrder = PlaceOrder((ENUM_ORDER_TYPE)Type, symbol, Volume,
      Distance2SLTP, Expiration, Until, Magic, Comment);
}
```

Для удаления написана простая функция *RemoveOrder*, создающая объект *request* и вызывающая для него последовательно методы *remove* и *completed*.

```
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));
}
```

В следующем фрагменте журнала показаны записи, появившиеся в результате размещения эксперта на графике EURUSD, после чего символ был переключен на XAUUSD, а затем эксперт удален.

```
(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
```

Мы рассмотрим другой пример удаления ордеров для реализации стратегии "One Cancel Other" (OCO) в разделе о событиях *OnTrade*.

6.4.22 Получение списка действующих ордеров

Программам-экспертам часто требуется выполнять перебор имеющихся активных ордеров и анализировать их свойства. В частности, в разделе о модификации отложенного ордера, в примере *PendingOrderModify.mq5* была написана специальная функция *GetMyOrder* для нахождения "собственных" ордеров эксперта, подлежащих периодическому исправлению. Там анализ проводился по имени символа и идентификатору эксперта (*Magic*-номеру). В принципе, такой же подход должен был бы применяться и в примере удаления отложенного ордера *PendingOrderDelete.mq5* из предыдущего раздела.

В последнем случае мы для простоты создавали ордер и запоминали его тикет в глобальной переменной, однако в общем случае так поступать нельзя, потому что эксперт, да и терминал целиком, может быть остановлен или перезапущен в любой момент. Поэтому в эксперте должен присутствовать алгоритм восстановления внутреннего состояния, включающий анализ всего торгового окружения, вместе с ордерами, сделками, позициями, балансом счета и так далее.

В данном разделе мы изучим функции MQL5 для получения перечня действующих ордеров и выделения любого из них в торговом окружении, что дает возможность читать все его свойства.

int OrdersTotal()

Функция *OrdersTotal* возвращает количество активных в данный момент ордеров. В их число входят отложенные ордера, а также рыночные, которые еще не исполнены. Как правило, рыночный ордер исполняется оперативно и потому застать его в активной фазе можно не часто, но на тонком рынке, при отсутствии ликвидности, такое может случиться. Как только ордер исполняется (по нему заключается сделка), он переводится из разряда действующих в историю. О работе с историей ордеров мы поговорим в отдельном разделе.

> Обратите внимание, что только ордера бывают активными и историческими. Это существенно отличает ордера от сделок, которые всегда создаются уже по факту, то есть в истории, и от позиций, которые существуют только онлайн. Для восстановления истории позиций следует анализировать историю сделок.

ulong OrderGetTicket(uint index)

Функция *OrderGetTicket* возвращает тикет ордера по его номеру в списке ордеров в торговом окружении терминала. Параметр *index* должен быть в пределах от 0 до значения *OrdersTotal()-1* включительно. Порядок следования ордеров не регламентирован.

Функция *OrderGetTicket* выбирает ордер, то есть копирует данные о нем в некий внутренний кеш, так что MQL-программа может читать все его свойства с помощью последующих вызовов функций *OrderGetDouble*, *OrderGetInteger*, *OrderGetString*, о которых речь пойдет в отдельном разделе.

Наличие такого кеша означает, что полученные из него данные могут рано или поздно устареть: самого ордера может уже не быть или в нем могут измениться какие-то свойства (например, статус, цена открытия, уровни *Stop Loss* или *Take Profit*, срок истечения). Поэтому для гарантированного получения свежих данных об ордере рекомендуется вызывать функцию *OrderGetTicket* непосредственно перед обращением за ними. Напомним, как это происходило в примере *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;
}
```

Для каждой MQL-программы поддерживается свой собственный кеш (контекст торговой среды), куда входит и выбранный ордер. В следующих разделах мы узнаем, что помимо ордеров MQL-программа может выбирать в активный контекст позиции и фрагменты истории со сделками и ордерами.

Аналогичное выделение ордера с копированием его данных во внутренний кеш выполняет и функция *OrderSelect*.

bool OrderSelect(ulong ticket)

Функция проверяет наличие ордера и подготавливает возможность дальнейшего чтения его свойств. В данном случае ордер задается не по порядковому номеру, а по тикету, который должен быть так или иначе получен MQL-программой ранее, в частности, в результате выполнения *OrderSend/OrderSendAsync*.

Функция возвращает *true* в случае успеха. Значение *false*, как правило, означает, что ордера с указанным тикетом не существует. Наиболее частая причина для этого — переход ордера из разряда действующих в исторические, например, в результате исполнения или отмены (точный статус мы научимся определять позднее). Для выделения ордеров в истории существуют другие функции.

Ранее мы использовали функцию *OrderSelect* в структуре *MqlTradeResultSync* для отслеживания создания и удаления отложенного ордера.

6.4.23 Свойства ордеров (действующих и в истории)

Как мы видели в разделах о торговых операциях, в частности о совершении покупки/продажи, закрытии позиции или установке отложенного ордера, отправка запросов на сервер фактически основана на заполнении специфических полей структуры *MqlTradeRequest*, большинство из которых напрямую определяет свойства создаваемых в результате ордеров. MQL5 API позволяет узнать эти и некоторые другие свойства, устанавливаемые самой торговой системой, такие как тикет, время регистрации и статус.

Важно отметить, что перечень свойств ордеров является общим как для действующих, так и исторических ордеров, хотя значения многих свойств у них, разумеется, будут отличаться.

Свойства ордеров сгруппированы в MQL5 по уже знакомому нам принципу, базирующемся на типе значений: целочисленные (совместимые с *long/ulong*), вещественные (*double*) и строковые. Каждая группа свойств имеет собственное перечисление.

Целочисленные свойства сведены в ENUM_ORDER_PROPERTY_INTEGER и представлены в следующей таблице.

Идентификатор	Описание	Тип
ORDER_TYPE	Тип ордера	ENUM_ORDER_TYPE
ORDER_TYPE_FILLING	Тип исполнения по объему	ENUM_ORDER_TYPE_FILLING
ORDER_TYPE_TIME	Время жизни ордера (отложенного)	ENUM_ORDER_TYPE_TIME
ORDER_TIME_EXPIRATION	Время истечения ордера (отложенного)	datetime
ORDER_MAGIC	Произвольный идентификатор, заданный экспертом, выставившим ордер	ulong
ORDER_TICKET	Тикет ордера — уникальное число, которое присваивается сервером каждому ордеру	ulong
ORDER_STATE	Статус ордера	ENUM_ORDER_STATE (см.ниже)
ORDER_REASON	Причина или источник выставления ордера	ENUM_ORDER_REASON (см.ниже)
ORDER_TIME_SETUP	Время постановки ордера	datetime
ORDER_TIME_DONE	Время исполнения или снятия ордера	datetime
ORDER_TIME_SETUP_MSC	Время установки ордера на исполнение в миллисекундах	ulong
ORDER_TIME_DONE_MSC	Время исполнения/снятия ордера в миллисекундах	ulong
ORDER_POSITION_ID	Идентификатор позиции, которую породил или изменил ордер, если он исполнен	ulong
ORDER_POSITION_BY_ID	Идентификатор встречной позиции для ордеров типа ORDER_TYPE_CLOSE_BY	ulong

Каждый исполненный ордер порождает сделку, которая открывает новую или изменяет уже существующую позицию. Идентификатор этой позиции проставляется исполненному ордеру в свойстве ORDER_POSITION_ID.

Перечисление ENUM_ORDER_STATE содержит элементы, описывающие состояния ордера. Далее для наглядности приведена упрощенная схема (диаграмма состояний) ордеров.

Идентификатор	Описание
ORDER_STATE_STARTED	Ордер проверен на корректность, но еще не принят сервером
ORDER_STATE_PLACED	Ордер принят сервером
ORDER_STATE_CANCELED	Ордер отменен клиентом (пользователем или MQL-программой)
ORDER_STATE_PARTIAL	Ордер выполнен частично
ORDER_STATE_FILLED	Ордер выполнен полностью
ORDER_STATE_REJECTED	Ордер отклонен сервером
ORDER_STATE_EXPIRED	Ордер снят по истечении срока его действия
ORDER_STATE_REQUEST_ADD	Ордер в состоянии регистрации (выставление в торговую систему)
ORDER_STATE_REQUEST_MODIFY	Ордер в состоянии модификации (изменение его параметров)
ORDER_STATE_REQUEST_CANCEL	Ордер в состоянии удаления (удаление из торговой системы)

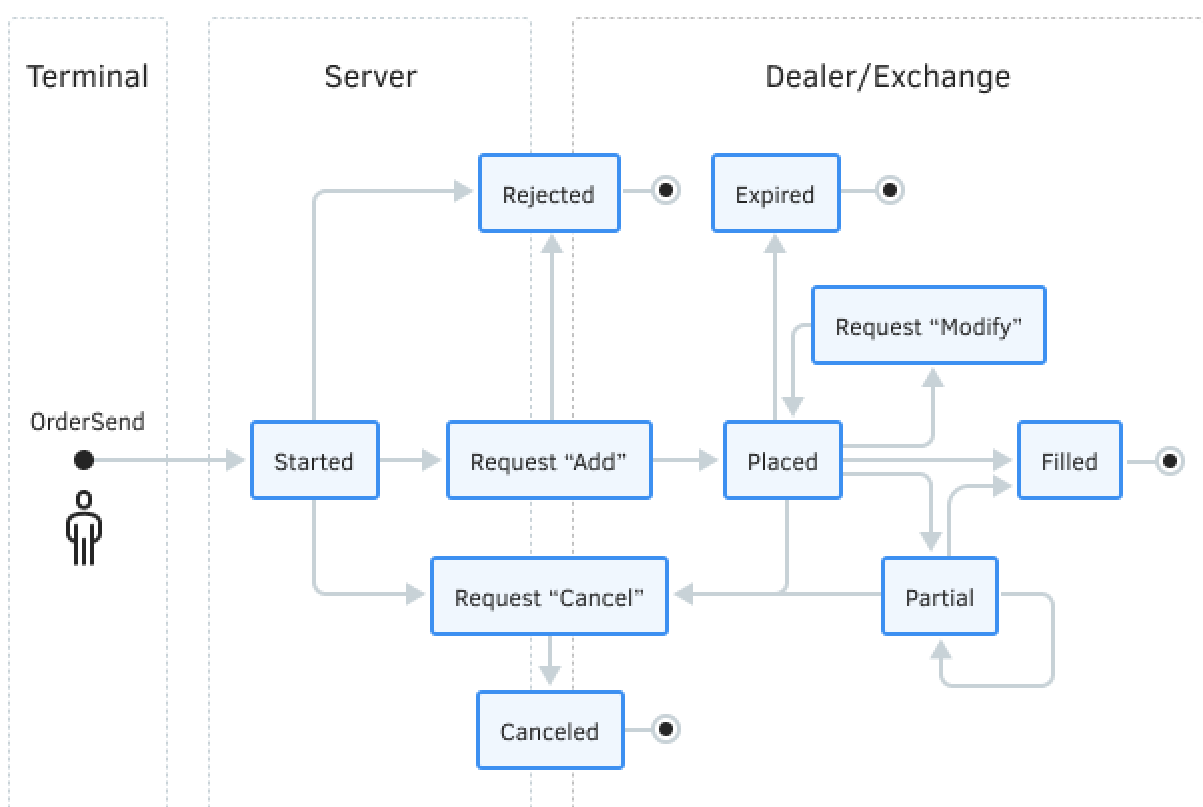


Диаграмма состояний ордеров

Следует понять, что изменение состояния возможно только у действующих ордеров. У ордеров в истории (исполненных или отмененных) состояние зафиксировано.

Вы можете отменить ордер, который уже частично исполнился, и тогда его статус в истории будет ORDER_STATE_CANCELED.

ORDER_STATE_PARTIAL бывает только у действующих ордеров. Исполненные (исторические) ордера всегда имеют статус ORDER_STATE_FILLED.

В перечислении ENUM_ORDER_REASON указаны возможные варианты происхождения ордера.

Идентификатор	**Описание**
ORDER_REASON_CLIENT	Ордер выставлен из десктопного терминала вручную
ORDER_REASON_EXPERT	Ордер выставлен из десктопного терминала советником или скриптом
ORDER_REASON_MOBILE	Ордер выставлен из мобильного приложения
ORDER_REASON_WEB	Ордер выставлен из web-терминала (браузера)
ORDER_REASON_SL	Ордер выставлен сервером в результате срабатывания Stop Loss
ORDER_REASON_TP	Ордер выставлен сервером в результате срабатывания Take Profit
ORDER_REASON_SO	Ордер выставлен сервером в результате наступления события Stop Out

Вещественные свойства собраны в перечислении ENUM_ORDER_PROPERTY_DOUBLE.

Идентификатор	**Описание**
ORDER_VOLUME_INITIAL	Первоначальный объем при постановке ордера
ORDER_VOLUME_CURRENT	Текущий объем (начальный или оставшийся после частичного исполнения)
ORDER_PRICE_OPEN	Цена, указанная в ордере
ORDER_PRICE_CURRENT	Текущая цена по символу еще неисполненного ордера или цена исполнения
ORDER_SL	Уровень Stop Loss
ORDER_TP	Уровень Take Profit
ORDER_PRICE_STOPLIMIT	Цена постановки Limit-ордера при срабатывании StopLimit-ордера

Свойство ORDER_PRICE_CURRENT содержит для действующих отложенных ордеров текущую цену *Ask* для покупки или цену *Bid* для продажи. Под "текущей" понимается цена, известная в торговом окружении на момент выбора ордера с помощью *OrderSelect* или *OrderGetTicket*. Для исполненных ордеров в истории данное свойство содержит цену исполнения — она может отличаться от заданной в ордере из-за проскальзывания.

Свойства ORDER_VOLUME_INITIAL и ORDER_VOLUME_CURRENT не равны друг другу, только если статус ордера — ORDER_STATE_PARTIAL.

Если ордер исполнялся частями, то в истории его свойство ORDER_VOLUME_INITIAL будет равно размеру последней исполненной части, а все остальные "заливки", относящиеся к исходному полному объему, будут оформлены отдельными ордерами (и сделками).

Строковые свойства описаны в перечислении ENUM_ORDER_PROPERTY_STRING.

Идентификатор	Описание
ORDER_SYMBOL	Символ, по которому выставлен ордер
ORDER_COMMENT	Комментарий
ORDER_EXTERNAL_ID	Идентификатор ордера во внешней торговой системе (на бирже)

Для чтения всех вышеупомянутых свойств существует два разных набора функций: для активных ордеров и для исторических. Сначала рассмотрим функции для активных ордеров, а к историческим вернемся после того, как познакомимся с принципами выделения требуемого фрагмента истории.

6.4.24 Функции для чтения свойств действующих ордеров

Наборы функций, с помощью которых можно получить значения всех свойств ордера, отличаются для активных ордеров и ордеров из истории. В данном разделе описаны функции для чтения свойств активных ордеров. Функции для доступа к свойствам ордеров в истории смотрите в соответствующем разделе.

Целочисленные свойства можно прочитать с помощью функции *OrderGetInteger*, имеющей 2 формы: первая возвращает непосредственно значение свойства, вторая — логический признак успеха (*true*) или ошибки (*false*), а значением свойства заполняется переданный по ссылке второй параметр.

long OrderGetInteger(ENUM_ORDER_PROPERTY_INTEGER property)
bool OrderGetInteger(ENUM_ORDER_PROPERTY_INTEGER property, long &value)

Обе функции позволяют получить запрошенное свойство ордера совместимого с целым типа (*datetime*, *long/ulong* или перечисление). Хотя в прототипе упоминается *long*, с технической точки зрения значение хранится как 8-байтовая ячейка, которую можно приводить к совместимым типам без какой-либо конвертации внутреннего представления, в частности, к *ulong*, который используется для всех тикетов.

Аналогичная пара функций предназначена для свойств вещественного типа *double*.

double OrderGetDouble(ENUM_ORDER_PROPERTY_DOUBLE property)
bool OrderGetDouble(ENUM_ORDER_PROPERTY_DOUBLE property, double &value)

Наконец, строковые свойства доступны посредством пары функций *OrderGetString*.

string OrderGetString(ENUM_ORDER_PROPERTY_STRING property)
bool OrderGetString(ENUM_ORDER_PROPERTY_STRING property, string &value)

Все функции принимают в качестве первого параметра идентификатор интересующего нас свойства — это должен быть элемент одного из перечислений ENUM_ORDER_PROPERTY_INTEGER, ENUM_ORDER_PROPERTY_DOUBLE, ENUM_ORDER_PROPERTY_STRING, рассмотренных в предыдущем разделе.

Напомним, что перед вызовом любой из предыдущих функций необходимо предварительно выделить ордер с помощью *OrderSelect* или *OrderGetTicket*.

Для чтения всех свойств конкретного ордера разработаем класс *OrderMonitor* (*OrderMonitor.mqh*), действующий по такому же принципу, как уже знакомые нам мониторы символов (*SymbolMonitor.mqh*) или торгового счета (*AccountMonitor.mqh*).

Напомним, что эти и другие классы-мониторы, рассмотренные в книге, предлагают унифицированный способ анализа свойств за счет перегруженных версий виртуальных *get*-методов.

Несколько забегая вперед, скажем, что сделки и позиции имеют такую же группировку свойств по трём основным типам значений, и для них нам также потребуется реализовать мониторы. В связи с этим имеет смысл выделить общий алгоритм в базовый абстрактный класс *MonitorInterface* (*TradeBaseMonitor.mqh*). Он является шаблоном, три параметра которого предназначены для указания типов конкретных перечислений: для целочисленных (I), вещественных (D) и строковых (S) групп свойств.

```
#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;
   }
   ...
```

Из-за того, что ордер (или сделка, или позиция) могут быть не найдены в торговом окружении по разным причинам, в классе зарезервирована переменная *ready*, в которую производные классы должны будут записать признак успешной инициализации, то есть выбора объекта для чтения его свойств.

Несколько чисто виртуальных методов декларируют доступ к свойствам соответствующих типов.

```
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;
...
```

В первых трех методах тип свойства задан одним из шаблонных параметров, а во вторых трех — вторым параметром самого метода: это требуется, потому что последние методы принимают первым параметром не константы конкретного перечисления, а просто целое число. С одной стороны это удобно для сквозной нумерации идентификаторов (константы перечислений трех типов не пересекаются). Но с другой стороны нам нужен другой источник определения типа значения — напомним, что тип возвращаемый функцией/методом не участвует в процессе выбора подходящей перегрузки.

Такой подход позволяет получать свойства, руководствуясь различной исходной информацией, доступной в вызывающем коде. Далее мы создадим классы на основе *OrderMonitor* (а также будущих *DealMonitor* и *PositionMonitor*) для отбора объектов по набору произвольных условий, и там все эти методы будут востребованы.

Довольно часто в программах требуется получить строковое представление любых свойств, например, для вывода в журнал. В новых мониторах это поручено методам *stringify*. Очевидно, что они получают значения запрашиваемых свойств через вызовы вышеупомянутых методов *get*.

```
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);
}
...
```

Единственный из этих методов, который не получил реализации, это первый вариант *stringify* для типа *long*. Это связано с тем, что в группе целочисленных свойств, как мы видели в предыдущем разделе, перемешаны свойства фактически разных прикладных типов — дата и время, перечисления, целые числа. Поэтому их преобразование в понятные строки смогут предоставить только производные классы. Данная ситуация является общей для всех торговых сущностей — не только ордеров, но и сделок, и позиций, свойства которых мы рассмотрим позднее.

Когда в целочисленном свойстве содержится элемент некоего перечисления (например, ENUM_ORDER_TYPE, ORDER_TYPE_FILLING и т.д.), для его преобразования в строку следует

применять функцию *EnumToString*. Эту задачу решает вспомогательный метод *enumstr* — скоро мы увидим его широкое использования в классах конкретных мониторов, начиная с *OrderMonitor* через пару абзацев.

```
template<typename E>
static string enumstr(const long v)
{
   return EnumToString((E)v);
}
```

Для вывода в журнал всех свойств конкретного типа создан метод *list2log*, использующий в цикле *stringify*.

```
template<typename E>
void list2log() const
{
   E e = (E)0; // подавляем предупреждение '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));
   }
}
```

Наконец, для упрощения вывода в журнал свойств всех трех групп предоставлен метод *print*, вызывающий *list2log* трижды — для каждой группы свойств.

```
virtual void print() const
{
   if(!ready) return;

   Print(typename(this));
   list2log<I>();
   list2log<D>();
   list2log<S>();
}
```

Имея в своем распоряжении базовый шаблонный класс *MonitorInterface*, опишем *OrderMonitorInterface*, где укажем конкретные типы перечислений для ордеров из предыдущего раздела и предоставим реализацию *stringify* для целочисленных свойств ордеров.

```
class OrderMonitorInterface:
   public MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,
   ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PROPERTY_STRING>
{
public:
   // описание свойств согласно подтипам
   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;
   }
};
```

Макрос STR_TIME_MSC для отображения времени с учетом миллисекунд определен следующим образом:

```
#define STR_TIME_MSC(T) (TimeToString((T) / 1000, TIME_DATE | TIME_SECONDS) \
    + StringFormat("'%03d", (T) % 1000))
```

Теперь мы готовы описать окончательный класс для чтения свойств любого ордера — *OrderMonitor*, наследник *OrderMonitorInterface*. В конструктор передается тикет ордера, и тот выбирается в торговом окружении с помощью *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;
      }
   }
   ...
```

Главная рабочая часть монитора состоит из переопределений виртуальных функций для чтения свойств. Здесь мы видим обращение к функциям *OrderGetInteger*, *OrderGetDouble*, *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);
   }
};
```

Данный фрагмент кода приведен с некоторыми сокращениями: из него убраны операторы для работы с ордерами в истории. После того как мы познакомимся с этим аспектом в последующих разделах, *OrderMonitor* будет показан в полном виде.

Важно отметить, что объект монитора не хранит в себе копии свойств. Поэтому доступ к методам *get* должен осуществляться сразу после создания объекта и, соответственно, вызова *OrderSelect*. Для чтения свойств в более поздний период потребуется вновь выделить ордер во внутреннем кеше MQL-программы, например, вызвав метод *refresh*.

```
   void refresh()
   {
      ready = OrderSelect(ticket);
   }
```

Протестируем работу *OrderMonitor*, добавив его в эксперт *MarketOrderSend.mq5*. Новая версия с именем *MarketOrderSendMonitor.mq5* подключает файл *OrderMonitor.mqh* директивой *#include*, а в теле функции *OnTimer* (в блоке успешного подтверждения открытия позиции по ордеру) создает объект монитора и вызывает его метод *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();
         ...
      }
   }
}
```

В журнале мы должны увидеть новые строки, содержащие все свойства ордера.

```
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
```

В четвертой строке начинается вывод из метода *print*, который включает полное название объекта монитора *MonitorInterface* вместе с типами-параметрами (в данном случае, тройка ENUM_ORDER_PROPERTY) и далее все свойства конкретного ордера.

Однако печать свойств — не самое интересное действие, которое может обеспечить монитор. Гораздо более востребованной в экспертах является задача отбора ордеров по условиям (значениям произвольных свойств). Используя монитор как вспомогательный инструмент, создадим механизм фильтрации ордеров, по аналогии с тем, что мы делали для символов: *SymbolFilter.mqh*.

6.4.25 Отбор ордеров по свойствам

В одном из разделов о свойствах символов мы представили класс *SymbolFilter* для отбора финансовых инструментов с заданными характеристиками. Теперь мы применим такой же подход для ордеров.

Поскольку нам предстоит похожим образом анализировать не только ордера, но также сделки и позиции, выделим общую часть алгоритма фильтрации в базовый класс *TradeFilter* (*TradeFilter.mqh*). Он практически один в один повторяет исходный код *SymbolFilter*. Поэтому мы не станем его здесь пояснять еще раз.

Желающие могут выполнить контекстное сравнение файлов *SymbolFilter.mqh* и *TradeFilter.mqh*, чтобы убедиться, насколько они похожи, и локализовать незначительные правки.

Основное отличие заключается в том, что класс *TradeFilter* является шаблоном, так как ему предстоит иметь дело со свойствами разных объектов: ордеров, сделок и позиций.

```
enum IS // поддерживаемые условия сравнения в фильтрах
{
   EQUAL,
   GREATER,
   NOT_EQUAL,
   LESS
};

enum ENUM_ANY // фиктивное перечисление для приведения к нему всех перечислений
{
};

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:
   // методы добавления условий в фильтр
   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);
   // методы получения в массивы подходящих по фильтру записей
   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;
   ...
}
```

Параметры шаблона I, D и S — это перечисления для групп свойств трех основных типов (целочисленные, вещественные, строковые): для ордеров они были описаны в предыдущих разделах, поэтому для наглядности вы можете представлять, что I=ENUM_ORDER_PROPERTY_INTEGER, D=ENUM_ORDER_PROPERTY_DOUBLE, S=ENUM_ORDER_PROPERTY_STRING.

Тип T предназначен для указания класса монитора. В данный момент у нас готов только один монитор — *OrderMonitor*. Позднее мы реализуем *DealMonitor* и *PositionMonitor*.

Ранее, в классе *SymbolFilter* мы обходились без параметров шаблона, потому что для символов неизменно известны все типы перечислений свойств и существует единственный класс *SymbolMonitor*.

Напомним структуру класса-фильтра. Группа методов *let* позволяет зарегистрировать в фильтре сочетание пар "свойство=значение", по которым будет затем осуществляться отбор объектов в методах *select*. Идентификатор свойства указывается в параметре *property*, а значение — в параметре *value*.

Самих методов *select* также несколько. Они позволяют заполнить для вызывающего кода массив с отобранными тикетами, а также при необходимости, дополнительные массивы со значениями запрошенных свойств объектов. Конкретные идентификаторы запрашиваемых свойств задаются в первом параметре метода *select* — это может быть одно свойство или несколько. В зависимости от этого приемный массив должен быть одномерным или двумерным.

Сочетание свойства и значения может проверяться не только на равенство (EQUAL), но и операции больше/меньше (GREATER/LESS). Для строковых свойств допустимо указывать шаблон поиска с символом "*", обозначающим любую последовательность знаков (например, "*[tp]*" для свойства ORDER_COMMENT совпадет со всеми комментариями, в которых с любом месте встретится "[tp]", хотя это только демонстрация возможности и для поиска ордеров в результате сработавшего *Take Profit* следует анализировать ORDER_REASON).

Поскольку алгоритм требует организации цикла с перебором всех объектов, а объекты могут быть разного типа (пока это ордера, но затем появится поддержка сделок и позиций), в классе *TradeFilter* потребовалось описать два абстрактных метода: *total* и *get*:

```
virtual int total() const = 0;
virtual ulong get(const int i) const = 0;
```

Первый возвращает количество объектов, а второй — тикет объекта по его номеру. Это должно напомнить вам пару функций *OrdersTotal* и *OrderGetTicket*. И действительно, они применены в конкретных реализациях методов для фильтра ордеров.

Ниже показан класс *OrderFilter* (*OrderFilter.mqh*) целиком.

```
#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);
   }
};
```

Эта простота особенно важна, учитывая, что аналогичные фильтры будут созданы для сделок и позиций без усилий.

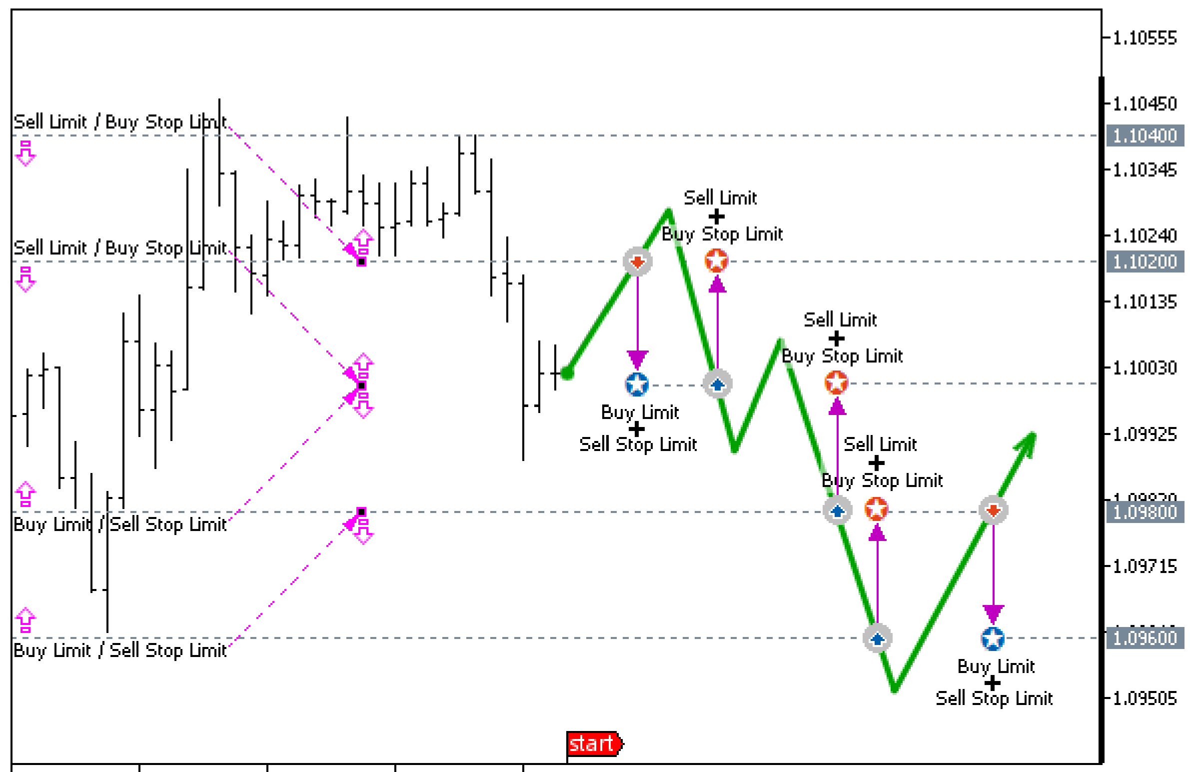
С помощью нового класса мы можем гораздо проще проверять наличие ордеров, принадлежащих нашему эксперту, то есть заменить любые самописные варианты функции *GetMyOrder*, использовавшейся в примере *PendingOrderModify.mq5*.

```
   OrderFilter filter;
   ulong tickets[];

   // задаем условие на ордера по текущему символу и нашему "магику"
   filter.let(ORDER_SYMBOL, _Symbol).let(ORDER_MAGIC, Magic);
   // отбираем подходящие тикеты в массив
   if(filter.select(tickets))
   {
      ArrayPrint(tickets);
   }
```

Под "любыми вариантами" здесь имеется в виду, что благодаря классу фильтра мы можем составлять произвольные условия для отбора ордеров и менять их "на ходу" (например, по указанию пользователя, а не программиста).

В качестве примера применения фильтра возьмем эксперт, создающий сетку отложенных ордеров для торговли на отбой от уровней внутри некоторого диапазона цен, то есть рассчитанный на флуктуирующий рынок. Начиная с этого раздела и в течение нескольких следующих, мы будем видоизменять эксперт в контексте изучаемого материала.

Первая версия эксперта *PendingOrderGrid1.mq5* строит сетку заданного размера из лимитных и стоп-лимитных ордеров. Параметрами будет количество ценовых уровней и шаг в пунктах между ними. Принцип действия иллюстрируется следующей схемой.



Сетка отложенных ордеров на 4 уровня с шагом 200 пунктов

В некий начальный момент времени, который может определяться внутрисуточным расписанием и соответствовать, например, "ночному флету", текущая цена округляется до размера шага сетки, и от этого уровня вверх вниз откладывается заданное количество уровней.

На каждом верхнем уровне ставится лимитный ордер на продажу и *stoplimit*-ордер на покупку с ценой будущего лимитного ордера на один уровень ниже. На каждом нижнем уровне ставится лимитный ордер по покупку и *stoplimit*-ордер на продажу с ценой будущего лимитного ордера на один уровень выше.

Когда цена касается одного из уровней, стоящий там лимитный ордер превращается в покупку или продажу (позицию). Одновременно с этим стоп-лимитный ордер того же уровня автоматически преобразуется системой в лимитный ордер противоположного направления на соседнем уровне.

Например, если цена при движении вверх пробила уровень, мы получим короткую позицию, а на дистанции шага ниже неё создастся лимитный ордер на покупку.

Эксперт будет отслеживать, чтобы на каждом уровне в паре с лимитным ордером существовал стоп-лимитный. Поэтому после обнаружения нового лимитного ордера на покупку программа добавит к нему на тот же уровень стоп-лимитный на продажу, причем целевой ценой будущего лимитного ордера назначается соседний сверху уровень, то есть тот, где открыта позиция.

Допустим, цена развернулась вниз и активировала лимитный ордер на уровень ниже — мы получим длинную позицию. Одновременно с этим стоп-лимитный ордер преобразуется в лимитный ордер на продажу на соседнем уровне сверху. Теперь эксперт снова обнаружит "голый" лимитный ордер и создаст ему в пару на том же уровне стоп-лимитный ордер на покупку с ценой будущего лимитного ордера на уровень ниже.

При наличии встречных позиций будем закрывать их. Также предусмотрим настройку внутрисуточного периода, когда торговая система включена, а на остальное время все ордера и позиции будут удаляться. Это, в частности, пригодится для "ночного флета", когда особенно выражены возвратные колебания рынка.

Разумеется, это лишь одна из множества потенциально возможных реализаций сеточной стратегии, и в ней отсутствуют многие настройки, присущие сеткам, но мы не станем усложнять пример.

Анализ ситуации эксперт будет проводить на каждом баре (предположительно, таймфрейма H1 или меньше). В принципе, такую логику работы советника можно и нужно усовершенствовать за счет оперативного реагирования на торговые события, но мы их еще не изучили. Поэтому вместо постоянного отслеживания и моментального "ручного" восстановления лимитных ордеров на вакантных уровнях сетки мы поручили эту работу серверу за счет применения стоп-лимитных ордеров. Однако здесь есть нюанс.

Дело в том, что лимитные и стоп-лимитные ордера, стоящие на каждом уровне относятся к противоположным типам (*buy/sell*), а потому активируются разными типами цен.

Получается, что если рынок двигался вверх к очередному уровню в верхней половине сетки, *Ask*-цена может задеть уровень и активирует стоп-лимитный ордер на покупку, но *Bid*-цена не дойдет до уровня, и лимитный ордер на продажу останется как есть (не превратится в позицию). В нижней половине сетки — при движении рынка вниз — ситуация зеркальная. Любого уровня первой касается *Bid*-цена и активирует стоп-лимитный ордер на продажу, и только при дальнейшем снижении до уровня также доходит *Ask*-цена. Если движения не произойдет, лимитный ордер на покупку останется как есть.

Данная проблема становится критичной с увеличением спреда. Поэтому в эксперте потребуется дополнительный контроль за "лишними" лимитными ордерами. Иными словами, эксперт не станет генерировать отсутствующий на уровне стоп-лимитный ордер, если на его предполагаемой целевой цене (соседний уровень) уже стоит лимитный ордер.

Исходный код прилагается в файле *PendingOrderGrid1.mq5*. Во входных параметрах можно задать объем каждой сделки *Volume* (по умолчанию, если оставить его равным 0, берется минимальный лот символа графика), количество уровней сетки *GridSize* (должно быть четным) и шаг между уровнями в пунктах *GridStep*. Начальное и конечное время внутрисуточного отрезка, на котором разрешена работа стратегии, указывается в параметрах *StartTime* и *StopTime*: в обоих важно только время.

```
#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 c
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;
```

Отрезок рабочего времени может быть как внутри суток (*StartTime* < *StopTime*), так и пересекать границу суток (*StartTime* > *StopTime*), например, с 22:00 до 09:00. Если два времени равны, предполагается круглосуточная торговля.

Прежде чем приступать к реализации торговой идеи упростим себе задачу по настройке запросов и выводу диагностической информации в журнал. Для этого опишем собственную структуру *MqlTradeRequestSyncLog*, производную от *MqlTradeRequestSync*.

```
const ulong DAYLONG = 60 * 60 * 24; // размер суток в секундах

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));
   }
};
```

В конструкторе мы заполняем все поля с неизменными значениями. В деструкторе выводим в журнал значащие поля запроса и результата. Очевидно, что деструктор автоматических объектов будет вызываться всегда в момент выхода из блока кода, где производилось формирование и отправка приказа, то есть в печать попадут отправленные и полученные данные.

В *OnInit* выполним некоторые проверки на корректность входных переменных, в частности, на четный размер сетки.

```
int OnInit()
{
   if(GridSize < 2 || !!(GridSize % 2))
   {
      Alert("GridSize should be 2, 4, 6+ (even number)");
      return INIT_FAILED;
   }
   return INIT_SUCCEEDED;
}
```

Основной точкой входа алгоритма является обработчик *OnTick*. В нем мы опустим для краткости тот же механизм обработки ошибок на базе TRADE_RETCODE_SEVERITY, что и в примере *PendingOrderModify.mq5*.

Для побаровой работы в функции заведена статическая переменная *lastBar*, в которую мы сохраняем время последнего успешно обработанного бара. Все последующие тики на том же баре пропускаются.

```
void OnTick()
{
   static datetime lastBar = 0;
   if(iTime(_Symbol, _Period, 0) == lastBar) return;
   uint retcode = 0;

   ... // основной алгоритм (см. далее)

   const TRADE_RETCODE_SEVERITY severity = TradeCodeSeverity(retcode);
   if(severity < SEVERITY_RETRY)
   {
      lastBar = iTime(_Symbol, _Period, 0);
   }
}
```

Вместо многоточия последует основной алгоритм, разделенный на несколько вспомогательных функций в целях систематизации. Первым делом определим, задан ли рабочий отрезок суток и если да, то включена ли в данный момент стратегия. Этот признак хранится в переменной *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;
   }
}
...
```

При разрешенной торговле сначала проверим, есть ли уже сеть ордеров, с помощью функции *CheckGrid*. Если сети нет, функция вернет константу GRID_EMPTY и нам следует создать сеть с помощью вызова *SetupGrid*. Если сеть уже построена, имеет смысл проверить, нет ли встречных позиций для закрытия: этим занимается функция *CompactPositions*.

```
if(tradeScheduled)
{
   retcode = CheckGrid();

   if(retcode == GRID_EMPTY)
   {
      retcode = SetupGrid();
   }
   else
   {
      retcode = CompactPositions();
   }
}
...
```

Как только торговый период заканчивается, необходимо удалить ордера и закрыть все позиции (если есть) — это поручено, соответственно, функции *RemoveOrders* и всё той же *CompactPositions*, но с логическим флагом (*true*): этот единственный, необязательный аргумент предписывает после встречного закрытия применить простое закрытие для оставшихся позиций.

```
else
{
   retcode = CompactPositions(true);
   if(!retcode) retcode = RemoveOrders();
}
```

Все функции возвращают код сервера, который анализируется на успех или ошибку с помощью *TradeCodeSeverity*. Специальные прикладные коды GRID_EMPTY и GRID_OK также расцениваются штатными согласно TRADE_RETCODE_SEVERITY.

```
#define GRID_OK    +1
#define GRID_EMPTY  0
```

Теперь разберем функции по отдельности.

В функции *CheckGrid* как раз используется класс *OrderFilter*, представленный в начале этого раздела. С помощью фильтра запрашиваются все отложенные ордера по текущему символу и с "нашим" идентификационным номером, и в массиве сохраняются тикеты найденных ордеров.

```
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;
   ...
```

Для анализа полноты сетки применяется уже знакомый класс MapArray, хранящий пары "ключ=значение". В данном случае в качестве ключа выступает уровень (цена, переведенная в пункты), а в качестве значения — битовая маска (суперпозиция) типов ордеров на данном уровне. Попутно в переменных *limits* и *stops* подсчитываются количества, соответственно, лимитных и стоп-лимитных ордеров.

```
   // ценовые уровни => маски типов существующих там ордеров
   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) / poir
         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));
         }
      }
   }
   ...
```

Если количество ордеров каждого типа совпадает и равно заданному размеру сетки, значит всё в порядке.

```
   if(limits == stops)
   {
      if(limits == GridSize) return GRID_OK; // полная сетка

      Alert("Error: Order number does not match requested");
      return TRADE_RETCODE_ERROR;
   }
   ...
```

Ситуация, когда количество лимитных ордеров больше стоп-лимитных является штатной: она означает, что за счет движения цены один или несколько стоп-лимитных ордеров превратились в лимитные. Программа должна в таком случае добавить стоп-лимитные ордера на уровни, где их не хватает. Отдельный ордер конкретного типа для конкретного уровня умеет выставлять функция *RepairGridLevel*.

```
      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) // на этом уровне нет стоп-лимитного ордера
            {
               // направление лимитного требуется для установки обратного стоп-лимитного
               const bool buyLimit = (levels[i] & (1 << ORDER_TYPE_BUY_LIMIT));
               // здесь опущены проверки "лишних" ордеров из-за спреда (см. исходный код
               ...
               // создаем стоп-лимитный ордер нужного направления
               const uint retcode = RepairGridLevel(levels.getKey(i), point, buyLimit);
               if(TradeCodeSeverity(retcode) > SEVERITY_NORMAL)
               {
                  return retcode;
               }
            }
         }
         return GRID_OK;
      }
      ...
```

Ситуация, когда количество стоп-лимитных ордеров больше, чем лимитных, трактуется как ошибка (вероятно, сервер по какой-то причине пропустил цену).

```
      Alert("Error: Orphaned Stop-Limit orders found");
      return TRADE_RETCODE_ERROR;
   }
```

Функция *RepairGridLevel* выполняет следующие действия.

```
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";

   // если существует непарный buy-limit, устанавливаем к нему sell-stop-limit
   // если существует непарный sell-limit, устанавливаем к нему buy-stop-limit
   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;
}
```

Обратите внимание, что нам не требуется фактически заполнять структуру (кроме комментария, который можно сделать более информативным при необходимости), поскольку часть полей заполняется автоматически конструктором, а объем и цену мы передаем непосредственно в метод *sellStopLimit* или *buyStopLimit*.

Похожий подход используется и в функции *SetupGrid*, устанавливающий новую полную сеть ордеров. В начале функции подготавливаем переменные для расчетов и описываем массив структур *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;
   // центральная цена диапазона с округлением до шага,
   // от неё вверх и вниз отложим уровни
   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]; // лимитный и стоп-лимитный - одна пара
   ArrayResize(request, GridSize);      // 2 отложенных ордера на каждый уровень
```

Далее генерируем ордера для нижней и верхней половины сетки, расходясь от центра в стороны.

```
   for(int i = 0; i < (int)GridSize / 2; ++i)
   {
      const int k = i + 1;
      
      // нижняя половина сетки
      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;
      }
      
      // верхняя половина сетки
      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;
      }
   }
```

Затем проверяем готовность.

```
    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;
 }
```

Хотя проверка (вызов *completed*) разнесена с отправкой приказов, наша структура по прежнему использует внутри синхронную форму *OrderSend*. На самом деле для ускорения отправки пакета приказов (как в нашем сеточном эксперте) лучше использовать асинхронную версию *OrderSendAsync*. Но тогда статус исполнения ордеров следует инициировать из обработчика события *OnTradeTransaction*. А его мы изучим позднее.

Ошибка при отправке любого приказа приводит к досрочному выходу из цикла и возврату кода с сервера. Данный тестовый эксперт просто прекратит свою дальнейшую работу в случае ошибки. Для реального робота желательно предусмотреть интеллектуальный анализ сути ошибки и, при необходимости, удалить все ордера и закрыть позиции.

Закрытием позиций, которые будут порождаться отложенными ордерами, занимается функция *CompactPositions*.

```
 uint CompactPositions(const bool cleanup = false)
```

Параметр *cleanup*, равный по умолчанию *false*, означает штатную "подчистку" позиций внутри торгового периода, то есть закрытие встречных позиций (если они есть). Значение *cleanup=true* используется для принудительного закрытия всех позиций в конце торгового периода.

Функция заполняет массивы *ticketsLong* и *ticketsShort* тикетами длинных и коротких позиций с помощью вспомогательной функции *GetMyPositions*. Мы уже использовали последнюю в примере *TradeCloseBy.mq5* в разделе про полное и частичное закрытие встречных позиций. Там же была показана и функция *CloseByPosition*. В новом эксперте она претерпела минимальные изменения: возвращает код с сервера вместо логического признака успеха или ошибки.

```
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;
      }
   }
   ...
```

Вторая часть *CompactPositions* работает только при *cleanup=true*. Она далека от совершенства и будет скоро переделана.

```
   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;
}
```

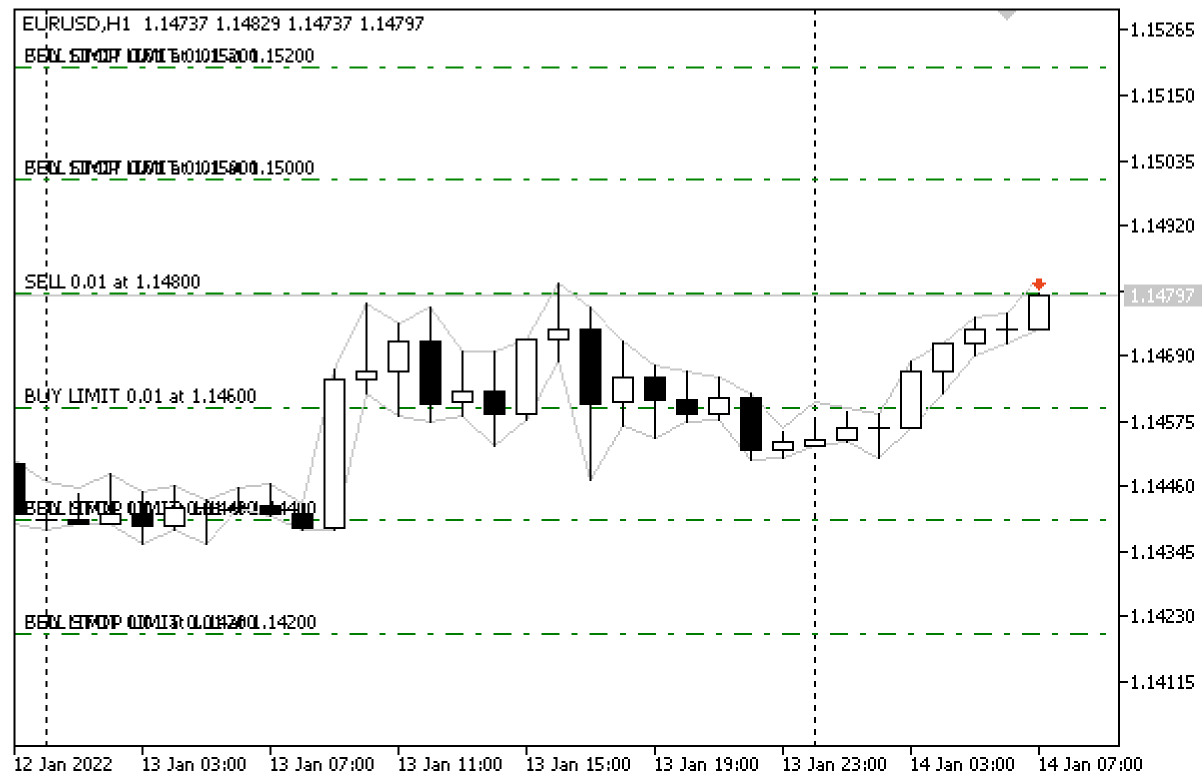
Для всех найденных оставшихся позиций выполняется обычное закрытие вызовом *CloseAllPositions*.

```
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
}
```

Нам осталось рассмотреть функцию *RemoveOrders*. Здесь также используется фильтр ордеров для получения их списка, и затем вызов метода *remove* в цикле.

```
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;
}
```

Проверим, как эксперт работает в тестере с настройками по умолчанию (торговый период с 00:00 до 09:00). Ниже представлен скриншот для запуска на EURUSD,H1.



Сеточная стратегия PendingOrderGrid1.mq5 в тестере

В журнале помимо периодических записей о пакетном создании нескольких ордеров (в начале суток) и их удалении под утро, мы будем регулярно видеть восстановление сети (добавление ордеров вместо сработавших) и закрытие позиций.

```
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
```

Теперь настало время изучить функции MQL5 для работы с позициями и усовершенствовать их выборку и анализ в нашем эксперте. Этому посвящены следующие разделы.

6.4.26 Получение списка позиций

Во многих примерах экспертов нам уже приходилось использовать функции MQL5 API, предназначенные для анализа открытых торговых позиций. В данном разделе представлено их формальное описание.

Важно отметить, что функции данной группы не способны управлять позициями: создавать, модифицировать, удалять. Как мы видели ранее, все такие действия выполняются опосредованно через отправку ордеров. При их успешном исполнении совершаются сделки, в результате которых и формируются позиции.

Другой особенностью является то, что функции применимы только к онлайн позициям. Для восстановления истории позиций необходимо анализировать историю сделок.

Узнать общее количество открытых позиций на счете (по всем финансовым инструментам) позволяет функция *PositionsTotal*.

int PositionsTotal()

При "неттинговом" учете позиций (ACCOUNT_MARGIN_MODE_RETAIL_NETTING и ACCOUNT_MARGIN_MODE_EXCHANGE) по каждому символу в любой момент может быть только одна позиция, которая является результатом одной или более сделок.

При независимом представлении позиций (ACCOUNT_MARGIN_MODE_RETAIL_HEDGING) по каждому символу одновременно может быть открыто несколько позиций, в том числе разнонаправленных. Каждая сделка входа в рынок формирует отдельную позицию, поэтому частичное поэтапное исполнение одного ордера способно породить несколько позиций.

Выяснить символ позиции по её номеру позволяет функция *PositionGetSymbol*.

string PositionGetSymbol(int index)

Индекс должен быть в пределах от 0 до N-1, где N — значение, полученное предварительным вызовом *PositionsTotal*. Порядок следования позиций не регламентирован.

Если позиция не найдена, то вернется пустая строка, а код ошибки будет доступен в *_LastError*.

Примеры использования этих двух функций встречались в нескольких тестовых экспертах (*TrailingStop.mq5*, *TradeCloseBy.mq5* и других) в функциях с говорящими названиями *GetMyPosition/GetMyPositions*.

Открытая позиция характеризуется уникальным тикетом — номером, который отличает её от других позиций, но может меняться в течение её жизни в некоторых случаях, таких как переворот позиции в режиме неттинга одной сделкой или в результате служебных операций на сервере (переоткрытие для начисления свопов, клиринга).

Для получения тикета позиции по её номеру предназначена функция *PositionGetTicket*.

ulong PositionGetTicket(int index)

Дополнительно функция выделяет позицию в торговом окружении терминала, что позволяет затем читать её свойства с помощью группы специальных *PositionGet*-функций. Иными словами, по аналогии с ордерами, терминал поддерживает для каждой MQL-программ внутренний кеш для хранения свойств одной позиции. Для выделения позиции, помимо *PositionGetTicket*, существует две функции: *PositionSelect* и *PositionSelectByTicket* — они рассмотрены ниже.

В случае ошибки функция *PositionGetTicket* вернет 0.

> Тикет следует отличать от другого свойства — идентификатора, который присваивается каждой позиции и никогда не меняется. Именно идентификаторы используются для увязки позиций с ордерами и сделками, но об этом мы поговорим чуть позже.
>
> Тикеты же нужны для выполнения запросов с участием позиций: именно тикеты задаются в

полях *position* и *position_by* структуры *MqlTradeRequest*. Кроме того, сохранив тикет в переменной, программа сможет впоследствии выделить конкретную позицию с помощью функции *PositionSelectByTicket* (см. ниже) и работать с ней, не прибегая к повторному перебору позиций в цикле.

При перевороте позиции на неттинговом счете POSITION_TICKET изменяется на тикет ордера, инициировавшего эту операцию. Однако такую позицию можно по-прежнему отследить с помощью идентификатора. В режиме хеджинга переворот позиции не поддерживается.

bool PositionSelect(const string symbol)

Функция выбирает открытую позицию по имени финансового инструмента.

При независимом представлении позиций (ACCOUNT_MARGIN_MODE_RETAIL_HEDGING) по каждому символу одновременно может быть несколько открытых позиций. В этом случае, *PositionSelect* выберет позицию с наименьшим тикетом.

Возвращаемый результат сигнализирует успешное (*true*) или неудачное (*false*) выполнение функции.

Тот факт, что свойства выделенной позиции кешируются означает, что самой позиции может уже не быть или она может быть изменена, если программа считывает её свойства через некоторое время. Рекомендуется вызывать функцию *PositionSelect* непосредственно перед обращением к данным.

bool PositionSelectByTicket(ulong ticket)

Функция выбирает открытую позицию для дальнейшей работы по указанному тикету.

Примеры использования функций мы рассмотрим позднее, в процессе изучения свойств и связанных с ними *PositionGet*-функций.

При построении алгоритмов с использованием функций *PositionsTotal*, *OrdersTotal* и аналогичных следует учитывать асинхронные принципы работы терминала. Мы уже касались этой темы в ходе написания классов *MqlTradeSync.mqh* и реализовали ожидание результатов выполнения исходящих торговых запросов. Однако это ожидание не всегда возможно выполнить на клиентской стороне. В частности, если мы установим отложенный ордер, то его превращение в рыночный и последующее исполнение будет происходить на сервере. В этот момент ордер может перестать числиться среди активных (*OrdersTotal* вернет 0), но позиция еще не отобразится (*PositionsTotal* также равен 0). Поэтому MQL-программа, имеющая условие на постановку ордера в отсутствии позиции, может ошибочно инициировать новый ордер, в результате чего позиция, в конечном счете, задвоится.

Чтобы решить эту проблему, MQL-программа должна анализировать торговое окружение более глубоко, нежели просто одномоментно проверять количество ордеров и позиций. Например, можно хранить слепок последнего корректного состояния торгового окружения и не допускать, чтобы какие-либо сущности исчезали без того или иного подтверждения. Только после этого можно формировать новый слепок. Так, ордер может удалиться только совместно с изменением позиции (создание, закрытие) или переводом в историю со статусом отмены. Одно из возможных решений предложено в виде класса *TradeGuard* в файле *TradeGuard.mqh*. Демонстрационный скрипт — *TradeGuardExample.mq5* — также поставляется вместе с книгой. Данный пример оставлен для самостоятельного изучения.

6.4.27 Свойства позиций

Все свойства позиций делятся на 3 группы по типу значений: целочисленные и совместимые с ними, вещественные числа и строки. Для их чтения используются *PositionGet*-функции, аналогичные *OrderGet*-функциям. Сами функции мы опишем в следующем разделе, а здесь приведем идентификаторы всех свойств, которые доступны для указания в первом параметре этих функций.

Целочисленные свойства объединены в перечисление ENUM_POSITION_PROPERTY_INTEGER.

Идентификатор	Описание	Тип
POSITION_TICKET	Тикет позиции	ulong
POSITION_TIME	Время открытия позиции	datetime
POSITION_TIME_MSC	Время открытия позиции в миллисекундах	ulong
POSITION_TIME_UPDATE	Время изменения позиции (объема)	datetime
POSITION_TIME_UPDATE_MSC	Время изменения позиции (объема) в миллисекундах	ulong
POSITION_TYPE	Тип позиции	ENUM_POSITION_TYPE
POSITION_MAGIC	Magic-число для позиции (на основе ORDER_MAGIC)	ulong
POSITION_IDENTIFIER	Идентификатор позиции — уникальное число, которое присваивается каждой вновь открытой позиции и не изменяется в течение всей ее жизни.	ulong
POSITION_REASON	Причина открытия позиции	ENUM_POSITION_REASON

Как правило, POSITION_IDENTIFIER соответствует тикету ордера, которым была открыта позиция. Идентификатор позиции указывается в каждом ордере (ORDER_POSITION_ID) и сделке (DEAL_POSITION_ID), которая ее открыла, изменила или закрыла. Поэтому его удобно использовать для поиска ордеров и сделок, связанных с позицией.

Если ордер заливается частично, то может одновременно существовать и позиция, и действующий отложенный ордер на оставшийся объем с совпадающими тикетами. Более того, такую позицию можно успеть закрыть, а при следующей заливке остатков отложенного ордера снова появится позиция с тем же тикетом.

В режиме неттинга переворот позиции одной сделкой считается изменением позиции, а не созданием новой, поэтому идентификатор POSITION_IDENTIFIER сохраняется. Новая позиция по символу возможна только после закрытия предыдущей в нулевой объем.

Свойство POSITION_TIME_UPDATE отвечает только на изменение объема (например, в результате частичного закрытия или "доливки"), но не других параметров вроде уровней *Stop Loss/Take Profit* или начисления свопов.

Типов позиций всего два (ENUM_POSITION_TYPE).

Идентификатор	Описание
POSITION_TYPE_BUY	Покупка
POSITION_TYPE_SELL	Продажа

Варианты происхождения позиции, то есть способы её открытия, раскрывает перечисление ENUM_POSITION_REASON.

Идентификатор	Описание
POSITION_REASON_CLIENT	Срабатывание ордера, выставленного из десктопного терминала
POSITION_REASON_MOBILE	Срабатывание ордера, выставленного из мобильного приложения
POSITION_REASON_WEB	Срабатывание ордера, выставленного из веб-платформы (браузера)
POSITION_REASON_EXPERT	Срабатывание ордера, выставленного советником или скриптом

Вещественные свойства собраны в ENUM_POSITION_PROPERTY_DOUBLE.

Идентификатор	Описание
POSITION_VOLUME	Объем позиции
POSITION_PRICE_OPEN	Цена позиции
POSITION_SL	Цена Stop Loss
POSITION_TP	Цена Take Profit
POSITION_PRICE_CURRENT	Текущая цена по символу
POSITION_SWAP	Накопленный своп
POSITION_PROFIT	Текущая прибыль

Тип текущей цены соответствует операции закрытия позиции. Например, длинная позиция должна закрываться продажей, и потому для неё в POSITION_PRICE_CURRENT отслеживается цена *Bid*.

Наконец, для позиций поддерживаются следующие строковые свойства (ENUM_POSITION_PROPERTY_STRING).

Идентификатор	Описание
POSITION_SYMBOL	Символ, по которому открыта позиция
POSITION_COMMENT	Комментарий к позиции
POSITION_EXTERNAL_ID	Идентификатор позиции во внешней системе (на бирже)

После знакомства с перечнем свойств позиций мы готовы рассмотреть функции для чтения этих свойств.

6.4.28 Функции для чтения свойств позиций

Получить свойства позиции MQL-программа может с помощью нескольких *PositionGet*-функций, зависящих от типа свойств. Во всех функциях конкретное запрашиваемое свойство определяется в первом параметре, принимающем идентификатор одного из ENUM_POSITION_PROPERTY-перечислений, рассмотренных в предыдущем разделе.

Для каждого типа свойств имеется краткая и полная форма функции: первая возвращает значение свойства напрямую, вторая — записывает его во второй параметр, передаваемый по ссылке.

Целочисленные свойства и свойства совместимых с ними типов (*datetime*, перечисления) можно получить функцией *PositionGetInteger*.

```
long PositionGetInteger(ENUM_POSITION_PROPERTY_INTEGER property)
bool PositionGetInteger(ENUM_POSITION_PROPERTY_INTEGER property, long &value)
```

В случае неудачного выполнения функция возвращает 0 или *false*.

Для получения вещественных свойств предназначена функция *PositionGetDouble*.

```
double PositionGetDouble(ENUM_POSITION_PROPERTY_DOUBLE property)
bool PositionGetDouble(ENUM_POSITION_PROPERTY_DOUBLE property, double &value)
```

Наконец, строковые свойства возвращает функция *PositionGetString*.

```
string PositionGetString(ENUM_POSITION_PROPERTY_STRING property)
bool PositionGetString(ENUM_POSITION_PROPERTY_STRING property, string &value)
```

В случае неудачного выполнения первая форма функции возвращает пустую строку.

Для чтения свойств позиций у нас уже готов абстрактный интерфейс *MonitorInterface* (*TradeBaseMonitor.mqh*) — мы его использовали для написания монитора ордеров. Теперь будет легко реализовать аналог и для позиций. Результат прилагается в файле *PositionMonitor.mqh*.

Класс *PositionMonitorInterface* наследуется от *MonitorInterface* с назначением шаблонным типам I, D, S рассмотренных ENUM_POSITION_PROPERTY-перечислений и переопределяет пару методов *stringify* с учетом специфики свойств позиций.

```
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);
   }
```

Конкретный класс монитора, полностью готовый к просмотру позиций, является следующим в цепочке наследования и основан на *PositionGet*-функциях. Выделение позиции по тикету производится в конструкторе.

```
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);
   }
   ...
};
```

Простой скрипт позволит вывести в журнал все характеристики первой позиции (если хотя бы одна имеется).

```
void OnStart()
{
   PositionMonitor pm(PositionGetTicket(0));
   pm.print();
}
```

В журнале мы должны получить что-то вроде этого.

```
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=
```

Если открытых позиций в данный момент нет, увидим сообщение об ошибке.

```
Error: PositionSelectByTicket(0) failed: TRADE_POSITION_NOT_FOUND
```

Однако монитор полезен не только и не столько выводом свойств в журнал. На основе *PositionMonitor* создадим класс для выборки позиций по условиям, аналогичный тому, что мы сделали для ордеров (*OrderFilter*). Конечной целью является усовершенствование нашего эксперта-сеточника.

Благодаря ООП, создание класса нового фильтра практически не требует усилий. Ниже приведен исходный код целиком (файл *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);
   }
};
```

Теперь мы можем написать, например, такой скрипт для получения удельной прибыли по позициям с заданным магическим числом.

```
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]);
   }
}
```

В данном случае нам пришлось вызвать метод *select* дважды, потому что типы интересующих нас свойств отличаются: вещественные прибыль и лот, но строковое название инструмента. В одном из разделов в начале главы, когда мы разрабатывали класс-фильтр для символов, была описана концепция кортежей. В MQL5 мы можем реализовать её в виде шаблонов структур с полями произвольных типов. Подобные кортежи пришлись бы очень кстати для доработки иерархии классов-фильтров, так как тогда можно было бы описать метод *select*, заполняющий массив кортежей с полями любых типов.

Сами кортежи описаны в файле *Tuples.mqh*. Все структуры в нем имеют название *TupleN<T1,...>*, где N — число от 2 до 8, и оно соответствует количеству параметров шаблона (типов Ti). Например, *Tuple2*:

```
template<typename T1,typename T2>
struct Tuple2
{
   T1 _1;
   T2 _2;

   static int size() { return 2; };

   // M - класс монитора ордеров, позиций, сделок, любой 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);
   }
};
```

В классе *TradeFilter* (*TradeFilter.mqh*) добавим вариант функции *select* с кортежами.

```
template<typename T,typename I,typename D,typename S>
class TradeFilter
{
   ...
   template<typename U> // тип U должен быть Tuple<>, например 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() не компилируется
      if(q != u.size()) return false;   // обязательное условие
      
      const int n = total();
      // цикл по ордерам/позициям/сделкам
      for(int i = 0; i < n; ++i)
      {
         const ulong t = get(i);
         // доступ к свойствам через монитор T
         T m(t);
         // проверяем все условия фильтра по разным типам свойств
         if(match(m, longs)
         && match(m, doubles)
         && match(m, strings))
         {
            // для подходящего объекта сохраняем свойства в массиве кортежей
            const int k = EXPAND(data);
            data[k].assign(property, m);
         }
      }
      
      if(sort)
      {
         sortTuple(data, u._1);
      }
      
      return true;
   }
```

Массив кортежей можно опционально сортировать по первому полю _1, но вспомогательный метод *sortTuple* оставлен для самостоятельного изучения.

Благодаря кортежам можно запросить у объекта-фильтра свойства трех разных типов за один вызов *select*.

Ниже позиции с некоторым *Magic*-номером выводятся упорядоченными по прибыли, для каждой дополнительно получается символ и тикет.

```
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);
   }
```

Разумеется, типы-параметры в описании массива кортежей (в данном случае, *Tuple3<double,string,ulong>*) должны соответствовать типам перечислений запрашиваемых свойств (POSITION_PROFIT, POSITION_SYMBOL, POSITION_TICKET).

Теперь мы можем слегка упростить сеточный эксперт (имеется в виду не просто более короткий, но и более понятный код). Новая версия имеет название *PendingOrderGrid2.mq5*. Изменения коснутся всех функций, связанных с управлением позициями.

Функция *GetMyPositions* заполняет переданный по ссылке массив кортежей *types4tickets*. В каждом кортеже *Tuple2* предполагается хранить тип и тикет позиции. В принципе, в данном конкретном случае мы могли бы обойтись и двумерным массивом *ulong* вместо кортежей, потому что оба свойства — одного базового типа. Однако мы используем кортежи для демонстрации работы с ними в вызывающем коде.

```
#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);
}
```

Обратите внимание, что последний, третий параметр метода *select* равен *true*, что предписывает отсортировать массив по первому полю, то есть типу позиций. Таким образом, покупки у нас будут в начале, а продажи — в конце. Это потребуется для встречного закрытия.

Реинкарнация метода *CompactPositions* выглядит следующим образом.

```
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) // пока типы отличаются
         {
            retcode = CloseByPosition(types4tickets[i]._2, types4tickets[j]._2);
            if(retcode) return retcode; // ошибка
         }
         else
         {
            break;
         }
      }
   }

   if(cleanup && j < n)
   {
      retcode = CloseAllPositions(types4tickets, i, j + 1);
   }

   return retcode;
}
```

Функция *CloseAllPositions* почти не изменилась.

```
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; // ошибка
      }
   }
   return 0; // успех
}
```

Вы можете сравнить работу экспертов *PendingOrderGrid1.mq5* и *PendingOrderGrid2.mq5* в тестере.

Отчеты будут слегка отличаться, потому что при наличии нескольких позиций они встречно закрываются в других сочетаниях, из-за чего закрытие прочих, непарных позиций происходит с учетом их индивидуальных спредов.

6.4.29 Свойства сделок

Сделка является отражением факта совершения торговой операции на основании ордера.

Один ордер может породить несколько сделок за счет исполнения по частям или встречного закрытия позиций.

Сделки характеризуются свойствами трех базовых типов: целочисленные (и совместимые с ними), вещественные, строковые. Каждое свойство описывается собственной константой в одном из перечислений: ENUM_DEAL_PROPERTY_INTEGER, ENUM_DEAL_PROPERTY_DOUBLE, ENUM_DEAL_PROPERTY_STRING.

Для чтения свойств сделок используются *HistoryDealGet*-функции. Все они предполагают, что необходимый участок истории был предварительно запрошен с помощью специальных функций выборки ордеров и сделок из истории.

Целочисленные свойства описаны в перечислении ENUM_DEAL_PROPERTY_INTEGER.

Идентификатор	Описание	Тип
DEAL_TICKET	Тикет сделки — уникальное число, которое присваивается каждой сделке	ulong
DEAL_ORDER	тикет ордера, на основании которого выполнена сделка	ulong
DEAL_TIME	Время совершения сделки	datetime
DEAL_TIME_MSC	Время совершения сделки в миллисекундах	ulong
DEAL_TYPE	Тип сделки	ENUM_DEAL_TYPE (см. ниже)
DEAL_ENTRY	Направление сделки — вход в рынок, выход из рынка или разворот	ENUM_DEAL_ENTRY (см. ниже)
DEAL_MAGIC	Magic-число для сделки (на базе ORDER_MAGIC)	ulong
DEAL_REASON	Причина или источник проведения сделки	ENUM_DEAL_REASON (см.ниже)
DEAL_POSITION_ID	Идентификатор позиции, в открытии, изменении или закрытии которой участвовала сделка	ulong

Возможные типы сделок представляет перечисление ENUM_DEAL_TYPE.

Идентификатор	Описание
DEAL_TYPE_BUY	Покупка
DEAL_TYPE_SELL	Продажа
DEAL_TYPE_BALANCE	Начисление баланса
DEAL_TYPE_CREDIT	Начисление кредита
DEAL_TYPE_CHARGE	Дополнительные сборы
DEAL_TYPE_CORRECTION	Корректирующая запись
DEAL_TYPE_BONUS	Перечисление бонусов
DEAL_TYPE_COMMISSION	Дополнительные комиссии
DEAL_TYPE_COMMISSION_DAILY	Комиссия, начисляемая в конце торгового дня
DEAL_TYPE_COMMISSION_MONTHLY	Комиссия, начисляемая в конце месяца

Идентификатор	Описание
DEAL_TYPE_COMMISSION_AGENT_DAILY	Агентская комиссия, начисляемая в конце торгового дня
DEAL_TYPE_COMMISSION_AGENT_MONTHLY	Агентская комиссия, начисляемая в конце месяца
DEAL_TYPE_INTEREST	Начисления процентов на свободные средства
DEAL_TYPE_BUY_CANCELED	Отмененная сделка покупки
DEAL_TYPE_SELL_CANCELED	Отмененная сделка продажи
DEAL_DIVIDEND	Начисление дивиденда
DEAL_DIVIDEND_FRANKED	Начисление франкированного дивиденда (освобожденного от уплаты налога)
DEAL_TAX	Начисление налога

Варианты DEAL_TYPE_BUY_CANCELED и DEAL_TYPE_SELL_CANCELED отражают ситуацию, когда ранее совершенная сделка отменяется. В таком случае тип ранее совершенной сделки (DEAL_TYPE_BUY или DEAL_TYPE_SELL) меняется на DEAL_TYPE_BUY_CANCELED или DEAL_TYPE_SELL_CANCELED, а ее прибыль/убыток обнуляется. Ранее полученная прибыль/убыток начисляется/списывается со счета отдельной балансовой операцией.

Сделки различаются по способу изменения позиции. Это может быть простое открытие позиции (вход в рынок) или наращивание объема ранее открытой позиции, закрытие позиции сделкой противоположного направления соответствующим объемом (выход из рынка) или переворот позиции в том случае, когда объем сделки в противоположном направлении перекрывает объем ранее открытой позиции. Последняя операция поддерживается только при неттинговом учете.

Все эти ситуации описаны элементами перечисления ENUM_DEAL_ENTRY.

Идентификатор	Описание
DEAL_ENTRY_IN	Вход в рынок
DEAL_ENTRY_OUT	Выход из рынка
DEAL_ENTRY_INOUT	Разворот
DEAL_ENTRY_OUT_BY	Закрытие встречной позицией

Причины проведения сделки сведены в перечисление ENUM_DEAL_REASON.

Идентификатор	Описание
DEAL_REASON_CLIENT	Срабатывание ордера, выставленного из десктопного терминала
DEAL_REASON_MOBILE	Срабатывание ордера, выставленного из мобильного приложения
DEAL_REASON_WEB	Срабатывание ордера, выставленного из веб-платформы
DEAL_REASON_EXPERT	Срабатывания ордера, выставленного советником или скриптом
DEAL_REASON_SL	Срабатывание ордера Stop Loss
DEAL_REASON_TP	Срабатывание ордера Take Profit
DEAL_REASON_SO	Наступление события Stop Out
DEAL_REASON_ROLLOVER	Перенос позиции между сутками
DEAL_REASON_VMARGIN	Начисление/списание вариационной маржи
DEAL_REASON_SPLIT	Сплит (понижение цены) инструмента, по которому имелась позиция

Свойства вещественного типа представлены перечислением ENUM_DEAL_PROPERTY_DOUBLE.

Идентификатор	Описание
DEAL_VOLUME	Объем сделки
DEAL_PRICE	Цена сделки
DEAL_COMMISSION	Комиссия по сделке
DEAL_SWAP	Накопленный своп при закрытии
DEAL_PROFIT	Финансовый результат сделки
DEAL_FEE	Оплата за проведение сделки, начисляется сразу после совершения сделки
DEAL_SL	Уровень Stop Loss
DEAL_TP	Уровень Take Profit

Два последних свойства заполняются по такому принципу. Для сделки входа или разворота берется значение *Stop Loss/Take Profit* из ордера, которым была открыта или развернута позиция. Для сделки выхода берется значение *Stop Loss/Take Profit* из позиции на момент её закрытия.

Строковые свойства сделок доступны по константам перечисления ENUM_DEAL_PROPERTY_STRING.

Идентификатор	Описание
DEAL_SYMBOL	Имя символа, по которому произведена сделка
DEAL_COMMENT	Комментарий к сделке
DEAL_EXTERNAL_ID	Идентификатор сделки во внешней торговой системе (на бирже)

Чтение свойств мы протестируем в разделе о *HistoryDealGet*-функциях, где будут представлены классы *DealMonitor* и *DealFilter*.

6.4.30 Выборка ордеров и сделок из истории

MetaTrader 5 позволяет создать для эксперта или скрипта слепок истории за конкретный период времени. Слепок представляет собой списки ордеров и сделок для дальнейшего обращения к ним посредством соответствующих функций. Кроме того история может запрашиваться в привязке к конкретным ордерам, сделкам или позициями.

Выделение нужного периода в явном виде (по датам) выполняет функция *HistorySelect*. После этого размер списка сделок можно узнать с помощью функции *HistoryDealsTotal*, а размер списка ордеров — с помощью *HistoryOrdersTotal*. Перебор элементов списка ордеров можно проводить функцией *HistoryOrderGetTicket*, для элементов списка сделок подходит функция *HistoryDealGetTicket*.

> Следует различать активные (действующие) ордера и ордера в истории — исполненные, отмененные или получившие отказ (rejected). Для анализа действующих ордеров используйте функции, рассмотренные в разделах о получении списка действующих ордеров и чтении их свойств.

bool HistorySelect(datetime from, datetime to)

Функция запрашивает историю сделок и ордеров за указанный период серверного времени (*from* и *to* — включительно, *to* >= *from*) и возвращает *true* в случае успеха.

Даже если в запрошенном периоде нет ордеров и сделок, функция вернет *true* в отсутствие ошибок. Ошибкой может быть, например, недостаток памяти для построения списка ордеров или сделок.

> Обратите внимание, что у ордеров есть два времени: установки (ORDER_TIME_SETUP) и исполнения (ORDER_TIME_DONE). Функция *HistorySelect* отбирает ордера по времени исполнения.

Для выделения всей истории счета можно использовать синтаксис *HistorySelect(0, LONG_MAX)*.

Другой способ выделения части истории — по идентификатору позиции.

bool HistorySelectByPosition(ulong positionID)

Функция запрашивает историю сделок и ордеров с указанным идентификатором позиции в свойствах ORDER_POSITION_ID, DEAL_POSITION_ID.

> Внимание! Функция не выбирает ордера по идентификатору встречной позиции, если использовалось встречное закрытие. Иными словами, свойство ORDER_POSITION_BY_ID

игнорируется, несмотря на то, что данные ордера участвовали в формировании позиции.

Например, эксперт мог совершить покупку (ордер #1) и продажу (ордер #2) на счету с поддержкой хеджирования. Тогда это приведет к образованию позиций #1 и #2. Для встречного закрытия позиций потребуется ордер ORDER_TYPE_CLOSE_BY (#3). В результате, вызов *HistorySelectByPosition(#1)* выделит ордера #1 и #3, что ожидаемо. Однако вызов *HistorySelectByPosition(#2)*, выделит только ордер #2 (несмотря на то, что в ордере #3 в свойстве ORDER_POSITION_BY_ID значится #2, и строго говоря, ордер #3 участвовал в закрытии позиции #2).

При успешном выполнении любой из двух функций *HistorySelect* или *HistorySelectByPosition* терминал формирует для MQL-программы некий внутренний список ордеров и сделок. Также изменить исторический контекст позволяют функции *HistoryOrderSelect* и *HistoryDealSelect*, для вызова которых необходимо заранее знать тикет соответствующего объекта (например, сохранить из результата запроса).

Важно отметить, что *HistoryOrderSelect* влияет только на список ордеров, а *HistoryDealSelect* — только на список сделок.

Все функции выделения контекста возвращают значение *bool*, которое обозначает успех (*true*) или ошибку (*false*). Код ошибки можно прочитать во встроенной переменной *_LastError*.

bool HistoryOrderSelect(ulong ticket)

Функция *HistoryOrderSelect* выбирает в истории ордер по его тикету для дальнейшей работы с ним (чтения свойств).

В ходе применения функции *HistoryOrderSelect*, если поиск ордера по тикету завершился успешно, новый список ордеров, выбранных в истории, будет состоять из единственного только что найденного ордера. Иными словами, прежний список выбранных ордеров (если он был) сбрасывается. Однако функция не сбрасывает выбранную до того историю сделок, то есть не выбирает сделку (сделки), связанную с ордером.

bool HistoryDealSelect(ulong ticket)

Функция *HistoryDealSelect* выбирает в истории сделку для дальнейших обращений к ней через соответствующие функции. Функция не сбрасывает историю ордеров, то есть не выбирает ордер, связанный с выбранной сделкой.

После того как в истории выбран некий контекст за счет вызова одной из вышеприведенных функций, MQL-программа может обращаться к функциям для перебора ордеров и сделок, попавших в этот контекст, и чтения их свойств.

int HistoryOrdersTotal()

Функция *HistoryOrdersTotal* возвращает количество ордеров в истории (в выборке).

ulong HistoryOrderGetTicket(int index)

Фукнция *HistoryOrderGetTicket* позволяет получить тикет ордера по его порядковому номеру в выбранном контексте истории. Индекс должен лежать в пределах от 0 до N-1, где N получено из функции *HistoryOrdersTotal*.

Имея "на руках" тикет ордера, далее легко получить все необходимые его свойства с помощью *HistoryOrderGet*-функций. Свойства исторических ордеров — точно такие же, как и у действующих.

Для работы со сделками существует аналогичная пара функций.

int HistoryDealsTotal()

Функция *HistoryDealsTotal* возвращает количество сделок в истории (в выборке).

ulong HistoryDealGetTicket(int index)

Функция *HistoryDealGetTicket* позволяет получить тикет сделки по её порядковому номеру в выбранном контексте истории. Это нужно для дальнейшей обработки сделки *HistoryDealGet-*функциями. Перечень доступных через них свойств сделок был описан в предыдущем разделе.

Пример применения функций мы рассмотрим после изучения *HistoryOrderGet-* и *HistoryDealGet-*функций.

6.4.31 Функции для чтения свойств ордеров из истории

Функции чтения свойств исторических ордеров разделены на 3 группы по базовому типу значений свойств, в соответствии с делением идентификаторов доступных свойств по трем перечислениям ENUM_ORDER_PROPERTY_INTEGER, ENUM_ORDER_PROPERTY_DOUBLE и ENUM_ORDER_PROPERTY_STRING, рассмотренных ранее в отдельном разделе при изучении действующих ордеров.

Перед вызовом этих функций нужно тем или иным способом выбрать соответствующий набор тикетов в истории.

Если попытаться читать свойства ордера или сделки, тикеты которых не попали в выбранный контекст истории, среда может сгенерировать ошибку WRONG_INTERNAL_PARAMETER (4002), доступную для анализа через *_LastError*.

Для каждого базового типа свойств существует 2 формы функции: одна непосредственно возвращает значение запрашиваемого свойства, вторая — записывает его в передаваемый по ссылке параметр, и возвращает признак успеха (*true*) или ошибки (*false*).

Для целочисленных и совместимых типов (*datetime*, перечисления) свойств выделена функция *HistoryOrderGetInteger*.

```
long HistoryOrderGetInteger(ulong ticket, ENUM_ORDER_PROPERTY_INTEGER property)
bool HistoryOrderGetInteger(ulong ticket, ENUM_ORDER_PROPERTY_INTEGER property,
  long &value)
```

Функция позволяет узнать свойство *property* ордера из выбранной истории по номеру его тикета.

Для вещественных свойств предназначена функция *HistoryOrderGetDouble*.

```
double HistoryOrderGetDouble(ulong ticket, ENUM_ORDER_PROPERTY_DOUBLE property)
bool HistoryOrderGetDouble(ulong ticket, ENUM_ORDER_PROPERTY_DOUBLE property,
  double &value)
```

Наконец строковые свойства можно прочитать с помощью *HistoryOrderGetString*.

```
string HistoryOrderGetString(ulong ticket, ENUM_ORDER_PROPERTY_STRING property)
bool HistoryOrderGetString(ulong ticket, ENUM_ORDER_PROPERTY_STRING property,
  string &value)
```

Эти новые знания дают возможность дополнить класс *OrderMonitor* (*OrderMonitor.mqh*) для работы с историческими ордерами. Прежде всего, добавим в класс логическую переменную *history*, которую будем заполнять в конструкторе на основании того, в каком сегменте удалось выделить ордер с переданным тикетом: среди действующих (*OrderSelect*) или в истории (*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;
      }
   }
   ...
```

Вызов функции *ResetLastError* в успешной ветке *if* нужен для сброса ошибки, которую могла взвести функция *OrderSelect* (если ордер в истории).

На самом деле, данный вариант конструктора содержит серьезную логическую ошибку, и мы вернемся к нему через несколько абзацев.

Для чтения свойств в get-методах мы теперь должны вызывать разные встроенные функции, в зависимости от значения переменной *history*.

```
   virtual long get(const ENUM_ORDER_PROPERTY_INTEGER property) const override
   {
      return history ? HistoryOrderGetInteger(ticket, property) : OrderGetInteger(pro
   }

   virtual double get(const ENUM_ORDER_PROPERTY_DOUBLE property) const override
   {
      return history ? HistoryOrderGetDouble(ticket, property) : OrderGetDouble(prope
   }

   virtual string get(const ENUM_ORDER_PROPERTY_STRING property) const override
   {
      return history ? HistoryOrderGetString(ticket, property) : OrderGetString(prope
   }
   ...
```

Основное предназначение класса *OrderMonitor* — снабжать данными другие аналитические классы. Напомним, что объекты *OrderMonitor* используются для фильтрации активных ордеров в классе *OrderFilter*, и нам потребуется аналогичный класс для выборки ордеров по произвольным условиям на истории — *HistoryOrderFilter*.

Напишем этот класс в том же файле — *OrderFilter.mqh*. Вполне логично в нем применены две новых функции для работы на истории: *HistoryOrdersTotal* и *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);
   }
};
```

Этот простой код наследуется от шаблонного класса *TradeFilter*, куда в качестве первого параметра шаблона передается класс *OrderMonitor* для чтения свойств соответствующих объектов (мы видели аналог для позиций, и скоро создадим для сделок).

Здесь и кроется проблема с конструктором *OrderMonitor*. Как мы узнали в разделе Выборка ордеров и сделок из истории, для анализа счета мы должны предварительно настроить контекст с помощью одной из функций, таких как *HistorySelect*. Поэтому здесь в исходном коде *HistoryOrderFilter* предполагается, что MQL-программа уже выделила нужный фрагмент истории. Однако в новом, промежуточном варианте конструктора *OrderMonitor* используется вызов *HistoryOrderSelect* для проверки существования тикета на истории. Между тем эта функция сбрасывает предыдущий контекст исторических ордеров и выбирает единственный ордер.

Поэтому нам требуется вспомогательный метод *historyOrderSelectWeak*, который мог бы проверить тикет "мягким" способом, не нарушая существующий контекст. Для этого достаточно проверить

равенство свойства ORDER_TICKET с самим переданным тикетом *t*: *(HistoryOrderGetInteger(t, ORDER_TICKET) == t)*. Если такой тикет уже отобран (доступен), проверка завершится успешно, и монитору не нужно манипулировать историей.

```
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;
      }
   }
```

Пример применения фильтрации ордеров на истории рассмотрим в следующем разделе, после подготовки аналогичного функционала для сделок.

6.4.32 Функции для чтения свойств сделок из истории

Для чтения свойств сделок предназначены группы функций, организованные по типу свойств: целочисленные, вещественные и строковые. Перед вызовом функций нужно выделить нужный фрагмент истории и тем самым обеспечить доступность сделок с тикетами, которые передаются в первом параметре (*ticket*) всех функций.

Для каждого типа свойств существует две формы: с непосредственным возвратом значения и путем записи в переменную по ссылке. Вторая форма возвращает *true* для индикации успеха. Первая форма просто вернет 0 в случае ошибки. Код ошибки — в переменной *_LastError*.

Целочисленные и совместимые типы свойств (*datetime*, перечисления) можно получить функцией *HistoryDealGetInteger*.

long HistoryDealGetInteger(ulong ticket, ENUM_DEAL_PROPERTY_INTEGER property)
bool HistoryDealGetInteger(ulong ticket, ENUM_DEAL_PROPERTY_INTEGER property,
  long &value)

Вещественные свойства читаются функцией *HistoryDealGetDouble*.

double HistoryDealGetDouble(ulong ticket, ENUM_DEAL_PROPERTY_DOUBLE property)
bool HistoryDealGetDouble(ulong ticket, ENUM_DEAL_PROPERTY_DOUBLE property,
  double &value)

Для строковых свойств предназначены функции *HistoryDealGetString*.

string HistoryDealGetString(ulong ticket, ENUM_DEAL_PROPERTY_STRING property)
bool HistoryDealGetString(ulong ticket, ENUM_DEAL_PROPERTY_STRING property,
  string &value)

Унифицированное чтение свойств сделки обеспечит класс *DealMonitor* (*DealMonitor.mqh*), организованный точно также, как *OrderMonitor* и *PositionMonitor*. В качестве базового класса выступает *DealMonitorInterface*, унаследованный от шаблона *MonitorInterface* (мы его описывали в разделе Функции для чтения свойств действующих ордеров). Именно на этом уровне задаются конкретные типы ENUM_DEAL_PROPERTY-перечислений в качестве параметров шаблона и специфическая реализация метода *stringify*.

```
#include <MQL5Book/TradeBaseMonitor.mqh>

class DealMonitorInterface:
   public MonitorInterface<ENUM_DEAL_PROPERTY_INTEGER,
   ENUM_DEAL_PROPERTY_DOUBLE,ENUM_DEAL_PROPERTY_STRING>
{
public:
   // описания свойств с учетом подтипов целого
   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;
   }
};
```

Нижеприведенный рабочий класс *DealMonitor* должен напомнить недавно модифицированный для работы с историей *OrderMonitor*. Помимо применения *HistoryDeal*-функций вместо *HistoryOrder*-функций следует отметить, что для сделок нет необходимости проверять тикет в онлайн-окружении, потому что сделки существуют только в истории.

```
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);
   }
   ...
};
```

На основе *DealMonitor* и *TradeFilter* легко создать фильтр сделок (*DealFilter.mqh*). Напомним, что *TradeFilter*, как базовый класс для многих сущностей, был описан в разделе Отбор ордеров по свойствам.

```
#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);
   }
};
```

В качестве обобщенного примера работы с историй рассмотрим скрипт восстановления истории позиций *TradeHistoryPrint.mq5*.

TradeHistoryPrint

Скрипт будет строить историю для текущего символа графика.

Нам для начала потребуются фильтры сделок и ордеров.

```
#include <MQL5Book/OrderFilter.mqh>
#include <MQL5Book/DealFilter.mqh>
```

Из сделок мы извлечем идентификаторы позиций и на их основе запросим подробности об ордерах.

Историю можно будет посмотреть целиком или для конкретной позиции, для чего во входных переменных предусмотрим выбор режима и поле ввода для идентификатора.

```
enum SELECTOR_TYPE
{
   TOTAL,    // Whole history
   POSITION, // Position ID
};

input SELECTOR_TYPE Type = TOTAL;
input ulong PositionID = 0; // Position ID
```

Следует помнить, что выборка длинной истории счета может быть накладной, поэтому в рабочих экспертах желательно предусмотреть кеширование полученных результатов обработки истории, вместе с последней временной меткой обработки. При каждом следующем анализе истории можно начинать процесс не с самого начала, а с запомненного момента.

Чтобы красиво, с выравниванием по колонкам выводить информацию о записях истории, имеет смысл представить её как массив структур. Однако наши фильтры уже поддерживают запрос данных с сохранением в особые структуры — кортежи. Поэтому мы применим хитрость: опишем свои прикладные структуры, соблюдая правила кортежей:

- первое поле должно иметь название _1 — оно опционально используется в алгоритме для сортировки;
- в структуре должна быть описана функция *size*, возвращающая количество полей в ней;
- в структуре должен быть шаблонный метод *assign* для заполнения полей из свойств переданного объекта-монитора, производного от *MonitorInterface*.

В стандартных кортежах метод *assign* описан так:

```
template<typename M>
void assign(const int &properties[], M &m);
```

Первым параметром он принимает массив с идентификаторами свойств, соответствующих полям, которые нас интересуют. Фактически этот тот массив, который передается вызывающим кодом в метод *select* фильтра (*TradeFilter::select*) и затем по ссылке попадает в *assign*. Но поскольку мы сейчас создадим не стандартные кортежи, а свои собственные структуры, "знающие" о прикладной сущности своих полей, мы можем оставить массив с идентификаторами свойств внутри самой структуры и не "гонять" его внутрь фильтра и обратно в метод *assign* той же структуры.

В частности, для запроса сделок опишем структуру *DealTuple* с 8-ю полями. Их идентификаторы укажем в статическом массиве *fields*.

```
struct DealTuple
{
   datetime _1;   // время сделки
   ulong deal;    // тикет сделки
   ulong order;   // тикет ордера
   string type;   // ENUM_DEAL_TYPE как строка
   string in_out; // ENUM_DEAL_ENTRY как строка
   double volume;
   double price;
   double profit;

   static int size() { return 8; }; // количество свойств
   static const int fields[]; // идентификаторы запрашиваемых свойств сделок
   ...
};

static const int DealTuple::fields[] =
{
   DEAL_TIME, DEAL_TICKET, DEAL_ORDER, DEAL_TYPE,
   DEAL_ENTRY, DEAL_VOLUME, DEAL_PRICE, DEAL_PROFIT
};
```

Данный подход сводит в едином месте идентификаторы и поля для хранения соответствующих значений, что облегчает понимание и сопровождение исходного кода.

Для заполнения полей значениями свойств потребуется слегка модифицированная (упрощенная) версия метода *assign* — она берет идентификаторы из массива *fields*, а не из входного параметра.

```
struct DealTuple
{
   ...
   template<typename M> // M производный от 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; // декларация типа по умолчанию (пустышка)
      
      _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);
   }
};
```

Заодно мы преобразуем числовые элементы перечислений ENUM_DEAL_TYPE и ENUM_DEAL_ENTRY в понятные для пользователя строки. Разумеется, это нужно только для вывода в журнал. Для программного анализа следовало бы оставить типы как есть.

Поскольку мы изобрели новый вариант метода *assign* в своих кортежах, требуется добавить для него новый вариант метода *select* в классе *TradeFilter*. Новшество наверняка будет полезно и для других программ, а потому внесем его непосредственно в *TradeFilter*, а не в некий новый производный класс.

```
template<typename T,typename I,typename D,typename S>
class TradeFilter
{
   ...
   template<typename U> // U должен иметь первое поле _1 и метод assign(T)
   bool select(U &data[], const bool sort = false) const
   {
      const int n = total();
      // цикл по элементам
      for(int i = 0; i < n; ++i)
      {
         const ulong t = get(i);
         // читаем свойства через объект-монитор
         T m(t);
         // проверяем все условия фильтрации
         if(match(m, longs)
         && match(m, doubles)
         && match(m, strings))
         {
            // для подходящего объекта складываем его свойства в массив
            const int k = EXPAND(data);
            data[k].assign(m);
         }
      }

      if(sort)
      {
         static const U u;
         sortTuple(data, u._1);
      }

      return true;
   }
```

Напомним, что все шаблонные методы не реализуются компилятором, пока к ним не будет обращение в коде с конкретным типом. Поэтому наличие подобных шаблонов в *TradeFilter* не обязывает вас подключать какие-либо заголовочные файлы с кортежами или описывать похожие структуры, если вы их не используете.

Итак, если раньше для выборки сделок с помощью стандартного кортежа нам нужно было бы писать так:

```
#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);
```

То с кастомизированной структурой все намного проще:

```
DealFilter filter;
DealTuple tuples[];
filter.let(DEAL_SYMBOL, _Symbol).select(tuples);
```

Аналогично структуре *DealTuple* опишем структуру для ордеров *OrderTuple* с 10-ю полями.

```
struct OrderTuple
{
   ulong _1;       // тикет (также используется как прототип 'ulong')
   datetime setup;
   datetime done;
   string type;
   double volume;
   double open;
   double current;
   double sl;
   double tp;
   string comment;

   static int size() { return 10; }; // количество свойств
   static const int fields[]; // идентификаторы запрашиваемых свойств ордеров

   template<typename M> // M производный от 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
};
```

Теперь все готово для реализации главной функции скрипта — *OnStart*. В самом начале опишем объекты фильтров сделок и ордеров.

```
void OnStart()
{
   DealFilter filter;
   HistoryOrderFilter subfilter;
   ...
```

В зависимости от входных переменных выберем либо всю историю, либо специфическую позицию.

```
if(PositionID == 0 || Type == TOTAL)
{
   HistorySelect(0, LONG_MAX);
}
else if(Type == POSITION)
{
   HistorySelectByPosition(PositionID);
}
...
```

Далее соберем в массив все идентификаторы позиций, или оставим одну заданную пользователем.

```
ulong positions[];
if(PositionID == 0)
{
   ulong tickets[];
   filter.let(DEAL_SYMBOL, _Symbol)
      .select(DEAL_POSITION_ID, tickets, positions, true); // true - сортировка
   ArrayUnique(positions);
}
else
{
   PUSH(positions, PositionID);
}

const int n = ArraySize(positions);
Print("Positions total: ", n);
if(n == 0) return;
...
```

Вспомогательная функция *ArrayUnique* оставляет в массиве неповторяющиеся элементы. Для её работы требуется, чтобы исходный массив был отсортирован.

Далее в цикле по позициям запрашиваем сделки и ордера, относящиеся к каждой из них. Сделки сортируются по первому полю структуры *DealTuple*, то есть по времени. Пожалуй, наиболее интересным является подсчет прибыли/убытка по позиции. Для этого мы суммируем значения поля *profit* всех сделок.

```
      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; // неверный идентификатор задан пользователем
         }
         double profit = 0; // TODO: нужно учесть комиссии, свопы и сборы
         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);
      }
   }
```

В данном коде не анализируются комиссии (DEAL_COMMISSION), свопы (DEAL_SWAP) и сборы (DEAL_FEE) в свойствах сделок. В реальных экспертах это, вероятно, следует делать (зависит от требований стратегии). Мы рассмотрим еще один пример анализа торговой истории в разделе, посвященном тестированию мультивалютных экспертов, и там учтем этот момент.

Вы можете сравнить результаты работы скрипта с таблицей на вкладке "История" в терминале: там в колонке "Прибыль" показывается чистая прибыль по каждой позиции (свопы, комиссии и сборы — в соседних столбцах, но их нужно включить).

Важно отметить, что ордер типа ORDER_TYPE_CLOSE_BY будет отображаться в обеих позициях только в том случае, если в настройках выбрана вся история. Если же была выбрана конкретная позиция, система включит такой ордер только в одну из них (ту, что была указана в торговом запросе первой, в поле *position*), но не во вторую (которая была указана в *position_by*).

Ниже показан пример результата работы скрипта для символа с небольшой историей.

```
Positions total: 3
Position:        1       1253500309 Profit:238.150000
                   [_1]     [deal]    [order] [type] [in_out] [volume]  [price]  [pro
[0] 2022.02.04 17:34:57 1236049891 1253500309 "BUY"  "IN"      1.00000 76.23900   0.0
[1] 2022.02.14 16:28:41 1242295527 1259788704 "SELL" "OUT"     1.00000 76.42100 238.1
Order details:
          [_1]             [setup]              [done] [type] [volume]   [open] [curr
   » [sl] [tp] [comment]
[0] 1253500309 2022.02.04 17:34:57 2022.02.04 17:34:57 "BUY"   1.00000 76.23900  76.2
   » 0.00 0.00 ""
[1] 1259788704 2022.02.14 16:28:41 2022.02.14 16:28:41 "SELL"  1.00000 76.42100  76.4
   » 0.00 0.00 ""
Position:        2       1253526613 Profit:878.030000
                   [_1]     [deal]    [order] [type] [in_out] [volume]  [price]  [pro
[0] 2022.02.07 10:00:00 1236611994 1253526613 "BUY"  "IN"      1.00000 75.75000   0.0
[1] 2022.02.14 16:28:40 1242295517 1259788693 "SELL" "OUT"     1.00000 76.42100 878.0
Order details:
          [_1]             [setup]              [done]      [type] [volume]   [open]
   » [sl] [tp] [comment]
[0] 1253526613 2022.02.04 17:55:18 2022.02.07 10:00:00 "BUY_LIMIT"  1.00000 75.75000
   » 0.00 0.00 ""
[1] 1259788693 2022.02.14 16:28:40 2022.02.14 16:28:40 "SELL"       1.00000 76.42100
   » 0.00 0.00 ""
Position:        3       1256280710 Profit:4449.040000
                   [_1]     [deal]    [order] [type] [in_out] [volume]  [price]   [pr
[0] 2022.02.09 13:17:52 1238797056 1256280710 "BUY"  "IN"      2.00000 74.72100    0.
[1] 2022.02.14 16:28:39 1242295509 1259788685 "SELL" "OUT"     2.00000 76.42100 4449.
Order details:
          [_1]             [setup]              [done] [type] [volume]   [open] [curr
   » [sl] [tp] [comment]
[0] 1256280710 2022.02.09 13:17:52 2022.02.09 13:17:52 "BUY"   2.00000 74.72100  74.7
   » 0.00 0.00 ""
[1] 1259788685 2022.02.14 16:28:39 2022.02.14 16:28:39 "SELL"  2.00000 76.42100  76.4
   » 0.00 0.00 ""
```

Случай с доливкой позиции (две сделки "IN") и её переворота (сделка "INOUT" большего объема) на неттинговом счете показан в следующем фрагменте.

```
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   1.1001
   » 0.00 0.00 ""
[1] 219087393 2022.03.29 08:04:05 2022.03.29 08:04:05 "BUY"   0.01000 0.0000   1.1000
   » 0.00 0.00 ""
[2] 219087400 2022.03.29 08:04:29 2022.03.29 08:04:29 "SELL"  0.03000 0.0000   1.1001
   » 0.00 0.00 ""
[3] 219087403 2022.03.29 08:04:34 2022.03.29 08:04:34 "BUY"   0.01000 0.0000   1.1001
   » 0.00 0.00 ""
```

Частичную историю на примере конкретных позиций рассмотрим для случая встречного закрытия на счете с хеджированием. Сперва можно просмотреть отдельно первую позицию: PositionID=1276109280. Она будет показана полностью вне зависимости от входного параметра *Type*.

```
Positions total: 1
Position:       1       1276109280 Profit:-0.040000
                   [_1]     [deal]    [order] [type] [in_out] [volume] [price] [profi
[0] 2022.03.07 12:20:53 1258725455 1276109280 "BUY"  "IN"      0.01000 1.08344  0.000
[1] 2022.03.07 12:20:58 1258725503 1276109328 "SELL" "OUT_BY"  0.01000 1.08340 -0.040
Order details:
          [_1]             [setup]              [done]     [type] [volume]  [open] [c
   » [sl] [tp]                    [comment]
[0] 1276109280 2022.03.07 12:20:53 2022.03.07 12:20:53 "BUY"       0.01000 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
   » 0.00 0.00 "#1276109280 by #1276109283"
```

Также можно посмотреть вторую: PositionID=1276109283. Однако если *Type* равен "*Position*", для выделения фрагмента истории используется функция *HistorySelectByPosition*, и в результате выходной ордер будет только один (несмотря на то, что сделок — две).

```
Positions total: 1
Position:       1       1276109283 Profit:0.000000
                   [_1]     [deal]    [order] [type] [in_out] [volume] [price] [profi
[0] 2022.03.07 12:20:53 1258725458 1276109283 "SELL" "IN"      0.01000 1.08340  0.000
[1] 2022.03.07 12:20:58 1258725504 1276109328 "BUY"  "OUT_BY"  0.01000 1.08344  0.000
Order details:
          [_1]             [setup]              [done] [type] [volume]  [open] [curre
   » [sl] [tp] [comment]
[0] 1276109283 2022.03.07 12:20:53 2022.03.07 12:20:53 "SELL"  0.01000 1.08340   1.08
   » 0.00 0.00 ""
```

Если поменять *Type* на "всю историю", ордер "CLOSE_BY" появится.

```
Positions total: 1
Position:        1       1276109283 Profit:0.000000
                   [_1]     [deal]    [order] [type] [in_out] [volume] [price] [profi
[0] 2022.03.07 12:20:53 1258725458 1276109283 "SELL" "IN"      0.01000 1.08340  0.000
[1] 2022.03.07 12:20:58 1258725504 1276109328 "BUY"  "OUT_BY"  0.01000 1.08344  0.000
Order details:
          [_1]             [setup]              [done]     [type] [volume]  [open] [c
   » [sl] [tp]                    [comment]
[0] 1276109283 2022.03.07 12:20:53 2022.03.07 12:20:53 "SELL"      0.01000 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
   » 0.00 0.00 "#1276109280 by #1276109283"
```

При таких настройках история выбирается полностью, но фильтр оставляет из неё только ордера, у которых в свойства ORDER_POSITION_ID или ORDER_POSITION_BY_ID встречается идентификатор заданной позиции. Для составления условий с логическим ИЛИ в классе *TradeFilter* добавлен элемент IS::OR_EQUAL — с ним предлагается разобраться самостоятельно.

6.4.33 Типы торговых транзакций

MQL-программы могут не только выполнять торговые операции, но и реагировать на торговые события. Важно отметить, что подобные события возникают не только в результате действия программ, но и по другим причинам, например, при ручном управлении пользователем или выполнении автоматических действий на сервере (срабатывание отложенного ордера, *Stop Loss*, *Take Profit*, *Stop Out*, перенос позиции через "ночь", начисление или списание средств со счета и многое другое).

Вне зависимости от инициатора действий, на счете выполняются торговые транзакции — неделимые шаги, включающие:

- обработку торгового запроса;
- изменение списка действующих ордеров (включая добавление нового ордера, исполнение и удаление сработавшего ордера);
- изменение истории ордеров;
- изменение истории сделок;
- изменение позиций.

В зависимости от сути операции, некоторые шаги могут быть опциональными. Например, модификация защитных уровней у позиции пропустит три средних пункта. А при отсылке приказа о покупке по рынку пройдет полный цикл: запрос обрабатывается, для счета создается соответствующий ордер, происходит исполнение ордера, его удаление из списка активных, добавление в историю ордеров, далее добавляется соответствующая сделка в историю и создается новая позиция. Все эти действия являются торговыми транзакциями.

Для получений уведомлений о таких событиях в эксперте или индикаторе следует описать специальную функцию-обработчик *OnTradeTransaction* — мы рассмотрим её подробно в следующем разделе. Дело в том, что один из её параметров — первый и самый важный — имеет тип предопределенной структуры *MqlTradeTransaction*. Поэтому давайте сначала поговорим о транзакциях как таковых.

```
struct MqlTradeTransaction
{
   ulong                         deal;             // Тикет сделки
   ulong                         order;            // Тикет ордера
   string                        symbol;           // Имя торгового инструмента
   ENUM_TRADE_TRANSACTION_TYPE   type;             // Тип торговой транзакции
   ENUM_ORDER_TYPE               order_type;       // Тип ордера
   ENUM_ORDER_STATE              order_state;      // Состояние ордера
   ENUM_DEAL_TYPE                deal_type;        // Тип сделки
   ENUM_ORDER_TYPE_TIME          time_type;        // Тип ордера по времени действия
   datetime                      time_expiration;  // Срок истечения ордера
   double                        price;            // Цена
   double                        price_trigger;    // Цена срабатывания стоп-лимитног
   double                        price_sl;         // Уровень Stop Loss
   double                        price_tp;         // Уровень Take Profit
   double                        volume;           // Объем в лотах
   ulong                         position;         // Тикет позиции
   ulong                         position_by;      // Тикет встречной позиции
};
```

В следующей таблице описаны все поля структуры.

Поле	Описание
deal	Тикет сделки
order	Тикет ордера
symbol	Имя торгового инструмента, по которому совершена транзакция
type	Тип торговой транзакции ENUM_TRADE_TRANSACTION_TYPE (см. ниже)
order_type	Тип ордера ENUM_ORDER_TYPE
order_state	Состояние ордера ENUM_ORDER_STATE
deal_type	Тип сделки ENUM_DEAL_TYPE
time_type	Тип ордера по истечению ENUM_ORDER_TYPE_TIME
time_expiration	Срок истечения отложенного ордера
price	Цена ордера, сделки или позиции, в зависимости от транзакции
price_trigger	Стоп-цена (цена срабатывания) стоп-лимитного ордера
price_sl	Цена *Stop Loss*, может относиться к ордеру, сделке или позиции, в зависимости от транзакции
price_tp	Цена *Take Profit*, может относиться к ордеру, сделке или позиции, в зависимости от транзакции
volume	Объем в лотах, может указывать на текущий объем ордера, сделки или позиции, в зависимости от транзакции

Поле	Описание
position	Тикет позиции, на которую повлияла транзакция
position_by	Тикет встречной позиции

Некоторые поля имеют смысл только в определенных случаях. В частности, поле *time_expiration* заполняется для ордеров, у которых в *time_type* указан один из типов истечения ORDER_TIME_SPECIFIED и ORDER_TIME_SPECIFIED_DAY. Поле *price_trigger* зарезервировано только для стоп-лимитных ордеров (ORDER_TYPE_BUY_STOP_LIMIT и ORDER_TYPE_SELL_STOP_LIMIT).

Также очевидно, что модификации позиции оперируют тикетом позиции (поле *position*), но не используют тикеты ордеров или сделок. Кроме того, поле *position_by* предназначено исключительно для закрытия позиции встречной — открытой по тому же инструменту, но в противоположном направлении.

Определяющей характеристикой для анализа транзакции является ее тип (поле *type*). В MQL5 API для его описания введено специальное перечисление ENUM_TRADE_TRANSACTION_TYPE, в котором собраны все возможные типы транзакций.

Идентификатор	Описание
TRADE_TRANSACTION_ORDER_ADD	Добавление нового ордера
TRADE_TRANSACTION_ORDER_UPDATE	Изменение действующего ордера
TRADE_TRANSACTION_ORDER_DELETE	Удаление действующего ордера
TRADE_TRANSACTION_DEAL_ADD	Добавление сделки в историю
TRADE_TRANSACTION_DEAL_UPDATE	Изменение сделки в истории
TRADE_TRANSACTION_DEAL_DELETE	Удаление сделки из истории
TRADE_TRANSACTION_HISTORY_ADD	Добавление ордера в историю в результате исполнения или отмены
TRADE_TRANSACTION_HISTORY_UPDATE	Изменение ордера в истории
TRADE_TRANSACTION_HISTORY_DELETE	Удаление ордера из истории
TRADE_TRANSACTION_POSITION	Изменение позиции
TRADE_TRANSACTION_REQUEST	Уведомление о том, что торговый запрос обработан сервером, и результат его обработки получен

Сделаем некоторые пояснения.

В транзакции типа TRADE_TRANSACTION_ORDER_UPDATE к изменениям ордера относятся не только явные изменения со стороны клиентского терминала или торгового сервера, но также и изменение его состояния (например, переход из состояния ORDER_STATE_STARTED в ORDER_STATE_PLACED или из ORDER_STATE_PLACED в ORDER_STATE_PARTIAL и т.д.).

При транзакции TRADE_TRANSACTION_ORDER_DELETE ордер может быть удален в результате соответствующего явного запроса или исполнения (заливки) на сервере. В обоих случаях он будет перенесен в историю, и должна также случиться транзакция TRADE_TRANSACTION_HISTORY_ADD.

Транзакция TRADE_TRANSACTION_DEAL_ADD осуществляется не только в результате исполнения ордера, но и проведения операций с балансом счета.

Некоторые транзакции, такие как TRADE_TRANSACTION_DEAL_UPDATE, TRADE_TRANSACTION_DEAL_DELETE, TRADE_TRANSACTION_HISTORY_DELETE довольно редки, так как описывают ситуации, когда сделка или ордер в истории меняется или удаляется на сервере "задним числом". Это, как правило, является следствием синхронизации с внешней торговой системой (биржей).

Важно отметить, что добавление или ликвидация позиции не влечет за собой появление транзакции TRADE_TRANSACTION_POSITION. Данный тип транзакции сообщает о том, что позиция была изменена на стороне торгового сервера, программно или пользователем вручную. В частности, у позиции может быть изменен объем (частичное встречное закрытие, переворот), цена открытия, а также уровни *Stop Loss* и *Take Profit*. Некоторые действия, например, доливки, не вызывают это событие.

Все торговые приказы, отдаваемые MQL-программами, находят отражение в транзакциях TRADE_TRANSACTION_REQUEST, что позволяет отложенным способом анализировать их выполнение. Это особенно важно при использовании функции *OrderSendAsync*, которая сразу возвращает управление в вызывающий код, из-за чего результат не известен. Вместе с тем транзакции точно также генерируются и при использовании "синхронной" *OrderSend*.

Кроме того, с помощью транзакций TRADE_TRANSACTION_REQUEST можно анализировать и торговые действия пользователя из интерфейса терминала.

6.4.34 Событие OnTradeTransaction

Эксперты и индикаторы могут получать уведомления о торговых событиях, если в их коде описана специальная функция обработки *OnTradeTransaction*.

```
void OnTradeTransaction(const MqlTradeTransaction &trans,
  const MqlTradeRequest &request, const MqlTradeResult &result)
```

Первым параметром идет структура *MqlTradeTransaction*, описанная в предыдущем разделе. Второй и третий параметры — это структуры *MqlTradeRequest* и *MqlTradeResult*, которые были представлены ранее в соответствующих разделах.

Структура *MqlTradeTransaction*, описывающая торговую транзакцию, заполняется по-разному в зависимости от типа транзакции, указанного в поле *type*. Например, для транзакций типа TRADE_TRANSACTION_REQUEST все остальные поля неважны, а для получения дополнительной информации необходимо анализировать второй и третий параметры функции (*request* и *result*). И напротив, для всех остальных типов транзакций два последних параметра функции следует игнорировать.

В случае TRADE_TRANSACTION_REQUEST поле *request_id* в переменной *result* содержит идентификатор (сквозной порядковый номер), под которым торговый запрос *request* зарегистрирован в терминале. Этот номер никак не связан с тикетами ордеров и сделок, а также идентификаторами позиций. В каждом сеансе работы с терминалом нумерация начинается с

начала (1). Наличие идентификатора запроса позволяет связать выполненное действие (вызов функций *OrderSend* или *OrderSendAsync*) с результатом этого действия, передаваемым в *OnTradeTransaction*. Позже мы рассмотрим примеры.

Для торговых транзакций, касающихся действующих ордеров (TRADE_TRANSACTION_ORDER_ADD, TRADE_TRANSACTION_ORDER_UPDATE и TRADE_TRANSACTION_ORDER_DELETE) и истории ордеров (TRADE_TRANSACTION_HISTORY_ADD, TRADE_TRANSACTION_HISTORY_UPDATE, TRADE_TRANSACTION_HISTORY_DELETE), в структуре *MqlTradeTransaction* заполняются следующие поля:

- order — тикет ордера;
- symbol — имя финансового инструмента в ордере;
- type — тип торговой транзакции;
- order_type — тип ордера;
- orders_state — текущее состояние ордера;
- time_type — тип истечения ордера;
- time_expiration — время истечения ордера (для ордеров с типом истечения ORDER_TIME_SPECIFIED и ORDER_TIME_SPECIFIED_DAY);
- price — цена в ордере, указанная клиентом/программой;
- price_trigger — стоп-цена срабатывания стоп-лимитного ордера (только для ORDER_TYPE_BUY_STOP_LIMIT и ORDER_TYPE_SELL_STOP_LIMIT);
- price_sl — цена *Stop Loss* ордера (заполняется, если указана в ордере);
- price_tp — цена *Take Profit* ордера (заполняется, если указана в ордере);
- volume — текущий объем ордера (не исполненный), изначальный объем ордера можно узнать из истории ордеров;
- position — тикет открытой, измененной или закрытой позиции;
- position_by — тикет встречной позиции (только для ордеров на закрытие встречной позицией).

Для торговых транзакций, касающихся сделок (TRADE_TRANSACTION_DEAL_ADD, TRADE_TRANSACTION_DEAL_UPDATE и TRADE_TRANSACTION_DEAL_DELETE), в структуре *MqlTradeTransaction* заполняются следующие поля:

- deal — тикет сделки;
- order — тикет ордера, на основе которого совершена сделка;
- symbol — имя финансового инструмента в сделке;
- type — тип торговой транзакции;
- deal_type — тип сделки;
- price — цена совершения сделки;
- price_sl — цена *Stop Loss* (заполняется, если указана в ордере, на основе которого совершена сделка);
- price_tp — цена *Take Profit* (заполняется, если указана в ордере, на основе которого совершена сделка);
- volume — объем сделки;
- position — тикет открытой, измененной или закрытой позиции;

- position_by — тикет встречной позиции (для сделок на закрытие встречной позицией).

Для торговых транзакций, касающихся изменений позиций (TRADE_TRANSACTION_POSITION), в структуре *MqlTradeTransaction* заполняются следующие поля:

- symbol — имя финансового инструмента позиции;
- type — тип торговой транзакции;
- deal_type — тип позиции (DEAL_TYPE_BUY или DEAL_TYPE_SELL);
- price — средневзвешенная цена открытия позиции;
- price_sl — цена *Stop Loss*;
- price_tp — цена *Take Profit*;
- volume — объем позиции в лотах;
- position — тикет позиции;

В описании торговой транзакции передается не вся доступная информация по ордерам, сделкам и позициям (например, комментарий). Для получения расширенной информации следует использовать *OrderGet*-, *HistoryOrderGet*-, *HistoryDealGet*- и *PositionGet*-функции.

Один торговый запрос, отправленный из терминала вручную или через торговые функции *OrderSend/OrderSendAsync*, может порождать на торговом сервере несколько последовательных торговых транзакций. При этом очередность поступления уведомлений об этих транзакциях в терминал не гарантирована, поэтому нельзя строить свой торговый алгоритм на ожидании одних торговых транзакций после других.

Торговые события обрабатываются асинхронно, то есть отложенно (по времени) относительно момента генерации. Каждое торговое событие посылается в очередь MQL-программы, и та последовательно "выбирает" их в порядке очереди (сам механизм "выборки" и вызова обработчиков в MQL-коде обеспечивает полностью ядро).

Во время обработки торговых транзакций экспертом внутри обработчика *OnTradeTransaction*, терминал продолжает принимать вновь поступающие торговые транзакции. Таким образом, состояние торгового счета может измениться уже в процессе работы *OnTradeTransaction*. В дальнейшем программа будет уведомлена обо всех этих событиях в порядке очереди событий.

Длина очереди транзакций составляет 1024 элемента. В случае, если *OnTradeTransaction* будет обрабатывать очередную транзакцию слишком долго, старые транзакции в очереди могут быть вытеснены более новыми.

Из-за параллельной многопоточной работы терминала с торговыми объектами к моменту вызова обработчика *OnTradeTransaction* все упомянутые в ней сущности — ордера, сделки, позиции — могут находиться уже в другом состоянии, нежели указанное в свойствах транзакции. Для получения их актуального состояния необходимо выбрать их в текущем окружении или в истории и запросить свойства с помощью соответствующих функций MQL5.

Для начала рассмотрим простой пример эксперта *TradeTransactions.mq5*, который выводит в журнал все торговые события *OnTradeTransaction*. Его единственный параметр *DetailedLog* позволяет опционально использовать классы *OrderMonitor*, *DealMonitor*, *PositionMonitor* для вывода всех свойств. По умолчанию эксперт выводит только содержимое заполненных полей структур *MqlTradeTransaction*, *MqlTradeRequest* и *MqlTradeResult*, поступающих в обработчик в виде параметров, причем *request* и *result* обрабатываются только для транзакций 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();
      }
   }
}
```

Запустим его на графике EURUSD и выполним вручную несколько действий — при этом в журнале будут появляться соответствующие записи (для чистоты эксперимента предполагается, что на торговом счете более никто и ничто не выполняет операций, в частности, не запущены другие эксперты).

Откроем длинную позицию минимальным лотом.

```
>>>      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=1
>>>      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
```

Продадим удвоенный минимальный лот.

```
>>>      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=1
>>>      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
```

Выполним операцию встречного закрытия.

```
>>>     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=1
>>>     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=1
>>>     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
```

У нас осталась короткая позиция минимального лота. Закроем её.

```
>>>     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=1
>>>     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
```

При желании вы можете включить опцию *DetailedLog* для вывода в журнал всех свойств торговых объектов на момент обработки события. В подробном логе можно заметить разночтения между состоянием объектов, сохраненном в структуре транзакции (на момент её инициирования), и текущим состоянием. Например, при добавлении приказа на закрытие позиции (встречное или обычное) в транзакции указан тикет, по которому объект-монитор уже не сможет ничего прочитать, так как позиция была удалена. В результате увидим в журнале строки вида:

```
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
```

Перезапустим эксперт *TradeTransaction.mq5* заново, чтобы сбросить счетчик событий, выводимый в журнал, для наглядности следующего теста. На этот раз будет достаточно стандартных настроек (без печати подробностей).

Теперь попробуем выполнить торговые действия программным способом в новом эксперте *OrderSendTransaction1.mq5*, и заодно опишем в нем свой обработчик *OnTradeTransaction* (такой же, как в предыдущем примере).

Этот эксперт предоставляет возможность выбрать направление сделки и объем: если оставить его нулевым, по умолчанию используется минимальный лот текущего символа. Также в параметрах есть дистанция до защитных уровней в пунктах. При заданных параметрах совершается вход в рынок, установка *Stop Loss* и *Take Profit*, и затем закрытие позиции — между всеми этапами делается пауза 5 секунд, так что пользователь может вмешаться (например, отредактировать стоплосс вручную), хотя это не обязательно, поскольку мы уже убедились, что ручные операции перехватываются программой.

```
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;
```

Стратегия запускается однократно, для чего используется 1-секундный таймер, который выключается в собственном обработчике.

```
int OnInit()
{
   EventSetTimer(1);
   return INIT_SUCCEEDED;
}

void OnTimer()
{
   EventKillTimer();
   ...
```

Все действия выполняются через уже знакомую структуру *MqlTradeRequestSync* с расширенными возможностями (*MqlTradeSync.mqh*): неявная инициализация полей правильными значениями, методы *buy/sell* для рыночных приказов, *adjust* — для защитных уровней, *close* — для закрытия позиции.

Шаг 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");
```

Шаг 2:

```
      Sleep(5000); // ждем 5 секунд (пользователь может редактировать позицию)
      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");
      }
```

Шаг 3:

```
      Sleep(5000); // ждем еще 5 секунд
      Print("Close down");
      if(request.close(request.result.position) && request.completed())
      {
         Print("Finish");
      }
      else
      {
         Print("Failed Close");
      }
   }
```

Промежуточные ожидания не только дают возможность успеть рассмотреть процесс, но и демонстрируют важный аспект программирования на MQL5 — однопоточность. Пока наш торгующий эксперт находится внутри *OnTimer*, генерируемые терминалом торговые события накапливаются в его очереди и будут переправлены во внутренний обработчик *OnTradeTransaction* в отложенном стиле — только после того, как произойдет выход из *OnTimer*.

В то же время, параллельно выполняющийся эксперт *TradeTransactions* не занят никакими вычислениями и будет получать торговые события максимально быстро.

Результат выполнения двух экспертов представлен в следующем логе с таймингом (для краткости *OrderSendTransaction1* помечен как *OS1*, а *TradeTransactions* — как *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_STA
                   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, @ 1.1091
                   P=1298021794
19:09:08.125  TTs  >>>     3
19:09:08.125  TTs  TRADE_TRANSACTION_ORDER_DELETE, #=1298021794(ORDER_TYPE_BUY/ORDER_
                   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_S
                   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_F
                   D=10, #=1298021794, M=1234567890
19:09:08.125  TTs  DONE, D=1280661362, #=1298021794, V=0.01, @ 1.10913, Bid=1.10913,
                   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=1.11
                   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_F
                   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_ST
                   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, @ 1.1090
                   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
                   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_
                   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_
                   D=10, #=1298022443, P=1298021794, M=1234567890
19:09:18.218  TTs  DONE, D=1280661967, #=1298022443, V=0.01, @ 1.10901, Bid=1.10901,
                   Req=11
19:09:18.218  OS1  >>>     1
```

```
19:09:18.218  OS1  TRADE_TRANSACTION_ORDER_ADD, #=1298021794(ORDER_TYPE_BUY/ORDER_STA
                   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_
                   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_S
                   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_F
                   D=10, #=1298021794, M=1234567890
19:09:18.218  OS1  DONE, D=1280661362, #=1298021794, V=0.01, @ 1.10913, Bid=1.10913,
                   Req=9
19:09:18.218  OS1  >>>     6
19:09:18.218  OS1  TRADE_TRANSACTION_POSITION, EURUSD, @ 1.10913, SL=1.09913, TP=1.11
                   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_F
                   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_ST
                   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, @ 1.1090
                   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
                   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_
                   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_
                   D=10, #=1298022443, P=1298021794, M=1234567890
19:09:18.218  OS1  DONE, D=1280661967, #=1298022443, V=0.01, @ 1.10901, Bid=1.10901,
                   Req=11
```

Нумерация событий в программах совпадает (при условии их чистого запуска, как рекомендовалось). Обратите внимание, что одно и то же событие печатается сначала из *TTs* сразу после выполнения запроса, а второй раз — уже в конце теста, где фактически единым пакетом выводятся все события из очереди в *OS1*.

Если убрать искусственные задержки, сценарий, разумеется, выполнится быстрее, но все равно обработчик *OnTradeTransaction* получит уведомления (несколько раз) после всех трех шагов, а не после каждого соответствующего запроса. Критично ли это?

Сейчас в примерах используется наша модификация структуры *MqlTradeRequestSync*, целенаправленно использующая "синхронный" вариант *OrderSend*, и более того — в ней реализован универсальный метод *completed*, проверяющий успешное завершение запроса. Благодаря этому контролю мы можем, в частности, устанавливать защитные уровни на позицию, потому что "умеем" дожидаться появления её тикета. В рамках такой синхронной концепции (взятой на вооружение в угоду удобству) анализ результатов запросов в *OnTradeTransaction* в общем-то не нужен. Однако так бывает не всегда.

Когда эксперту требуется отправить много запросов сразу, как в случае примера с установкой сетки ордеров *PendingOrderGrid2.mq5*, рассмотренной в разделе о свойствах позиций, ожидание "готовности" каждой позиции или ордера может снижать общую производительность эксперта. В подобных случаях рекомендуется использовать функцию *OrderSendAsync*, но она при успешном выполнении заполняет в структуре *MqlTradeResult* только поле *request_id*, с помощью которого затем нужно отслеживать появление ордеров, сделок и позиций в *OnTradeTransaction*.

Один из наиболее очевидных, но не особо изящных приемов для реализации этой схемы заключается в сохранении идентификаторов запросов или целиком структур отправляемых запросов в массиве, в глобальном контексте. Затем эти идентификаторы можно искать в приходящих транзакциях в *OnTradeTransaction*, находить тикеты в параметре *MqlTradeResult* и производить дальнейшие действия. В результате торговая логика оказывается разнесенной по разным функциям. Например, в контексте последнего эксперта *OrderSendTransaction1.mq5* данное "разнесение" заключается в том, что фрагменты кода после отправки первого приказа нужно перенести в *OnTradeTransaction* и обложить многочисленными проверками:

- на тип транзакции в *MqlTradeTransaction* (*transaction.type*);
- на тип запроса в *MqlTradeRequest* (*request.action*);
- на идентификатора запроса в *MqlTradeResult* (*result.request_id*);

И это все должно быть дополнено специфической прикладной логикой (например, проверкой на существование позиции), обеспечивающей ветвление по состояниям торговой стратегии. Чуть позже мы сделаем подобную модификацию эксперта *OrderSendTransaction* под другим номером, чтобы наглядно показать объем дополнительного исходного кода. А затем предложим способ организовать программу более линейно, но без отказа от транзакционных событий.

Пока лишь отметим, что выбор о построении алгоритма вокруг *OnTradeTransaction* или без него остается за разработчиком. Во многих случаях, когда массовая отправка приказов не нужна, можно оставаться в "синхронной" парадигме программирования. Вместе с тем, *OnTradeTransaction* является наиболее практичным способом для контроля срабатывания отложенных ордеров и защитных уровней, а также других событий, генерируемых сервером. После небольшой подготовки мы представим два соответствующих примера: финальную модификацию эксперта-сеточника и реализацию известного "сетапа" из двух ордеров OCO — "One Cancels Other" (см. раздел о событии *OnTrade*).

Альтернатива применению *OnTradeTransaction* заключается в периодическом анализе торгового окружения, то есть фактически в запоминании количества ордеров и позиций и поиске изменений среди них. Этот подход подойдет для стратегий, основанных на расписаниях или допускающих определенные временные задержки.

Еще раз подчеркнем, что использование *OnTradeTransaction* не означает, что в программе нужно непременно переходить с *OrderSend* на *OrderSendAsync*: вы можете использовать любую разновидность или обе. Напомним, функция *OrderSend* тоже не совсем синхронная, так как возвращает в лучшем случае тикет ордера и сделки, но не позиции. Скоро у нас появится

возможность измерить время исполнения пакета приказов в рамках одной и той же сеточной стратегии при использовании обоих вариантов функции: *OrderSend* и *OrderSendAsync*.

Для унификации разработки синхронных и асинхронных программ было бы здорово поддержать *OrderSendAsync* в нашей структуре *MqlTradeRequestSync* (несмотря на её название). Сделать это можно, внеся лишь пару исправлений. Во-первых, необходимо заменить все имеющиеся на данный момент вызовы *OrderSend* на собственный метод *orderSend*, а в нём переключать обращение к *OrderSend* или *OrderSendAsync* в зависимости от некоего флага.

```
struct MqlTradeRequestSync: public MqlTradeRequest
{
   ...
   static bool AsyncEnabled;
   ...
private:
   bool orderSend(const MqlTradeRequest &req, MqlTradeResult &res)
   {
      return AsyncEnabled ? ::OrderSendAsync(req, res) : ::OrderSend(req, res);
   }
};
```

Устанавливая публичную переменную *AsyncEnabled* в *true* или *false*, можно переключаться из одного режима в другой, например, в том фрагменте кода, где делаются массовые отправки приказов.

Во-вторых, для тех методов структуры, которые возвращали тикет (например, для входа в рынок), следует предусмотреть возврат поля *request_id* вместо *order*. Например, внутри методов *_pending* и *_market* у нас был оператор:

```
if(OrderSend(this, result)) return result.order;
```

Теперь он заменяется на:

```
if(orderSend(this, result)) return result.order ? result.order :
   (result.retcode == TRADE_RETCODE_PLACED ? result.request_id : 0);
```

Разумеется, когда включен асинхронный режим, мы уже не можем пользоваться методом *completed* для ожидания готовности результатов запроса сразу после его отправки. Но этот метод в принципе является опциональным — вы можете не использовать его даже при работе через *OrderSend*.

Итак, с учетом новой модификации файла *MqlTradeSync.mqh* создадим *OrderSendTransaction2.mq5*.

Этот эксперт будет отправлять первичный запрос как прежде из *OnTimer*, а установку защитных уровней и закрытие позиции — в *OnTradeTransaction* поэтапно. Хотя у нас между этапами не будет на этот раз искусственной задержки, сама последовательность состояний является стандартной для многих экспертов: открыл позицию, модифицировал, закрыл (при выполнении неких рыночных условий, которые здесь оставлены "за кадром").

Отслеживать состояние позволят 2 глобальные переменные: *RequestID* с идентификатором последнего отправленного запроса (результат которого мы ожидаем) и *PositionTicket* с тикетом открытой позиции. Когда позиции еще нет или уже нет, тикет равен 0.

```
uint RequestID = 0;
ulong PositionTicket = 0;
```

В обработчике *OnInit* включим асинхронный режим.

```
int OnInit()
{
   ...
   MqlTradeRequestSync::AsyncEnabled = true;
   ...
}
```

Функция *OnTimer* теперь значительно короче.

```
void OnTimer()
{
   ...
   // отсылаем запрос TRADE_ACTION_DEAL (асинхронно!)
   const ulong order = (Type == MARKET_BUY ? request.buy(volume) : request.sell(volum
   if(order) // в асихронном режиме это теперь request_id
   {
      Print("OK Open?");
      RequestID = request.result.request_id; // то же самое, что order
   }
   else
   {
      Print("Failed Open");
   }
}
```

При успешном выполнении запроса мы получаем только *request_id* и сохраняем его в переменной *RequestID*. Печать статусов теперь содержит вопросительный знак, например, "OK Open?", потому что фактический результат еще не известен.

OnTradeTransaction существенно усложнилась из-за проверки результатов и выполнения последующих торговых приказов по условиям. Рассмотрим её постепенно.

В данном случае вся торговая логика переехала внутрь ветви для транзакций типа TRADE_TRANSACTION_REQUEST. Разумеется, разработчик может по желанию использовать и другие типы, но мы используем этот, поскольку он содержит информацию в виде привычной структуры *MqlTradeResult*, то есть как бы представляет собой отложенную концовку асинхронного вызова *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));

      ...
      // здесь весь алгоритм
   }
}
```

Нас должны интересовать только запросы с идентификатором, который мы ожидаем. Поэтому следующим оператором будет вложенный *if*. В его блоке мы заранее описываем объект *MqlTradeRequestSync*, потому что потребуется отправлять очередные торговые запросы согласно плану.

```
      if(result.request_id == RequestID)
      {
         MqlTradeRequestSync next;
         next.magic = Magic;
         next.deviation = Deviation;
         ...
      }
```

Рабочих типов запроса у нас только два, поэтому добавляем для них еще один вложенный *if*.

```
         if(request.action == TRADE_ACTION_DEAL)
         {
            ... // здесь реакция на открытие и закрытие позиции
         }
         else if(request.action == TRADE_ACTION_SLTP)
         {
            ... // здесь реакция на установку SLTP у открытой позиции
         }
```

Обратите внимание, что TRADE_ACTION_DEAL используется и для открытия, и для закрытия позиции, а потому потребуется еще один *if*, в котором будем различать эти два состояния в зависимости от значения переменной *PositionTicket*.

```
            if(PositionTicket == 0)
            {
               ... // позиции нет, значит это уведомление об открытии
            }
            else
            {
               ... // позиция есть, значит это закрытие
```

```
      }
```

В рассматриваемой торговой стратегии нет "доливок" (для неттинга) или нескольких позиций (для хеджинга), из-за чего данная часть логически проста. В реальных экспертах потребуется гораздо больше различных оценок промежуточных состояний.

В случае уведомления об открытии позиции блок кода выглядит следующим образом:

```
      if(PositionTicket == 0)
      {
         // пытаемся получить результаты из транзакции: выделяем ордер по тикет
         if(!HistoryOrderSelect(result.order))
         {
            Print("Can't select order in history");
            RequestID = 0;
            return;
         }
         // получаем идентификатор и тикет позиции
         const ulong posid = HistoryOrderGetInteger(result.order, ORDER_POSITIC
         PositionTicket = TU::PositionSelectById(posid);
         ...
```

Для простоты мы опустили здесь проверку ошибок, в частности реквотов, но пример их обработки можно увидеть в прилагаемом исходном коде. Напомним, что все эти проверки уже были реализованы в методах структуры *MqlTradeRequestSync*, но работают они только в синхронном режиме, и поэтому нам приходится их повторять в явном виде.

Дальнейший фрагмент кода по установке защитных уровней почти не изменился.

```
      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);
         // посылаем запрос TRADE_ACTION_SLTP (асинхронно!)
         if(next.adjust(PositionTicket, SL, TP))
         {
            Print("OK Adjust?");
            RequestID = next.result.request_id;
         }
         else
         {
            Print("Failed Adjust");
            RequestID = 0;
         }
      }
```

Единственное отличие: мы заполняем переменную *RequestID* идентификатором нового запроса TRADE_ACTION_SLTP.

Получение уведомления о сделке при ненулевом *PositionTicket* подразумевает, что произошло закрытие позиции.

```
if(PositionTicket == 0)
{
   ... // см. выше
}
else
{
   if(!PositionSelectByTicket(PositionTicket))
   {
      Print("Finish");
      RequestID = 0;
      PositionTicket = 0;
   }
}
```

В случае успешного удаления позицию не удастся выделить с помощью *PositionSelectByTicket*, а потому мы обнуляем *RequestID* и *PositionTicket*. Эксперт при этом возвращается в начальное состояние и готов совершить следующий цикл покупки/продажи-модификации-закрытия.

Нам осталось рассмотреть отправку запроса на закрытие позиции. В нашей упрощенной до минимума стратегии это происходит сразу после успешной модификации защитных уровней.

```
if(request.action == TRADE_ACTION_DEAL)
{
   ... // см. выше
}
else if(request.action == TRADE_ACTION_SLTP)
{
   // посылаем запрос TRADE_ACTION_DEAL на закрытие (асинхронно!)
   if(next.close(PositionTicket))
   {
      Print("OK Close?");
      RequestID = next.result.request_id;
   }
   else
   {
      PrintFormat("Failed Close %lld", PositionTicket);
   }
}
```

Вот и вся функция *OnTradeTransaction*. Эксперт готов.

Запустим *OrderSendTransaction2.mq5* с настройками по умолчанию на EURUSD. Ниже приведен пример журнала.

```
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=1
>>>     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
```

Легко убедиться, что торговая логика работает по плану, и события о транзакциях прибывают строго после отправки каждого следующего приказа. Если теперь параллельно запустить наш новый эксперт и сторонний "перехватчик" транзакций *TradeTransactions.mq5*, сообщения в журнале от двух экспертов будут появляться синхронно.

Однако переделка из первой "прямопоточной" версии *OrderSendTransaction1.mq5* в асинхронную вторую *OrderSendTransaction2.mq5* потребовала существенного усложнения кода. Возникает вопрос: нельзя ли как-то совместить принципы последовательного описания торговой логики (прозрачность кода) и параллельной обработки (скорость)?

В принципе, это возможно, но потребует один раз потрудиться над созданием некоего вспомогательного механизма.

6.4.35 Синхронные и асинхронные запросы

Прежде чем углубляться в детали, напомним, что каждая MQL-программа выполняется в собственном потоке, и потому параллельная асинхронная обработка транзакций (и прочих событий) возможна только за счет того, что этим занималась бы другая MQL-программа. При этом необходимо обеспечить передачу информации между программами. Мы уже знаем пару способов для этого: глобальные переменные терминала и файлы. В 7-ой Части книги мы познакомимся с другими возможностями, такими как графические ресурсы и базы данных.

Действительно, представим себе, что эксперт аналогичный *TradeTransactions.mq5* выполняется параллельно с торговым экспертом и сохраняет полученные транзакции (не обязательно все поля, а лишь выборочные, влияющие на принятие решений) в глобальных переменных. Тогда торговый эксперт мог бы сразу после отправки очередного запроса проверять глобальные переменные и считывать из них результаты, не покидая текущей функции. Более того, в нем не нужен собственный обработчик *OnTradeTransaction*.

Однако запуск стороннего эксперта не очень просто организовать. Чисто технически это можно было сделать за счет создания объекта-графика и применения в нем шаблона с предопределенным экспертом-монитором транзакций. Но есть более легкий способ. Дело в том, что события *OnTradeTransaction* транслируются не только в эксперты, но и в индикаторы. А индикатор — наиболее просто запускаемый тип MQL-программы: достаточно вызвать *iCustom*.

Кроме того, использование индикатора дает еще один приятный бонус: в нём можно описать индикаторный буфер, доступный из внешних программ через *CopyBuffer*, и организовать в нем *кольцевой буфер* (таков термин, так что тавтология неизбежна) для хранения поступающих от терминала транзакций (результатов запросов). Таким образом, не потребуется связываться с глобальными переменными.

> Внимание! По каким-то причинам событие *OnTradeTransaction* не генерируется для индикаторов в тестере, поэтому проверить работу связки эксперт-индикатор можно только онлайн.

Назовем данный индикатор *TradeTransactionRelay.mq5*. Опишем в нем единственный буфер. В принципе, его можно сделать невидимым, потому что в него будут записываться данные, не подлежащие визуализации, но мы оставили его видимым для подтверждения самой концепции.

```
#property indicator_chart_window
#property indicator_buffers 1
#property indicator_plots   1

double Buffer[];

void OnInit()
{
   SetIndexBuffer(0, Buffer, INDICATOR_DATA);
}
```

Обработчик *OnCalculate* пустой.

```
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const int begin,
                const double &price[])
{
   return rates_total;
}
```

Нам потребуется в коде готовый конвертер из *double* в *ulong* и обратно, так как ячейки буфера могут повредить большие значения *ulong*, если их туда записывать с помощью простого приведения типов (см. Вещественные числа).

```
#include <MQL5Book/ConverterT.mqh>
Converter<ulong,double> cnv;
```

А вот и функция *OnTradeTransaction*.

```
#define FIELD_NUM 6  // наиболее важные поля в MqlTradeResult

void OnTradeTransaction(const MqlTradeTransaction &transaction,
   const MqlTradeRequest &request,
   const MqlTradeResult &result)
{
   if(transaction.type == TRADE_TRANSACTION_REQUEST)
   {
      ArraySetAsSeries(Buffer, true);

      // сохраняем FIELD_NUM полей результата в последовательные ячейки буфера
      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;
      // это присваивание должно идти последним,
      // потому что оно является флагом готовности результатов
      Buffer[offset + 0] = result.request_id;
   }
}
```

Мы решили сохранять только 6 наиболее важных полей структуры *MqlTradeResult*. При желании можно расширить механизм на всю структуру целиком, но для переноса строкового поля *comment* потребуется массив символов, под который придется резервировать довольно много элементов.

Таким образом, каждый результат у нас сейчас занимает 6 последовательных ячеек буфера. Индекс первой ячейки шестерки определяется исходя из идентификатора запроса: это число просто умножается на 6. Поскольку запросов может быть много, запись работает по принципу кольцевого буфера, то есть получившийся индекс нормализуется с помощью деления по остатку ('%') на размер индикаторного буфера — а это количество баров, округленное до 6. Когда номера запросов превысят размер, запись пойдет по кругу с начальных элементов.

Поскольку на нумерацию баров влияет формирование новых баров, рекомендуется ставить индикатор на крупные таймфреймы, такие как D1. Тогда только в начале суток вероятна (но даже маловероятна) ситуация, когда нумерация баров в индикаторе сдвинется непосредственно в процессе обработки очередной транзакции, и тогда записанные индикатором результаты не будут считаны экспертом (одна транзакция может быть пропущена).

Индикатор готов. Теперь приступим к реализации новой модификации тестового эксперта — *OrderSendTransaction3.mq5* (ура, это его последняя версия). Опишем в нем переменную *handle* под дескриптор индикатора и создадим индикатор в *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;
}
```

Для чтения результатов запросов из индикаторного буфера подготовим вспомогательную функцию *AwaitAsync*. Она принимает в качестве первого параметра ссылку на структуру *MqlTradeRequestSync*, куда будут в случае успеха записаны результаты, полученные из буфера индикатора с дескриптором *handle*. Идентификатор интересующего нас запроса уже должен быть во вложенной структуре, в поле *result.request_id*. Разумеется, здесь мы должны считывать данные по такому же принципу — шестерками баров.

```
#define FIELD_NUM   6  // наиболее важные поля в MqlTradeResult
#define TIMEOUT  1000  // 1 секунда

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();
   // ждем поступления результатов или таймаута
   while(!IsStopped() && GetTickCount() - start < TIMEOUT)
   {
      double array[];
      if((CopyBuffer(_handle, 0, offset, FIELD_NUM, array)) == FIELD_NUM)
      {
         ArraySetAsSeries(array, true);
         // когда найден request_id, заполняем остальные поля результатами
         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;
}
```

Имея данную функцию, напишем торговый алгоритм в асинхронно-синхронном стиле: как прямую последовательность шагов, каждый из которых дожидается готовности предыдущего за счет уведомлений из параллельной программы-индикатора, оставаясь при этом внутри одной функции.

```
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;
   ...
```

Шаг 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;
   }
   ...
```

Шаг 2.

```
   Print("SL/TP modification");
   ...
   if(request.adjust(SL, TP))
   {
      Print("OK Adjust?");
   }

   if(!(AwaitAsync(request, handle) && request.completed()))
   {
      Print("Failed Adjust");
   }
```

Шаг 3.

```
   Print("Close down");
   if(request.close(request.result.position))
   {
      Print("OK Close?");
   }

   if(!(AwaitAsync(request, handle) && request.completed()))
   {
      Print("Failed Close");
   }

   Print("Finish");
}
```

Обратите внимание, что вызовы метода *completed* теперь делаются не после отправки запроса, а после получения результата функцией *AwaitAsync*.

В остальном всё очень похоже на первый вариант этого алгоритма, но теперь он построен на асинхронных вызовах функций и реагирует на асинхронные события.

Вероятно, это не кажется существенным в данном конкретном примере с цепочкой манипуляций над одной позицией. Однако мы можем применить эту же технику для отправки и контроля пакета приказов. И тогда выгода станет очевидной. Спустя момент мы продемонстрируем это с помощью эксперта-сеточника и заодно сравним быстродействие двух функций: *OrderSend* и *OrderSendAsync*.

Но прямо сейчас, завершая серию экспертов *OrderSendTransaction*, запустим последнюю версию и увидим в журнале штатное, линейное выполнение всех шагов.

```
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
```

Тайминг с задержками ответов может существенно зависеть от сервера, времени суток, и инструмента. Разумеется, часть времени здесь уходит не на торговый запрос с подтверждением, а на выполнение функции *CopyBuffer*. По нашим наблюдениям, на него тратится не более 16 мс (в пределах одного такта стандартного системного таймера, желающие могут профилировать программы с помощью таймеров повышенной точности *GetMicrosecondCount*).

Не обращайте внимание на разночтение статуса (DONE) и строкового описания ("Order placed"). Дело в том, что комментарий (а также поля *ask/bid*) остается в структуре с момента отправки функцией *OrderSendAsync*, а окончательный статус в поле *retcode* записывает наша функция

AwaitAsync. Для нас важно, что в структуре с результатами являются актуальными номера тикетов (*deal* и *order*), цена исполнения (*price*) и объем (*volume*).

Используя наработки примера *OrderSendTransaction3.mq5*, создадим новую версию сеточного эксперта *PendingOrderGrid3.mq5* (напомним, что предыдущая описана в разделе Функции для чтения свойств позиций). Её особенностью будет возможность устанавливать полную сетку ордеров в синхронном или асинхронном режиме, по выбору пользователя. Также мы засечем времена установки полной сетки для сравнения.

Режимом управляет входная переменная *EnableAsyncSetup*, а под дескриптор индикатора выделена, как обычно, переменная *handle*.

```
input bool EnableAsyncSetup = false;

int handle;
```

При инициализации, в случае асинхронного режима, создаем экземпляр индикатора *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);
   }
   ...
}
```

В целях упрощения кодирования мы заменили в функции *SetupGrid* двумерный массив *request* на одномерный.

```
uint SetupGrid()
{
   ...                                  // было:
   MqlTradeRequestSyncLog request[];    // MqlTradeRequestSyncLog request[][2];
   ArrayResize(request, GridSize * 2);  // ArrayResize(request, GridSize);
   ...
}
```

Далее в цикле по массиву вместо обращений типа *request[i][1]* используется адресация *request[i * 2 + 1]*.

Это маленькое преобразование потребовалось по следующей причине. Поскольку при создании сетки мы используем этот массив структур для запросов, и требуется дождаться всех

результатов, функция *AwaitAsync* должна теперь принимать первым параметром ссылку на массив. А одномерный массив проще обрабатывать.

Для каждого запроса, согласно его *request_id* рассчитывается свое смещений в индикаторном буфере: все смещения помещаются в массив *offset*. По мере получения подтверждений запросов, соответствующие элементы массива помечаются отработанными путем записи туда значения -1. Количество выполненных запросов подсчитывается в переменной *done*. Когда оно сравняется с размером массива, вся сетка готова.

```
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; // пропускаем пустые элементы
      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; // помечаем обработанным
            done++;
         }
      }
   }
   return done == n;
}
```

Возвращаясь к функции *SetupGrid*, покажем, как вызов *AwaitAsync* производится после цикла отправки запросов.

```
uint SetupGrid()
{
   ...
   const uint start = GetTickCount();
   for(int i = 0; i < (int)GridSize / 2; ++i)
   {
      // вызовы 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));
   ...
}
```

Если при установке сетки случится таймаут (не все запросы получат подтверждение за отведенное время), мы вернем код TRADE_RETCODE_ERROR, и эксперт попытается "откатить" то, что успел создать.

Важно отметить, что асинхронный режим предполагается только для установки полной сетки, когда требуется отправить пакет запросов. В остальных случаях будет по-прежнему использоваться синхронный режим. Поэтому мы должны установить флаг *MqlTradeRequestSync::AsyncEnabled* в *true* перед циклом отправки и вернуть в *false* после. Однако здесь есть одна тонкость. Внутри цикла могут случиться ошибки, из-за которых он досрочно прерывается с возвратом последнего кода с сервера. Таким образом, если мы разместим сброс асинхронного режима после цикла, нет гарантии, что он будет сброшен.

Чтобы решить эту проблему в файле *MqlTradeSync.mqh* добавлен маленький класс *AsyncSwitcher*, управляющий включением и отключением асинхронного режима из своих конструктора и деструктора. Это в духе концепции по управлению ресурсами RAII, рассмотренной в разделе Управление дескрипторами файлов.

```
class AsyncSwitcher
{
public:
   AsyncSwitcher(const bool enabled = true)
   {
      MqlTradeRequestSync::AsyncEnabled = enabled;
   }
   ~AsyncSwitcher()
   {
      MqlTradeRequestSync::AsyncEnabled = false;
   }
};
```

Теперь для безопасного временного включения асинхронного режима достаточно описать в функции *SetupGrid* локальный объект *AsyncSwitcher*. Возврат в синхронный режим произойдет автоматически при любом выходе из функции.

```
uint SetupGrid()
{
   ...
   AsyncSwitcher sync(EnableAsyncSetup);
   ...
   for(int i = 0; i < (int)GridSize / 2; ++i)
   {
      ...
   }
   ...
}
```

Эксперт готов. Попробуем запустить его два раза: в синхронном и асинхронном режимах для сетки достаточно большого размера (10 уровней, шаг сетки 200).

Для сетки в 10 уровней мы получим 20 запросов, поэтому ниже показаны фрагменты журнала с сокращениями. Сначала был использован синхронный режим. Поясним, что надпись о готовности запросов выводится раньше сообщений о самих запросах, потому что последние генерируются деструкторами структур при выходе из функции. Скорость обработки: 51мс на запрос.

```
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
```

Середина сетки пришлась на цену 1.10400. Система нумерует запросы в порядке поступления, а их нумерация в массиве соответствует порядку, в котором мы выставляем ордера: от центрального базового уровня постепенно расходимся в стороны. Поэтому не удивляйтесь, что после пары 1 и 2 (для уровня 1.10200) идет 5 и 6 (1.10000) — просто мы отослали 3 и 4 (1.10600) раньше.

В асинхронном режиме перед деструкторами "вклиниваются" сообщения о готовности конкретных запросов, получаемые в *AwaitAsync* в реальном времени, причем не обязательно в том порядке, в котором запросы были отправлены (например, 49-й и 50-й запросы "обогнали" 47-й и 48-й).

```
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)
...
```

Из-за того, что все запросы выполнялись параллельно, общая длительность отправки (234мс) лишь незначительно больше чем время одного запроса (здесь в районе 100мс, но у вас будет свой тайминг). В результате мы получили скорость 11мс на запрос — в 5 раз быстрее, чем при синхронном способе. Поскольку запросы отправлялись практически одновременно, мы не можем узнать время выполнения каждого, и миллисекунды сообщают приход результата конкретного запроса с момента общего начала групповой отправки.

Дальнейший вывод в лог содержит, как и в предыдущем случае, все поля запросов и результатов, печатаемые из деструкторов структур. Напомним, что строка "Order placed" осталась неизменной после *OrderSendAsync*, так как наш вспомогательный индикатор *TradeTransactionRelay.mq5* не публикует структуру *MqlTradeResult* из сообщения TRADE_TRANSACTION_REQUEST полностью.

```
...
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
```

До сих пор в нашем сеточном эксперте на каждом уровне располагалась пара отложенных ордеров - лимитный и стоп-лимитный. В принципе, можно избавиться от этого дублирования и оставить только лимитные ордера. Это будет окончательная версия *PendingOrderGrid4.mq5*, которую также можно запускать в синхронном и асинхронном режиме. Мы не станем подробно рассматривать исходный код, а лишь отметим основные отличия от предыдущей версии.

В функции *SetupGrid* потребуется массив структур размером *GridSize*, а не удвоенный, и количество запросов также уменьшится в 2 раза: для них используются только методы *buyLimit* и *sellLimit*.

Функция *CheckGrid* проверяет целостность сетки по другому принципу. Ранее ошибкой считалось отсутствие парного стоп-лимитного ордера на уровне, где есть лимитный. Такое могло произойти при срабатывании на сервере стоп-лимитного ордера с соседнего уровня. Однако данная схема не способна восстановить сетку, если на одном баре случится сильное двустороннее движение цены (спайк): оно выбьет не только исходные лимитные ордера, но и новые — сгенерированные при этом из стоп-лимитных. Теперь же алгоритм честно проверяет вакантные уровни по обе стороны от текущей цены и создает там лимитные ордера с помощью *RepairGridLevel*. Эта вспомогательная функция ранее расставляла стоп-лимитные ордера.

Наконец, в *PendingOrderGrid4.mq5* появился обработчик *OnTradeTransaction*. Срабатывание отложенного ордера приведет к заключению сделки (и изменению конфигурации сетки, которую нужно поправить), поэтому мы контролируем сделки по заданному символу и магику. При обнаружении сделки функция *CheckGrid* вызывается моментально, в дополнение к тому, что она по-прежнему выполняется в начале каждого бара.

```
void OnTradeTransaction(const MqlTradeTransaction &transaction,
   const MqlTradeRequest &,
   const MqlTradeResult &)
{
   if(transaction.type == TRADE_TRANSACTION_DEAL_ADD)
   {
      if(transaction.symbol == _Symbol)
      {
         DealMonitor dm(transaction.deal); // выделяем сделку
         if(dm.get(DEAL_MAGIC) == Magic)
         {
            CheckGrid();
         }
      }
   }
}
```

Следует отметить, что наличие *OnTradeTransaction* недостаточно для написания экспертов, устойчивых к непредвиденным внешним воздействиям. Конечно, события позволяют оперативно реагировать на ситуацию, однако у нас нет гарантии, что эксперт не будет выключен (или перейдет в оффлайн) по тем или иным причинам на некоторое время и пропустит ту или иную транзакцию. Поэтому обработчик *OnTradeTransaction* должен лишь помогать ускорить процессы, которые программа умеет выполнять и без него. В частности, восстанавливать корректно свое состояние после старта.

Однако помимо события *OnTradeTransaction* MQL5 предоставляет и другое, более простое событие *OnTrade*.

6.4.36 Событие OnTrade

Событие *OnTrade* возникает при изменении списка выставленных ордеров и открытых позиций, истории ордеров и истории сделок. При любом торговом действии (выставлении/срабатывании/удалении отложенного ордера, открытии/закрытии позиции, установке защитных уровней, и т.п.) соответствующим образом изменяется история ордеров и сделок и/или список позиций и текущих ордеров. Инициатором действия может быть пользователь, программа или сервер.

Для получения события в программе необходимо описать соответствующий обработчик.

void OnTrade(void)

В случае отправки торговых запросов с помощью *OrderSend/OrderSendAsync* один запрос вызовет несколько событий *OnTrade*, так как обработка обычно происходит в несколько этапов и каждая операция может изменять состояние ордеров, позиций и торговой истории.

В общем случае нет точного соотношения по количеству вызовов *OnTrade* и *OnTradeTransaction*. *OnTrade* вызывается после соответствующих вызовов *OnTradeTransaction*.

Поскольку событие *OnTrade* носит обобщенный характер и не конкретизирует суть операции, оно менее популярно у разработчиков MQL-программ: ведь в коде нужно реализовать проверку всех аспектов состояния торгового счета и сравнить его с неким сохраненным состоянием — фактически с прикладным кэшем используемых в торговой стратегии торговых сущностей. В простейшем случае можно, например, запомнить тикет созданного ордера и в обработчике *OnTrade* опрашивать все его свойства, однако при этом вполне вероятен "холостой" анализ большого количества попутных событий, никак не связанных с конкретным ордером.

О возможности прикладного кэширования торгового окружения и истории мы поговорим в разделе о мультивалютных экспертах.

А сейчас для практического изучения *OnTrade* займемся экспертом, реализующим стратегию на двух отложенных ордерах OCO ("One Cancels Other"). Он будет выставлять пару стоп-ордеров на пробой диапазона и отслеживать срабатывание одного из них с тем, чтобы убрать другой. Для наглядности предусмотрим поддержку обоих типов торговых событий *OnTrade* и *OnTradeTransaction*, так что рабочая логика будет запускаться, по выбору пользователя, либо из одного обработчика, либо из другого.

Исходный код доступен в файле *OCO2.mq5*. Во входных параметрах: размер лота *Volume* (по умолчанию, 0, что означает минимальный), дистанция *Distance2SLTP* в пунктах до места установки каждого из ордеров, и она же определяет защитные уровни, срок истечения *Expiration* в секундах от времени установки, и переключатель событий *ActivationBy* (по умолчанию, *OnTradeTransaction*). Поскольку *Distance2SLTP* задает и отступ от текущей цены, и расстояние до стоплосса, стоплоссы двух ордеров совпадают и равны цене на момент установки.

```
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;
```

Для упрощения инициализации структур запросов опишем свою собственную структуру *MqlTradeRequestSyncOCO*, производую от *MqlTradeRequestSync*.

```
struct MqlTradeRequestSyncOCO: public MqlTradeRequestSync
{
   MqlTradeRequestSyncOCO()
   {
      symbol = _Symbol;
      magic = Magic;
      deviation = Deviation;
      if(Expiration > 0)
      {
         type_time = ORDER_TIME_SPECIFIED;
         expiration = (datetime)(TimeCurrent() + Expiration);
      }
   }
};
```

На глобальном уровне введем несколько объектов и переменных.

```
OrderFilter orders;         // объект для отбора ордеров
PositionFilter trades;      // объект для отбора позиций
bool FirstTick = false;     // для однократной обработки OnTick при старте
ulong ExecutionCount = 0;   // счетчик вызовов торговой стратегии RunStrategy()
```

Вся торговая логика за исключением момента старта будет запускаться торговыми событиями. В обработчика *OnInit* настраиваем объекты фильтров и ждем первого тика (ставим *FirstTick* в *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;
}
```

Нас интересуют только стоп-ордера (покупки/продажи) и позиции с конкретным магическим номером и текущим символом.

В функции *OnTick* однократно вызываем основную часть алгоритма, оформленную как *RunStrategy* (опишем её чуть ниже). Далее эта функция будет вызываться только из *OnTrade* или *OnTradeTransaction*.

```
void OnTick()
{
   if(FirstTick)
   {
      RunStrategy();
      FirstTick = false;
   }
}
```

Например, когда включен режим *OnTrade*, работает данный фрагмент.

```
void OnTrade()
{
   static ulong count = 0;
   PrintFormat("OnTrade(%d)", ++count);
   if(ActivationBy == ON_TRADE)
   {
      RunStrategy();
   }
}
```

Обратите внимание, что количество вызовов самого обработчика *OnTrade* подсчитывается независимо от того, активируется ли стратегия здесь или нет. Аналогичным образом в обработчике *OnTradeTransaction* считается своё количество событий (даже если они происходят вхолостую). Так сделано, чтобы иметь возможность увидеть в журнале одновременно оба события и их счетчики.

Когда включен режим *OnTradeTransaction*, очевидно, *RunStrategy* запускается оттуда.

```
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)
   {
      // почему не здесь? ответ см. в тексте
      /* // это не сработает онлайн: m.isReady() == false, т.к. ордер временно потеря
      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
      {
         // свойство ORDER_STATE неважно - в любом случае нужно удалить оставшийся
         // if(transaction.order_state == ORDER_STATE_FILLED
         // || transaction.order_state == ORDER_STATE_CANCELED ...)
         RunStrategy();
      }
   }
}
```

Следует отметить, что при торговле онлайн сработавший отложенный ордер на некоторое время может пропасть из торгового окружения в процессе переноса из действующих в историю. Когда мы получаем событие TRADE_TRANSACTION_ORDER_DELETE, ордер уже удален из списка действующих, но еще не попал в историю. Он оказывается там только в тот момент, когда мы получаем событие TRADE_TRANSACTION_HISTORY_ADD. Примечательно, что в тестере такой особенности нет, то есть удаленный ордер сразу попадает в историю и доступен там для выделения и чтения свойств уже в фазе TRADE_TRANSACTION_ORDER_DELETE.

В обоих обработчиках торговых событий мы подсчитываем и выводим в журнал их количество вызовов. Для случая *OnTrade* оно должно будет совпасть со счетчиком *ExecutionCount*, который мы скоро увидим внутри *RunStrategy*. А вот для *OnTradeTransaction* её счетчик и значение *ExecutionCount* будут существенно отличаться, потому что стратегия здесь вызывается сильно избирательно — по одному типу событий. Из этого можно сделать вывод, что *OnTradeTransaction* позволяет более эффективно расходовать ресурсы за счет вызова алгоритма, только когда это уместно.

Счетчик *ExecutionCount* выводится в журнал при выгрузке эксперта.

```
void OnDeinit(const int r)
{
   Print("ExecutionCount = ", ExecutionCount);
}
```

Теперь, наконец представим функцию *RunStrategy*. Обещанный счетчик инкрементируется в самом начале.

```
void RunStrategy()
{
   ExecutionCount++;
   ...
```

Далее описаны два массива для приема тикетов ордеров и их статусов из объекта-фильтра orders.

```
   ulong tickets[];
   ulong states[];
```

Для начала запросим ордера, подпадающие под наши условия. Если их 2 — всё хорошо, и делать ничего не надо.

```
   orders.select(ORDER_STATE, tickets, states);
   const int n = ArraySize(tickets);
   if(n == 2) return; // OK - штатное состояние
   ...
```

Если остался один ордер, значит другой сработал и оставшийся нужно удалить.

```
   if(n > 0)          // 1 или 2+ ордера это ошибка, нужно все удалить
   {
      // удаляем все подходящие ордера, кроме частично залитых
      MqlTradeRequestSyncOCO r;
      for(int i = 0; i < n; ++i)
      {
         if(states[i] != ORDER_STATE_PARTIAL)
         {
            r.remove(tickets[i]) && r.completed();
         }
      }
   }
   ...
```

В противном случае ордеров нет. Следовательно, нужно проверить, нет ли открытой позиции: для этого используем другой объект-фильтр *trades*, но результаты складываем в тот же приемный массив *tickets*. При отсутствии позиции выставляем новую пару ордеров.

```
      else // n == 0
      {
         // если нет открытых позиций, выставляем 2 ордера
         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();
         }
      }
   }
```

Запустим эксперт в тестере с настройками по умолчанию, на паре EURUSD. На следующем изображении показан процесс тестирования.

Эксперт с парой отложенных стоп-ордеров по стратегии OCO в тестере

На стадии установки пары ордеров увидим в журнале следующие записи.

```
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
```

Как только один из ордеров срабатывает, происходит вот что:

```
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
```

Ордер #3 удалился сам, а ордер #2 удален (отменен) нашим экспертом.

Если запустить эксперт, изменив в настройках только режим работы через событие *OnTrade*, мы должны получить полностью аналогичные финансовые результаты (при прочих равных условиях, то есть, например, если не включены случайные задержки в генерации тиков). Единственное, что будет отличаться: количество вызовов функции *RunStrategy*. Например, за 4 месяца 2022 года на EURUSD,H1 при 88 сделках получим такие приблизительные показатели *ExecutionCount* (важно соотношение, а не абсолютные величины, связанные с тиками вашего брокера):

- OnTradeTransaction — 132;

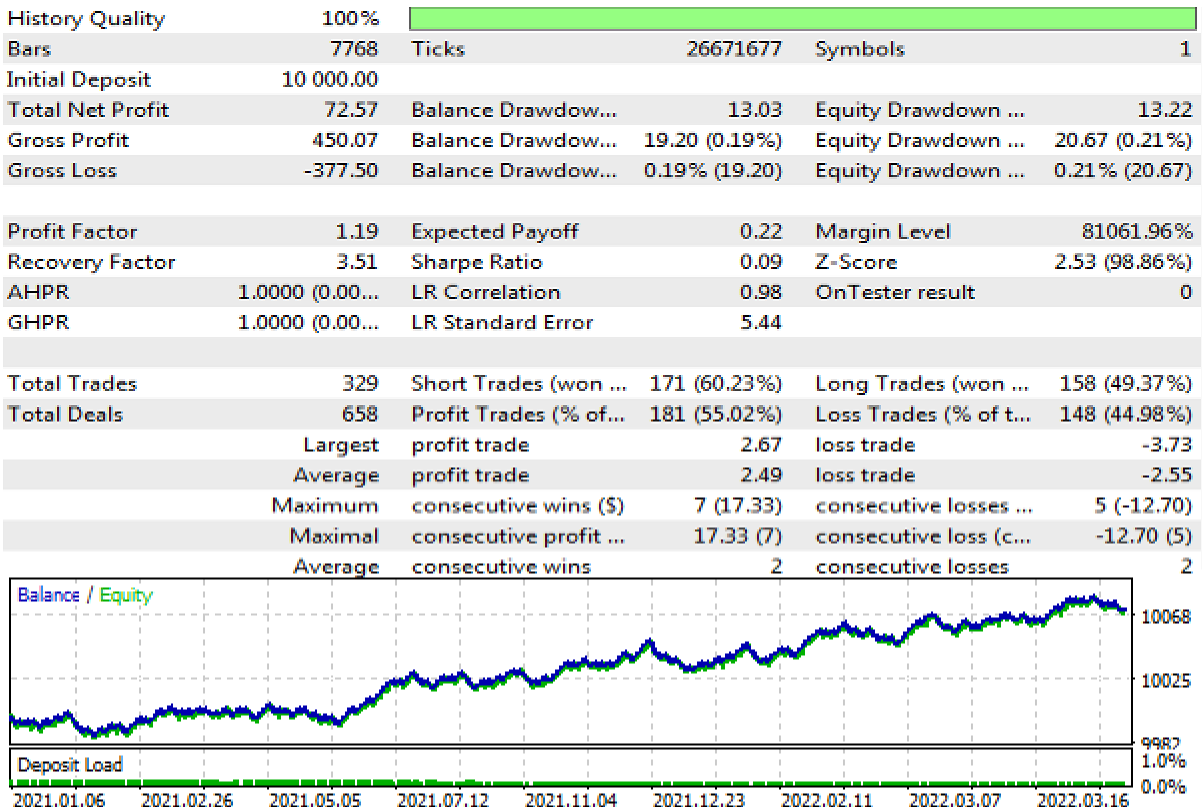
- OnTrade — 438;

Это практическое доказательство возможности строить более избирательные алгоритмы на базе *OnTradeTransaction* по сравнению с *OnTrade*.

Данная версия эксперта *OCO2.mq5* реагирует на действия с ордерами и позициями довольно прямолинейно. В частности, как только предыдущая позиция закроется по стоплоссу или тейкпрофиту, он выставит два новых ордера. Если удалить один из ордеров вручную, эксперт тут же удалит второй, а затем воссоздаст новую пару с отступом от текущей цены. Вы можете улучшить поведение за счет встраивания расписания, аналогичного тому, что сделано в эксперте-сеточнике, и не реагировать на отмененные ордера в истории (хотя, к сожалению, MQL5 не позволяет узнать, был ли ордер снят вручную или программно). Мы же представим иное направление по усовершенствованию этого эксперта в рамках изучения API экономического календаря.

Кроме того, уже в текущей версии доступен интересный режим, связанный с установкой срока истечения отложенных ордеров во входной переменной *Expiration*. Если пара ордеров не срабатывает, сразу по их истечении выставляется новая пара относительно изменившейся новой текущей цены. В качестве самостоятельного упражнения вы можете попробовать оптимизировать эксперт в тестере, меняя *Expiration* и *Distance2SLTP*. Программную работу с тестером, в том числе и в режиме оптимизации, мы затронем в следующей главе.

Ниже представлен один из вариантов настроек (*Distance2SLTP=250*, *Expiration=5000*), найденных на промежутке в 16 месяцев, с начала 2021 года по паре 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

Результаты тестового прогона эксперта OCO2

6.4.37 Контроль за изменениями торгового окружения

В предыдущем разделе про событие *OnTrade* мы упоминали о том, что некоторые подходы программирования торговых стратегий могут потребовать делать слепки окружения и сравнивать их друг с другом с течением времени. Это частый прием при использовании *OnTrade*, но его также можно активировать по расписанию, на каждом баре или даже тике. В наших классах-мониторах, способных читать свойства ордеров, сделок и позиций до сих пор отсутствовала способность сохранять состояние. В данном разделе мы представим один из вариантов кэширования торгового окружения.

Как вы знаете, свойства всех торговых объектов делятся по типам на 3 группы: целочисленные, вещественные и строковые. У каждого класса объектов эти группы свои (например, у ордеров целочисленные свойства описаны в перечислении ENUM_ORDER_PROPERTY_INTEGER, а у позиций — в ENUM_POSITION_PROPERTY_INTEGER), но суть деления общая. Поэтому введем перечисление PROP_TYPE, с помощью которого можно будет описывать для любого объекта, к какому типу относится то или иное свойство. Это обобщение напрашивается, поскольку механизмы хранения и обработки свойств одного типа должны быть одинаковыми, вне зависимости от того, принадлежит ли свойство ордеру, позиции или сделке.

```
enum PROP_TYPE
{
   PROP_TYPE_INTEGER,
   PROP_TYPE_DOUBLE,
   PROP_TYPE_STRING,
};
```

Наиболее простым способом хранения значений свойств являются массивы. Очевидно, что из-за наличия 3 базовых типов нам потребуется 3 разных массива. Опишем их внутри нового класса *TradeState*, вложенного в *MonitorInterface* (*TradeBaseMonitor.mqh*).

Напомним, что базовый шаблон *MonitorInterface<I,D,S>* составляет основу всех прикладных классов-мониторов (*OrderMonitor*, *DealMonitor*, *PositionMonitor*). Типы I, D, S здесь соответствуют конкретным перечислениям целых, вещественных и строковых свойств.

Вполне логично включить механизм хранения именно в базовый монитор, тем более что создаваемый кэш свойств будет наполняться данными посредством чтения свойств из объекта-монитора.

```
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)
      {
         ...
      }
   };
```

Весь класс *TradeState* сделан публичным, поскольку к его полям потребуется доступ из родительского объекта-монитора (который передается как указатель в конструктор), а кроме того объекты *TradeState* будут использованы только в защищенной части монитора (извне к ним достучаться нельзя).

Для того чтобы заполнять 3 массива значениями свойств 3-х разных типов необходимо предварительно выяснить распределение свойств по типам и индексы в каждом конкретном массиве.

Для каждого типа торговых объектов (ордеров, сделок, позиций) идентификаторы 3-х соответствующих перечислений со свойствами разных типов не пересекаются и составляют сквозную нумерацию. Продемонстрируем это.

В разделе Перечисления мы представили скрипт *ConversionEnum.mq5*, в котором реализована функция *process* для вывода в журнал всех элементов конкретного перечисления. В том скрипте изучалось перечисление ENUM_APPLIED_PRICE, но ничто не мешает нам сделать копию скрипта и выполнить анализ трех других перечислений, которые нас интересуют. Например, так:

```
void OnStart()
{
   process((ENUM_POSITION_PROPERTY_INTEGER)0);
   process((ENUM_POSITION_PROPERTY_DOUBLE)0);
   process((ENUM_POSITION_PROPERTY_STRING)0);
}
```

В результате его выполнения получим следующий лог. Левая колонка содержит нумерацию внутри перечислений, а значения справа (после знака '=') — встроенные константы (идентификаторы) элементов.

```
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
```

Например, свойство с константой 0 — это строковое POSITION_SYMBOL, с константами 1 и 2 — целочисленные POSITION_TIME и POSITION_TYPE, а с константой 3 — вещественное POSITION_VOLUME, и так далее.

Таким образом, константы представляют собой систему сквозных индексов по свойствам всех типов, и мы можем использовать тот же алгоритм (основанный на *EnumToArray.mqh*) для их получения.

Для каждого свойства нужно запомнить его тип (от этого зависит, в каком из трех массивов хранить значение) и порядковый номер среди свойств такого же типа (это будет индекс элемента в соответствующем массиве). Например, мы видим, что у позиций есть только 3 строковых свойства, значит массив *strings* в слепке одной позиции должен будет иметь такой размер, и под индексами 0, 1, 2 в него будут записываться POSITION_SYMBOL (0), POSITION_COMMENT (11), POSITION_EXTERNAL_ID (19).

Преобразование сквозных индексов свойств в их тип (один из PROP_TYPE) и в порядковый номер в массиве соответствующего типа можно сделать один раз при старте программы, так как перечисления со свойствами являются постоянными (встроены в систему). Полученную таблицу косвенной адресации запишем в статический двумерный массив *indices*. Его размер по первому измерению будет динамически определен как общее количество свойств (всех 3-х типов) — его запишем в статическую переменную *limit*. По второму измерению выделена пара ячеек: *indices[i][0]* — тип PROP_TYPE, *indices[i][1]* — индекс в одном из массивов *ulongs*, *doubles* или *strings* (в зависимости от *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];
   }
   ...
```

Переменные *j*, *d*, *s* будут использованы для последовательной индексации свойств внутри каждого из 3-х разных типов. Вот, собственно, как это делается в статическом методе *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;
      }
```

Метод *boundary* возвращает максимальную константу среди всех элементов заданного перечисления E.

```
   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];
   }
```

Наибольшее значение из всех трех типов перечислений определяет диапазон целых чисел, которые следует рассортировать по принадлежности к свойствам трех типов.

В этом помогает метод *detect*, который возвращает *true*, если целое число является элементом перечисления.

```
      template<typename E>
      static bool detect(const int v)
      {
         ResetLastError();
         const string s = EnumToString((E)v); // результат не используется
         if(_LastError == 0) // важно только отсутствие ошибки
         {
            return true;
         }
         return false;
      }
```

Последний вопрос в том, как запустить этот расчет при старте программы. Это достигается за счет статичности переменных и метода.

```
   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();
```

Обратите внимание, что *limit* инициализируется результатом вызова нашей функции *calcIndices*.

Имея таблицу с индексами, реализуем заполнение массивов значениями свойств в методе *cache*.

```
class TradeState
{
   ...
   TradeState(const MonitorInterface *ptr) : owner(ptr)
   {
      cache(); // при создании объекта сразу кешируем свойства
   }

   template<typename T>
   void _get(const int e, T &value) const // перегрузка с записью по ссылке
   {
      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;
         }
      }
   }
};
```

Мы проходимся в цикле по всему диапазону свойств от 0 до *limit* и в зависимости от типа свойства в *indices[i][0]* записываем его значение в элемент массива *ulongs*, *doubles* или *strings* под номером *indices[i][1]* (соответствующий элемент массива передается по ссылке в метод *_get*).

Вызов *owner.get(e, value)* обращается к одному из стандартных методов класса-монитора (здесь он виден как абстрактный указатель *MonitorInterface*). В частности, для позиций в классе *PositionMonitor* это приведет к вызовам *PositionGetInteger*, *PositionGetDouble* или *PositionGetString*. Нужный тип выберет компилятор. В мониторах ордеров и сделок существуют свои аналогичные реализации, которые автоматически подключаются этим базовым кодом.

Описание слепка одного торгового объекта логично унаследовать от класса-монитора. Поскольку нам предстоит кэшировать ордера, сделки и позиции, имеет смысл сделать новый класс шаблоном и собрать в нем все общие алгоритмы, подходящие для всех объектов. Назовем его *TradeBaseState* (файл *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)   // включение/отключение кеша по желанию
   {
      cached = b;
   }
   ...
```

Под буквой M здесь скрывается один из конкретных классов-мониторов, описанных ранее (*OrderMonitor.mqh*, *PositionMonitor.mqh*, *DealMonitor.mqh*). Основу составляет кэширующий объект *state* только что представленного класса *M::TradeState*. В зависимости от M внутри будет сформирована специфическая индексная таблица (одна для класса M) и распределены массивы свойств (собственные для каждого экземпляра M, то есть для каждого ордера, сделки, позиции).

Переменная *cached* содержит признак того, заполнены ли уже массивы в *state* значениями свойств и следует ли при запросе свойств у объекта возвращать значения из кэша. Это потребуется в дальнейшем для сравнения сохраненного и актуального состояний.

Иными словами, когда *cached* сброшено в *false*, объект будет вести себя как обычный монитор, считывая свойства из торгового окружения. Когда *cached* равно *true*, объект вернет предварительно сохраненные значения из внутренних массивов.

```
   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
   }
   ...
```

По умолчанию, кэширование, разумеется, включено.

Мы должны предусмотреть и метод, непосредственно выполняющий кэширование (заполнение массивов). Для этого достаточно вызвать метод *cache* у объекта *state*.

```
bool update()
{
   if(refresh())
   {
      cached = false; // отключаем чтение из кэша
      state.cache();  // читаем реальные свойства и записываем в кэш
      cached = true;  // включаем обратно внешний доступ к кэшу
      return true;
   }
   return false;
}
```

Но что такое метод *refresh*?

Дело в том, что до сих пор мы использовали объекты-мониторы в простом режиме: создавали, читали свойства и удаляли. При этом чтение свойств предполагает, что соответствующий ордер, сделка или позиция были выбраны в торговом контексте (внутри конструктора). Поскольку сейчас мы совершенствуем мониторы, привнося в них поддержку внутреннего состояния, необходимо обеспечить повторное выделение нужного элемента, чтобы прочитать свойства даже спустя неопределенное время (разумеется, с проверкой на то, что элемент еще существует). В связи с этим в шаблонный класс *MonitorInterface* и был добавлен виртуальный метод *refresh*.

```
// TradeBaseMonitor.mqh
template<typename I,typename D,typename S>
class MonitorInterface
{
   ...
   virtual bool refresh() = 0;
```

Он должен вернуть *true* при успешном выделении ордера, сделки или позиции. Если результат равен *false*, во встроенной переменной *_LastError* подразумевается наличие одной из ошибок:

- 4753 ERR_TRADE_POSITION_NOT_FOUND;
- 4754 ERR_TRADE_ORDER_NOT_FOUND;
- 4755 ERR_TRADE_DEAL_NOT_FOUND;

При этом переменная-член *ready*, которая сигнализирует доступность объекта, должна быть сброшена в *false* в реализациях этого метода в производных классах.

Например, в конструкторе *PositionMonitor* у нас была и остается такая инициализация. В мониторах ордеров и сделок ситуация похожа.

```
// 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;
      }
   }
   ...
```

Теперь мы добавим во все конкретные классы метод *refresh* такого вида (на примере *PositionMonitor*):

```
// PositionMonitor.mqh
   virtual bool refresh() override
   {
      ready = PositionSelectByTicket(ticket);
      return ready;
   }
```

Но заполнение массивов кэша значениями свойств еще полдела. Вторая половина заключается в сравнении этих значений с актуальным состоянием ордера, сделки или позиции.

Для выявления различий и записи индексов изменившихся свойств в массив *changes* предназначен метод *getChanges* в создаваемом классе *TradeBaseState*. Метод возвращает *true* при обнаружении изменений.

```
template<typename M,typename I,typename D,typename S>
class TradeBaseState: public M
{
   ...
   bool getChanges(int &changes[])
   {
      const bool previous = ready;
      if(refresh())
      {
         // элемент выбран в торговом окружении = можно читать и сравнивать свойства
         cached = false;    // читаем напрямую
         const bool result = M::diff(state, changes);
         cached = true;     // включаем обратно кэш по умолчанию
         return result;
      }
      // перестал быть "готовым" = скорее всего, удален
      return previous != ready; // если удален только что, это тоже изменение
   }
```

Как видно, основная работа поручается некоему методу *diff* в классе M. Это новый метод: нужно его написать. К счастью, благодаря ООП, можно сделать это единожды в базовом шаблоне *MonitorInterface*, и метод появится сразу для ордеров, сделок и позиций.

```
// 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;
   }
```

Итак, все готово для формирования конкретных кэширующих классов для ордеров, сделок и позиций. Например, позиции будут храниться в расширенном мониторе *PositionState* на базе *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) { }
};
```

Аналогичным образом в файле *TradeState.mqh* определен кэширующий класс для сделок.

```
class DealState: public TradeBaseState<DealMonitor,
   ENUM_DEAL_PROPERTY_INTEGER,
   ENUM_DEAL_PROPERTY_DOUBLE,
   ENUM_DEAL_PROPERTY_STRING>
{
public:
   DealState(const long t): TradeBaseState(t) { }
};
```

С ордерами все немного сложнее, потому что они могут быть действующими и историческими. У нас до сих пор был один универсальный класс монитора для ордеров *OrderMonitor*. Он пытается найти переданный тикет ордера и среди активных ордеров, и в истории. Для кэширования такой подход не подойдет, потому что в экспертах требуется отслеживать переход ордера из одного состояния в другое.

В связи с этим в файл *OrderMonitor.mqh* добавлены 2 более конкретных класса: *ActiveOrderMonitor* и *HistoryOrderMonitor*.

```
// OrderMonitor.mqh
class ActiveOrderMonitor: public OrderMonitor
{
public:
   ActiveOrderMonitor(const ulong t): OrderMonitor(t)
   {
      if(history) // если ордер в истории, то он уже неактивен
      {
         ready = false;   // сбрасываем флаг готовности
         history = false; // это объект только для активных ордеров по определению
      }
   }

   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; // работаем только с историей
      ready = historyOrderSelectWeak(ticket);
      return ready; // готовность определяется наличием тикета в истории
   }
};
```

Каждый из них ищет тикет только в своей области. На основе этих мониторов уже можно создать кэширующие классы.

```
// 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) { }
};
```

Последний штрих, который мы добавим для удобства в класс *TradeBaseState* — особый метод для преобразования значения свойства в строку. Хотя в мониторе есть несколько версий методов *stringify*, все они будут "печатать" либо значения из кэша (если переменная-член *cached* равна *true*), либо из оригинального объекта торгового окружения (если *cached* равно *false*). Нам же для визуализации отличий кэша от измененного объекта (когда эти отличия обнаружатся) нужно одновременно прочитать и значение из кэша, и минуя кэш. В связи с этим добавим метод *stringifyRaw*, всегда работающий со свойством напрямую (за счет того, что переменная *cached* временно сбрасывается и устанавливается обратно).

```
   // получить строковое представление свойства 'i' минуя кэш
   string stringifyRaw(const int i)
   {
      const bool previous = cached;
      cached = false;
      const string s = stringify(i);
      cached = previous;
   }
```

Проверим работоспособность кэширующего монитора на простом примере эксперта, отслеживающего состояние действующего ордера (*OrderSnapshot.mq5*). Позднее мы разовьем данную идею для кэширования любой совокупности ордеров, сделок или позиций, то есть создадим полноценный кэш.

Эксперт будет пытаться найти последний в списке действующих ордеров и создавать для него объект *OrderState*. Если ордеров нет, пользователь получит предложение создать ордер или открыть позицию (последнее сопряжено с постановкой и исполнением ордера по рынку). Как только ордер обнаружен, для него в обработчике *OnTrade* производится проверка на изменение состояния. Эксперт продолжит контролировать этот ордер, пока не будет выгружен.

```
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;
   // указатель на объект хранится в статическом AutoPtr
   static AutoPtr<OrderState> auto;
   // получаем "чистый" указатель (чтобы не разыменовывать auto[] везде)
   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(); // начальное состояние на момент "захвата" ордера
   }
   else if(state)
   {
      int changes[];
      if(state.getChanges(changes))
      {
         Print("Order properties changed:");
         ArrayPrint(changes);
         ...
      }
      if(_LastError != 0) Print(E2S(_LastError));
   }
}
```

В дополнение к выводу массива изменившихся свойств неплохо бы отобразить сами изменения. Поэтому вместо многоточия добавим такой фрагмент (он нам пригодится и в будущих классах полноценных кэшей).

```
      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;
         }
      }
```

Здесь нам уже пригодился новый метод *stringifyRaw*. После отображения изменений не забываем обновить состояние кэша.

```
      state.update();
```

Если запустить эксперт на счете без действующих ордеров и выставить новый, увидим в журнале следующие записи (в данном случае создавался *buy limit* для EURUSD, ниже текущей рыночной цены).

```
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)
```

Здесь видно, как статус ордера изменился со STARTED на PLACED. Если бы мы вместо отложенного ордера открылись по рынку мелким объемом, этих изменений могли бы не успеть получить, потому что такие ордера, как правило, очень быстро устанавливаются, и их наблюдаемый статус меняется со STARTED сразу на FILLED. А последнее уже означает, что ордер перенесен в историю. Поэтому для их отслеживания требуется параллельный мониторинг истории. Мы покажем это в следующем примере.

Обратите внимание, что событий *OnTrade* может быть много, но они не все связаны с нашим ордером.

Попробуем задать уровень *Take Profit* и посмотрим в журнал.

```
>>> OnTrade(5)
Order properties changed:
10 13
ORDER_PRICE_CURRENT: 1.09073 -> 1.09079
ORDER_TP: 0.0 -> 1.097
>>> OnTrade(6)
>>> OnTrade(7)
```

Далее поменяем срок истечения: с GTC до одного дня.

```
>>> 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)
```

Здесь в процессе изменения нашего ордера цена успела измениться, и потому мы "зацепили" промежуточное уведомление о новом значении в ORDER_PRICE_CURRENT. И только после этого в журнал попали ожидаемые перемены в ORDER_TYPE_TIME и ORDER_TIME_EXPIRATION.

Далее мы удалили ордер.

```
>>> OnTrade(12)
TRADE_ORDER_NOT_FOUND
```

Теперь при любых действиях со счетом, которые приводят к событиям *OnTrade*, наш эксперт будет выводить TRADE_ORDER_NOT_FOUND, потому что он рассчитан на отслеживание одного единственного ордера. Если эксперт перезапустить, он "подхватит" другой ордер при его наличии. Но мы остановим эксперт и займемся приготовлениями к решению более насущной задачи.

Кэшировать и контролировать изменения, как правило, требуется не для отдельного ордера или позиции, а для всех или их набора, отобранного по некоторым условиям. Для этих целей разработаем базовый класс-шаблон *TradeCache* (*TradeCache.mqh*) и на его основе — прикладные классы для списков ордеров, сделок и позиций.

```
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); // будем переопределять в производных классах для нагляд
   }
   ...
```

В данном шаблоне буквой T обозначен один из классов семейства *TradeState*. Как видно, массив таких объектов в виде авто-указателей зарезервирован под именем *data*.

Буква F описывает тип одного из классов-фильтров (*OrderFilter.mqh*, включая *HistoryOrderFilter*, *DealFilter.mqh*, *PositionFilter.mqh*), используемых для отбора кэшируемых элементов. В простейшем случае, когда в фильтре нет *let*-условий, будут кэшироваться все элементы (с учетом выборки истории для объектов из истории).

Буква E соответствует перечислению, в котором находится свойство *property*, идентифицирующее объекты. Поскольку обычно это свойство — КАКОЙ-ТО_TICKET, в качестве перечисления предполагается целочисленный ENUM_ЧТО-ТО_PROPERTY_INTEGER.

Переменная NOT_FOUND_ERROR предназначена для кода ошибки, возникающей при попытке выделить для чтения не существующий объект, например, ERR_TRADE_POSITION_NOT_FOUND для позиций.

Главный метод класса *scan* принимает в качестве параметра ссылку на настроенный фильтр (его настройкой должен заняться вызывающий код).

```
   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);
      }
      ...
```

В начале метода мы собираем идентификаторы уже кэшированных объектов в массив *tickets*. Очевидно, что при первом запуске он окажется пустым.

Далее заполняем другой массив *objects* тикетами актуальных объектов с помощью фильтра. Для каждого нового тикета создаем объект кэширующего монитора T и добавляем в массив *data*. Для старых объектов анализируем наличие изменений путем вызова *data[j][].getChanges(changes)* и затем обновляем кэш, вызвав *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; // помечаем как найденный
         break;
      }
   }

   if(j == existedBefore) // такого в кэше нет, надо добавить
   {
      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(_LastErrc
   }
}
...
```

Как нетрудно заметить, в каждой фазе изменения — то есть при добавлении объекта или после его изменения — вызываются некие методы *onAdded* и *onUpdated*. Это виртуальные методы-заглушки, с помощью которых сканирование может уведомить программу о соответствующих событиях. Предполагается, что прикладной код реализует класс-наследник с переопределенными версиями этих методов. Мы коснемся этого вопроса чуть ниже, а пока продолжим рассматривать метод *scan*.

В вышеприведенном цикле все найденные тикеты в массиве *tickets* обнулены, и потому оставшиеся элементы соответствуют отсутствующим объектам торговой среды. Далее делается их проверка путем вызова *getChanges* и сравнением кода ошибки с NOT_FOUND_ERROR. Если это действительно так, вызывается еще один виртуальный метод *onRemoved*. Он возвращает логический флаг (проставляемый вашим прикладным кодом), следует ли удалить элемент из кэша.

```
         for(int j = 0; j < existedBefore; ++j)
         {
            if(tickets[j] == 0) continue; // пропускаем обработанные элементы

            // этот тикет не нашелся, скорее всего удален
            int changes[];
            ResetLastError();
            if(data[j][].getChanges(changes))
            {
               if(_LastError == NOT_FOUND_ERROR) // например, ERR_TRADE_POSITION_NOT_FOUND
               {
                  if(onRemoved(data[j][]))
                  {
                     data[j] = NULL;             // освобождаем объект и элемент массива
                  }
                  continue;
               }

               // NB! обычно мы не должны сюда проваливаться
               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());
            }
         }
      }
```

В самом конце метода *scan* массив *data* очищается от нулевых элементов, но здесь данный фрагмент опущен для краткости.

Базовый класс предоставляет стандартные реализации методов *onAdded*, *onRemoved*, *onUpdated*, которые выводят суть событий в журнал. Определив макрос PRINT_DETAILS в своем коде до включения заголовочного файла *TradeCache.mqh*, можно заказать распечатку всех свойств каждого нового объекта.

```
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; // разрешаем удалить объект из кэша (false, чтобы сохранить)
}

virtual void onUpdated(T &state, const int &changes[],
   const bool unexpected = false)
{
   ...
}
```

Метод *onUpdated* не будем приводить, поскольку он практически повторяет код для вывода изменений из эксперта *OrderSnapshot.mq5*, показанный выше.

Разумеется, в базовом классе есть средства для получения размера кэша и доступа к конкретному объекту по номеру.

```
int size() const
{
   return ArraySize(data);
}

T *operator[](int i) const
{
   return data[i][]; // возвращаем указатель (T*) из объекта AutoPtr
}
```

На основе базового класса *TradeCache* легко создать конкретные классы для кэширования списков позиций, действующих ордеров и ордеров из истории. Кэширование сделок оставлено в качестве самостоятельного задания.

```
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) { }
};
```

Чтобы подвести итог процессу разработки представленного функционала приведем диаграмму основных классов. Это упрощенный вариант диаграмм UML, которые имеет смысл взять на вооружение при проектировании сложных программ на MQL5.

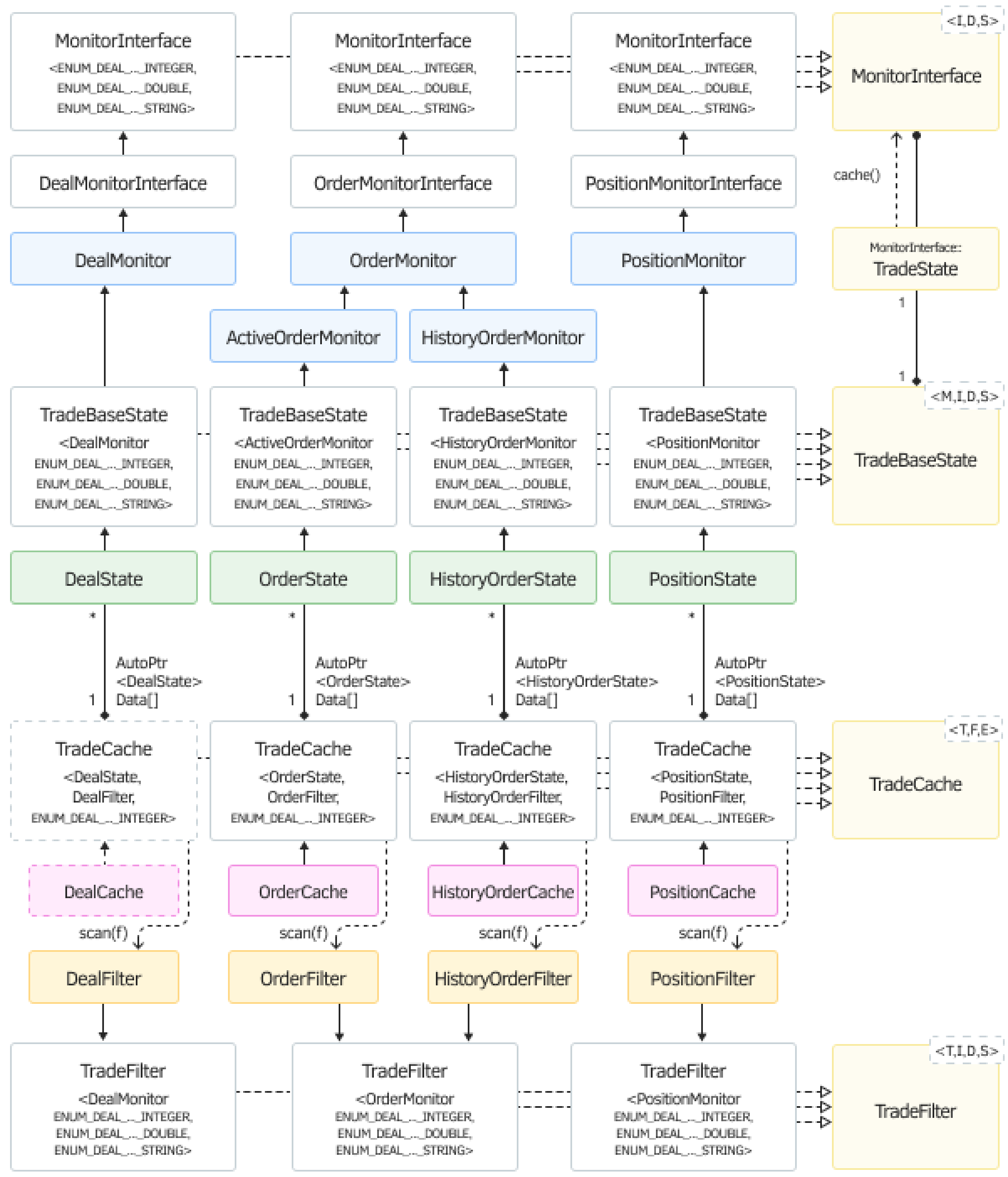


Диаграмма классов мониторов, фильтров и кэшей торговых объектов

Желтым цветом обозначены шаблоны, белым оставлены абстрактные классы, цветные — конкретные реализации. Сплошные стрелки с закрашенными наконечниками обозначают наследование, пунктирные с полыми — типизацию шаблонов. Пунктирные стрелки с открытыми наконечниками — это использование классами указанных методов друг друга. Связи с ромбами — это композиция (включение одних объектов в другие).

В качестве примера использования кэша создадим эксперт *TradeSnapshot.mq5*, который будет реагировать на любые изменения торгового окружения из обработчика *OnTrade*. Для фильтрации и кэширования в коде описано 6 объектов, по 2 (фильтр и кэш) для каждого типа элементов — позиций, действующих ордеров и исторических ордеров.

```
PositionFilter filter0;
PositionCache positions;

OrderFilter filter1;
OrderCache orders;

HistoryOrderFilter filter2;
HistoryOrderCache history;
```

Фильтрам не задается каких-либо условий через вызовы метода *let*, так что в кэш попадут все обнаруженные онлайн объекты. Для ордеров из истории существует дополнительная настройка.

Опционально при запуске можно загрузить в кэш прошлые ордера на заданную глубину истории. Для этого предусмотрена входная переменная *HistoryLookup*. В ней можно выбрать последние сутки, последнюю неделю (по длительности, а не календарную), месяц (30 дней) или год (360 дней). По умолчанию прошлая история не подгружается (точнее, подгружается только за 1 секунду). Поскольку в эксперте определен макрос PRINT_DETAILS, будьте осторожны со счетами, где большая история: они могут сгенерировать объемный журнал, если не ограничить период.

```
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;
```

В обработчике *OnInit* сбрасываем кэши (на тот случай, если эксперт перезапущен с новыми параметрами), вычисляем начальную дату истории в переменной *origin* и вызываем сами *OnTrade* первый раз.

```
int OnInit()
{
   positions.reset();
   orders.reset();
   history.reset();
   origin = HistoryLookup ? TimeCurrent() - HistoryLookup : 0;

   OnTrade(); // само-запуск
   return INIT_SUCCEEDED;
}
```

Обработчик *OnTrade* довольно минималистичен, поскольку все сложности теперь скрыты внутри классов.

```
void OnTrade()
{
   static int count = 0;

   PrintFormat(">>> OnTrade(%d)", count++);
   positions.scan(filter0);
   orders.scan(filter1);
   // делаем выборку истории непосредственно перед использованием фильтра
   // внутри метода 'scan'
   HistorySelect(origin, LONG_MAX);
   history.scan(filter2);
   PrintFormat(">>> positions: %d, orders: %d, history: %d",
      positions.size(), orders.size(), history.size());
}
```

Сразу после запуска эксперта на чистом счете увидим сообщение:

```
>>> OnTrade(0)
>>> positions: 0, orders: 0, history: 0
```

Попробуем выполнить простейший тест-кейс: совершим покупку или продажу на "пустом" счете без открытых позиций и отложенных ордеров. Журнал зафиксирует следующие события (происходящие практически моментально).

Сначала обнаружится активный ордер.

```
>>> 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=
```

Затем этот ордер будет перемещен в историю (при этом поменяются, как минимум, статус, время исполнения и идентификатор позиции).

```
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
```

Обратите внимание, что данные модификации произошли в течение одного вызова *OnTrade*. Иными словами, пока наша программа анализировала свойства нового ордера (с помощью вызова *orders.scan*), ордер параллельно был обработан терминалом, и к моменту проверки истории (с помощью вызова *history.scan*), уже попал в историю. Именно поэтому он числится и там, и там согласно последней строке этого фрагмента лога. Такое поведение нормально для многопоточных программ, и это следует учитывать при их разработке. Но оно не обязательно всегда будет проявляться. Здесь мы просто обращаем на это внимание. При быстром выполнении MQL-программы такую ситуацию обычно не удается застать.

Если бы мы выполняли сначала проверку истории, а потом онлайн-ордеров, то могли бы на первом этапе обнаружить, что ордера еще нет в истории, а на втором — что ордера уже нет онлайн. То есть он на какое-то мгновение мог бы теоретически потеряться. Или, более реальна ситуация — за счет синхронизации истории пропустить ордер в его активной фазе, то есть зафиксировать первый раз сразу в истории.

Напомним, что MQL5 не позволяет синхронизировать торговое окружение целиком, а лишь по частям:

- среди активных ордеров актуальна информация для ордера, для которого только что была вызвана функция *OrderSelect* или *OrderGetTicket*;
- среди позиций актуальна информация для позиции, для которой только что была вызвана функция *PositionSelect*, *PositionSelectByTicket* или *PositionGetTicket*;

- для ордеров и сделок истории доступна информация в контексте последнего вызова *HistorySelect*, *HistorySelectByPosition*, *HistoryOrderSelect*, *HistoryDealSelect*.

Кроме того, напомним, что торговые события (как и любые события MQL5) — это сообщения о произошедших изменениях, помещенные в очередь и извлеченные из очереди отложенным образом, а не непосредственно в момент совершения изменений. Тем более, событие *OnTrade* происходит после соответствующих событий *OnTradeTransaction*.

Пробуйте разные конфигурации программы, проводите отладку и формируйте подробные логи, чтобы выбрать наиболее надежный алгоритм для вашей торговой системы.

Вернемся к нашему логу. При следующем срабатывании *OnTrade* ситуация уже исправлена: кэш активных ордеров обнаружил удаление ордера. Попутно кэш позиций "увидел" открытую позицию.

```
>>> 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
```

Спустя некоторое время закроем позицию. Поскольку у нас в коде первым проверяется кэш позиций (*positions.scan*), в журнал попадают изменения закрываемой позиции.

```
>>> OnTrade(8)
PositionCache changed: 1311792104
POSITION_PRICE_CURRENT: 1.09214 -> 1.09222
POSITION_PROFIT: -0.04 -> 0.04
```

Далее в этом же вызове *OnTrade* засекается появление ордера на закрытие и его моментальный перенос в историю (опять же за счет его быстрой параллельной обработки терминалом).

```
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
```

В кэше истории числятся уже 2 ордера, но кэши позиций и активных ордеров, которые анализировались раньше кэша истории, еще не применили эти изменения.

Но уже в следующем событии *OnTrade* мы видим, что позиция закрыта и рыночный ордер исчез.

```
>>> OnTrade(9)
PositionCache removed: 1311792104
OrderCache removed: 1311796883
>>> positions: 0, orders: 0, history: 2
```

Если мониторить кэши на каждом тике (или раз в секунду, но не только по событиям OnTrade), мы увидим изменения свойств ORDER_PRICE_CURRENT и POSITION_PRICE_CURRENT "на лету". Также будет меняться POSITION_PROFIT.

Наши классы не обладают *персистентностью*, то есть "живут" только в оперативной памяти и не умеют сохранять и восстанавливать свое состояние в каких-либо долговременных хранилищах, таких как файлы. Это означает, что программа может пропустить изменение, которое произошло между сеансами работы в терминале. Если вам необходим такой функционал, его следует реализовать самостоятельно. В будущем, в 7-й Части книги мы рассмотрим встроенную в MQL5 поддержку базы данных SQLite, которая предоставляет наиболее эффективный и удобный способ для хранения кэша торгового окружения и подобных табличных данных.

6.4.38 Особенности создания мультисимвольных экспертов

До сих пор в рамках книги мы в основном разбирали примеры экспертов, торгующих на текущем рабочем символе графика. Однако MQL5 позволяет формировать торговые приказы для любых символов *Обзора рынка*, вне зависимости от рабочего символа графика.

На самом деле, многие примеры предыдущих разделов имели входной параметр *Symbol*, в котором можно задать произвольный символ. По умолчанию, там находится пустая строка, что трактуется как текущий символ графика. Напомним эти примеры и их разделы:

- *CustomOrderSend.mq5* — Отправка торгового запроса;
- *MarketOrderSend.mq5* — Совершение покупки или продажи;
- *MarketOrderSendMonitor.mq5* — Функции для чтения свойств действующих ордеров;
- *PendingOrderSend.mq5* — Установка отложенного ордера;
- *PendingOrderModify.mq5* — Модификация отложенного ордера;
- *PendingOrderDelete.mq5* — Удаление отложенного ордера;

Вы можете попробовать запустить эти примеры с чужим символом для графика и убедиться, что торговые операции выполняются точно также, как и с родным символом.

Более того, как мы видели в описании событий *OnBookEven* и *OnTradeTransaction*, они являются универсальными, сообщающими об изменениях торгового окружения, касающихся произвольных символов. Но этого нельзя сказать о событии *OnTick* — оно генерируется только по факту изменения новых цен текущего символа. Как правило, это не является проблемой, но для

высокочастотной мультивалютной торговли требуется предпринять какие-либо дополнительные технические приемы, например, подписаться на события *OnBookEvent* для "чужих" символов или установить высокочастотный таймер. Еще один вариант обхода этого ограничения в виде индикатора-шпиона *EventTickSpy.mq5* был представлен в разделе Генерация пользовательских событий.

В контексте разговора о поддержке мультисимвольной торговли следует отметить, что похожее понятие мультитаймфреймовых экспертов не совсем корректно. Дело в том, что торговля по открытию баров представляет собой лишь частный случай группировки тиков по произвольным периодам, не обязательно стандартным. Конечно, анализ появления нового бара на конкретном таймфрейме упрощен ядром системы за счет функций вроде *iTime(_Symbol, PERIOD_XX, 0)*, но этот анализ в любом случае отталкивается от тиков.

Вы можете строить внутри своего эксперта виртуальные бары по количеству тиков (эквиобъемные), по размаху цен (ренко, рейндж) и так далее. В некоторых случаях, в том числе и для наглядности, имеет смысл такие "таймфреймы" генерировать явным образом вне эксперта — в виде пользовательских символов, но данный подход имеет свои ограничения: о них мы поговорим в соответствующем разделе, уже в следующей Части книги.

Однако если торговая система все же требует анализа котировок по открытию баров или использует мультивалютный индикатор, следует тем или иным образом дожидаться синхронизации баров на всех задействованных инструментах. Мы приводили пример класса, выполняющего эту задачу, в разделе Отслеживание формирования баров.

При разработке мультисимвольного эксперта на первый план выходит задача выделения универсального торгового алгоритма в некий блок, который можно затем применять для разных символов и при разных настройках. Разумеется, наиболее логичный способ сделать это — описать один или несколько классов в рамках концепции ООП.

Продемонстрируем данный подход на примере эксперта по известной стратегии мартингейла. Как известно, это довольно рискованная стратегия из-за удвоения лотов после каждого убыточного трейда в надежде компенсировать прошлые потери. И один из самых простых способов нивелирования этого риска — ведение торговли сразу по нескольким, желательно слабо коррелированным, символам. Тогда временные просадки на одном инструменте могут компенсироваться доходами на других.

Чем больше различных инструментов (или разных настроек одной торговой системы, или даже разных торговых систем) мы используем в эксперте, тем меньше будет зависимость общего результата от отдельных неудач её составных частей.

Новый эксперт назовем *MultiMartingale.mq5*. Настройки торгового алгоритма включают:

- *UseTime* — логический флаг включения/отключения торговли по расписанию;
- *HourStart* и *HourEnd* — диапазон часов, в пределах которого разрешена торговля, если *UseTime* равно *true*;
- *Lots* — объем первой сделки в серии;
- *Factor* — коэффициент увеличения объема для последующих сделок после убытка;
- *Limit* — максимальное количество сделок в убыточной серии с умножением объемов (после него возврат к начальному лоту);
- *StopLoss* и *TakeProfit* — дистанция до защитных уровней в пунктах;
- *StartType* — тип первой сделки (покупка или продажа);

- *Trailing* — признак сопровождения стоп-лосса.

В исходном коде они описаны таким образом.

```
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
```

В принципе, защитные уровни имеет смысл задавать не в пунктах, а в долях ATR (индикатор Average True Range), но сейчас это не приоритетная задача.

Помимо прочего, в эксперте предусмотрен механизм заморозки торговых операций на заданный пользователем период (параметр *SkipTimeOnError*) в случае ошибок. Здесь мы опустим данный аспект — с ним можно ознакомиться в исходных кодах.

Для хранения всей группы настроек как единого целого описана структура *Settings* с полями, аналогичными входным переменным. Плюс ко всему в структуре имеется и поле *symbol*, в силу мультивалютности стратегии (т.е. символ может быть произвольным, отличным от рабочего символа графика).

```
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;
   ...
};
```

На первом этапе разработки мы будем заполнять структуру из входных переменных. Однако этого достаточно только для торговли на одном символе. Позднее, когда мы приступим к масштабированию алгоритма на несколько символов, нам потребуется считывать несколько вариантов настроек (неким другим способом) и складывать их в массив структур.

Структура также имеет несколько полезных методов. В частности, метод *validate* проверяет корректность настроек, включая существование указанного символа, и возвращает признак успеха (*true*).

```
struct Settings
{
   ...
   bool validate()
   {
      ... // проверки размера лота и защитных уровней (см. исходный код)

      double rates[1];
      const bool success = CopyClose(symbol, PERIOD_CURRENT, 0, 1, rates) > -1;
      if(!success)
      {
         Print("Unknown symbol: ", symbol);
      }
      return success;
   }
   ...
};
```

Вызов *CopyClose* не только проверяет наличие символа в *Обзоре рынка* онлайн, но и инициирует подгрузку его котировок (нужного таймфрейма) и тиков в тестере. Если этого не сделать, в тестере по умолчанию доступны котировки и тики (в режиме реальных тиков) только текущего выбранного инструмента и таймфрейма. Поскольку мы пишем мультивалютный эксперт, нам потребуются сторонние котировки и тики.

```
struct Settings
{
   ...
   void print() const
   {
      Print(symbol, (startType == POSITION_TYPE_BUY ? "+" : "-"), (float)lots,
        "*", (float)factor,
        "^", limit,
        "(", stopLoss, ",", takeProfit, ")",
        useTime ? "[" + (string)hourStart + "," + (string)hourEnd + "]": "");
   }
};
```

Метод *print* в сокращенном виде одной строкой выводит в журнал совокупность всех полей. Например,

```
EURUSD+0.01*2.0^5(500,1000)[2,22]
|      | |   |   |  |     |   |  |
|      | |   |   |  |     |   |  `до этого часа торговля разрешена
|      | |   |   |  |     |   `от этого часа торговля разрешена
|      | |   |   |  |     `тейк-профит в пунктах
|      | |   |   |  `стоп-лосс в пунктах
|      | |   |   `максимальный размер серии убыточных сделок (после '^')
|      | |   `фактор умножения лотов (после '*')
|      | `начальный лот в серии
|      `+ начинаем с покупки
|      `- начинаем с продажи
`инструмент
```

Другие методы в структуре *Settings* потребуются нам, когда мы перейдем к мультивалютности. Пока представим себе упрощенный вариант того, как мог бы выглядеть обработчик *OnInit* эксперта, торгующего на одном символе.

```
int OnInit()
{
   Settings settings =
   {
      UseTime, HourStart, HourEnd,
      Lots, Factor, Limit,
      StopLoss, TakeProfit,
      StartType, Magic, SkipTimeOnError, Trailing, _Symbol
   };
   
   if(settings.validate())
   {
      settings.print();
      ...
      // здесь нужно будет инициализировать торговый алгоритм с этими настройками
   }
   ...
}
```

Придерживаясь ООП, торговую систему в обобщенном виде следует описать как программный интерфейс. Опять же в целях упрощения примера ограничимся единственным методом в этом интерфейсе — *trade*.

```
interface TradingStrategy
{
   virtual bool trade(void);
};
```

В конце концов, основная задача алгоритма — торговать, и даже не важно, откуда мы затем решим вызывать данный метод — на каждом тике из *OnTick*, на открытии бара или, может быть по таймеру.

В ваших рабочих экспертах, скорее всего, потребуются дополнительные методы интерфейса для настройки и поддержки различных режимов, но здесь они не нужны.

На базе интерфейса начнем создавать класс конкретной торговой системы. В нашем случае все экземпляры будут одного класса *SimpleMartingale*, но в перспективе никто не мешает внутри

одного эксперта реализовать множество различных классов-наследников интерфейса и затем единообразным образом пускать их в дело в произвольном сочетании. Портфель стратегий (желательно, сильно различающихся по своей сути), как правило, характеризуется повышенной устойчивостью финансовых показателей.

```
class SimpleMartingale: public TradingStrategy
{
protected:
   Settings settings;
   SymbolMonitor symbol;
   AutoPtr<PositionState> position;
   AutoPtr<TrailingStop> trailing;
   ...
};
```

Внутри класса мы видим знакомую структуру с настройками *Settings* и монитор для рабочего символа *SymbolMonitor*. Кроме того нам потребуется контролировать наличие позиций и выполнять для них сопровождение уровня стоп-лосса, для чего заведены переменные с авто-указателями на объекты *PositionState* и *TrailingStop*. Авто-указатели позволяют нам в своем коде не заботиться о явном удалении объектов — это будет сделано автоматически при выходе управления из области определения или когда авто-указателю присваивается новый указатель.

> Класс *TrailingStop* является базовым, с наиболее простой реализацией сопровождения цены, от которого можно унаследовать массу более сложных алгоритмов, пример чего мы рассматривали в виде производного *TrailingStopByMA*. Поэтому для придания программе гибкости в перспективе желательно предусмотреть, чтобы вызывающий код мог передавать в стратегию свой специфический, настроенный объект "трала", производный от *TrailingStop*. Это можно сделать, например, передачей указателя в конструктор или сделав *SimpleMartingale* шаблонным (тогда класс "трала" будет задаваться параметром шаблона).
>
> Данный принцип ООП называется *внедрением зависимости* (*dependency injection*) и широко применяется наравне со многими другими, которые мы вскользь упомянули в разделе Теоретические основы ООП: композиция.

Настройки передаются в класс стратегии как параметр конструктора. Исходя из них, присваиваем все внутренние переменные.

```
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);

      // вычисляем максимальный лот в серии (после заданного количества умножений)
      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;
      }
      ...
```

Далее используем объект *PositionFilter* для поиска существующих "своих" позиций (по магику и символу). Если такая находится, создаем для неё объект *PositionState* и при необходимости объект *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);
         }
      }
   }
```

В методе *trade* оставим пока "за кадром" работу по расписанию (поля настроек *useTime*, *hourStart*, *hourEnd*) и обратимся непосредственно к торговому алгоритму.

Если позиций еще нет и не было, указатель *PositionState* будет нулевым, и нам нужно открыть длинную или короткую позицию в соответствии с выбранным направлением *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);
         }
      }
      ...
```

Здесь используются вспомогательные методы *openBuy* и *openSell* — мы до них доберемся через пару абзацев. Пока достаточно знать, что они возвращают номер тикета при успешном открытии или 0 в случае ошибки.

Если в объекте *position* уже есть информация о подопечной позиции, проверяем "жива" ли она с помощью вызова *refresh*, и в случае успеха (*true*) обновляем информацию о позиции вызовом *update*, а также сопровождаем стоп-лосс, если это было запрошено настройками.

```
      else // position[] != NULL
      {
         if(position[].refresh()) // позиция все еще существует?
         {
            position[].update();
            if(trailing[]) trailing[].trail();
         }
         ...
```

Если позиция закрылась, *refresh* вернет *false*, и мы окажемся в другой ветке *if*, где требуется открыть новую позицию: либо в прежнем направлении, если была зафиксирована прибыль, либо в противоположном, если получился убыток. Обратите внимание, что в кэше у нас остался слепок прежней позиции.

```
         else // позиция закрыта - нужно открыть новую
         {
            if(position[].get(POSITION_PROFIT) >= 0.0)
            {
               // сохраняем прежнее направление:
               // BUY в случае прибыльного предыдущего BUY
               // SELL в случае прибыльного предыдущего SELL
               if(position[].get(POSITION_TYPE) == POSITION_TYPE_BUY)
                  ticket = openBuy(settings.lots);
               else
                  ticket = openSell(settings.lots);
            }
            else
            {
               // увеличиваем лот в оговоренных пределах
               double lots = MathFloor((position[].get(POSITION_VOLUME) * settings.fa

               if(lotsLimit < lots)
               {
                  lots = settings.lots;
               }

               // меняем направление торговли:
               // SELL в случае предыдущего убыточного BUY
               // BUY в случае предыдущего убыточного SELL
               if(position[].get(POSITION_TYPE) == POSITION_TYPE_BUY)
                  ticket = openSell(lots);
               else
                  ticket = openBuy(lots);
            }
         }
      }
      ...
```

Наличие ненулевого тикета на данной завершающей стадии означает, что мы должны начать его контролировать с помощью новых объектов *PositionState* и *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;
   }
```

Теперь представим с некоторыми сокращениями метод *openBuy* (*openSell* во всем аналогичен). Его суть заключается в трех шагах:

- подготовка структуры *MqlTradeRequestSync* с помощью метода *prepare* (здесь не показан, в нем заполняются *deviation* и *magic*);
- отправка приказа с помощью вызова метода *request.buy*;
- проверка результата с помощью метода *postprocess* (здесь не показан, в нем вызывается *request.completed* и в случае ошибки начинается отсчет периода приостановки торговли в ожидании лучших условий);

```
   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;
   }
```

Обычно позиции будут закрываться по стоп-лоссу или тейк-профиту. Однако у нас поддерживается и работа по расписанию, которая может привести к закрытию. Вернемся в начало метода *trade* для знакомства с работой по расписанию.

```
virtual bool trade() override
{
   if(settings.useTime && !scheduled(TimeCurrent())) // время вне расписания?
   {
      // если есть открытая позиция - закроем её
      if(position[] && position[].isReady())
      {
         if(close(position[].get(POSITION_TICKET)))
         {
                               // по желанию проектировщика:
            position = NULL; // затираем кэш или можно было бы...
            // не делать это обнуление, то есть сохранить позицию в кэше,
            // чтобы перенести направление и лот следующего трейда в новую серию
         }
         else
         {
            position[].refresh(); // гарантируем сброс флага 'ready'
         }
      }
      return false;
   }
   ... // открытие позиций (было приведено выше)
}
```

Рабочий метод *close* во многом схож с *openBuy* — нет смысла его представлять. Еще один метод *scheduled* просто возвращает *true* или *false*, в зависимости от того, попадает ли текущее время в заданный диапазон рабочих часов (*hourStart*, *hourEnd*).

Итак, торговый класс готов. Но для мультивалютной работы потребуется создать несколько его экземпляров. Управлять ими будет класс *TradingStrategyPool*, в котором опишем массив указателей на *TradingStrategy* и методы для его пополнения: параметрический конструктор и *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;
   }
};
```

То, что пул сделан производным от интерфейса *TradingStrategy*, в принципе не обязательно, но позволяет в будущем упаковывать пулы стратегий в другие более крупные пулы стратегий и так далее. Метод *trade* просто вызывает аналогичный метод у всех объектов массива.

В глобальном контексте добавим автоуказатель на торговый пул, а в обработчике *OnInit* обеспечим его заполнение — для начала одной единственной стратегией (мультивалютностью займемся чуть позже).

```
AutoPtr<TradingStrategyPool> pool;

int OnInit()
{
   ... // инициализация настроек была приведена ранее
   if(settings.validate())
   {
      settings.print();
      pool = new TradingStrategyPool(new SimpleMartingale(settings));
      return INIT_SUCCEEDED;
   }
   else
   {
      return INIT_FAILED;
   }
   ...
}
```

Нетрудно догадаться, что для запуска торговли нам достаточно написать следующий небольшой обработчик *OnTick*.

```
void OnTick()
{
   if(pool[] != NULL)
   {
      pool[].trade();
   }
}
```

А как же быть с поддержкой мультивалютности?

Текущий набор входных параметров рассчитан лишь на один инструмент. Мы можем использовать это для тестирования и оптимизации эксперта на отдельном символе, но после того как оптимальные настройки найдены для всех символов, их нужно каким-то образом объединить и передать в алгоритм.

В данном случае применим наиболее простое решение. Чуть выше была представлена строка с описанием настроек, которую формирует метод *print* структуры *Settings*. Реализуем в структуре метод *parse*, который делает обратную операцию: по строке с описанием восстанавливает состояние полей. Кроме того, поскольку нам потребуется объединить несколько настроек для разных символов, договоримся, что они могут быть состыкованы в общую длинную строку через специальный символ разделитель, скажем ';'. Тогда для чтения объединенного набора настроек легко написать статический метод *parseAll*, который с помощью вызовов *parse* заполнит переданный по ссылке массив структур *Settings*. С полным исходным кодом методов можно ознакомиться в прилагаемом файле.

```
struct Settings
{
   ...
   bool parse(const string &line);
   void static parseAll(const string &line, Settings &settings[])
   ...
};
```

Например, следующая объединенная строка содержит настройки для трех символов.

```
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
```

Именно строки такого вида сможет парсить метод *parseAll*. Для ввода такой строки в эксперт опишем входную переменную *WorkSymbols*.

```
input string WorkSymbols = ""; // WorkSymbols (name±lots*factor^limit(sl,tp)[start,stop];...)
```

Если она пуста, эксперт будет работать с настройками из отдельных входных переменных, представленных ранее. Если же строка указана, обработчик *OnInit* заполнит пул торговых систем по результатам разбора этой строки.

```
int OnInit()
{
   if(WorkSymbols == "")
   {
      ... // работа с текущим одним символом, как ранее
   }
   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;
         // поддержим несколько систем на одном символе для счетов с хеджингом
         settings[i].magic = Magic + i;  // разный магик для каждой подсистемы
         pool[].push(new SimpleMartingale(settings[i]));
      }
   }
   return INIT_SUCCEEDED;
}
```

Следует иметь в виду, что длина входной строки ограничена в MQL5 250-ю символами. Более того, при оптимизации в тестере строки урезаются еще сильнее — до 63 символов. Поэтому для оптимизации одновременной торговли на большом количестве символов потребуется реализовать альтернативный вариант загрузки настроек, например, из текстового файла. Это легко сделать на основе той же входной переменной, если в ней задано имя файла, а не строка с настройками.

Этот подход реализован в упомянутом методе *Settings::parseAll*. Имя текстового файла для предоставления эксперту входной строки без ограничения длины решено устанавливать по

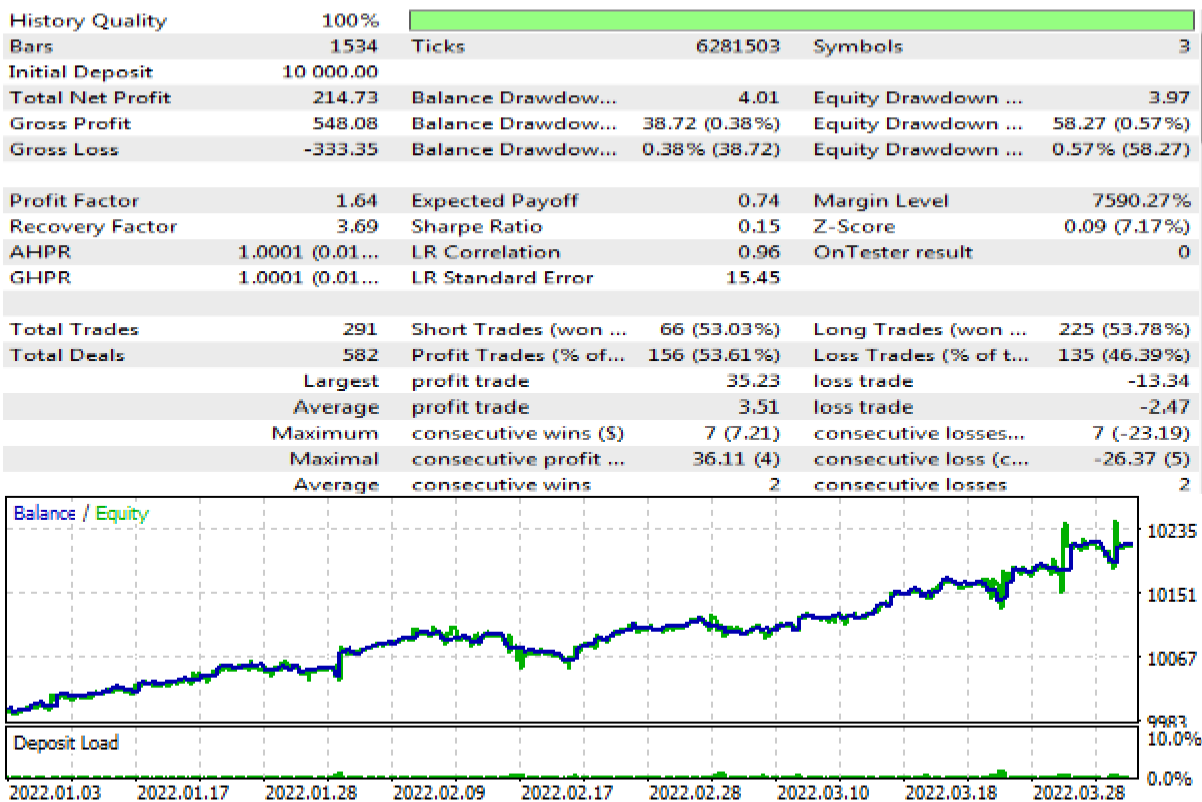
универсальному принципу, походящему для всех аналогичных случаев: имя файла начинается с имени эксперта и далее, после дефиса, должно идти имя переменной, данные для которой содержит файл. Например, в нашем случае во входной переменной *WorkSymbols* можно опционально указать имя файла "MultiMartingale-WorkSymbols.txt". Тогда метод *parseAll* попробует прочитать текст из файла (тот должен быть в стандартной "песочнице" *MQL5/Files*).

> Передача имен файлов во входных параметрах требует предпринять дополнительные действия для последующего тестирования и оптимизации такого эксперта: в исходный код следует добавить директиву *#property tester_file "MultiMartingale-WorkSymbols.txt"*. Подробно про это будет рассказано в разделе Директивы препроцессора для тестера. Когда данная директива добавлена, эксперт будет требовать наличия файла и не запустится без него в тестере!

Эксперт готов. Мы можем протестировать его на разных символах по отдельности, подобрать удачные настройки для каждого и собрать торговый портфель. В следующей Главе мы займемся изучением API тестера, включая оптимизацию, и наличие такого "подопытного кролика" придется весьма кстати. А пока проверим его мультивалютную работу на настройках, которые определились как бы сами собой.

```
WorkSymbols=EURUSD+0.01*1.2^4(300,600)[9,11];GBPCHF+0.01*2.0^7(300,400)[14,16];AUDJPY
```

На первом квартале 2022 года получим следующий отчет (в отчетах MetaTrader 5 не предусмотрены показатели в разбивке по символам, поэтому отличить одновалютный отчет от многовалютного возможно только по таблице сделок/ордеров/позиций).



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

Отчет тестера для мультивалютного эксперта по стратегии Мартингейла

Следует отметить, что из-за того, что стратегия запускается из обработчика *OnTick*, её прогоны на разных основных символах (то есть тех, что выбираются в выпадающем списке настроек тестера) будут давать слегка отличающиеся результаты. В нашем тесте мы просто использовали EURUSD, как наиболее ликвидный инструмент, по которому тики происходят наиболее часто, и для большинства применений этого достаточно. Однако если требуется реагировать на тики всех

инструментов, можно использовать индикатор вроде *EventTickSpy.mq5*. Или можно запускать торговую логику по таймеру, не привязываясь к тикам конкретного инструмента.

А вот как торговая стратегия выглядит для отдельно взятого символа, в данном случае AUDJPY.

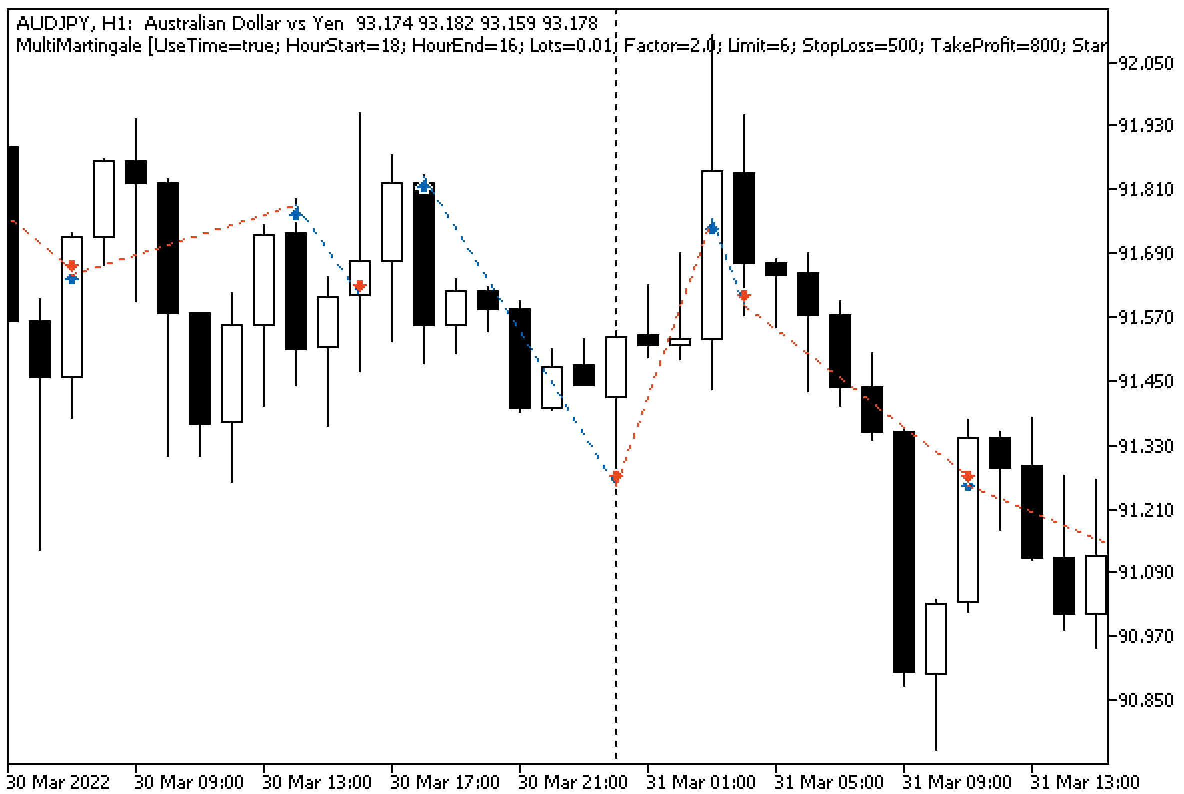


График с тестом мультивалютного эксперта по стратегии Мартингейла

Кстати говоря, для всех мультивалютных экспертов актуален другой важный вопрос, оставленный здесь без внимания. Речь о способе подбора размера лота, например, исходя из загрузки депозита или риска. Ранее мы показали примеры подобных расчетов в неторгующем эксперте *LotMarginExposureTable.mq5*. В эксперте *MultiMartingale.mq5* мы упростили задачу, выбрав фиксированный лот и выведя его в настройки для каждого символа. Однако в рабочих мультивалютных экспертах имеет смысл выбирать лоты пропорционально стоимости инструментов (по марже или волатильности).

В завершение обратим внимание, что мультивалютные стратегии могут требовать разных принципов оптимизации. Рассмотренная стратегия позволяет раздельно находить параметры для символов и затем совмещать их. Однако некоторые арбитражные и кластерные стратегии (например, парный трейдинг) построены на одновременном анализе всех инструментов для принятия торговых решений. В таком случае во входные параметры должны быть раздельно вынесены настройки, связанные со всеми символами.

6.4.39 Ограничения и преимущества экспертов

В силу своей специфики эксперты имеют некоторые ограничения, но также и преимущества по сравнению с другими типами MQL-программ. В частности в экспертах запрещены все функции, предназначенные для индикаторов:

- *SetIndexBuffer*;
- *IndicatorSetDouble*;

- *IndicatorSetInteger*;
- *IndicatorSetString*;
- *PlotIndexSetDouble*;
- *PlotIndexSetInteger*;
- *PlotIndexSetString*;
- *PlotIndexGetInteger*.

Также в экспертах не следует описывать обработчики событий, характерных для других типов программ: *OnStart* (скрипты и сервисы), *OnCalculate* (индикаторы).

Напомним, что в отличие от индикаторов, на каждом графике допускается размещать только один эксперт.

Вместе с тем, эксперты являются единственным типом MQL-программ, которые можно не только тестировать (что мы уже делали и для индикаторов, и для экспертов), но и оптимизировать, то есть находить лучшие входные параметры по различным критериям, как торговым, так и абстрактно-математическим. Для этих целей в API включены дополнительные функции и несколько специфических обработчиков событий. Мы изучим этот материал в следующей главе.

Кроме того, в экспертах (впрочем, как и в скриптах, и сервисах, то есть во всех типах программ кроме индикаторов) доступны группы встроенных функций MQL5 для работы с сетью на уровне сокетов и различных интернет протоколов (HTTP, FTP, SMTP). Их мы рассмотрим в заключительной 7-ой Части книги.

6.4.40 Создание заготовки эксперта в Мастере MQL

Итак, мы завершаем изучение торговых API для разработки экспертов. На протяжении этой главы было представлено немало примеров, которые можно использовать в качестве отправной точки для собственного проекта. Однако если требуется начать эксперт с нуля, не обязательно делать это буквально "с чистого листа". Напомним, что в редактор MetaEditor встроен Мастер MQL, позволяющий, кроме всего прочего, создать и заготовку эксперта. Причем в случае экспертов данный Мастер предлагает два разных способа генерации исходного кода.

Мы уже знакомились с первым шагом Мастера в разделе Мастер MQL и эскиз программы. Очевидно, что именно на первом шаге происходит выбор типа создаваемого проекта. Тогда мы создавали заготовку скрипта. Затем в главе про индикаторы мы провели экскурсию по созданию заготовки индикатора. Сейчас наше внимание будет обращено к первым двум вариантам:

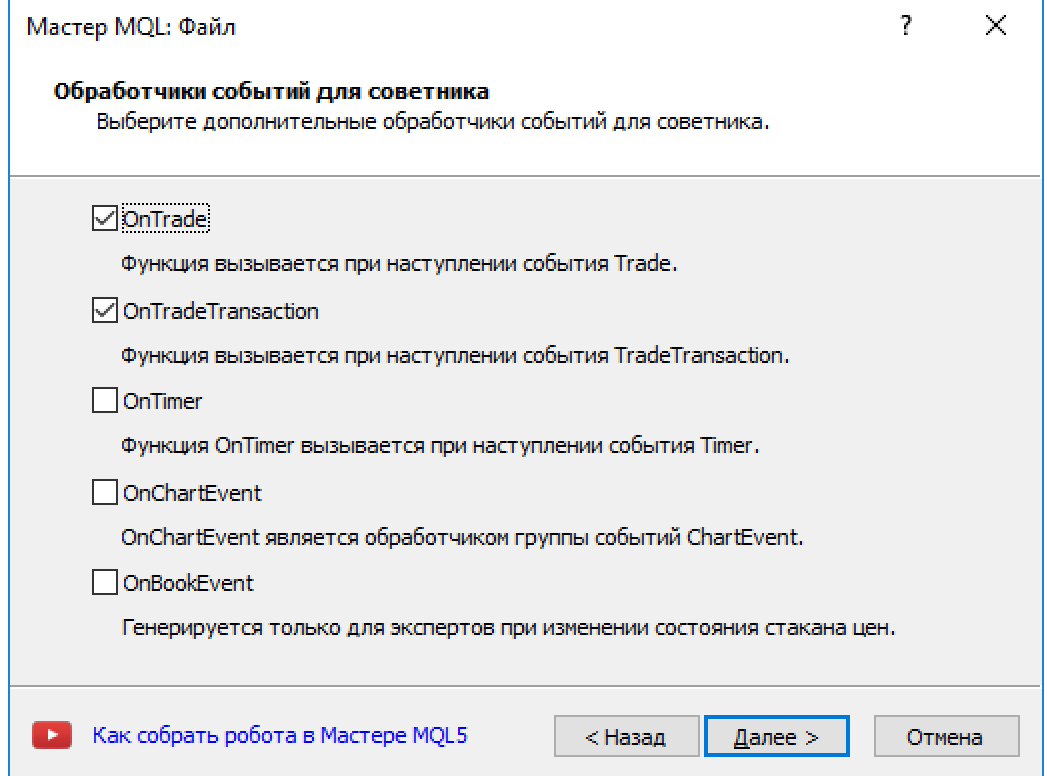
- Советник (шаблон)
- Советник (сгенерировать)

Первый из них — более простой. Он позволяет выбрать название, входные параметры и требуемые обработчики событий, — ниже приведена пара скриншотов, — но никакой торговой логики и готовых алгоритмов в полученном исходном файле не будет.

Второй вариант более сложный. Результатом его работы является готовый советник на основе стандартной библиотеки — набора классов в заголовочных файлах, поставляемых вместе с MetaTrader 5. Файлы располагаются в папках *MQL5/Include/Expert/*, *MQL5/Include/Trade*, *MQL5/Include/Indicators* и нескольких других. В классах библиотеки реализованы наиболее популярные сигналы индикаторов, механизмы выполнения торговых операций по комбинациям сигналов, а также алгоритмы мани-менеджмента и трейлинг-стопа. Рассмотрение стандартной библиотеки выходит за рамки данной книги.

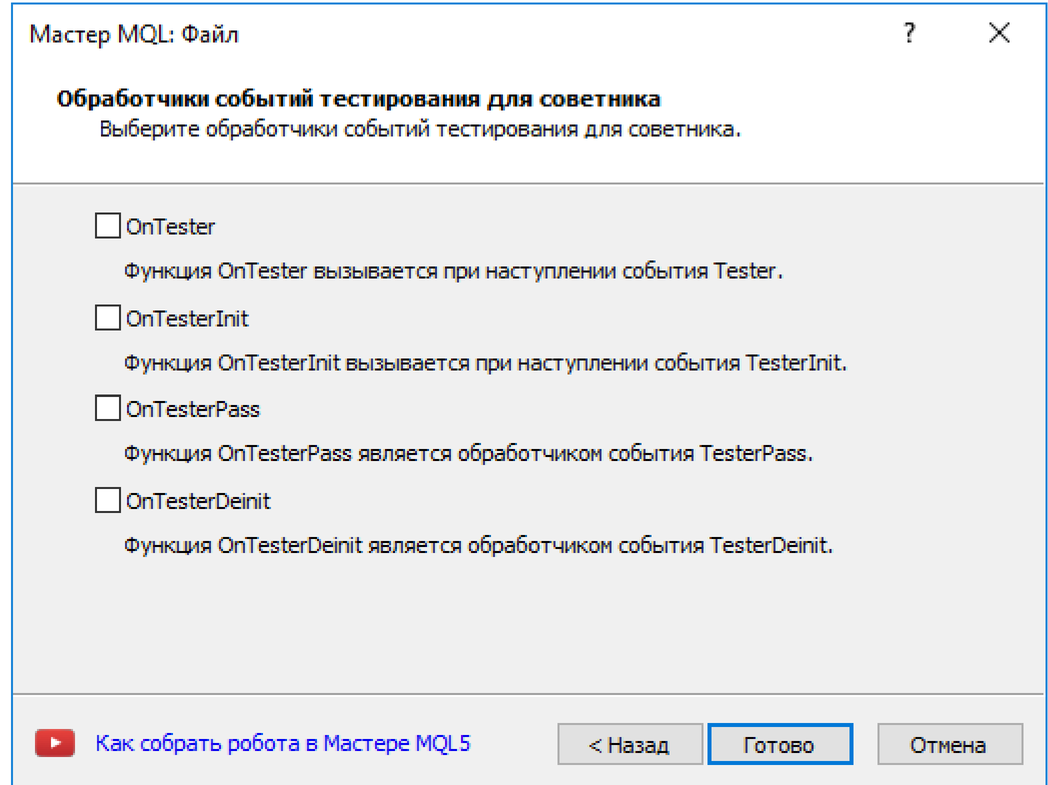
Вне зависимости от того, какой из вариантов генерации эксперта выбран, на втором шаге Мастера нужно ввести его название и входные параметры. Внешний вид этого шага аналогичен тому, что также был уже показан в разделе Мастер MQL и эскиз программы. Единственный нюанс заключается, в том, что советники на основе стандартной библиотеки должны иметь 2 обязательных (неудаляемых) параметра: *Symbol* и *TimeFrame*.

Для простого шаблона на 3-м шаге предлагается выбрать дополнительные обработчики событий, которые будут добавлены в исходный код, помимо *OnTick* (*OnTick* вставляется всегда).



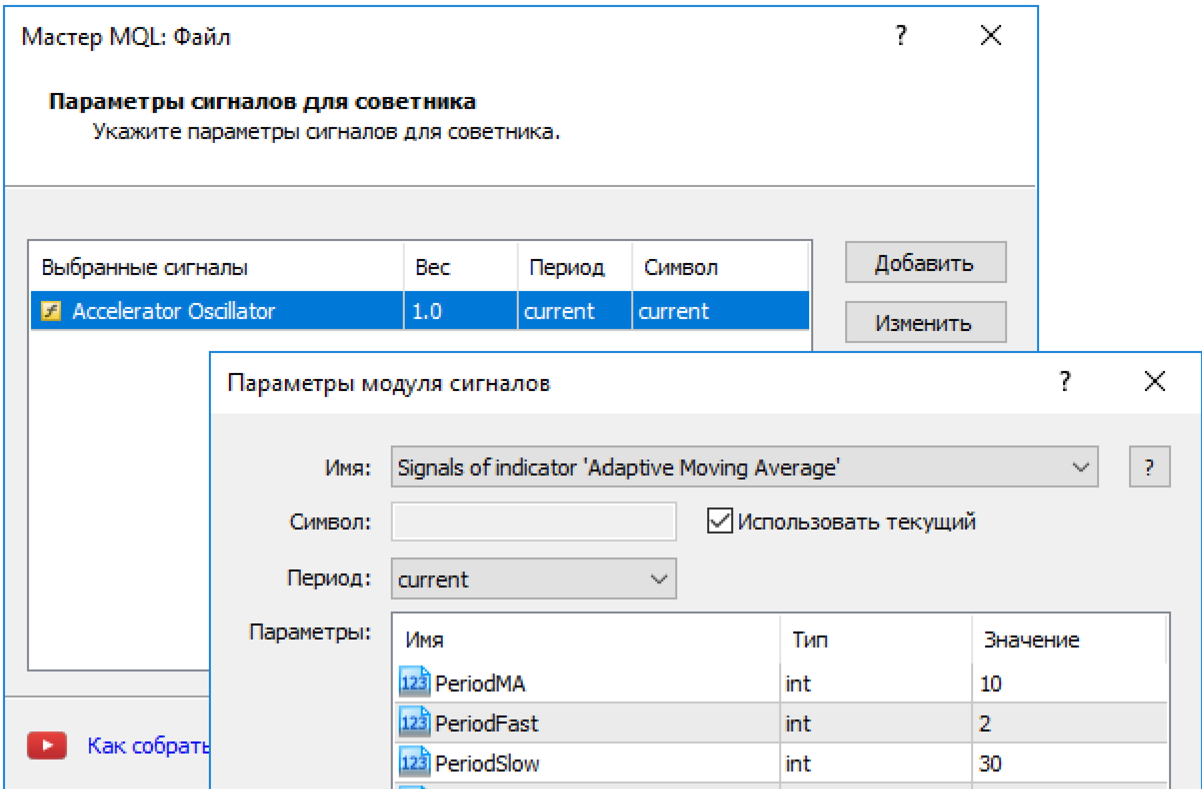
Создание шаблона эксперта. Шаг 3. Дополнительные обработчики событий

Завершающий 4-й шаг позволяет указать один или несколько опциональных обработчиков событий для тестера. О них речь пойдет в следующей главе.



Создание шаблона эксперта. Шаг 4. Обработчики событий тестера

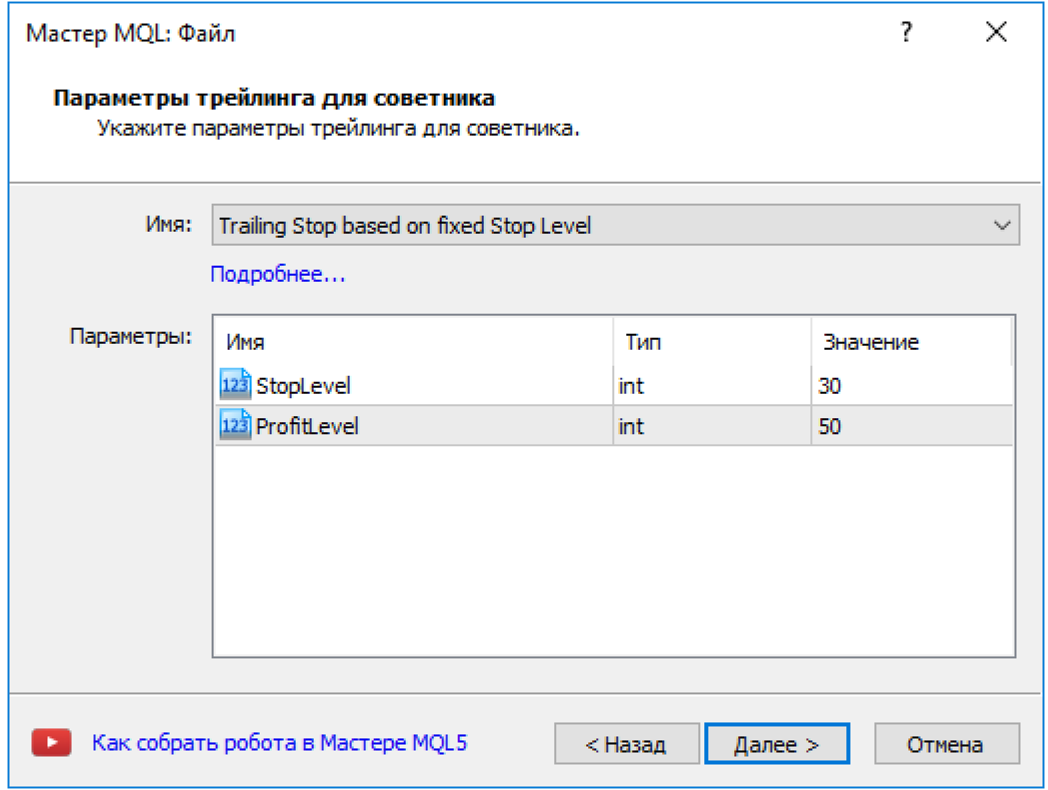
Если пользователь выбрал на первом шаге Мастера генерацию программы на основе стандартной библиотеки, то 3-й шаг представляет собой настройку торговых сигналов.



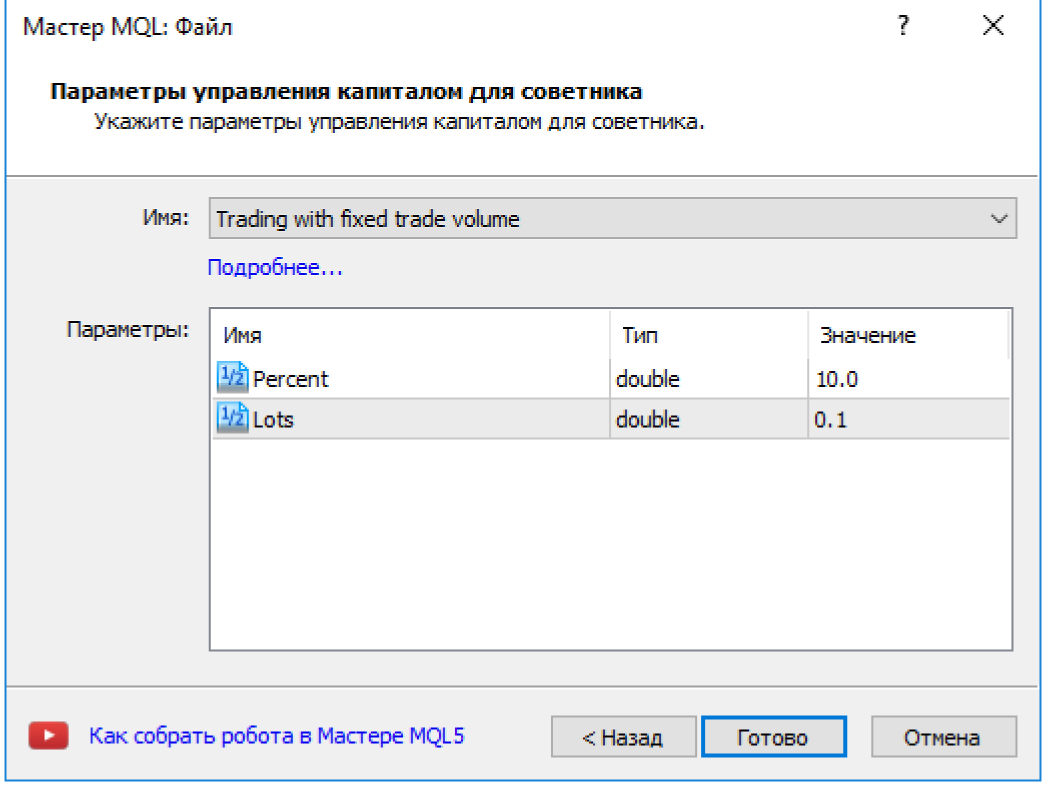
Генерация готового эксперта. Шаг 3. Настройка торговых сигналов

Подробнее про неё можно почитать в документации.

Шаги 4 и 5 предназначены для включения в советник трейлинга и автоматического подбора лотов согласно одному из предопределенных методов.



Генерация готового эксперта. Шаг 4. Выбор метода трейлинг-стопа



Генерация готового эксперта. Шаг 5. Подбор лотов

Мастер, конечно, не является универсальным средством, и получившийся прототип программы, как правило, требует доработки. Однако знания, почерпнутые в данной Главе, позволят вам увереннее чувствовать себя в сгенерированных исходных кодах и расширять их в соответствии с требованиями.

6.5 Тестирование и оптимизация экспертов

Разработка экспертов подразумевает не только и не столько реализацию торговой стратегии на MQL5, а в большей степени — тестирование её финансовых показателей, нахождение оптимальных настроек и отладку (поиск и исправление ошибок) в различных ситуациях. Все это позволяет делать штатный тестер MetaTrader 5.

Тестер является мультивалютным и поддерживает несколько режимов генерации тиков: по ценам открытия выбранного таймфрейма, по ценам OHLC таймфрейма M1, по искусственно генерируемым тикам и по истории реальных тиков. Таким образом, можно выбрать оптимальное соотношение быстродействия и точности моделирования торговли.

Настройки тестера дают возможность задать временной интервал тестирования в прошлом, размер депозита, плечо, эмулировать реквоты и специфические особенности счета (включая размер комиссий, маржи, расписания сессий, ограничения количества лотов). Все подробности работы с тестером с точки зрения пользователя можно найти в документации терминала.

Ранее мы вскользь уже касались работы с тестером, в частности в разделе Тестирование индикаторов. Напомним, что индикаторам недоступны функции управления тестером и также их оптимизация, в отличие от экспертов. Хотя лично я хотел бы видеть опцию адаптивной самонастройки индикаторов — для этого достаточно поддержать в них обработчик *OnTester*, который мы представим в отдельном разделе.

Как известно, для оптимизации доступны различные режимы: прямой перебор сочетаний входных параметров эксперта, ускоренный генетический алгоритм, математические расчеты или последовательные прогоны по символам в *Обзоре рынка*. В качестве критерия оптимизации можно использовать как известные показатели вроде прибыльности, коэффициента Шарпа, фактора восстановления, матожидания выигрыша, так и "пользовательские", заложенные в исходный код разработчиком эксперта. В контексте данной книги предполагается, что читатель уже знаком с принципами настройки, запуска и интерпретации результатов оптимизации, потому что в данной главе мы начнем изучать программное API управления тестером. Желающие могут освежить свои знания в соответствующем разделе справки.

Особенно важная функция тестера — многопоточная оптимизация, которую можно выполнять с привлечением локальных и распределенных (сетевых) программ-агентов, в том числе и в MQL5 Cloud Network. Единичный прогон тестирования (с конкретными входными параметрами эксперта), запущенный пользователем вручную, или один из множества прогонов, вызванных оптимизацией (когда делается перебор значений параметров в заданных диапазонах) производится в отдельной программе — агенте. Технически — это файл metatester64.exe, и копии его процессов можно увидеть в диспетчере задач Windows во время тестирования и оптимизации. Именно за счет этого тестер является многопоточным.

Терминал является диспетчером, который раздает задачи локальным и удаленным агентам. Локальные агенты он запускает при необходимости сам. При оптимизации по умолчанию запускается несколько агентов — их количество соответствует количеству ядер процессора. После выполнения очередного задания по тестированию советника с заданными параметрами агент возвращает терминалу результаты.

В каждом агенте создается свое собственное торговое и программное окружение. Все агенты изолированы друг от друга и от клиентского терминала.

В частности, у агента свои глобальные переменные и собственная файловая песочница, включая и папку, в которую записываются подробные логи агента — *Tester/Agent-IPaddress-Port/Logs*. Здесь *Tester* — каталог установки тестера (при стандартной установке вместе с MetaTrader 5 это подпапка, куда установлен терминал). В имени каталога *Agent-IPaddress-Port* вместо *IPaddress* и *Port* будут указаны конкретные значения сетевого адреса и порта, которые используются для обмена данными с терминалом. Для локальных агентов это адрес 127.0.0.1 и диапазон портов, по умолчанию, начиная с 3000 (например, на компьютере с 4-мя ядрами мы увидим агенты на портах 3000, 3001, 3002, 3003).

Напомним, что все файловые операции при тестировании эксперта происходят в папке *Tester/Agent-IPaddress-Port/MQL5/Files*. Однако можно реализовать взаимодействие между локальными агентами и клиентским терминалом (а также между разными копиями терминала на одном компьютере) через общую папку — для этого при открытии файла функцией *FileOpen* следует указать флаг FILE_COMMON. Другой способ передачи данных с агентов в терминал предоставляет механизм фреймов.

Локальная песочница агента автоматически очищается перед каждым тестом из соображений безопасности (чтобы разные эксперты не могли читать данные друг друга).

Рядом с файловой песочницей для каждого агента создается папка с историей котировок — *Tester/Agent-IPaddress-Port/bases/ServerName/Symbol/*. В следующем разделе мы вкратце напомним, как она формируется.

Результаты отдельных тестовых прогонов и оптимизаций сохраняются терминалом в особом кэше — в каталоге установки, в подпапке *Tester/cache/*. Результаты тестов хранятся в файлах с расширением *tst*, а результаты оптимизаций — в *opt*-файлах. Оба формата открыты разработчиками MetaQuotes, поэтому вы можете реализовать собственную пакетную аналитическую обработку данных или воспользоваться готовыми исходными кодами из кодовой базы на сайте mql5.com.

В данной главе мы сначала рассмотрим основные принципы работы MQL-программ в тестере, а затем на практике изучим способы взаимодействия с ним.

6.5.1 Генерация тиков в тестере

Наличие обработчика *OnTick* в эксперте не является обязательным для того, чтобы его можно было подвергнуть проверке в тестере. Советник может использовать одну или несколько других знакомых нам функций:

- *OnTick* — обработчик события прихода нового тика;
- *OnTrade* — обработчик торгового события;
- *OnTradeTransaction* — обработчик торговой транзакции;
- *OnTimer* — обработчик сигнала таймера;
- *OnChartEvent* — обработчик событий на графике, в том числе пользовательских.

Вместе с тем, внутри тестера основным эквивалентом хода времени является поток тиков, которые содержат не только изменение цены, но и время с точностью до миллисекунд. Поэтому для тестирования экспертов необходимо генерировать тиковые последовательности. В тестере MetaTrader 5 реализовано 4 режима генерации тиков:

- Реальные тики (если их историю предоставляет брокер);
- Все тики (эмуляция на основе доступных котировок таймфрейма M1);
- Цены OHLC с минутных баров (1 Minute OHLC);
- Только цены открытия (1 тик на бар);

Еще один режим работы — математические вычисления — мы разберем позднее, так как он не связан с котировками и тиками.

Какой бы из 4-х режимов пользователь ни выбрал, терминал осуществляет загрузку доступных исторических данных для тестирования. Если был выбран режим реальных тиков, а их у брокера по данному инструменту нет, то используется режим "Все тики". Качество генерации тиков тестер указывает в своем отчете графически и в процентах (100% — все тики реальные).

История по выбранному в настройках тестера инструменту синхронизируется и закачивается терминалом с торгового сервера перед запуском процесса тестирования. При этом в первый раз терминал скачивает с торгового сервера историю на требуемую глубину (с некоторым запасом, в зависимости от таймфрейма, как минимум 1 год до старта теста), чтобы впоследствии не обращаться за ней. В дальнейшем происходит лишь докачка новых данных. Все это сопровождается соответствующими сообщениями в журнале тестера.

Агент тестирования получает от клиентского терминала историю по тестируемому инструменту сразу же после запуска тестирования. Если в процессе тестирования используются данные по другим инструментам (например, это мультивалютный эксперт), то в этом случае агент тестирования запрашивает у клиентского терминала требуемую историю при первом же обращении. Если исторические данные имеются на терминале, они сразу передаются на агенты тестирования. Если данные отсутствуют, терминал запросит и скачает их с сервера, а затем передаст на агенты тестирования.

Обращение к дополнительным инструментам происходит и в том случае, когда вычисляется цена кросс-курса при торговых операциях. Например, при тестировании стратегии на EURCHF с валютой депозита в долларах США перед обработкой первой же торговой операции агент тестирования запросит у клиентского терминала историю по EURUSD и USDCHF, хотя в стратегии нет прямого обращения к этим инструментам.

В связи с этим перед началом тестирования мультивалютной стратегии рекомендуется предварительно скачать все необходимые исторические данные на клиентском терминале. Это позволит избежать задержек при тестировании/оптимизации, связанных с докачкой данных. Закачать историю можно, например, путем открытия соответствующих графиков и прокрутки их к началу истории.

Теперь рассмотрим режимы генерации тиков более подробно.

Реальные тики из истории

Тестирование и оптимизация на реальных тиках являются максимально приближенными к реальным условиям. Это — тики с бирж и от поставщиков ликвидности.

> Если в истории символа есть минутный бар, но тиковых данных за эту минуту нет, тестер сгенерирует тики в режиме "Все тики" (см. далее). Это позволяет выстроить правильный график в тестере в случае неполных тиковых данных у брокера. Более того, тиковые данные могут не совпадать с минутными барами по различным причинам. Например, из-за обрывов связи или иных сбоев при передаче данных от источника в клиентский терминал. При тестировании минутные данные считаются более достоверными.

Тики хранятся в кэше символа в тестере стратегий. Размер кэша — не более 128 000 тиков. При поступлении новых тиков самые старые данные из него выталкиваются. Однако при помощи функции *CopyTicks* можно получить тики и за пределами кэша (только при тестировании по реальным тикам). В этом случае данные будут запрошены из базы тиков тестера, которая полностью соответствует аналогичной базе клиентского терминала. В эту базу никакие корректировки по минутным барам не вносятся. Поэтому тики в ней могут отличаться от тиков, находящихся в кэше.

Все тики (эмуляция)

При отсутствии истории реальных тиков или в целях минимизации сетевого трафика (т.к. архив реальных тиков может занимать значительный объем), пользователь может выбрать искусственную генерацию тиков на основе доступных котировок таймфрейма M1.

История котировок по финансовым инструментам передается от торгового сервера в клиентский терминал MetaTrader 5 в виде экономно упакованных блоков минутных баров. Как происходит запрос и построение требуемых таймфреймов мы рассматривали более подробно в разделе Технические особенности организации и хранения таймсерий.

Минимальным элементом ценовой истории является минутный бар, из которого можно получить информацию о четырех значениях цены OHLC: Open, High, Low, Close.

Новый минутный бар открывается не в тот момент, когда начинается новая минута (количество секунд становится равным 0), а когда происходит тик — изменение цены хотя бы на один пункт. Точно также мы не можем по бару определить с точностью до секунды, когда пришел тик, соответствующий цене закрытия этого минутного бара: известно только одно — это последняя цена на минутном баре, которая и была записана как цена Close.

Таким образом, для каждого минутного бара нам известны 4 контрольные точки, о которых можно точно сказать, что цена там побывала. Если бар имеет только 4 тика, то для тестирования этой информации достаточно, но обычно тиковый объем больше 4. Значит, необходимо сгенерировать дополнительные контрольные точки для тиков, которые приходили между ценами *Open*, *High*, *Low* и *Close*. Принцип генерации тиков в режиме "Все тики" описан в документации.

При тестировании в режиме "Все тики" функция *OnTick* эксперта будет вызываться на каждом сгенерированном тике. Эксперт будет получать время и цены *Ask/Bid/Last* так же, как и при работе онлайн.

Режим тестирования "Все тики" является самым точным (после режима на основе реальных тиков), но при этом и самым затратным по времени. Для первичной оценки большинства торговых стратегий обычно достаточно использовать один из двух упрощенных режимов тестирования: по ценам OHLC M1 или по открытию баров выбранного таймфрейма.

1 minute OHLC

В режиме "1 minute OHLC" тиковая последовательность строится только по OHLC ценам минутных баров, количество вызовов функции *OnTick* существенно уменьшается — следовательно, уменьшается и время тестирования. Это очень эффективный, полезный режим, предлагающий компромисс между точностью тестирования и скоростью. Однако с ним нужно быть осторожным, когда речь идет о чужом эксперте.

Отказ от генерации дополнительных промежуточных тиков между ценами *Open*, *High*, *Low* и *Close* приводит к появлению жесткой детерминированности в развитии цены с того момента, как

определена цена *Open*. Это дает возможность для создания "Грааля тестирования", который показывает красивый восходящий график баланса при тестировании.

> Для минутного бара известны 4 цены, из которых первой идет *Open*, а последней *Close*. Между ними случились цены *High* и *Low*, причем информация о порядке их наступления утрачена, однако мы знаем, что цена *High* больше или равна цене *Open*, а цена *Low* меньше или равна цене *Open*.
>
> Достаточно после поступления цены *Open* анализировать следующий тик, чтобы определить, что перед нами — *High* или *Low*. Если цена ниже цены *Open*, значит это цена *Low* — покупаем на этом тике, так как следующий тик будет соответствовать цене *High*, на котором закрываем покупку и открываем продажу. Следующий тик последний на баре, это цена *Close*, на нем закрываем продажу.
>
> Если после цены пришел тик с ценой больше цены открытия, то последовательность сделок обратная. В таком мошенническом режиме можно торговать на каждом баре. При тестировании такого эксперта на истории все идет идеально, но в онлайне его ждет полный крах.

Похожий эффект может случиться и непреднамеренно - из-за сочетания особенностей алгоритма расчетов (например, подсчет статистики) и генерации тиков.

Таким образом, всегда важно после нахождения оптимальных настроек эксперта на грубых режимах тестирования ("1 minute OHLC" и "Только цены открытия") протестировать его в режиме "Все тики", а еще лучше — на реальных тиках.

Только цены открытия

В данном режиме происходит генерация тиков по четырем ценам OHLC таймфрейма, выбранного для тестирования, причем функция *OnTick* запускается только один раз в начале каждого бара. Из-за этой особенности стоп-уровни и отложенные ордера могут срабатывать не по заявленной цене (особенно при тестировании на старших таймфреймах). В обмен за это мы получаем возможность быстро провести оценочное тестирование эксперта.

Например, производится тестирование советника на EURUSD H1 в режиме "Только цены открытия". В этом случае общее количество тиков (контрольных точек) будет в 4 раза больше количества часовых баров, попавших в тестируемый интервал. Но при этом вызов обработчика *OnTick* произойдет только на открытии часовых баров. На остальных ("скрытых" от эксперта) тиках происходят проверки, необходимые для корректного тестирования:

- вычисление маржевых требований;
- срабатывание *Stop Loss* и *Take Profit*;
- срабатывание отложенных ордеров;
- удаление отложенных ордеров с истекшим временем.

Если нет открытых позиций или отложенных ордеров, то необходимости в данных проверках на скрытых тиках нет, и прирост скорости может оказаться существенным.

Исключением при генерации тиков в режиме "Только цены открытия" являются периоды W1 и MN1: для этих таймфреймов тики генерируются для цен OHLC каждого дня, а не недели или месяца соответственно.

Данный режим "Только цены открытия" хорошо подходит для тестирования стратегий, которые совершают сделки только на открытии бара и не используют отложенные ордера, а также не

используют уровни *Stop Loss* и *Take Profit*. Для класса таких стратегий сохраняется вся необходимая точность тестирования.

MQL5 API не позволяет программе узнать, в каком режиме она запущена в тестере. Вместе с тем, это может быть важно для экспертов или используемых ими индикаторов, которые не рассчитаны, например, на корректную работу по ценам открытия или OHLC. В связи с этим реализуем простой механизм детекции режима. Исходный код прилагается в файле *TickModel.mqh*.

Объявим свое перечисление с обозначениями существующих режимов.

```
enum TICK_MODEL
{
   TICK_MODEL_UNKNOWN = -1,    /*Unknown (any)*/    // неизвестный/еще не определен
   TICK_MODEL_REAL = 0,        /*Real ticks*/       // лучшее качество
   TICK_MODEL_GENERATED = 1,   /*Generated ticks*/  // хорошее качество
   TICK_MODEL_OHLC_M1 = 2,     /*OHLC M1*/          // приемлемое качество и быстро
   TICK_MODEL_OPEN_PRICES = 3, /*Open prices*/      // худшее качество, но очень быст
   TICK_MODEL_MATH_CALC = 4,   /*Math calculations*/// без тиков (не определяется)
};
```

За исключением первого элемента, который зарезервирован для случая, когда режим еще не определен или не может быть по каким-то причинам определен, все остальные элементы расположены в порядке ухудшения качества моделирования, начиная от реальных и заканчивая ценами открытия (для них разработчик должен проверить совместимость стратегии с тем, что её торговля ведется только по открытию нового бара). Последний режим TICK_MODEL_MATH_CALC обходится вообще без тиков, мы его рассмотрим отдельно.

Принцип определения режима заключается в проверке доступности тиков и их времени на первых двух тиках при запуске теста. Сама проверка обернута в функцию *getTickModel*, которую эксперт должен вызывать из обработчика *OnTick*. Поскольку проверка делается один раз, внутри функции описана статическая переменная model, установленная изначально в значение TICK_MODEL_UNKNOWN. В ней будет храниться и переключаться текущее состояние проверки, что потребуется для различения режимов OHLC и по ценам открытия.

```
TICK_MODEL getTickModel()
{
   static TICK_MODEL model = TICK_MODEL_UNKNOWN;
   ...
```

На первом анализируемом тике модель равна TICK_MODEL_UNKNOWN, и производится попытка получить реальные тики с помощью вызова *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:     // эмуляция тиков
            model = TICK_MODEL_GENERATED;
            break;

         case ERR_FUNCTION_NOT_ALLOWED: // цены открытия или 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;
      }
   }
   ...
```

Если она завершится успешно, детекция тут же заканчивается установкой модели в TICK_MODEL_REAL. Если же реальные тики недоступны, система вернет нам некий код ошибки, по которому можно сделать следующие заключения. Код ошибки ERR_NOT_ENOUGH_MEMORY соответствует режиму эмуляции тиков. Почему код именно такой — не совсем понятно, но это характерная особенность, и мы её здесь используем. В двух других режимах генерации тиков мы получим ошибку ERR_FUNCTION_NOT_ALLOWED.

Различить один от другого можно по времени тика. Если оно у тика окажется некратным таймфрейму, значит речь о режиме OHLC. Однако проблема здесь в том, первый тик в обоих режимах может быть выровнен по времени открытия бара. Таким образом, мы получим значение TICK_MODEL_OPEN_PRICES, но оно требует уточнения. Поэтому для окончательного заключения следует проанализировать еще один тик (вызвать на нем функцию еще раз в случае получения перед этим TICK_MODEL_OPEN_PRICES). Внутри функции для этого случая предусмотрена следующая ветвь оператора *if*.

```
      else if(model == TICK_MODEL_OPEN_PRICES)
      {
         if(TimeCurrent() != iTime(_Symbol, _Period, 0))
         {
            model = TICK_MODEL_OHLC_M1;
         }
      }
      return model;
   }
```

Работу детектора проверим в простом эксперте *TickModel.mq5*. Во входном параметре *TickCount* укажем максимальное количество анализируемых тиков, то есть сколько раз вызвать функцию *getTickModel*. Мы знаем, что достаточно двух, но для того, чтобы убедиться, что модель впоследствии не меняется, по умолчанию предложено 5 тиков. Также предусмотрим параметр *RequireTickModel*, который предписывает эксперту завершить работу, если уровень моделирования окажется ниже запрошенного. По умолчанию его значение TICK_MODEL_UNKNOWN, что означает отсутствие ограничения по режиму.

```
input int TickCount = 5;
input TICK_MODEL RequireTickModel = TICK_MODEL_UNKNOWN;
```

В обработчике *OnTick* запускаем свой код только при условии работы в тестере.

```
void OnTick()
{
   if(MQLInfoInteger(MQL_TESTER))
   {
      static int count = 0;
      if(count++ < TickCount)
      {
         // выводим информацию о тике для справки
         static MqlTick tick[1];
         SymbolInfoTick(_Symbol, tick[0]);
         ArrayPrint(tick);
         // определяем и выводим модель (предварительно)
         const TICK_MODEL model = getTickModel();
         PrintFormat("%d %s", count, EnumToString(model));
         // если счетчик тиков 2+, заключение окончательное и на его основе действуем
         if(count >= 2)
         {
            if(RequireTickModel != TICK_MODEL_UNKNOWN
            && RequireTickModel < model) // качество ниже запрошенного
            {
               PrintFormat("Tick model is incorrect (%s %sis required), terminating",
                  EnumToString(RequireTickModel),
                  (RequireTickModel != TICK_MODEL_REAL ? "or better " : ""));
               ExpertRemove(); // завершаем работу
            }
         }
      }
   }
}
```

Попробуем запустить эксперт в тестере при различных режимах генерации тиков, выбрав расхожее сочетание EURUSD H1.

Установим в эксперте параметр *RequireTickModel* в OHLC M1. Если режим тестера "Все тики", получим в журнале соответствующее сообщение, и эксперт продолжит работу.

```
                 [time]   [bid]   [ask]  [last] [volume]    [time_msc] [flags] [volum
[0] 2022.04.01 00:00:30 1.10656 1.10679 1.10656        0 1648771230000      14
NOT_ENOUGH_MEMORY
1 TICK_MODEL_GENERATED
                 [time]   [bid]   [ask]  [last] [volume]    [time_msc] [flags] [volum
[0] 2022.04.01 00:01:00 1.10656 1.10680 1.10656        0 1648771260000      12
2 TICK_MODEL_GENERATED
                 [time]   [bid]   [ask]  [last] [volume]    [time_msc] [flags] [volum
[0] 2022.04.01 00:01:30 1.10608 1.10632 1.10608        0 1648771290000      14
3 TICK_MODEL_GENERATED
```

Аналогично подойдут режимы непосредственно OHLC M1 и реальные тики, причем в последнем случае не будет никакого кода ошибки.

```
                  [time]    [bid]    [ask] [last] [volume]     [time_msc] [flags] [volume
[0] 2022.04.01 00:00:00 1.10656 1.10687 0.0000        0 1648771200122     134        0
1 TICK_MODEL_REAL
                  [time]    [bid]    [ask] [last] [volume]     [time_msc] [flags] [volume
[0] 2022.04.01 00:00:00 1.10656 1.10694 0.0000        0 1648771200417       4        0
2 TICK_MODEL_REAL
                  [time]    [bid]    [ask] [last] [volume]     [time_msc] [flags] [volume
[0] 2022.04.01 00:00:00 1.10656 1.10691 0.0000        0 1648771200816       4        0
3 TICK_MODEL_REAL
```

Однако если в тестере поменять режим на "Только цены открытия", эксперт остановится после второго тика.

```
                  [time]    [bid]    [ask]  [last] [volume]     [time_msc] [flags] [volum
[0] 2022.04.01 00:00:00 1.10656 1.10679 1.10656        0 1648771200000      14
FUNCTION_NOT_ALLOWED
1 TICK_MODEL_OPEN_PRICES
                  [time]    [bid]    [ask]  [last] [volume]     [time_msc] [flags] [volum
[0] 2022.04.01 01:00:00 1.10660 1.10679 1.10660        0 1648774800000      14
2 TICK_MODEL_OPEN_PRICES
Tick model is incorrect (TICK_MODEL_OHLC_M1 or better is required), terminating
ExpertRemove() function called
```

К сожалению, данный способ требует запуска теста и пары тиков для определения режима. Иными словами, мы не можем остановить тест раньше, вернув ошибку из *OnInit*. Более того, при запуске оптимизации с неправильным типом генерации тиков мы не сможем остановить оптимизацию, что можно делать только из функции *OnTesterInit*. Таким образом, тестер будет пытаться выполнить все проходы по время оптимизации, хотя они будут пресекаться в самом начале. Это текущее ограничение платформы.

6.5.2 Управление ходом времени в тестере: таймер, Sleep, GMT

При разработке экспертов следует учитывать, что в тестере имеются некоторые нюансы моделирования хода времени на основе генерируемых тиков и работы связанных со временем функций.

При тестировании локальное время, возвращаемое функцией *TimeLocal*, всегда равно серверному времени согласно *TimeTradeServer*. В свою очередь, серверное время всегда равно времени GMT — *TimeGMT*. Таким образом, все эти функции при тестировании выдают одно и то же время. Такова техническая особенность платформы, обусловленная тем, что информацию о серверном времени было решено не хранить локально, а всегда брать с сервера, с которым может не быть связи в конкретный момент.

Данная особенность создает трудности при реализации стратегий, связанных с глобальным временем, в частности, с привязкой к выходу новостей. В подобных случаях приходится предусматривать указание часового пояса котировок в настройках тестируемого эксперта или изобретать способы автоопределения таймзоны (см. раздел Переход на летнее время).

Обратимся теперь к другим функциям для работы со временем.

Как мы знаем, в MQL5 возможна обработка событий таймера. Вызов обработчика *OnTimer* производится независимо от режима тестирования. Это означает, что если тестирование запущено в режиме "Только цены открытия" на периоде H4 и внутри эксперта установлен таймер с вызовом каждую секунду, то на открытии каждого бара H4 один раз будет вызван

обработчик *OnTick* и далее "в течение" бара будет 14400 раз (3600 секунд * 4 часа) вызван обработчик *OnTimer*. Насколько при этом увеличится время тестирования эксперта, зависит от его алгоритма.

Другой функцией, влияющей на ход времени внутри программы, является функция *Sleep*. Она позволяет приостановить выполнение эксперта на некоторое время. Это может понадобиться при запросе каких-либо данных, которые в момент запроса еще не готовы и необходимо дождаться момента их готовности.

Важно понять, что *Sleep* влияет только на вызвавшую его программу и не задерживает процесс тестирования. Фактически при вызове *Sleep* "проигрываются" сгенерированные тики в пределах указанной задержки, в результате чего могут сработать отложенные ордера, стоп-уровни и т.д. После вызова *Sleep* смоделированное в тестере время увеличивается на интервал, указанный в параметре функции.

Чуть позже, в разделе про тестирование мультивалютных советников мы покажем, как можно использовать таймер и функцию *Sleep* для синхронизации баров.

6.5.3 Визуализация тестирования: график, объекты, индикаторы

Тестер позволяет проводить тестирование двумя различными способами: с визуализацией и без неё. За выбор способа отвечает соответствующая опция на закладке основных настроек тестера.

При включенной визуализации тестер открывает отдельное окно, в котором воспроизводит торговые операции, выводит индикаторы и объекты. Это наглядно, но нужно не всегда, а лишь для программ с пользовательским интерфейсом (например, торговых панелей или управляемых разметкой, сделанной графическими объектами). Для прочих экспертов важно только исполнение алгоритма согласно заложенной стратегии. А это можно проверить и без визуализации, что позволяет существенно ускорить процесс. Кстати говоря, именно в таком режиме делаются тестовые прогоны при оптимизации.

Во время такого "фонового" тестирования и оптимизации не осуществляется построение графических объектов. Поэтому при обращении к свойствам объектов эксперт получит нулевые значения. Таким образом, проверить работу с объектами и графиком можно только при тестировании в визуальном режиме.

Также напомним особенность поведении индикаторов в тестере, о которой уже говорилось в соответствующем разделе Тестирование индикаторов. Для повышения эффективности невизуального тестирования и оптимизации экспертов (использующих индикаторы), индикаторы могут рассчитываться не на каждом тике, а только по запросу данных от них. Пересчет на каждом тике происходит только при наличии в индикаторе функций *EventChartCustom*, *OnChartEvent*, *OnTimer* или директивы *tester_everytick_calculate* (см. Директивы препроцессора для тестера). В визуальном окне тестера и онлайн индикаторы всегда получают события *OnCalculate* на каждом тике.

Если тестирование проводится в невизуальном режиме, после его окончания в терминале автоматически открывается график инструмента, на котором отображаются совершенные сделки и индикаторы, которые использовались в эксперте. Это помогает сопоставить моменты входа в рынок и выхода из него со значениями индикаторов. Однако здесь имеются в виду только индикаторы, работающие на символе и таймфрейме тестирования. Если эксперт создавал индикаторы на других символах или таймфреймах, они не будут показаны.

> Важно отметить, что индикаторы, отображаемые на графике, автоматически открытом после завершения тестирования, рассчитываются заново уже после окончания тестирования. Это происходит, даже если эти индикаторы использовались в тестируемом эксперте и до этого рассчитывались "на лету" — по мере формирования баров.

В некоторых случаях программисту может понадобиться скрыть информацию о том, какие индикаторы задействованы в торговом алгоритме, и потому их визуализация на графике нежелательна. Для этого подойдет функция *IndicatorRelease*.

Функция *IndicatorRelease* изначально предназначена для освобождения расчетной части индикатора, если он больше не нужен. Это позволяет экономить память и ресурсы процессора. Второе ее предназначение — запретить показ индикатора на графике тестирования после окончания одиночного прогона.

Чтобы запретить показ индикатора на графике по окончании тестирования, достаточно вызвать *IndicatorRelease* с дескриптором индикатора в обработчике *OnDeinit*. Функция *OnDeinit* всегда вызывается в экспертах после завершения и перед показом графика тестирования. В самих индикаторах в тестере не вызывается ни OnDeinit, ни деструкторы глобальных и статических объектов — так решили сделать разработчики MetaTrader 5.

Кроме того, в MQL5 API входит специальная функция с аналогичным назначением *TesterHideIndicators*, которую мы рассмотрим позднее.

При этом следует учитывать, что на внешнее представление графика тестирования дополнительно могут влиять tpl-шаблоны (если они созданы).

Так при наличии шаблона *tester.tpl* в каталоге *MQL5/Profiles/Templates*, именно он будет применен к открываемому графику. Если эксперт в своей работе использовал другие индикаторы и не запретил их показ, то на графике будут объединены индикаторы из шаблона и из эксперта.

При отсутствии *tester.tpl* применяется шаблон по умолчанию (*default.tpl*).

Если в папке *MQL5/Profiles/Templates* присутствует tpl-шаблон с названием, совпадающим с именем эксперта (например, *ExpertMACD.tpl*), то при визуальном тестировании или на графике, открываемом после тестирования, будут показаны только индикаторы из данного шаблона. В этом случае никакие индикаторы, используемые в тестируемом эксперте, показаны не будут.

6.5.4 Мультивалютное тестирование

Как известно, тестер MetaTrader 5 позволяет проверять стратегии, торгующие на нескольких инструментах. Чисто технически, с оглядкой на доступные "железные" ресурсы компьютера, можно моделировать одновременную торговлю по всем доступным инструментам.

Тестирование таких стратегий налагает на тестер несколько дополнительных технических требований:

- генерация последовательностей тиков для всех инструментов;
- расчет индикаторов для всех инструментов;
- расчет маржевых требований и эмуляция прочих торговых условий по всем инструментам.

История по используемым инструментам закачивается тестером из терминала автоматически при первом обращении к ней. Если в терминале не окажется требуемой истории, он в свою очередь запросит её с торгового сервера. Поэтому перед началом тестирования мультивалютного

эксперта рекомендуется выбрать требуемые инструменты в *Обзоре рынка* терминала и подкачать данные на нужную глубину.

Агент закачивает недостающую историю с небольшим запасом, чтобы обеспечить необходимые данные для расчета индикаторов или копирования экспертом на момент начала тестирования. Минимальный объем истории, скачиваемой с торгового сервера, зависит от таймфрейма. Так для таймфреймов D1 и меньше он составляет один год. Иными словами, предварительная история закачивается от начала предыдущего года относительно стартовой даты тестера. Это дает как минимум 1 год истории, если тестирование задано с первого января, и как максимум чуть меньше двух лет, если тестирование идет с декабря. Для недельного таймфрейма запрашивается история в 100 баров, то есть примерно два года (в году 52 недели). Для тестирования на месячном таймфрейме агент запросит 100 месяцев (то есть историю примерно за 8 лет: 12 месяцев * 8 лет = 96). В любом случае, на более младших таймфреймах, чем рабочий, будет доступно пропорционально большее количество баров. Если существующих данных не хватает для предопределенной глубины предварительной истории, об этом факте будет запись в журнале тестирования.

Настроить (изменить) данное поведение нельзя. Поэтому, если нужно с самого начала обеспечить заданное количество исторических баров текущего таймфрейма, следует ставить более раннюю стартовую дату теста и затем "дожидаться" в коде эксперта наступления нужной даты активации торговли или достаточного количества баров, а до того пропускать все события "вхолостую".

В тестере также эмулируется свой *Обзор рынка*, из которого программа может получать информацию по инструментам. По умолчанию в начале тестирования в *Обзоре рынка* тестера есть только один символ — символ, на котором запущено тестирование. Все дополнительные символы добавляются в *Обзор рынка* тестера автоматически при обращении к ним через функции API. При первом же обращении к "чужому" символу из MQL-программы агент тестирования произведет синхронизацию по этому символу с терминалом.

Обращение к данным чужого символа происходят в следующих случаях:

- использование технических индикаторов, *iCustom*, *IndicatorCreate* на паре символ/таймфрейм
- запрос к *Обзору рынка* по чужому символу:
  - SeriesInfoInteger
  - Bars
  - SymbolSelect
  - SymbolIsSynchronized
  - SymbolInfoDouble
  - SymbolInfoInteger
  - SymbolInfoString
  - SymbolInfoTick
  - SymbolInfoSessionQuote
  - SymbolInfoSessionTrade
  - MarketBookAdd
  - MarketBookGet
- запрос к таймсерии по паре символ/таймфрейм функциями:
  - CopyBuffer

- CopyRates
- CopyTime
- CopyOpen
- CopyHigh
- CopyLow
- CopyClose
- CopyTickVolume
- CopyRealVolume
- CopySpread

Кроме того, можно явно запросить историю для нужных символов с помощью вызова функции *SymbolSelect* в обработчике *OnInit* — загрузка истории будет произведена заранее, до начала тестирования советника.

В тот момент, когда происходит первое обращение к чужому символу, процесс тестирования останавливается и происходит подкачка истории по паре символ/период от терминала к агенту тестирования. Одновременно включается генерация тиковой последовательности.

Для каждого инструмента генерируется собственная тиковая последовательность согласно установленному режиму генерации тиков.

Особую важность при реализации мультивалютных советников играет синхронизация баров разных символов, так как от этого зависит правильность расчетов. Синхронизированным считается состояние, когда последние бары всех используемых символов имеют одно и то же время открытия.

Тестер генерирует и проигрывает для каждого инструмента его тиковую последовательность. При этом новый бар на каждом инструменте открывается независимо от того, как открываются бары на других инструментах. Это означает, что при тестировании мультивалютного эксперта возможна ситуация (и чаще всего так и бывает), когда на одном инструменте новый бар уже открылся, а на другом еще нет.

Например, если мы тестируем эксперт на символе EURUSD, и здесь открылась новая часовая свеча, то мы получим событие *OnTick*. Но при этом нет никакой гарантии, что новая свеча открылась по символу GBPUSD, который, допустим, нас тоже интересует.

Таким образом, алгоритм синхронизации подразумевает, что нужно проверять котировки всех инструментов и дождаться равенства времен открытия последних баров.

Это не вызывает вопросов до тех пор, пока используются режимы тестирования на реальных тиках, эмуляции всех тиков или OHLC M1. При этих режимах в пределах одной свечи генерируется достаточное количество тиков, чтобы дождаться момента синхронизации баров с разных символов. Достаточно завершить работу функции *OnTick* и проверить появление нового бара на GBPUSD на следующем тике. Но при тестировании в режиме "Только цены открытия" другого тика не будет, так как эксперт вызывается только один раз за бар, и может показаться, что этот режим не годится для тестирования мультивалютных экспертов. На самом деле тестер позволяет засечь момент, когда на другом символе откроется новый бар с помощью функции *Sleep* (в цикле) или таймера.

Для начала рассмотрим пример эксперта *SyncBarsBySleep.mq5*, демонстрирующего синхронизацию баров через *Sleep*.

Пара входных параметров позволяет задать размер паузы (*Pause*) в секундах для ожидания "чужих" баров, а также название "чужого" символа (*OtherSymbol*) — он должен отличаться от символа графика.

```
input uint Pause = 1;                    // Pause (seconds)
input string OtherSymbol = "USDJPY";
```

Для выявления закономерностей в запаздывании времен открытия баров опишем простой класс *BarTimeStatistics*. Он содержит поле для подсчета общего числа баров (*total*) и числа баров, на которых не было изначально синхронизации (*late*), то есть "чужой" символ запаздывал.

```
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);
   }
};
```

Полученную статистику объект данного класса печатает в своем деструкторе. Поскольку мы собираемся сделать этот объект статическим, отчет выведется в самом конце теста.

Если выбранный в тестере режим генерации тиков будет отличаться от цен открытия, обнаружим это с помощью рассмотренной ранее функции *getTickModel* и выдадим пользователю предупреждение.

```
void OnTick()
{
   const TICK_MODEL model = getTickModel();
   if(model != TICK_MODEL_OPEN_PRICES)
   {
      static bool shownOnce = false;
      if(!shownOnce)
      {
         Print("This EA is intended to run in \"Open Prices\" mode");
         shownOnce = true;
      }
   }
```

Далее в *OnTick* идет непосредственно рабочий алгоритм синхронизации.

```
      // время последнего известного бара для _Symbol
      static datetime lastBarTime = 0;
      // признак синхронизированности
      static bool synchronized = false;
      // счетчики баров
      static BarTimeStatistics stats;

      const datetime currentTime = iTime(_Symbol, _Period, 0);

      // если выполняемся первый раз или бар изменился, сохраняем бар
      if(lastBarTime != currentTime)
      {
         stats.total++;
         lastBarTime = currentTime;
         PrintFormat("Last bar on %s is %s", _Symbol, TimeToString(lastBarTime));
         synchronized = false;
      }

      // время последнего известного бара для другого символа
      datetime otherTime;
      bool late = false;

      // ждем пока времена двух баров не станут одинаковыми
      while(currentTime != (otherTime = iTime(OtherSymbol, _Period, 0)))
      {
         late = true;
         PrintFormat("Wait %d seconds...", Pause);
         Sleep(Pause * 1000);
      }
      if(late) stats.late++;

      // здесь мы оказываемся после синхронизации, сохраним новый статус
      if(!synchronized)
      {
         // используем TimeTradeServer() т.к. TimeCurrent() не меняется в отсутствие тик
         Print("Bars are in sync at ", TimeToString(TimeTradeServer(),
            TIME_DATE | TIME_SECONDS));
         // больше не выводим сообщение до следующей рассинхронизации
         synchronized = true;
      }
      // здесь будет ваш синхронный алгоритм
      // ...
   }
```

Настроим тестер для работы эксперта на EURUSD, H1, как наиболее ликвидном инструменте. Параметры эксперта оставим по умолчанию, то есть "чужим" символом будет USDJPY.

В результате теста журнал будет содержать примерно такие записи (в журнале намеренно оставлены строки о подкачке истории USDJPY, которая произошла при первом обращении к *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%
```

Видно, что бары на USDJPY регулярно запаздывают. Если выбрать в настройках тестера USDJPY, H1, а в параметрах эксперта EURUSD, получим обратную картину.

```
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%
```

Здесь в большинстве случаев ждать не приходилось — бары на EURUSD уже существовали в момент формирования бара на USDJPY.

Есть и другой способ синхронизации баров — с помощью таймера. Пример такого эксперта *SyncBarsByTimer.mq5* прилагается к книге. Обратите внимание, что в нем события таймера, как правило, приходятся внутрь бара (т.к. вероятность попасть точно на начало ничтожно мала). Из-за этого бары оказываются синхронизированными почти всегда.

Мы могли бы еще напомнить о возможности синхронизации баров с помощью индикатора шпиона *EventTickSpy.mq5*, но он основан на пользовательских событиях, которые работают только при визуальном тестировании. Кроме того, для подобных индикаторов, требующих реагировать на каждый тик, важно использовать директиву *#property tester_everytick_calculate*. Мы уже говорили о ней в разделе о Тестировании индикаторов и еще раз напомним в разделе о специфических директивах для тестера.

6.5.5 Критерии оптимизации

Критерий оптимизации — некий показатель, значение которого определяет качество тестируемого набора входных параметров. Чем больше значение критерия оптимизации, тем

лучше оценивается результат тестирования с данным набором параметров. Выбор данного показателя осуществляется на вкладке "Настройки" справа от поля "Оптимизация".

Критерий важен не только для того, чтобы пользователь мог сравнить между собой результаты. Без критерия оптимизации невозможно использование генетического алгоритма, поскольку он на основе критерия "решает", как отбирать кандидатов для новых поколений. При полном переборе вариантов критерий не участвует в процессе перебора.

Напомним, что в тестере доступны следующие встроенные критерии оптимизации:

- Максимальный баланс;
- Максимальная прибыльность;
- Максимальное матожидание выигрыша (средние прибыль/убыток на сделку);
- Минимальная просадка в процентах по эквити;
- Максимальный фактор восстановления;
- Максимальный коэффициент Шарпа;
- Пользовательский критерий оптимизации;

При выборе последнего варианта в качестве критерия оптимизации будет учитываться значение функции *OnTester*, реализованной в советнике — мы рассмотрим её в одном из следующих разделов. Данный параметр позволяет программисту использовать любой собственный показатель для оптимизации.

Также в MetaTrader 5 доступен особый "комплексный критерий". Это интегральный показатель качества прохода тестирования, который учитывает сразу несколько параметров:

- Количество сделок;
- Просадка;
- Фактор восстановления;
- Матожидание выигрыша;
- Коэффициент Шарпа;

Формула не раскрывается разработчиками, но известно, что диапазон возможных значений — от 0 до 100. Важно, что значения комплексного показателя влияют на цвет ячеек колонки *Result* таблицы оптимизации вне зависимости от критерия, то есть подсветка то такому принципу действует, даже когда для отображения в колонке *Result* выбран другой критерий. Слабые комбинации со значениями ниже 20 подсвечиваются красным, сильные, выше 80 — темно-зеленым.

Поиск универсального критерия добротности торговой системы — насущная и сложная задача для большинства трейдеров, поскольку выбор настроек по максимальному значению одного показателя (например, прибыли), как правило, является далеко не лучшим вариантом с точки зрения стабильного и предсказуемого поведения эксперта в обозримом будущем.

Наличие комплексного показателя позволяет нивелировать слабые места каждого отдельного показателя (а они обязательно имеются и широко известны) и дает ориентир при разработке собственных пользовательских показателей для расчета в *OnTester*. Мы займемся этим в ближайшее время.

6.5.6 Получение финансовых показателей теста: TesterStatistics

Обычно мы оцениваем качество эксперта по торговому отчету, аналогом которого в тестере является отчет тестирования. В нем представлено большое количество показателей, характеризующих стиль торговли, стабильность и, разумеется, прибыльность. Все эти показатели, за некоторым исключением, доступны MQL-программе посредством специальной функции *TesterStatistics*. Таким образом, разработчик эксперта имеет возможность анализировать в коде отдельные показатели и конструировать из них собственные комбинированные критерии качества оптимизации.

double TesterStatistics(ENUM_STATISTICS statistic)

Функция *TesterStatistics* возвращает значение указанного статистического показателя, рассчитанного по результатам отдельного прогона эксперта в тестере. Функцию можно вызывать в обработчике *OnDeinit* или *OnTester*, о котором речь еще впереди.

Все доступные показатели сведены в перечисление ENUM_STATISTICS. Часть показателей представляет собой качественные характеристики, то есть вещественные числа (как правило, итоговые суммы прибыли, просадки, коэффициенты и так далее), а другая часть — количественные, то есть целые числа (например, количество сделок). Однако обе группы управляются одной и той же функцией, несмотря на её тип результат *double*.

В следующей таблице приведены показатели вещественные (денежные суммы и коэффициенты). Все денежные суммы выражаются в валюте депозита.

Идентификатор	Описание
STAT_INITIAL_DEPOSIT	Начальный депозит
STAT_WITHDRAWAL	Размер выведенных со счета средств
STAT_PROFIT	Чистая прибыль или убыток по окончании тестирования, сумма STAT_GROSS_PROFIT и STAT_GROSS_LOSS
STAT_GROSS_PROFIT	Общая прибыль, сумма всех прибыльных трейдов (больше или равно нулю)
STAT_GROSS_LOSS	Общий убыток, сумма всех убыточных трейдов (меньше или равно нулю)
STAT_MAX_PROFITTRADE	Максимальная прибыль — наибольшее значение среди всех прибыльных трейдов (больше или равно нулю)
STAT_MAX_LOSSTRADE	Максимальный убыток — наименьшее значение среди всех убыточных трейдов (меньше или равно нулю)
STAT_CONPROFITMAX	Общая максимальная прибыль в серии прибыльных трейдов (больше или равно нулю)
STAT_MAX_CONWINS	Общая прибыль в самой длинной серии прибыльных трейдов
STAT_CONLOSSMAX	Общий максимальный убыток в серии убыточных трейдов (меньше или равно нулю)
STAT_MAX_CONLOSSES	Общий убыток в самой длинной серии убыточных трейдов

Идентификатор	Описание
STAT_BALANCEMIN	Минимальное значение баланса
STAT_BALANCE_DD	Максимальная просадка баланса в деньгах
STAT_BALANCEDD_PERCENT	Просадка баланса в процентах, которая была зафиксирована в момент максимальной просадки баланса в деньгах (STAT_BALANCE_DD)
STAT_BALANCE_DDREL_PERCENT	Максимальная просадка баланса в процентах
STAT_BALANCE_DD_RELATIVE	Просадка баланса в деньгах, которая была зафиксирована в момент максимальной просадки баланса в процентах (STAT_BALANCE_DDREL_PERCENT)
STAT_EQUITYMIN	Минимальное значение собственных средств
STAT_EQUITY_DD	Максимальная просадка средств в деньгах
STAT_EQUITYDD_PERCENT	Просадка средств в процентах, которая была зафиксирована в момент максимальной просадки средств в деньгах (STAT_EQUITY_DD)
STAT_EQUITY_DDREL_PERCENT	Максимальная просадка средств в процентах
STAT_EQUITY_DD_RELATIVE	Просадка средств в деньгах, которая была зафиксирована в момент максимальной просадки средств в процентах (STAT_EQUITY_DDREL_PERCENT)
STAT_EXPECTED_PAYOFF	Математическое ожидание выигрыша (среднее арифметические общей прибыли и количества сделок)
STAT_PROFIT_FACTOR	Прибыльность — отношение STAT_GROSS_PROFIT/STAT_GROSS_LOSS (если STAT_GROSS_LOSS = 0, прибыльность принимает значение DBL_MAX)
STAT_RECOVERY_FACTOR	Фактор восстановления — отношение STAT_PROFIT/STAT_BALANCE_DD
STAT_SHARPE_RATIO	Коэффициент Шарпа
STAT_MIN_MARGINLEVEL	Минимальное достигнутое значение уровня маржи
STAT_CUSTOM_ONTESTER	Значение пользовательского критерия оптимизации, возвращенного функцией OnTester

В следующей таблице приведены показатели целочисленные (количества).

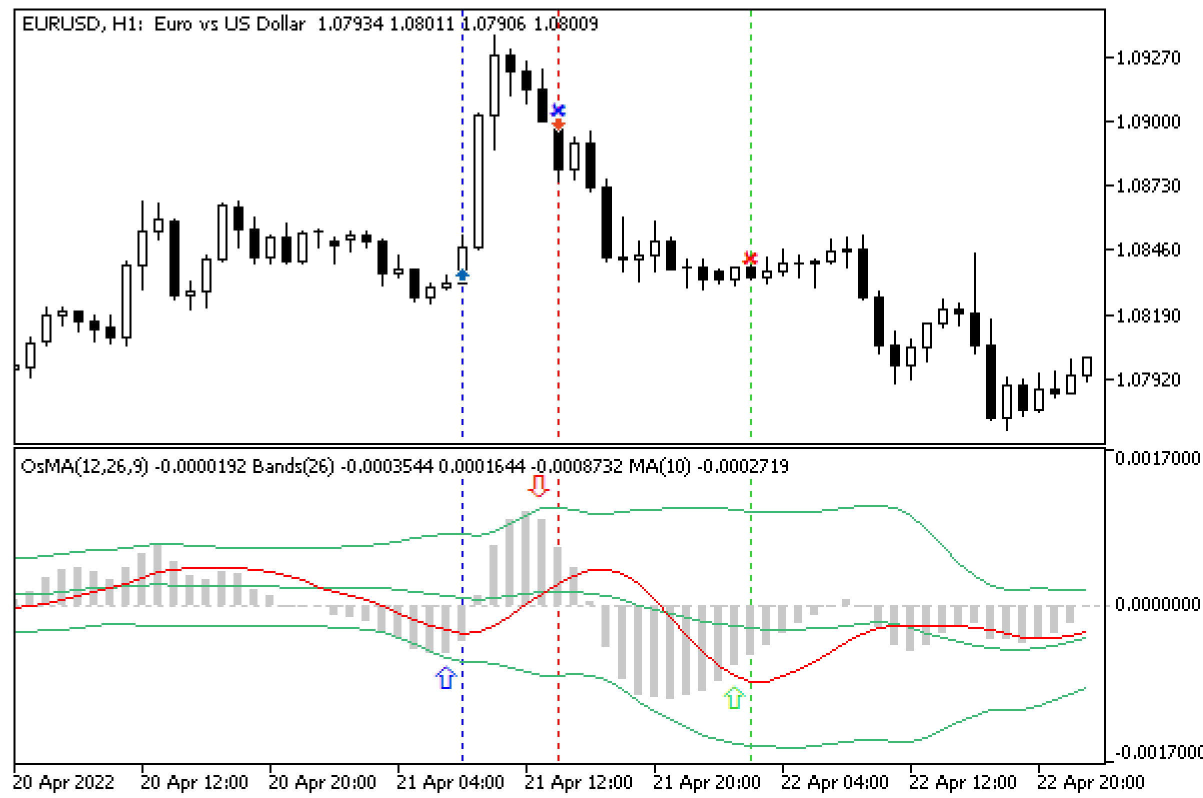
Идентификатор	Описание
STAT_DEALS	Общее количество совершенных сделок
STAT_TRADES	Количество трейдов (сделок выхода из рынка)

Идентификатор	Описание
STAT_PROFIT_TRADES	Прибыльные трейды
STAT_LOSS_TRADES	Убыточные трейды
STAT_SHORT_TRADES	Короткие трейды
STAT_LONG_TRADES	Длинные трейды
STAT_PROFIT_SHORTTRADES	Короткие прибыльные трейды
STAT_PROFIT_LONGTRADES	Длинные прибыльные трейды
STAT_PROFITTRADES_AVGCON	Средняя длина прибыльной серии трейдов
STAT_LOSSTRADES_AVGCON	Средняя длина убыточной серии трейдов
STAT_CONPROFITMAX_TRADES	Количество трейдов, сформировавших STAT_CONPROFITMAX (максимальная прибыль в последовательности прибыльных трейдов)
STAT_MAX_CONPROFIT_TRADES	Количество трейдов в самой длинной серии прибыльных трейдов STAT_MAX_CONWINS
STAT_CONLOSSMAX_TRADES	Количество трейдов, сформировавших STAT_CONLOSSMAX (максимальный убыток в последовательности убыточных трейдов)
STAT_MAX_CONLOSS_TRADES	Количество трейдов в самой длинной серии убыточных трейдов STAT_MAX_CONLOSSES

Попробуем воспользоваться представленными показателями, чтобы создать свой собственный комплексный критерий качества эксперта. Для этого нам нужен некий "подопытный" пример MQL-программы. Возьмем за отправную точку эксперт *MultiMartingale.mq5*, но упростим его: уберем мультивалютность, встроенную обработку ошибок и работу по расписанию. Более того, выберем для него сигнальную торговую стратегию с однократным расчетом на баре, то есть по ценам открытия. Это позволит ускорить оптимизацию и расширить поле для экспериментов.

Стратегия будет основываться на состояниях перекупленности и перепроданности, определяемых индикатором OsMA. Динамически находить границы избыточной волатильности, означающей торговые сигналы, поможет индикатор Bollinger Bands, наложенный на OsMA.

Когда OsMA будет возвращаться внутрь коридора, пересекая нижнюю границу снизу вверх, будем открывать покупку. Когда OsMA будет аналогичным образом пересекать сверху вниз верхнюю границу, будем продавать. Для выхода из позиций используем еще один индикатор — скользящую среднюю, также примененную к OsMA. Если OsMA продемонстрирует обратное движение (вниз для длинной позиции или вверх для короткой) и коснется MA, позиция будет закрыта. Данная стратегия иллюстрируется следующим скриншотом.



Торговая стратегия на индикаторах OsMA, BBands и MA

Синяя вертикальная линия соответствует бару, на котором открыта покупка, так как на двух предыдущих барах произошло пересечение нижней линии Боллинджера гистограммой OsMA снизу вверх (в подокне это место помечено полой синей стрелкой). Красная вертикальная линия — это место возникновения обратного сигнала, поэтому покупка была закрыта и открыта продажа. В подокне в этом месте (а точнее, на двух предыдущих барах, где стоит полая красная стрелка) гистограмма OsMA пересекает верхнюю линию Боллинджера сверху вниз. Наконец зеленая линия обозначает закрытие продажи, из-за того, что гистограмма стала подниматься выше красной MA.

Дадим эксперту говорящее имя *BandOsMA.mq5*. В общие настройки войдут магическое число, фиксированный лот и дистанция стоплосса в пунктах. Для стоплосса оставим сопровождение с помощью *TrailingStop* из прошлого примера. Тейкпрофит здесь не используется.

```
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;
```

Три группы настроек предназначены для индикаторов.

```
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;
```

В эксперте *MultiMartingale.mq5* у нас фактически не было торговых сигналов: в какую сторону открываться, задавал пользователь. Здесь у нас появились торговые сигналы, и имеет смысл оформить их отдельным классом. Для начала опишем абстрактный интерфейс *TradingSignal*.

```
interface TradingSignal
{
   virtual int signal(void);
};
```

Он такой же простой, как и другой наш интерфейс *TradingStrategy*. И это хорошо. Чем проще интерфейсы и объекты, тем вероятнее, что они отвечают за один единственный род деятельности, что является хорошим стилем программирования, поскольку минимизирует ошибки и делает более понятными крупные программные проекты. Благодаря абстракции в любой программе, использующий *TradingSignal*, можно будет заменить один сигнал на другой. Впрочем, и как одну стратегию — на другую. У нас сейчас стратегии отвечают за подготовку и отправку приказов, а сигналы их инициируют за счет анализа рынка.

В нашем случае конкретную реализацию *TradingSignal* упакуем в класс *BandOsMaSignal*. Разумеется, нам потребуются переменные для хранения дескрипторов 3-х индикаторов. Создание экземпляров индикаторов и их удаление производится, соответственно, в конструкторе и деструкторе. Все параметры будут передаваться из входных переменных. Обратите внимание, что *iBands* и *iMA* строятся на дескрипторе *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);
   }
   ...
```

Направление текущего торгового сигнала находится в переменной *direction*: 0 — нет сигналов (неопределенная ситуация), +1 — покупка, -1 — продажа. Заполнение этой переменной произведем в методе *signal*. Его код повторяет на MQL5 приведенное выше словесное описание сигналов.

```
   virtual int signal(void) override
   {
      double osma[2], upper[2], lower[2], ma[2];
      // получаем два значения каждого индикатора на барах 1 и 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;

      // если уже был сигнал, проверяем не закончился ли он
      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;
            }
         }
      }

      // в любом случае проверяем, нет ли нового сигнала
      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;
   }
};
```

Как легко заметить, значения индикаторов считываются для баров 1 и 2, поскольку мы будем работать по открытию бара, и 0-й бар не то что незавершенный, а он только что открылся, когда мы вызовем метод *signal*.

Новый класс, реализующий интерфейс *TradingStrategy*, назовем *SimpleStrategy*.

В нем мы найдем кое-что новое, но и кое-что старое. В частности, в нем остались автоуказатели для *PositionState* и *TrailingStop*, зато добавился автоуказатель на сигнал *TradingSignal*. Также, поскольку мы собираемся торговать только по открытию баров, потребовалась переменная *lastBar*, в которой будет храниться время последнего обработанного бара.

```
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;
   ...
```

В конструктор *SimpleStrategy* передаются глобальные параметры, а также указатель на объект *TradingSignal* — очевидно, что это в данном случае будет *BandOsMaSignal*, и его должен будет создать вызывающий код. Далее конструктор пытается найти среди существующих позиций те, что имеют нужные *magic*-число и символ, и в случае успеха подключает к ней сопровождение. Это пригодится в случае, если в работе эксперт возник по тем или иным причинам перерыв, а позиция уже была открыта.

```
public:
   SimpleStrategy(TradingSignal *signal, const ulong m, const int sl, const double v)
      command(signal), magic(m), stopLoss(sl), lots(v), lastBar(0)
   {
      // подбираем "свою" позицию среди существующих (если есть подходящая)
      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: закрыть лишние позиции - это не допускается стратегией
      }
      else if(n > 0)
      {
         position = new PositionState(tickets[0]);
         if(stopLoss)
         {
           trailing = new TrailingStop(tickets[0], stopLoss, stopLoss / 50);
         }
      }
   }
```

Реализация метода *trade* во многом похожа на пример с мартингейлом, но здесь отсутствуют умножения лотов и добавился вызов метода *signal*.

```
      virtual bool trade() override
      {
         // работаем только один раз при появлении нового бара
         if(lastBar == iTime(_Symbol, _Period, 0)) return false;

         int s = command[].signal(); // получаем сигнал

         ulong ticket = 0;

         if(position[] != NULL)
         {
            if(position[].refresh()) // позиция существует
            {
               // сигнал изменился на противоположный или пропал
               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
                  {
                     // актуализируем внутренний флаг 'ready'
                     // согласно тому, было или нет закрытие
                     position[].refresh();
                  }
               }
               else
               {
                  position[].update();
                  if(trailing[]) trailing[].trail();
               }
            }
            else // позиция закрыта
            {
               position = NULL;
            }
         }

         if(position[] == NULL && s != 0)
         {
            ticket = (s == +1) ? openBuy() : openSell();
         }

         if(ticket > 0) // новая позиция только что открыта
         {
            position = new PositionState(ticket);
            if(stopLoss)
```

```
         {
            trailing = new TrailingStop(ticket, stopLoss, stopLoss / 50);
         }
      }
      // запоминаем текущий бар
      lastBar = iTime(_Symbol, _Period, 0);
      
      return true;
   }
```

Вспомогательные методы *openBuy*, *openSell* и другие претерпели минимальные изменения, поэтому мы не станем их приводить (полный исходный код прилагается).

Поскольку в данном эксперте у нас всегда только одна стратегия, в отличие от мультивалютного мартингейла, где каждый символ требовал собственных настроек, исключим пул стратегий и будем управлять объектом стратегии напрямую.

```
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();
   }
}
```

Теперь, вооружившись экспертом как инструментом, перейдем вплотную к исследованию тестера. Для начала создадим вспомогательную структуру для запроса и хранения всех статистических данных *TesterRecord*.

```
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);
      }
   }
};
```

В данном случае строковое поле *feature* нужно только для информативного вывода в журнал. Для сохранения всех показателей (например, чтобы иметь возможность позднее сгенерировать собственную форму отчета) достаточно просто массива типа *double* соответствующей длины.

С помощью структуры проверим в обработчике *OnDeinit*, что MQL5 API возвращает нам те же значения, что и отчет тестера.

```
void OnDeinit(const int)
{
   TesterRecord stats[];
   TesterRecord::fill(stats);
   ArrayPrint(stats, 2);
}
```

Например, при запуске на EURUSD,H1 с депозитом 10000 и без всяких оптимизаций (с настройками по умолчанию) получим за 2021 год примерно такие величины (фрагмент):

```
                       [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
```

Зная все эти величины, мы можем изобретать собственную формулу комбинированного показателя качества эксперта и заодно целевую функцию оптимизации. Но значение этого показателя в любом случае нужно будет сообщать тестеру. И именно для этого служит функция *OnTester*.

6.5.7 Событие OnTester

Событие *OnTester* генерируется по окончании тестирования эксперта на исторических данных (имеется в виду как отдельный прогон тестера, инициированный пользователем, так и один из множества прогонов, автоматически запускаемых тестером во время оптимизации). Для обработки события *OnTester* MQL-программа должна иметь в исходном коде соответствующую функцию, но это не обязательно. Даже без функции *OnTester* эксперты могут успешно оптимизироваться на основе стандартных критериев.

Функция может быть использована только в экспертах.

double OnTester()

Функция предназначена для расчета некоторого значения типа *double*, используемого в качестве пользовательского критерия оптимизации (*Custom max*). Подбор критерия важен в первую очередь для успешной генетической оптимизации, но также позволяет пользователю оценивать и сравнивать влияние разных настроек.

При генетической оптимизации сортировка результатов в пределах одного поколения производится по убыванию критерия. То есть, лучшими с точки зрения критерия оптимизации считаются результаты с наибольшим значением. Худшие значения при такой сортировке впоследствии отбрасываются и не принимают участия в формировании следующего поколения.

> Обратите внимание: значения, возвращаемые функцией *OnTester*, принимаются в расчет, только когда в настройках тестера выбран пользовательский критерий. Наличие функции *OnTester* не означает автоматически её использование "генетикой".
>
> К сожалению, MQL5 API не содержит средств, чтобы программно узнать, какой оптимизационный критерий в настройках тестера выбрал пользователь. А иногда это очень важно знать для реализации собственных аналитических алгоритмов постобработки результатов оптимизации.

Функция вызывается ядром только в тестере, непосредственно перед вызовом функции *OnDeinit*.

Для расчета возвращаемого значения мы можем использовать как стандартные статистические показатели, доступные через функцию *TesterStatistics*, так и свои произвольные расчеты.

Создадим в эксперте *BandOsMA.mq5* обработчик *OnTester*, учитывающий несколько показателей: прибыль, прибыльность, количество трейдов и коэффициент Шарпа. Все показатели перемножим, предварительно взяв от каждого квадратный корень. Разумеется, у каждого разработчика могут быть свои предпочтения и идеи по конструированию подобных обобщенных критериев качества.

```
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)));
}
```

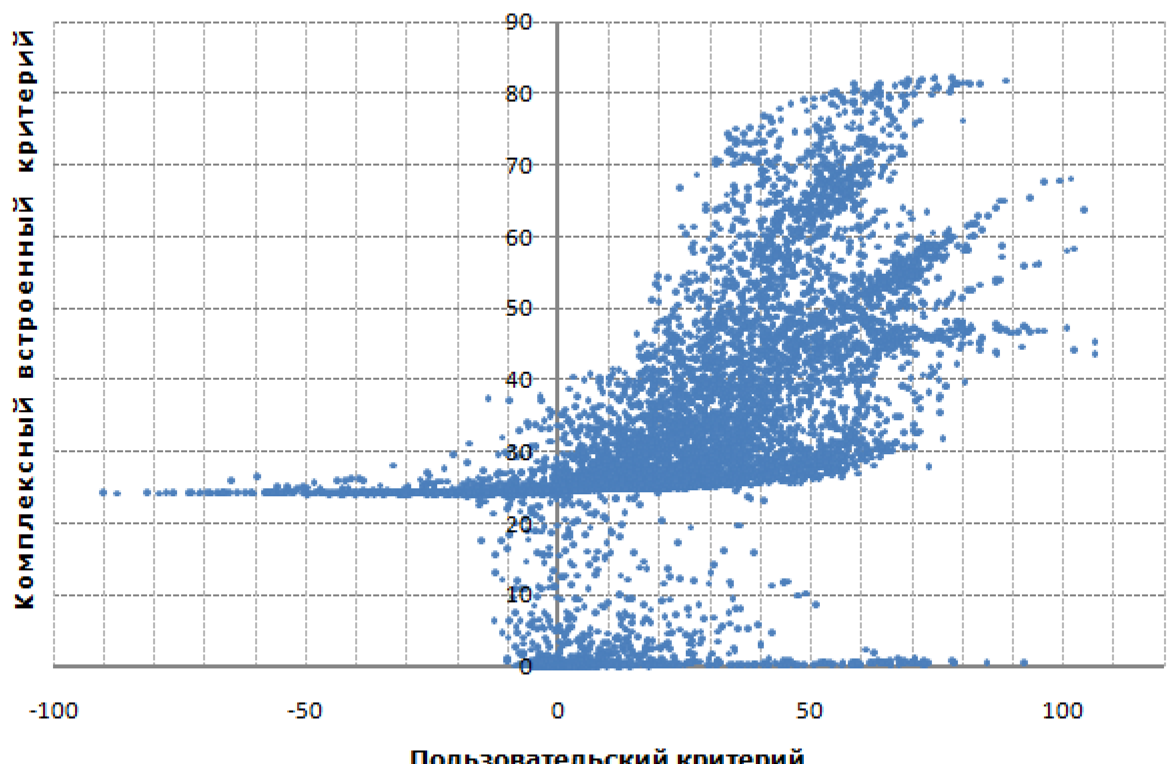
В журнале единичного теста выводится строка со значением функции *OnTester*.

Запустим генетическую оптимизацию эксперта за 2021 год на EURUSD,H1 с подбором параметров индикаторов и размера стоплосса (к книге прилагается файл *MQL5/Presets/MQL5Book/BandOsMA.set*). Для проверки качества оптимизации включим также форвард тесты с начала 2022 года (5 месяцев).

Сначала сделаем оптимизацию по нашему критерию.

Как известно, MetaTrader 5 сохраняет в результатах оптимизации все стандартные критерии в дополнение к текущему, используемому в ходе оптимизации. Это позволяет по завершению оптимизации смотреть на результаты, так сказать, под разными углами зрения: достаточно в правом верхнем углу панели с таблицей выбрать другой критерий из выпадающего списка. Таким образом, хотя мы делали оптимизацию по своему критерию, нам также доступен и наиболее интересный встроенный комплексный критерий.

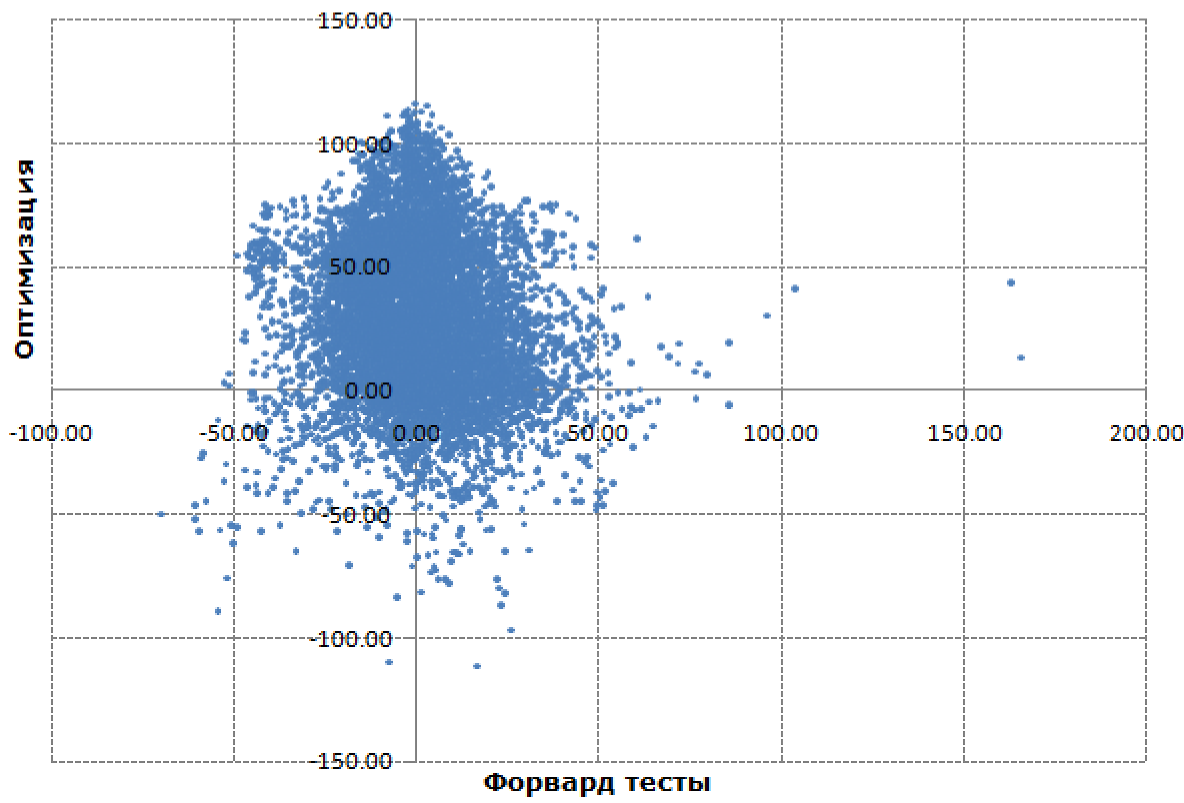
Мы можем экспортировать таблицу оптимизации в XML-файл сначала при выбранном нашем критерии, а затем под другим именем — с комплексным критерием (к сожалению, в файл экспорта записывается лишь один критерий; важно не менять сортировку между двумя операциями экспорта). Это дает возможность во внешней программе объединить 2 таблицы и построить диаграмму, на которой по осям отложены два критерия, а каждая точка обозначает сочетание критериев в одном прогоне.



Сопоставление пользовательского и комплексного критерия оптимизации

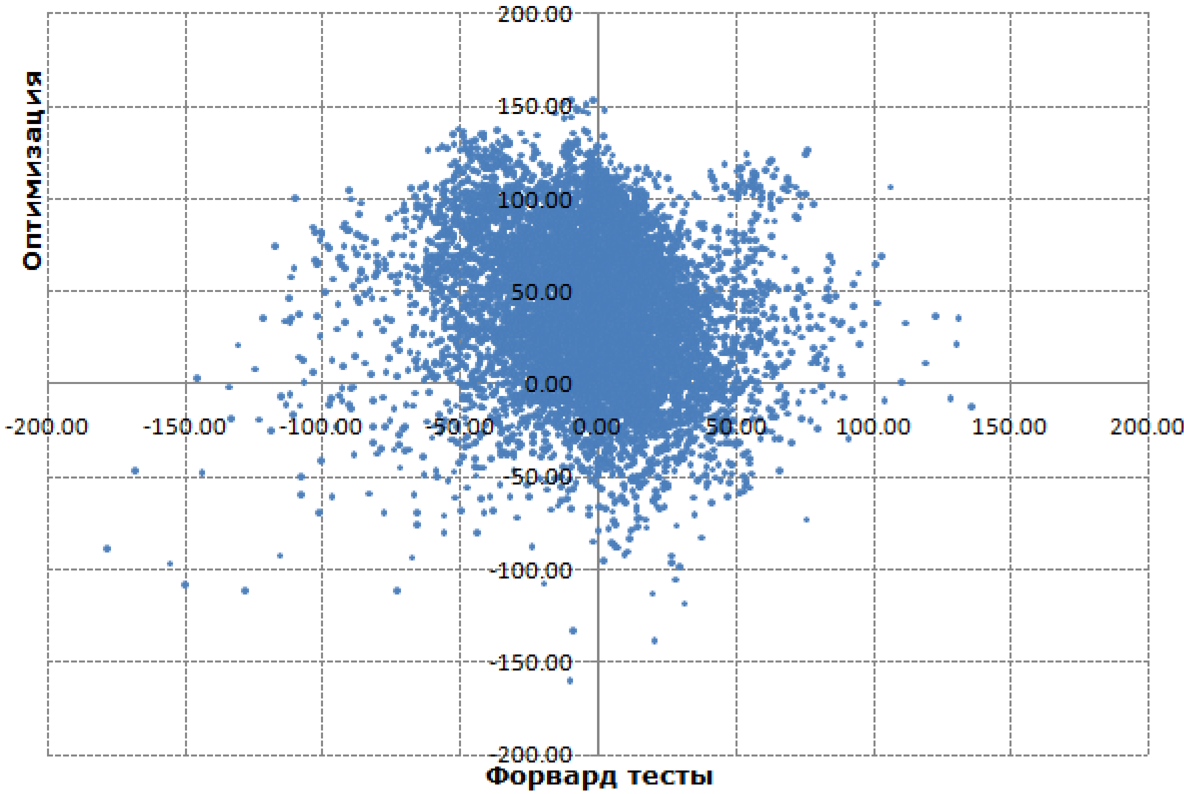
В комплексном критерии мы наблюдаем многоуровневую структуру, так как он считается по формуле с условиями: где-то срабатывает одна ветвь, а где-то — другая. Наш пользовательский критерий всегда считается по одной формуле. Также отметим наличие отрицательных значений в нашем критерии (это ожидаемо) и заявленный диапазон 0-100 у комплексного критерия.

Проверим, насколько хорош наш критерий за счет анализа его значений на форвард периоде.



Значения пользовательского критерия на периодах оптимизации и форвард-тестов

Как и следовало ожидать, лишь часть хороших показателей оптимизации сохранилась на форварде. Но нас больше интересует не критерий, а прибыль. Посмотрим на её распределение в связке оптимизация-форвард.



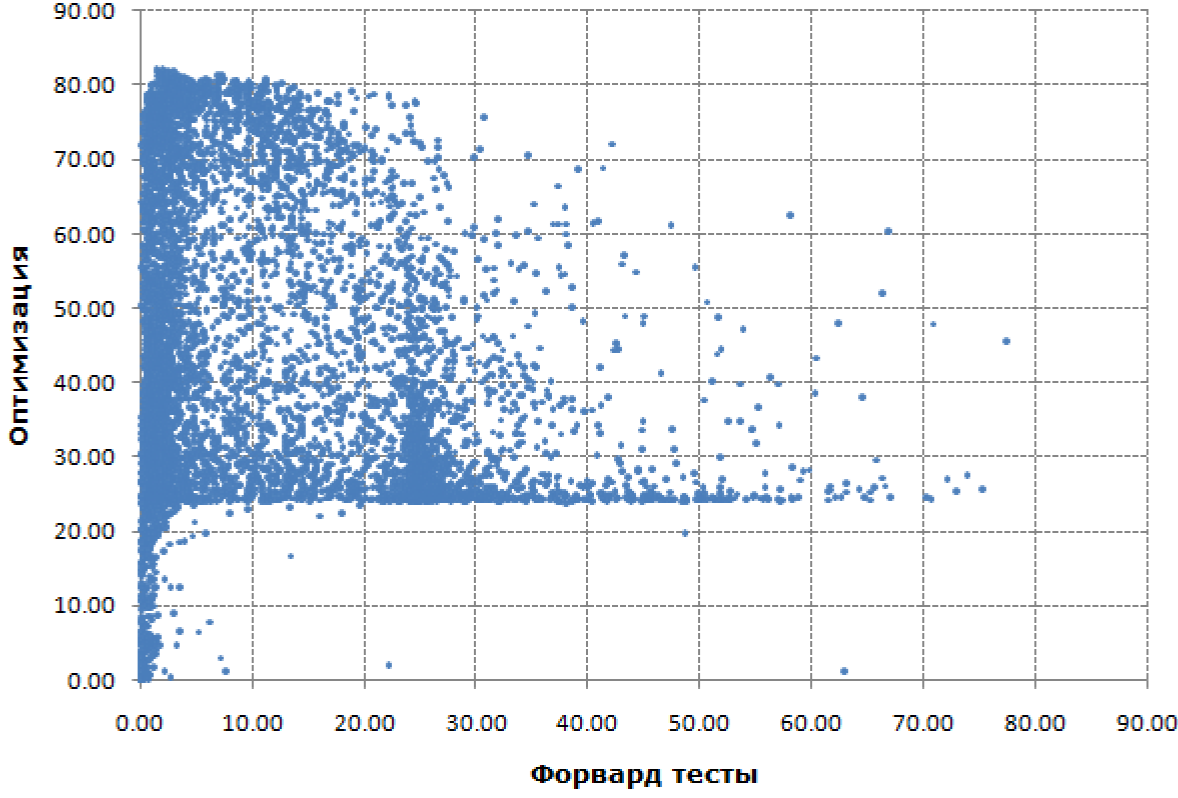
Прибыль на периодах оптимизации и форвард-тестов

Здесь картина похожая. Из 6850 проходов с прибылью на периоде оптимизации — 3123 оказались прибыльными и на форварде (45%). А из первой 1000 наилучших — лишь 323, что не очень хорошо. Следовательно, для выявления стабильных прибыльных настроек у этого эксперта потребуется еще много работы. Но может быть дело в критерии оптимизации?

Повторим оптимизацию, на этот раз — по встроенному комплексному критерию.

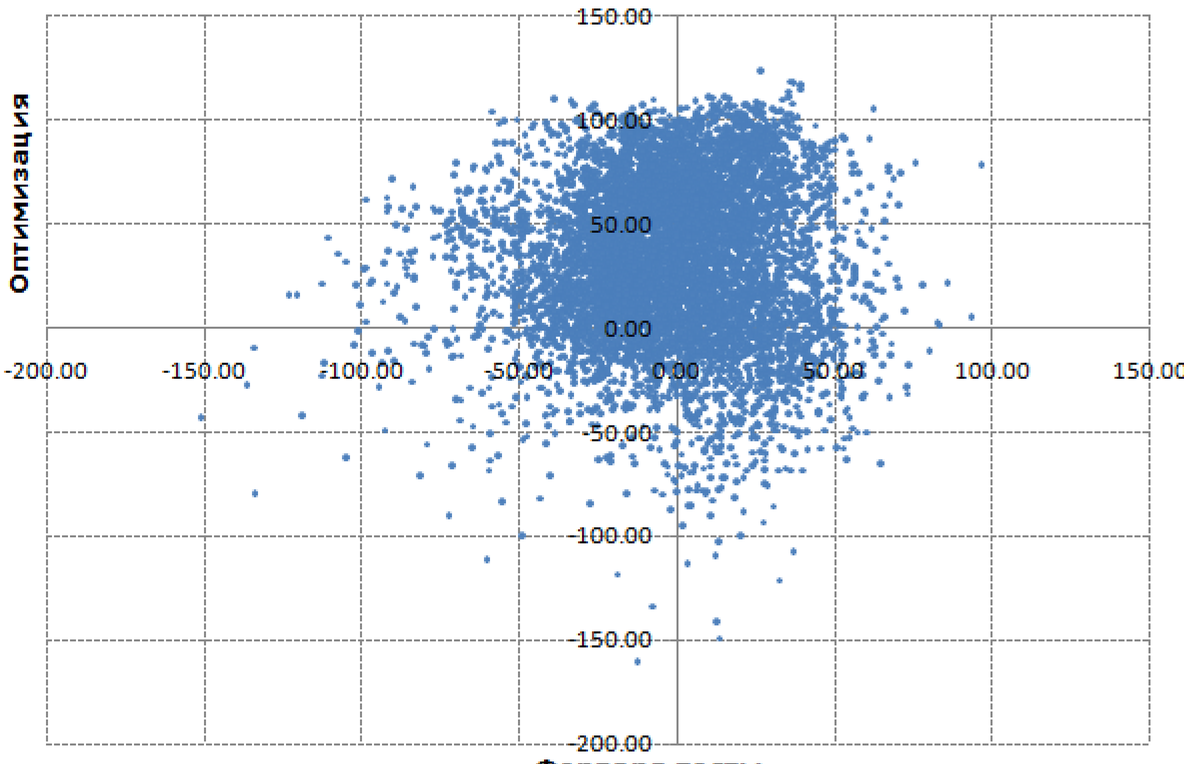
> Внимание! MetaTrader 5 в ходе оптимизаций генерирует кэши оптимизаций — opt-файлы по пути *Tester/cache*. При запуске очередной оптимизации он ищет подходящие кэши, чтобы продолжить оптимизацию. При наличии файла кэша с прежними настройками процесс запускается не с самого начала, а с учетом предыдущих результатов. Это позволяет выстраивать генетические оптимизации в цепочки, в предположении о нахождении лучших результатов (ведь каждая генетическая оптимизация — это случайный процесс).
>
> MetaTrader 5 не учитывает критерий оптимизации как отличительный фактор в настройках. Кому-то это может пригодиться, исходя из вышеизложенного, но нам это сейчас помешает. Нам для проведения чистого эксперимента требуется оптимизация с чистого листа (извините за тавтологию). Поэтому мы не можем сразу после первой оптимизации с использованием нашего критерия запустить вторую с использованием комплексного критерия.
>
> Отключить текущее поведение из интерфейса терминала нельзя. Поэтому следует либо удалить, либо переименовать (сменить расширение) предыдущий opt-файл вручную в любом файловом менеджере. Чуть позже мы познакомимся с директивой препроцессора для тестера *tester_no_cache*, которая может быть указана в исходном коде конкретного эксперта — она позволяет отключить считывание кэша.

Сопоставление значений комплексного критерия на периодах оптимизации и форвард-периода принимает следующий вид.



Комплексный критерий на периодах оптимизации и форвард-тестов

А вот как обстоит дело со стабильностью прибыли на форвардах.



Прибыль на периодах оптимизации и форвард-тестов

Из 5952 положительных результатов на истории лишь 2655 (тоже примерно 45%) остались в плюсе. Но из первой 1000 удачными на форварде оказались уже 581.

Итак, мы убедились, что чисто технически использовать *OnTester* довольно просто, но наш критерий работает хуже, чем встроенный (при прочих равных условиях), хотя и он вряд ли идеален. Таким образом, с точки зрения поиска формулы самого критерия и последующего обоснованного выбора параметров без заглядывания в будущее, вопросов к содержимому *OnTester* больше, чем ответов.

Здесь программирование плавно перетекает в исследовательско-научную деятельность и за рамки данной книги. Но мы приведем один пример критерия, рассчитываемого на собственной метрике, а не на готовых показателях *TesterStatistics*. Речь пойдет о критерии R2, известном также как коэффициент детерминации (*RSquared.mqh*).

Создадим функцию для расчета R2 по кривой баланса. Известно, что при торговле постоянным лотом идеальная торговая система должна показывать баланс в виде прямой линии. Мы сейчас используем постоянный лот, и потому нам это подойдет. А как быть с R2 в случае переменных лотов, мы разберемся чуть позже.

По сути R2 представляет собой обратную меру дисперсии данных относительно построенной по ним линейной регрессии. Диапазон значений R2 лежит от минус бесконечности до +1 (правда большие отрицательные значения в нашем случае очень маловероятны). Очевидно, что найденная линия попутно характеризуется углом наклона, поэтому с целью универсализации кода будем сохранять в качестве промежуточного результата и R2, и тангенс угла в структуре R2A.

```
struct R2A
{
   double r2;    // квадрат коэффициента корреляции
   double angle; // тангенс угла наклона
   R2A(): r2(0), angle(0) { }
};
```

Расчет показателей выполняется в функции *RSquared*, принимающей на вход массив данных и возвращающий структуру 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;
}
```

Для оптимизации нам нужно одно значение критерия, и угол здесь важен, потому что хорошую оценку R2 может получить и ровная спадающая кривая баланса с отрицательным уклоном. Поэтому напишем еще одну функцию, которая будет "минусовать" любые оценки R2 с отрицательным углом. Значение R2 мы берем по модулю, потому что оно и само может быть отрицательным в случае очень плохих (разрозненных) данных, которые не укладываются в нашу линейную модель. Таким образом, мы должны предотвратить ситуацию, когда минус на минус дают плюс.

```
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;
}
```

Дополнительно в нашем критерии учитывается размер ряда, который соответствует количеству трейдов. За счет этого увеличение числа сделок будет увеличивать показатель.

Имея в распоряжении данный инструмент, реализуем в эксперте функцию вычислении линии баланса, и найдем для неё R2. В конце умножим величину на 100, тем самым преобразовав масштаб к диапазону встроенного комплексного критерия.

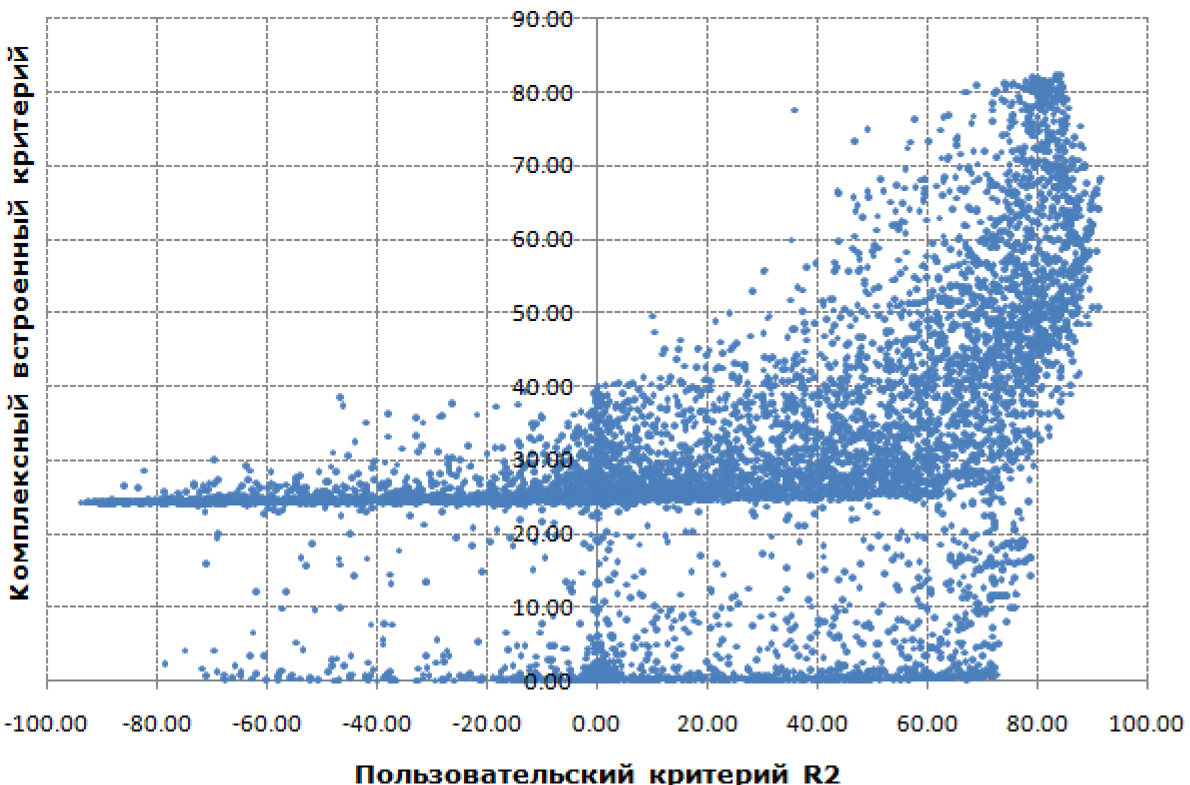
```
#define STAT_PROPS 4

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[]; // нужно только из-за прототипа 'select', но полезно для отладки

   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;
}
```

В обработчике *OnTester* будем использовать новый критерий под директивой условной компиляции, поэтому нужно раскомментировать директиву *#define USE_R2_CRITERION* в начале исходного кода.

```
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
}
```

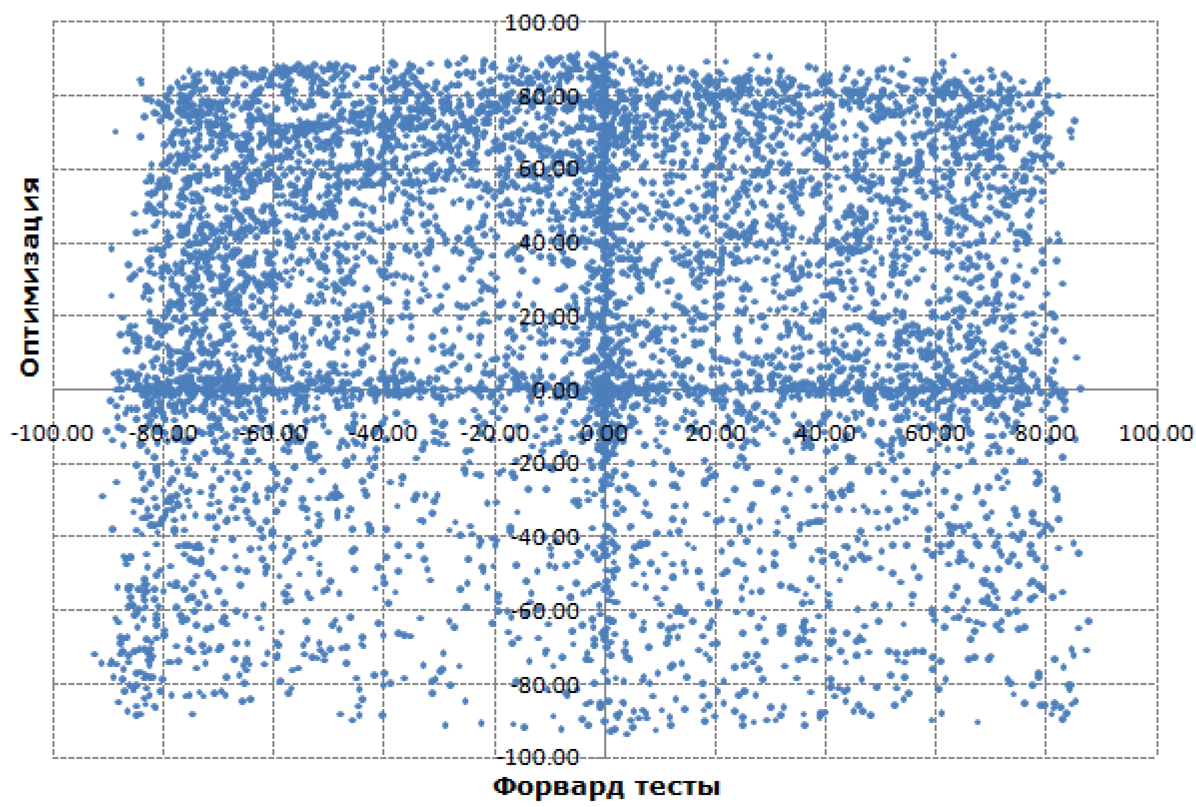
Удалим прежние результаты оптимизаций (opt-файлы с кэшем) и запустим новую оптимизацию эксперта — по критерию R2.

При сравнении значений критерия R2 с комплексным критерием можно сказать, что "конвергенция" между ними увеличилась.



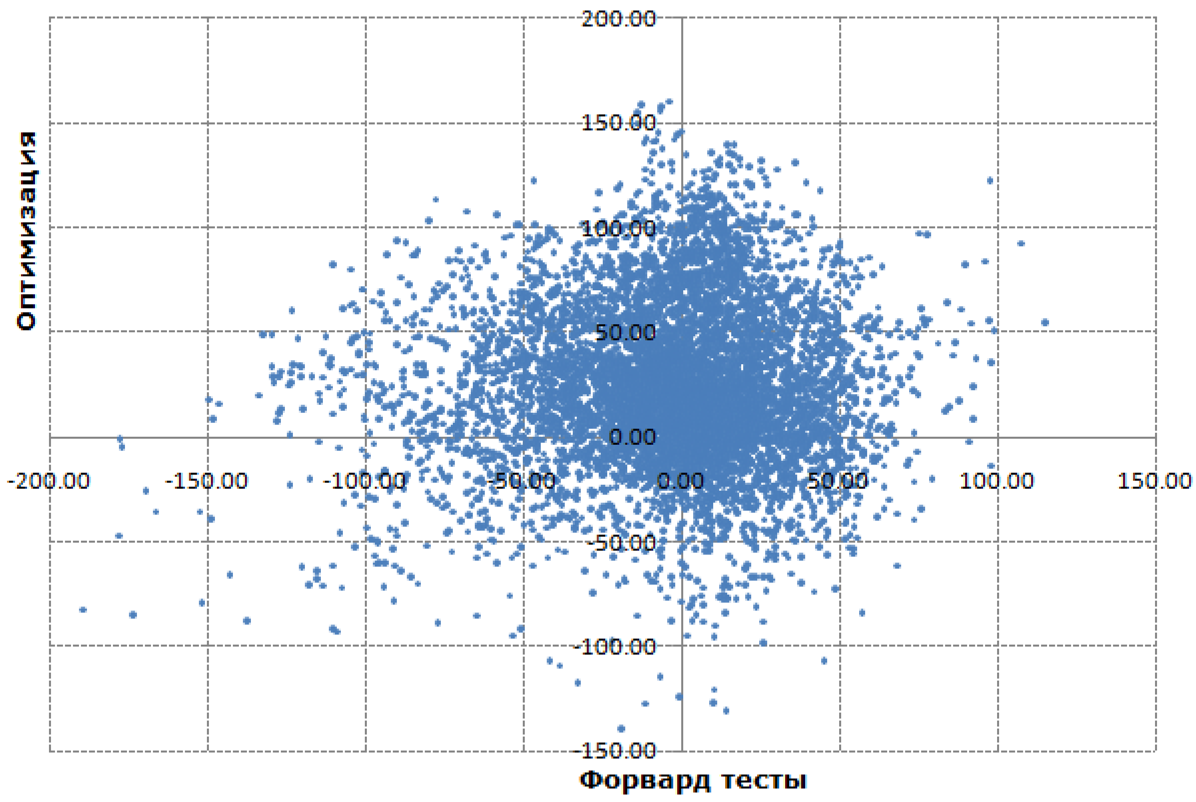
Сравнение пользовательского критерия R2 и комплексного встроенного критерия

Значения критерия R2 в окне оптимизации и на форвард-периоде для соответствующих наборов параметров выглядят следующим образом.



Критерий R2 на периодах оптимизации и форвард-тестов

А вот как сочетаются прибыли в прошлом и в будущем.



Прибыль на периодах оптимизации и форвард-тестов для R2

Статистика такова: из числа прошлых прибыльных 5582 проходов таковыми остались 2638 (47%), а из первой 1000 наиболее прибыльных проходов — 566, что сопоставимо со встроенным комплексным критерием.

Как уже было сказано выше, данная статистика пока представляет собой сырой исходный материал для следующих, более интеллектуальных этапов оптимизации экспертов, далеких от "сухого" программирования. Мы же сконцентрируемся на прочих, чисто программных аспектах оптимизации.

6.5.8 Авто-настройка: ParameterGetRange и ParameterSetRange

В предыдущем разделе мы научились передавать тестеру критерий оптимизации, но обошли вниманием один важный нюанс. Если заглянуть в журналы наших оптимизаций, там можно в большом количестве встретить такие сообщения об ошибках.

```
...
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
...
```

Иными словами, каждые несколько тестовых проходов что-то неправильно со входными параметрами, и такой прогон не выполняется. Дело в том, что в обработчике *OnInit* имеется проверка:

```
if(FastOsMA >= SlowOsMA) return INIT_PARAMETERS_INCORRECT;
```

С нашей стороны вполне логично налагать такое ограничение, чтобы период медленной MA был больше периода быстрой. Но тестер подобных прикладных тонкостей нашего алгоритма не знает, и потому пытается перебирать самые разные сочетания периодов, включая и некорректные. В принципе, это обычная ситуация для оптимизации, но она имеет одно негативное последствие.

Поскольку мы применяем генетическую оптимизацию, в каждом поколении оказывается несколько забракованных образцов, которые не участвуют в дальнейших мутациях. По тем или иным причинам оптимизатор MetaTrader 5 не восполняет эти потери, то есть не генерирует им замену. А меньший размер популяции может негативно сказываться на качестве. Таким образом, следует придумать способ, как обеспечить перебор входных настроек только в корректных сочетаниях. Тут нам на помощь приходят две функции MQL5 API: *ParameterGetRange* и *ParameterSetRange*.

Обе функции имеют по два перегруженных прототипа, отличающихся типами параметров: *long* и *double*. Вот как описаны 2 варианта функции *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)

Функция получает для заданной по имени входной переменной информацию о её текущем значении (*value*), диапазоне значений (*start*, *stop*) и шаге изменения (*step*) при оптимизации.

Кроме того в переменную *enable* записывается признак того, включена ли оптимизация по входной переменной с именем name.

Функция возвращает признак успеха (*true*) или ошибки (*false*).

Функция может вызываться только из трех специальных обработчиков, связанных с оптимизацией: *OnTesterInit*, *OnTesterPass* и *OnTesterDeinit*. Мы расскажем про них в следующем разделе. Но как можно догадаться из названий, *OnTesterInit* вызывается перед началом оптимизации, *OnTesterDeinit* — по окончании оптимизации, а *OnTesterPass* — после каждого прохода в процессе оптимизации. Нас пока интересует только *OnTesterInit*. Она, также как и две другие функции, не имеет параметров и может быть описана с типом *void*, то есть ничего не возвращать.

Два варианта функции *ParameterSetRange* имеют похожие прототипы и выполняют обратное действие: задают оптимизационные свойства входного параметра эксперта.

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)

Функция устанавливает правила модификации *input*-переменной с названием *name* при оптимизации: значение, шаг изменения, начальное и конечное значения.

Эта функция может вызываться только из обработчика *OnTesterInit* при запуске оптимизации в тестере стратегий.

Таким образом, с помощью функций *ParameterGetRange* и *ParameterSetRange* можно анализировать и задавать новые значения диапазона и шага, а также полностью исключать или наоборот включать те или иные параметры из оптимизации, несмотря на настройки в тестере стратегий. Это позволяет создавать собственные сценарии для управления пространством входных параметров при оптимизации.

Функция позволяет использовать в оптимизации даже переменные, объявленные с модификатором *sinput* (они недоступны для включения в оптимизацию пользователем).

> Внимание! После вызова *ParameterSetRange* с изменением настроек конкретной входной переменной, последующие вызовы *ParameterGetRange* не "увидят" этих изменений и будут по-прежнему возвращать начальные настройки. Это делает невозможным использование функций совместно в сложных программных продуктах, где настройками могут заниматься разные классы и библиотеки от независимых разработчиков.

Усовершенствуем эксперт *BandOsMA* с использованием новых функций. Обновленная версия прилагается под именем *BandOsMApro.mq5* ("pro" можно условно расшифровать как "parameter range optimization").

Итак, у нас появляется обработчик *OnTesterInit*, в котором мы считываем настройки для параметров *FastOsMA* и *SlowOsMA* и проверяем, включены ли они в оптимизацию. Если да, требуется их выключить, и предложить что-то взамен.

```
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));
   }
}
```

К сожалению, из-за добавления *OnTesterInit* компилятор требует также добавить *OnTesterDeinit*, хотя эта функция нам ни к чему. Но мы вынуждены согласиться и добавить пустой обработчик.

```
void OnTesterDeinit()
{
}
```

Наличие в коде функций *OnTesterInit/OnTesterDeinit* приведет к тому, что при старте оптимизации в терминале откроется дополнительный график с запущенной на нем копией нашего эксперта. Она работает в особом режиме, позволяющем принимать дополнительные данные "*фреймы*" от тестируемых копий на агентах, но мы изучим эту возможность позднее. Пока для нас важно отметить, что все операции с файлами, журналами, графиком, объектами работают в этой вспомогательной копии эксперта непосредственно в терминале, как обычно (а не на агенте). В частности, все сообщения об ошибках и вызовы *Print* будут отображаться в журнале на закладке *Эксперты* терминала.

Имея информацию о диапазонах изменения и шаге этих параметров, мы можем буквально пересчитать все правильные сочетания. Эта задача поручена отдельной функции *Iterate*, потому что аналогичную операцию должны будут воспроизвести и копии эксперта на агентах, в обработчике *OnInit*.

В функции *Iterate* мы пробегаем в двух вложенных циклах по периодам быстрой и медленной MA и подсчитываем количество допустимых сочетаний, т.е. когда период *i* меньше периода *j*. Необязательный параметр *find* потребуется нам при вызове *Iterate* из *OnInit*, чтобы по порядковому номеру сочетания вернуть пару *i* и *j*. И поскольку требуется возвращать 2 числа, мы объявили для них структуру *PairOfPeriods*.

```
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;
}
```

При вызове *Iterate* из *OnTesterInit* мы не используем параметр *find* и ведем подсчет до самого конца, а получившееся количество возвращаем в первом поле структуры. Это и будет диапазон значений некоего нового теневого параметра, для которого мы должны разрешить оптимизацию. Назовем его *FastSlowCombo4Optimization* и добавим в новую группу вспомогательных входных параметров. В одиночестве этот параметр тут будет оставаться недолго.

```
input group "A U X I L I A R Y"
sinput int FastSlowCombo4Optimization = 0;   // (reserved for optimization)
...
```

А пока вернемся в *OnTesterInit* и организуем на MQL5 оптимизацию по параметру *FastSlowCombo4Optimization* в нужном диапазоне с помощью *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);
   ...
}
```

Обратите внимание, в журнале терминала должно вывестись получившееся количество итераций для нового параметра.

Во время теста на агенте по номеру в *FastSlowCombo4Optimization* следует получить пару периодов, вызвав вновь *Iterate*, на этот раз с заполненным параметром *find*. Но проблема в том, что для этой операции требуется знать изначальные диапазоны и шаг изменения параметров *FastOsMA* и *SlowOsMA*. А эта информация есть только в терминале. Значит, нам нужно как-то передать её на агент.

Сейчас мы применим пока единственное известное нам решение: добавим еще 3 теневых параметра оптимизации и установим для них некие значения. В будущем мы познакомимся с технологией передачи файлов на агенты (см. Директивы препроцессора для тестера). Тогда мы сможем записать в файл весь массив посчитанных функцией *Iterate* индексов и отправить его на агенты. Тогда мы избавимся от трех лишних теневых параметров оптимизации.

Итак, добавим 3 входных параметра:

```
sinput ulong FastShadow4Optimization = 0;    // (reserved for optimization)
sinput ulong SlowShadow4Optimization = 0;    // (reserved for optimization)
sinput ulong StepsShadow4Optimization = 0;   // (reserved for optimization)
```

Мы используем тип *ulong* для экономии, чтобы в каждое значение упаковать по 2 целых *int*-числа. А вот как они заполняются в *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);
   ...
}
```

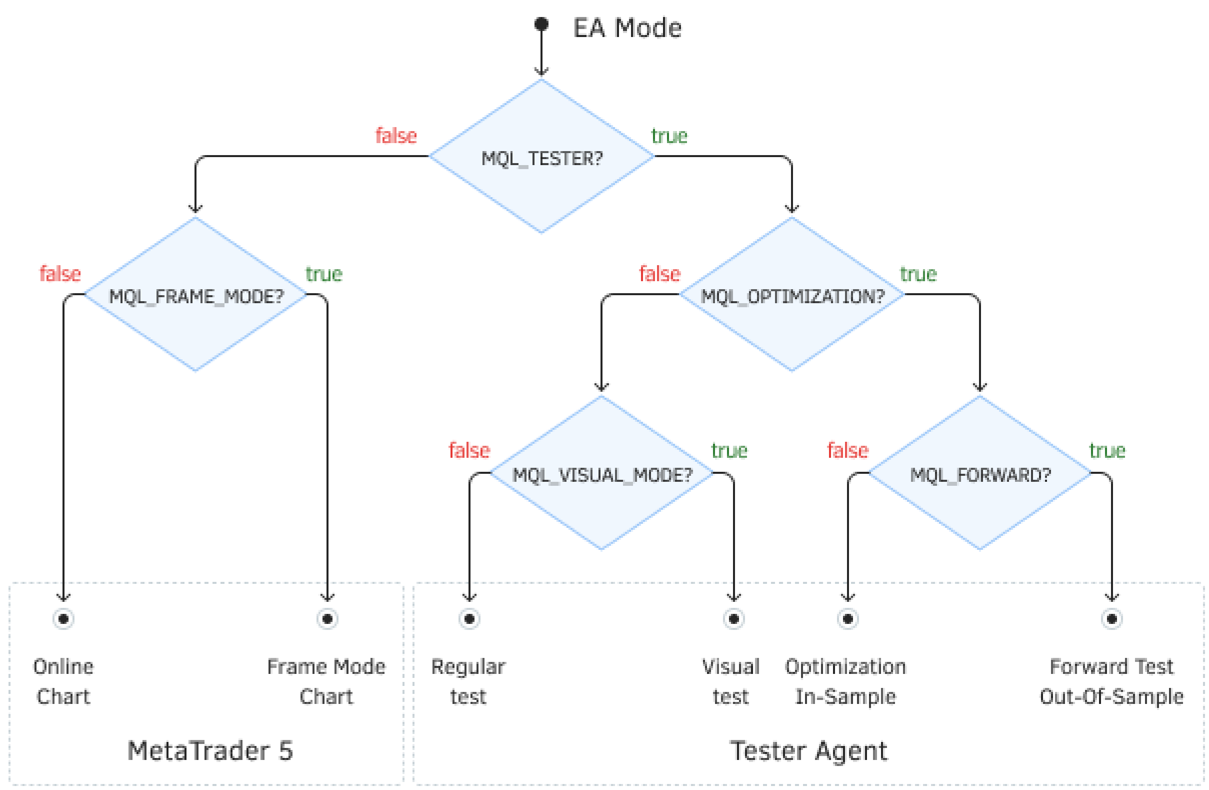
Все 3 параметра — неоптимизируемые (*false* во втором аргументе).

На этом мы разобрались с функцией *OnTesterInit* и должны обратиться к приемной стороне — обработчику *OnInit*.

```
int OnInit()
{
   // оставим проверку для одиночных тестов
   if(FastOsMA >= SlowOsMA) return INIT_PARAMETERS_INCORRECT;

   // при оптимизации требуем наличия теневых параметров
   if(MQLInfoInteger(MQL_OPTIMIZATION) && StepsShadow4Optimization == 0)
   {
      return INIT_PARAMETERS_INCORRECT;
   }

   PairOfPeriods p = {FastOsMA, SlowOsMA}; // по умолчанию работаем с обычными параме
   if(FastShadow4Optimization && SlowShadow4Optimization && StepsShadow4Optimization)
   {
      // если теневые параметра заполнены, раскодируем их в периоды
      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;
}
```

Напоминаем, что мы можем с помощью функции *MQLInfoInteger* определить все режимы работы эксперта, включая и те, что связаны с тестером и оптимизацией. Задав в качестве параметра один из элементов перечисления ENUM_MQL_INFO_INTEGER, мы получим в результате логический признак (*true/false*):

- MQL_TESTER - программа работает в тестере;
- MQL_VISUAL_MODE - тестер запущен в визуальном режиме;
- MQL_OPTIMIZATION - тестовый проход выполняется в ходе оптимизации (а не отдельно);
- MQL_FORWARD - тестовый проход выполняется на форвард-периоде после оптимизации (если задано настройками оптимизации);
- MQL_FRAME_MODE - эксперт запущен в особом сервисном режиме на графике терминала (а не на агенте) для управления оптимизацией (об этом подробнее в следующем разделе).



Режимы работы MQL-программ, связанные с тестером

Все готово для запуска оптимизации. Сразу в момент её начала, при упоминавшихся настройках *Presets/MQL5Book/BandOsMA.set*, мы увидим сообщение в журнале *Эксперты* терминала:

```
Parameter FastSlowCombo4Optimization is enabled with maximum: 698
```

На этот раз в журнале оптимизации не должно быть ошибок, и все поколения генерируются без сбоев.

```
...
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
...
```

Это можно определить даже по увеличившемуся общему времени оптимизации: раньше часть проходов отбраковывалась на ранней стадии, а теперь они все обрабатываются целиком.

Но у нашего решения есть и один минус. Теперь в рабочих настройках эксперта фигурирует не пара периодов в параметрах *FastOsMA* и *SlowOsMA*, а порядковый номер их комбинации среди всех возможных (*FastSlowCombo4Optimization*). Единственное, что мы можем сделать, это выводить раскодированные периоды в функции *OnInit*, что и было продемонстрировано выше.

Таким образом, найдя с помощью оптимизации хорошие настройки, пользователь, как обычно, выполнит одиночный прогон, чтобы уточнить поведение торговой системы. И в начале журнала тестирования должна появиться надпись вида:

```
MA periods are restored from shadow: FastOsMA=27 SlowOsMA=175
```

Тогда можно ввести указанные периоды в одноименные параметры, а все теневые параметры обнулить.

6.5.9 Группа OnTester-событий для контроля оптимизации

Для управления ходом оптимизации и передачи с агентов на терминал произвольных прикладных результатов (помимо показателей торговли) в MQL5 существует 3 особых события: *OnTesterInit*, *OnTesterDeinit*, *OnTesterPass*. Описав для них обработчики в коде, программист получит возможность выполнять нужные ему действия перед запуском оптимизации, после завершения оптимизации и при завершении каждого из отдельных проходов оптимизации (если с агента поступили прикладные данные — об этом чуть ниже).

Все обработчики являются опциональными. Как мы видели, оптимизация работает и без них. Также следует понять, что все 3 события работают только в ходе оптимизации, но не одиночного теста.

Эксперт с данными обработчиками автоматически загружается на отдельном графике терминала с указанными в тестере символом и периодом. Эта копия эксперта не торгует, а выполняет исключительно сервисные действия. В ней не работают все прочие обработчики событий, в частности, *OnInit*, *OnDeinit*, *OnTick*.

Чтобы в коде эксперта отличить, выполняется ли он в штатном торговом режиме на агенте или в сервисном режиме в терминале, следует вызвать функцию *MQLInfoInteger(MQL_FRAME_MODE)* — получим *true* или *false*. Как можно понять, этот сервисный режим также называемся режимом "фреймов" — пакетов прикладных данных, которые могут отправляться в терминал из экземпляров эксперта на агентах. Как это делается, мы покажем чуть позже.

В ходе оптимизации только один экземпляр эксперта работает в терминале и, при необходимости, принимает поступающие фреймы. Но еще раз уточним, что такой экземпляр эксперта запускается только при наличии в его коде одного из трех описываемых обработчиков событий.

Событие *OnTesterInit* генерируется при запуске оптимизации в тестере стратегий перед самым первым проходом. Обработчик имеет 2 варианта: с возвращаемым типом *int* и *void*.

```
int OnTesterInit(void)
void OnTesterInit(void)
```

В варианте с возвратом *int* значение ноль (INIT_SUCCEEDED) означает успешную инициализацию эксперта, запущенного на графике в терминале, что разрешает начать оптимизацию. Любое другое значение означает код ошибки, и оптимизация не начнется.

Вторая версия функции всегда подразумевает успешную подготовку эксперта к оптимизации.

На выполнение *OnTesterInit* отводится ограниченное время, по превышении которого будет произведено принудительное завершение работы эксперта, а сама оптимизация будет отменена. При этом в журнал тестера будет выведено соответствующее сообщение.

В предыдущем разделе мы видели пример того, как обработчик *OnTesterInit* использовался для модификации параметров оптимизации с помощью функций *ParameterGetRange/ParameterSetRange*.

void OnTesterDeinit(void)

Функция *OnTesterDeinit* вызывается по окончании оптимизации эксперта.

Функция предназначена для финальной обработки прикладных результатов оптимизации. Например, если в *OnTesterInit* был открыт файл для записи содержимого фреймов, то в *OnTesterDeinit* его нужно закрыть.

void OnTesterPass(void)

Событие *OnTesterPass* автоматически генерируется при поступлении фрейма данных во время оптимизации. Функция позволяет обработать прикладные данные, получаемые от экземпляров эксперта, выполняющихся на агентах во время оптимизации. Отправку фрейма с агента тестирования необходимо выполнять из обработчика *OnTester* с помощью функции *FrameAdd*.

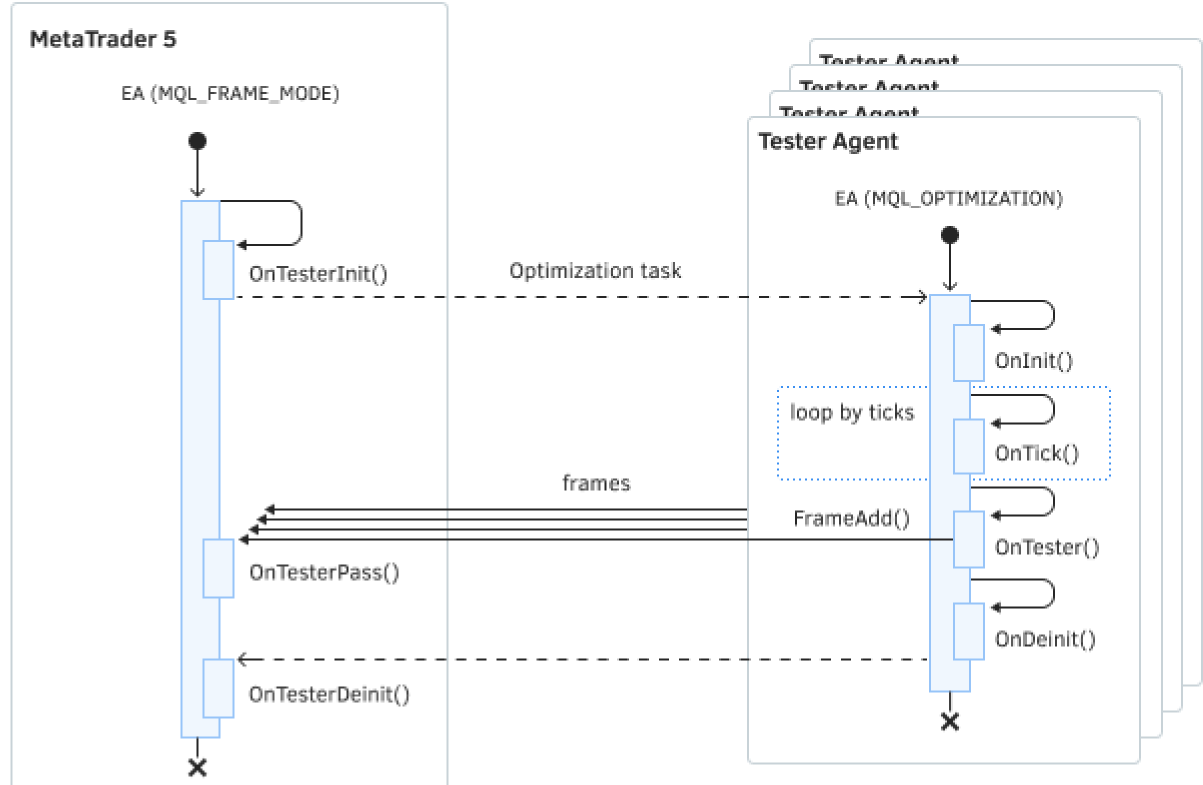


Диаграмма последовательности событий при оптимизации экспертов

Стандартный набор финансовых показателей о каждом проходе теста отправляется из агентов в терминал автоматически. Эксперт не обязан что-либо отправлять с помощью *FrameAdd*, если ему это не требуется. Если фреймы не используются, обработчик *OnTesterPass* не будет вызываться.

С помощью *OnTesterPass* можно динамически обрабатывать результаты оптимизации "на лету", например, отображать на графике в терминале, или складывать в файл для последующей пакетной обработки.

Для демонстрации возможностей обработчиков *OnTester*-событий нам нужно сначала изучить функции для работы фреймами. Они представлены в следующих разделах.

6.5.10 Отправка фреймов данных с агентов в терминал

MQL5 предоставляет группу функций для организации передачи и обработки собственных (прикладных) результатов оптимизации, в дополнение к стандартным финансовым показателям и статистике. Одна из этих — *FrameAdd* — предназначена для отправки данных с агентов тестирования, а остальные — для приема данных в терминале.

Формат обмена данными построен на так называемых фреймах (или кадрах). Это специальная внутренняя структура, которую эксперт в тестере может заполнить из массива простого типа (который не содержит строк, объектов классов или динамических массивов) или из файла с указанным именем (файл предварительно должен быть создан в "песочнице" агента). С помощью многократного вызова функции *FrameAdd* эксперт может посылать в терминал серию фреймов. Ограничений на количество фреймов нет.

Существует 2 варианта функции *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[])
```

Функция добавляет фрейм с данными в буфер для отправки на терминал. Параметры *name* и *id* представляют собой публичные метки, которые могут использоваться для фильтрации фреймов в функции *FrameFilter*. Параметр *value* позволяет передать произвольное числовое значение, которое можно задействовать, когда одного значения достаточно. Более объемные данные указываются либо в массиве *data* (может быть и массивом простых структур), либо в файле с именем *filename*.

Если объемных данных для передачи нет (например, нужно передать только статус процесса), используйте первую форму функции и вместо строки с именем файла укажите NULL, или вторую форму с фиктивным массивом нулевого размера.

Функция возвращает *true* в случае успеха.

Функцию можно вызывать только в обработчике *OnTester*.

Функция не имеет эффекта при вызове во время простого теста, то есть вне оптимизации.

> Отправлять данные можно только из агентов в терминал. В MQL5 нет механизмов для пересылки данных в обратном направлении в ходе оптимизации. Все данные, которые эксперт желает переслать на агенты, должны быть подготовлены и доступны (в виде входных параметров или файлов, подключенных директивами) перед началом оптимизации.

Мы рассмотрим пример использования *FrameAdd*, после того как познакомимся с функциями принимающей стороны в следующем разделе.

6.5.11 Получение фреймов данных в терминале

Фреймы, посланные с агентов тестирования функцией *FrameAdd*, попадают в терминал и записываются в порядке поступления в mqd-файл с именем эксперта в папку *каталог_терминала/MQL5/Files/Tester*. Поступление одного или сразу нескольких фреймов генерирует событие *OnTesterPass*.

Для анализа и чтения фреймов MQL5 API предоставляет 4 функции *FrameFirst*, *FrameFilter*, *FrameNext*, *FrameInputs*. Все функции возвращают логическое значение с признаком успеха (*true*) или ошибки (*false*).

Для доступа к имеющимся фреймам ядро поддерживает метафору внутреннего указателя на текущий фрейм. Указатель автоматически сдвигается вперед при чтении очередного фрейма функцией *FrameNext*, но его можно вернуть на начало всех фреймов с помощью *FrameFirst* или *FrameFilter*. Таким образом, MQL-программа может организовать перебор фреймов в цикле, пока не просмотрит все фреймы. И этот процесс можно при необходимости повторять, например, налагая разные фильтры в *OnTesterDeinit*.

bool FrameFirst()

Функция *FrameFirst* переводит внутренний указатель чтения фреймов на начало и сбрасывает фильтр (если он был ранее установлен с помощью функции *FrameFilter*).

В принципе, для однократного приема и обработки всех фреймов не требуется вызывать *FrameFirst*, так как указатель и так находится в начале при старте оптимизации.

bool FrameFilter(const string name, ulong id)

Устанавливает фильтр чтения фреймов и переводит внутренний указатель фреймов на начало. Фильтр будет влиять на то, какие фреймы попадут в последующие вызовы *FrameNext*.

Если в качестве первого параметра передана пустая строка, фильтр будет работать только по числовому параметру, то есть будут просматриваться все фреймы с указанным *id*. Если значение второго параметра равно ULONG_MAX, то работает только текстовый фильтр.

Вызов *FrameFilter("", ULONG_MAX)* эквивалентен вызову *FrameFirst()*, то есть равнозначен отсутствию фильтра.

> Если вызываете *FrameFirst* или *FrameFilter* в *OnTesterPass*, проверьте, действительно это то, что нужно: вероятно код содержит логическую ошибку — возможно зацикливание, чтение одного и того же фрейма, или увеличение вычислительной нагрузки в геометрической прогрессии.

bool FrameNext(ulong &pass, string &name, ulong &id, double &value)
bool FrameNext(ulong &pass, string &name, ulong &id, double &value, void &data[])

Функция *FrameNext* читает один фрейм и перемещает указатель на следующий. В параметр *pass* будет записан номер прохода оптимизации. Параметры *name*, *id* и *value* получат значения, переданные в соответствующих параметрах функции *FrameAdd*.

Важно отметить, что функция может вернуть *false* вполне штатно, когда больше нет фреймов для чтения. В этом случае во встроенной переменной *_LastError* содержится значение 4000 (у него нет встроенного обозначения).

Вне зависимости от того, какая форма функции *FrameAdd* была использована для отправки данных, содержимое файла или массива будет помещено в приемный массив *data*. Тип приемного массива должен совпадать с типом отправляемого массива, а для случая отправки файла существуют нюансы.

Бинарный файл (FILE_BIN) желательно принимать в байтовый массив *uchar*, чтобы быть совместимым с любым размером (поскольку другие типы большего размера могут оказаться некратными размеру файла). Если размер файла (а фактически размер блока с данными в принятом фрейме) не будет кратен размеру типа приемного массива, функция *FrameNext* не прочитает данные и вернет ошибку INVALID_ARRAY (4006).

Текстовый файл *Unicode* (FILE_TXT или FILE_CSV без модификатора FILE_ANSI) необходимо принимать в массив типа *ushort* и затем конвертировать в строку с помощью вызова

ShortArrayToString. Текстовый файл ANSI следует принимать в массив типа *uchar* и конвертировать с помощью *CharArrayToString*.

bool FrameInputs(ulong pass, string ¶meters[], uint &count)

Функция *FrameInputs* позволяет получить описания и значения *input*-параметров эксперта, на которых сформирован проход с указанным номером pass. Строковый массив *parameters* будет заполнен строками вида "ИмяПараметраN=значениеПараметраN". В параметр *count* записывается количество элементов в массиве *parameters*.

Вызовы всех 4-х функций разрешено делать только внутри обработчиков *OnTesterPass и OnTesterDeinit*.

Фреймы могут приходить в терминал пачками и для их доставки требуется время, поэтому не обязательно, что все из них успеют сгенерировать событие *OnTesterPass* и будут обработаны до окончания оптимизации. В связи с этим для гарантированного получения всех запоздавших фреймов необходимо поместить блок кода с их обработкой (с использованием функции *FrameNext*) в *OnTesterDeinit*.

Рассмотрим простой пример *FrameTransfer.mq5*.

В эксперте имеется 4 тестовых параметра. Все они, кроме последнего строкового, могут быть включены в оптимизацию.

```
input bool Parameter0;
input long Parameter1;
input double Parameter2;
input string Parameter3;
```

Однако для упрощения примера количество шагов для параметров *Parameter1* и *Parameter2* ограничено 10-ю (для каждого). Таким образом, если не использовать *Parameter0*, максимальное количество проходов равно 121. *Parameter3* служит примером параметра, который нельзя включить в оптимизацию.

Эксперт не торгует, а генерирует случайные данные, которые имитируют прикладные данные произвольного назначения. Не используйте в своих рабочих проектах такую рандомизацию, как здесь: она подходит только для демонстрации.

```
ulong startup; // засекаем время одного прогона (просто как демо-данные)

int OnInit()
{
   startup = GetMicrosecondCount();
   MathSrand((int)startup);
   return INIT_SUCCEEDED;
}
```

Данные отправляются фреймами двух типов: из файла и из массива. Для каждого типа выделен свой идентификатор.

```
#define MY_FILE_ID 100
#define MY_TIME_ID 101

double OnTester()
{
   // посылаем файл в одном фрейме
   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);

   // посылаем массив в другом фрейме
   ulong dummy[1];
   dummy[0] = GetMicrosecondCount() - startup;
   FrameAdd("timing", MY_TIME_ID, 0, dummy);

   return (Parameter2 + 1) * (Parameter1 + 2);
}
```

Файл записывается как бинарный с простыми строками. Результатом (критерием) *OnTester* является простое арифметическое выражение с участием *Parameter1* и *Parameter2*.

На принимающей стороне, в экземпляре эксперта, выполняющемся в сервисном режиме на графике терминала, мы собираем данные всех фреймов с файлами и складываем их в общий CSV-файл. Файл открывается в обработчике *OnTesterInit*.

```
int handle; // файл для сбора прикладных результатов
void OnTesterInit()
{
   handle = FileOpen("output.csv", FILE_WRITE | FILE_CSV | FILE_ANSI, ",");
}
```

Как было сказано ранее, все фреймы могут не успеть попасть в обработчик *OnTesterPass*, и их нужно дополнительно проверить в *OnTesterDeinit*. Поэтому мы реализовали одну вспомогательную функцию *ProcessFileFrames*, которую будем вызывать и из *OnTesterPass*, и из *OnTesterDeinit*.

Внутри *ProcessFileFrames* мы ведем свой внутренний счетчик обработанных фреймов *framecount*. На его примере мы убедимся, что порядок прихода фреймов и нумерация тестовых проходов часто не совпадают.

```
void ProcessFileFrames()
{
   static ulong framecount = 0;
   ...
```

Для приема фреймов в функции описаны переменные, необходимые согласно прототипу *FrameNext*. Приемный массив данных здесь описан типа *uchar*. Если бы мы записывали в свой двоичный файл некие структуры, то могли бы принимать их непосредственно в массив структур того же типа.

```
   ulong    pass;
   string   name;
   long     id;
   double   value;
   uchar    data[];
   ...
```

Далее описаны переменные для получения входных переменных эксперта для текущего прохода, которому принадлежит фрейм.

```
   string   params[];
   uint     count;
   ...
```

Затем мы в цикле читаем фреймы с помощью *FrameNext*. Напомним, что в обработчик может поступить сразу несколько фреймов, поэтому нужен цикл. Для каждого фрейма мы выводим в журнал (терминала) номер прохода, название фрейма и полученное значение *double*. Фреймы с идентификатором, отличным от MY_FILE_ID мы пропускаем, и будем их обрабатывать потом.

```
   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));
   }
}
```

Для фреймов с MY_FILE_ID мы выполняем следующие действия: запрашиваем входные переменные, узнаем, какие из них включены в оптимизацию, и сохраняем их значения в общий CSV-файл вместе с информацией из фрейма. Когда счетчик фреймов равен 0, мы формируем заголовок CSV-файла в переменной *header*. Во всех фреймах текущая (новая) запись для CSV-файла формируется в переменной *record*.

```
void ProcessFileFrames()
{
   ...
      if(FrameInputs(pass, params, count))
      {
         string header, record;
         if(framecount == 0) // готовим CSV заголовок
         {
            header = "Counter,Pass ID,";
         }
         record = (string)framecount + "," + (string)pass + ",";
         // собираем оптимизируемые параметры и их значения
         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) // готовим CSV заголовок
                     {
                        header += name2value[0] + ",";
                     }
                     record += name2value[1] + ","; // поле данных
                  }
               }
            }
         }
         if(framecount == 0) // готовим CSV заголовок
         {
            FileWriteString(handle, header + "Value,File Content\n");
         }
         // записываем данные в CSV
         FileWriteString(handle, record + DoubleToString(value) + ","
            + CharArrayToString(data) + "\n");
      }
      framecount++;
   ...
}
```

Вызов *ParameterGetRange* можно было также сделать более эффективно — только при нулевом значении счетчика *framecount*. Это оставлено как самостоятельное упражнение.

В обработчике *OnTesterPass* просто вызываем *ProcessFileFrames*.

```
void OnTesterPass()
{
   ProcessFileFrames(); // стандартная обработка фреймов на лету
}
```

Дополнительно вызываем ту же функцию из *OnTesterDeinit* и закрываем CSV-файл.

```
void OnTesterDeinit()
{
   ProcessFileFrames(); // подбираем припозднившиеся фреймы
   FileClose(handle);   // закрываем CSV-файл
   ..
}
```

Кроме того в *OnTesterDeinit* делаем обработку фреймов с MY_TIME_ID. В данных фреймах к нам приходят длительности тестовых проходов, и здесь рассчитывается средняя длительность одного прохода. В принципе, это имеет смысл делать только для анализа в своей программе, так как для пользователя длительности проходов и так выводятся тестером в журнал.

```
void OnTesterDeinit()
{
   ...
   ulong   pass;
   string  name;
   long    id;
   double  value;
   ulong   data[]; // тот же тип массива, что и при отправке

   FrameFilter("timing", MY_TIME_ID); // перемотка на первый фрейм

   ulong count = 0;
   ulong total = 0;
   // цикл только по фреймам 'timing'
   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);
   }
}
```

Эксперт готов. Включим для него оптимизацию полным перебором (потому что общее количество вариантов искусственно ограничено и мало для генетики), можно в режиме только по ценам

открытия, так как эксперт не торгует. Из-за этого, кстати говоря, следует выбрать пользовательский критерий (все остальные критерии дадут 0). Например, установим диапазон изменения *Parameter1* от 1 до 10 с единичным шагом, а *Parameter2* от -0.5 до +0.5 с шагом 0.1.

Запустим оптимизацию. В журнале экспертов в терминале увидим записи о получаемых фреймах вида:

```
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
```

В файле output.csv появятся соответствующие строки с номерами проходов, значениями параметров и содержимым фреймов:

```
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
```

Очевидно, что наша внутренняя нумерация (колонка *Count*) идет по порядку, а номера проходов *Pass ID* могут быть перемешаны (это зависит от многих факторов параллельной обработки пакетов заданий агентами). В частности, пакет заданий может первым закончить тот агент, которому были присвоены задания с бОльшими порядковыми номерами: в таком случае нумерация в файле начнется со старших проходов.

В журнале тестера можно проверить служебную статистику по фреймам.

```
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'
```

Важно отметить, что при генетической оптимизации номера проходов представляются в отчете оптимизации как пара *(номер поколения, номер экземпляра)*, в то время как номер прохода, получаемый в функции *FrameNext* — по-прежнему число *ulong* — фактически номер прохода в пакетных заданиях в контексте текущего запуска оптимизации. MQL5 не предоставляет средств для сопоставления нумерации проходов с "генетическим" отчетом. Для этой цели следует рассчитывать контрольные суммы входных параметров каждого прохода. Opt-файлы с кэшем оптимизации уже содержат такое поле с MD5-хэшем.

6.5.12 Директивы препроцессора для тестера

В разделе об Общих свойствах программ мы впервые познакомились с директивами *#property* в MQL-программах. Затем нам встречались директивы, предназначенные для скриптов, сервисов и индикаторов. Есть своя группа директив и для тестера. Некоторые из них мы уже упоминали, в частности, *tester_everytick_calculate* влияет на расчет индикаторов.

В следующей таблице представлены все директивы тестера. Ниже приведены пояснения.

Директива	Описание
tester_indicator "строка"	Имя пользовательского индикатора в формате "имя_индикатора.ex5"
tester_file "строка"	Имя файла в формате "имя_файла.расширение" с исходными данными, необходимыми для теста программы
tester_library "строка"	Имя библиотеки с расширением, например, "библиотека.ex5" или "библиотека.dll"
tester_set "строка"	Имя файла в формате "имя_файла.set" с настройками значений и диапазонов оптимизации входных параметров программы
tester_no_cache	Отключение чтения имеющегося кэша предыдущих оптимизаций (opt-файлов)
tester_everytick_calculate	Отключение экономного режима расчета индикаторов в тестере

Две последних директивы не имеют аргументов. Все остальные ожидают указания строки в двойных кавычках с названием файла того или иного типа. Из этого также следует, что директивы могут повторяться с разными файлами, то есть можно подключить несколько файлов настроек или несколько индикаторов.

Директива *tester_indicator* требуется для подключения к процессу тестирования тех индикаторов, которые не упомянуты в исходном коде тестируемой программы в виде константных строк (литералов). Как правило, необходимый индикатор может быть определен компилятором автоматически из вызова функций *iCustom*, если его имя в явном виде задано в соответствующем параметре, например, *iCustom(symbol, period, "indicator_name",...)*. Однако так бывает не всегда.

Допустим, мы пишем универсальный эксперт, который способен использовать разные индикаторы скользящего среднего, а не только стандартные встроенные. Тогда мы можем завести входную переменную для указания имени индикатора пользователем. И вызов *iCustom* превратится в *iCustom(symbol, period, CustomIndicatorName,...)*, где *CustomIndicatorName* — входная переменная эксперта, содержимое которой не известно в момент компиляции. Более того, разработчик в таком случае, скорее всего, применит *IndicatorCreate* вместо *iCustom*, так как количество и типы параметров индикатора должны также настраиваться. В подобных случаях, для отладки программы или её демонстрации с конкретным индикатором, тестеру нужно сообщить его имя с помощью директивы *tester_indicator*.

> Необходимость сообщать имена индикаторов в исходном коде существенно ограничивает возможности тестирования подобных универсальных программ, способных подключать различные индикаторы онлайн.

Без директивы *tester_indicator* терминал не сможет отправить на агент индикатор, который явно не задекларирован в исходном коде, в результате чего зависимая программа утратит часть или весь функционал.

Директива *tester_file* позволяет указать файл, который будет передан на агенты и помещен в песочницу перед началом тестирования. Содержимое и тип файла не регламентирован. Например, это могут быть веса предварительно обученной нейронной сети, данные "стакана", собранные загодя онлайн (т.к. они не воспроизводятся самим тестером) и так далее.

> Файл из директивы *tester_file* считывается только в том случае, если он существовал на момент компиляции. Если исходный код скомпилирован, когда не было соответствующего файла, то его появление в дальнейшем уже не поможет: откомпилированная программа отправится на агент без вспомогательного файла. Поэтому, например, если файл, указанный в *tester_file*, генерируется в *OnTesterInit*, следует убедиться, что файл с заданным именем уже был в момент компиляция, хотя бы и пустой. Мы продемонстрируем это ниже.

Обратите внимание, что компилятор не выдает предупреждений, если файла, указанного в директиве *tester_file*, не существует.

Подключаемые файлы должны находиться в "песочнице" терминала *MQL5/Files/*.

Директива *tester_library* сообщает тестеру о необходимости передать на агенты библиотеку — вспомогательную программу, способную работать только в контексте другой MQL-программы. О библиотеках мы подробно поговорим в отдельном разделе.

Необходимые для тестирования библиотеки определяются автоматически по директивам *#import* в исходном коде. Однако, если какая-либо библиотека используется внешним индикатором, то необходимо включить данное свойство. Библиотека может быть как с расширением *dll*, так и с расширением *ex5*.

Директива *tester_set* оперирует *set*-файлами с настройками MQL-программы. Указанный в директиве файл станет доступен из контекстного меню тестера и позволит пользователю быстро применить настройки.

Если имя указано без пути, *set*-файл должен лежать в том же каталоге, где эксперт. Это несколько неожиданно, т.к. по умолчанию каталог *set*-файлов другой — *Presets*, и именно туда они сохраняются по командам из интерфейса терминала. Чтобы подключить *set*-файл из данного каталога, нужно явно указать его в директиве и предварить косой чертой, которая обозначает абсолютный путь внутри папки MQL5.

```
#property tester_set "/Presets/xyz.set"
```

Когда ведущей косой черты нет, путь считается относительно места размещения исходного текста.

> Сразу после добавления файла и перекомпиляции программы нужно перевыбрать эксперт в тестере, потому что иначе файл не подхватится!

Если в названии *set*-файла указать имя эксперта и номер версии как "*<expert_name>_<number>.set*", то он автоматически добавится в меню загрузки версий параметров под номером версии *<number>*. Например, имя "*MACD Sample_4.set*" означает, что это *set*-файл для эксперта "*MACD Sample.mq5*" с номером версии равным 4.

Желающие могут изучить формат *set*-файлов: для этого достаточно вручную сохранить настройки тестирования/оптимизации в тестере стратегий и затем открыть созданный таким образом файл в текстовом редакторе.

Теперь обратимся к директиве *tester_no_cache*. Тестер стратегий при выполнении оптимизации сохраняет все результаты выполненных проходов в кэш оптимизации (файлы с расширением *opt*), в котором для каждого набора входных параметров сохраняется результат тестирования. Это позволяет при повторной оптимизации на тех же параметрах брать готовые результаты без повторного вычисления и затрат времени.

Но для некоторых задач — например, при математических вычислениях — может потребоваться проводить расчеты независимо от наличия готовых результатов в кэше оптимизации. В этом случае в исходном коде необходимо включить свойство *tester_no_cache*. При этом сами результаты тестирования все равно будут сохраняться в кэше, чтобы можно было в тестере стратегий посмотреть все данные по выполненным проходам.

Директива *tester_everytick_calculate* предназначена для включения режима расчета индикатора на каждом тике в тестере.

По умолчанию индикаторы рассчитываются в тестере только при обращении к ним за данными — то есть, когда запрашиваются значения индикаторных буферов. Это даёт существенное ускорение при тестировании и оптимизации, если не требуется получать значения индикатора на каждом тике.

Однако некоторые программы могут требовать пересчета индикаторов на каждом тике. Именно в таких случаях и пригодится свойство *tester_everytick_calculate*.

Индикаторы в тестере стратегий также принудительно считаются на каждом тике в следующих случаях:

- при тестировании в визуальном режиме;
- при наличии в индикаторе функций *EventChartCustom*, *OnChartEvent*, *OnTimer*;

Данное свойство касается только работы в тестере стратегий — в терминале индикаторы всегда считаются на каждом поступившем тике.

В эксперте *FrameTransfer.mq5* уже была на самом деле использована директива:

```
#property tester_set "FrameTransfer.set"
```

Просто мы не акцентировали на этом внимание. Файл "*FrameTransfer.set*" находится рядом с исходным кодом. В том же эксперте нам пригодилась и другая директива из вышеприведенной таблицы:

```
#property tester_no_cache
```

В дополнение рассмотрим пример директивы *tester_file*. Ранее в разделе про автонастройку параметров экспертов при оптимизации мы представили эксперт *BandOsMApro.mq5*, в котором потребовалось ввести несколько теневых параметров для передачи диапазонов оптимизации в наш исходный код, выполняющийся на агентах.

Директива *tester_file* позволит нам избавиться от этих лишних параметров. Новую версию эксперта назовем *BandOsMAprofile.mq5*.

Поскольку мы теперь знакомы с директивой *tester_set*, добавим в новую версию уже упоминавшийся ранее файл */Presets/MQL5Book/BandOsMA.set*.

```
#property tester_set "/Presets/MQL5Book/BandOsMA.set"
```

Информацию о диапазоне и шаге изменения периодов *FastOsMA* и *SlowOsMA* будем сохранять в файл "*BandOsMAprofile.csv*" вместо трех дополнительных входных параметров *FastShadow4Optimization*, *SlowShadow4Optimization*, *StepsShadow4Optimization*.

```
#define SETTINGS_FILE "BandOsMAprofile.csv"
#property tester_file SETTINGS_FILE

const string SettingsFile = SETTINGS_FILE;
```

Теневой параметр *FastSlowCombo4Optimization* по-прежнему нужен для полного перебора разрешенных комбинаций периодов.

```
input group "A U X I L I A R Y"
sinput int FastSlowCombo4Optimization = 0;   // (reserved for optimization)
```

Напомним, его диапазон для оптимизации мы находим в функции *Iterate*. Первый раз мы её вызываем в *OnTesterInit* с полным перебором сочетаний быстрого и медленного периодов.

В принципе, мы могли бы сохранить все допустимые сочетания в массив структур *PairOfPeriods* и записать его в двоичный файл для передачи на агенты. Тогда на агентах наш эксперт мог бы прочитать из файла готовый массив и по индексу *FastSlowCombo4Optimization* извлечь из массива соответствующую пару *FastOsMA* и *SlowOsMA*.

Вместо этого мы остановимся на минимальном изменении рабочей логики программы: будем по прежнему восстанавливать пару периодов за счет второго вызова *Iterate* в обработчике *OnInit*. Только на этот раз диапазон и шаг перебора значений периодов мы получим не из теневых параметров, а из CSV-файла.

Вот изменения в *OnTesterInit*.

```
int OnTesterInit()
{
   ...
         // проверим есть ли файл уже до компиляции
         // - если нет, тестер не сможет отослать его на агенты
         const bool preExisted = FileIsExist(SettingsFile);

         // запишем настройки в файл для передачи программам-копиям на агенты
         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;
}
```

Обратите внимание, мы сделали обработчик *OnTesterInit* с типом возврата *int*, что дает возможность отменить оптимизацию, если файл не существует. Однако в файл в любом случае записываются актуальные данные, так что если его не было, он создается, и последующий запуск оптимизации уже пройдет успешно.

Если вы хотите исключить этот шаг, можете заранее создать пустой файл *MQL5/Files/BandOsMAprofile.csv*.

Обработчик *OnInit* преобразился следующим образом.

```
int OnInit()
{
   if(FastOsMA >= SlowOsMA) return INIT_PARAMETERS_INCORRECT;

   PairOfPeriods p = {FastOsMA, SlowOsMA}; // исходные параметры по умолчанию
   int handle = FileOpen(SettingsFile, FILE_READ | FILE_TXT | FILE_ANSI);

   // во время оптимизации нужен файл с теневыми параметрами
   if(MQLInfoInteger(MQL_OPTIMIZATION) && handle == INVALID_HANDLE)
   {
      return INIT_PARAMETERS_INCORRECT;
   }

   if(handle != INVALID_HANDLE)
   {
      if(FastSlowCombo4Optimization != -1)
      {
         // если теневая копия есть, считываем значения периодов из неё
         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);
   }
```

При запуске одиночных тестов после оптимизации мы увидим в журнале раскодированные значения периодов *FastOsMA* и *SlowOsMA* на основе оптимизированного значения *FastSlowCombo4Optimization*. В дальнейшем мы можем подставить эти значение в параметры-периоды, а csv-файл удалить. Также мы предусмотрели, что файл не будет учитываться, если в *FastSlowCombo4Optimization* поставить значение -1.

6.5.13 Управление видимостью индикаторов: TesterHideIndicators

По умолчанию на графике визуального тестирования показываются все индикаторы, которые создаются в тестируемом эксперте. Также эти индикаторы показываются на графике, который

автоматически открывается по окончании тестирования. Это все касается только тех индикаторов, которые непосредственно создаются в вашем коде: вложенные индикаторы, которые могут использоваться в расчетах основных индикаторов, сюда не относятся.

Видимость индикаторов не всегда желательна с точки зрения разработчика, который может хотеть скрыть детали реализации эксперта. В подобных случаях функция *TesterHideIndicators* позволит запретить показ используемых индикаторов на графике.

void TesterHideIndicators(bool hide)

Логический параметр *hide* предписывает либо скрывать (по значению *true*), либо отображать (по значению *false*) индикаторы. Установленное состояние запоминается средой исполнения MQL-программы до тех пор, пока не будет изменено повторным вызовом функции с обратным значением параметра. Текущее состояние данной настройки влияет на все вновь создаваемые индикаторы.

Иными словами функцию *TesterHideIndicators* с необходимым значением флага *hide* следует вызывать перед созданием дескрипторов соответствующих индикаторов. В частности, после вызова функции с параметром *true* новые индикаторы будут помечены флагом скрытия и не будут показаны при визуальном тестировании и на графике, который автоматически открывается при завершении тестирования.

Для отключения режим скрытия вновь создаваемых индикаторов нужно вызвать *TesterHideIndicators* с параметром *false*.

Функция применима только в тестере.

Функция имеет особенности работы при условии, что для тестера или эксперта созданы специальные tpl-шаблоны в папке */MQL5/Profiles/Templates*.

Если в папке присутствует специальный шаблон *<имя_эксперта>.tpl*, то при визуальном тестировании и на графике тестирования будут показаны только индикаторы из данного шаблона. В этом случае никакие индикаторы, используемые в тестируемом эксперте, показаны не будут, даже если в коде советника вызывалась функция *TesterHideIndicators* с параметром *false*.

Если же в папке есть шаблон *tester.tpl*, то при визуальном тестировании и на графике тестирования будут показаны индикаторы из шаблона *tester.tpl* и те индикаторы из советника, которые не запрещены вызовом *TesterHideIndicators*. Функция *TesterHideIndicators* не влияет на индикаторы в шаблоне.

Если шаблона *tester.tpl* нет, но есть шаблон *default.tpl*, то индикаторы из него обрабатываются по аналогичному принципу.

Мы продемонстрируем работу функции в большом примере эксперта далее.

6.5.14 Эмуляция пополнения депозита и снятия средств

Тестер MetaTrader 5 позволяет эмулировать операции пополнения счета и снятия с него средств. Это позволяет проводить эксперименты с некоторыми системами управления капиталом.

bool TesterDeposit(double money)

Функция *TesterDeposit* пополняет счет в процессе тестирования на размер вносимой суммы в параметре money. Сумма указывается в валюте тестового депозита.

bool TesterWithdrawal(double money)

Функция *TesterWithdrawal* производит снятие средств в размере *money*.

Обе функции возвращают *true* как признака успеха.

В качестве примера рассмотрим эксперт на основе стратегии "carry trade". Для неё нам потребуется выбрать символ с большими положительными свопами в одном из торговых направлений, например, покупка AUDUSD. Эксперт будет открывать одну или более позиций в указанном направлении. Убыточные позиции будут удерживаться ради накопления по ним свопов. Прибыльные позиции будут закрываться по достижении предопределенного размера прибыли в расчете на лот. Заработанные свопы будут сниматься со счета.

Исходный код доступен в файле *CrazyCarryTrade.mq5*. Название выбрано не случайно, так как пересиживание убытков является рискованной практикой.

Во входных параметрах пользователь может выбрать направление торговли, размер одной сделки (по умолчанию 0, что означает минимальный лот) и минимальную прибыль на лот, при которой прибыльная позиция закроется.

```
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;
```

Для начала протестируем в обработчике *OnInit* работу функций *TesterWithdrawal* и *TesterDeposit*. В частности, попытка снять двойной баланс приведет к ошибке 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
   */
   ...
```

Зато последующие снятие и зачисление обратно по 100 единиц валюты счета пройдут успешно.

```
      PRTF(TesterWithdrawal(100));
      /*
      deal #2 balance -100.00 [withdrawal] done
      TesterWithdrawal(100)=true / ok
      */
      PRTF(TesterDeposit(100)); // вернем деньги
      /*
      deal #3 balance 100.00 [deposit] done
      TesterDeposit(100)=true / ok
      */
      return INIT_SUCCEEDED;
   }
```

В обработчике *OnTick* проверим наличие позиций с помощью *PositionFilter* и заполним массив *values* значениями их текущих прибылей/убытков и накопленных свопов.

```
   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);
      ...
```

Когда позиций нет, откроем одну в предопределенном направлении.

```
      if(ArraySize(tickets) == 0) // позиций нет
      {
         MqlTradeRequestSync request1;
         (Type == MARKET_BUY ? request1.buy(volume) : request1.sell(volume));
      }
      else
      {
         ... // позиции есть - см. следующую врезку
      }
```

Когда позиции есть, проходимся по ним в цикле и закрываем те, по которым появилась достаточная прибыль (с поправкой на свопы). Попутно суммируем свопы закрытых позиций и общие убытки. Поскольку свопы растут пропорционально времени, мы используем их как усиливающий коэффициент на закрытие "древних" позиций. Таким образом, возможно закрытие и с убытком.

```
      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];
         }
      }
      ...
```

Если общие убытки нарастают, периодически открываем дополнительные позиции, но тем реже, чем больше позиций, чтобы хоть как-то контролировать риски.

```
      if(loss / ArraySize(tickets) <= -MinProfitPerLot * volume * sqrt(ArraySize(tick
      {
         MqlTradeRequestSync request1;
         (Type == MARKET_BUY ? request1.buy(volume) : request1.sell(volume));
      }
      ...
```

Наконец, снимаем со счета свопы.

```
      if(swaps >= 0)
      {
         TesterWithdrawal(swaps);
      }
```

В обработчике *OnDeinit* выведем статистику по отчислениям.

```
void OnDeinit(const int)
{
   PrintFormat("Deposit: %.2f Withdrawals: %.2f",
      TesterStatistics(STAT_INITIAL_DEPOSIT),
      TesterStatistics(STAT_WITHDRAWAL));
}
```

Например, при запуске эксперта с настройками по умолчанию на периоде 2021-начало 2022-го года получим для AUDUSD такой результат:

```
   final balance 10091.19 USD
   Deposit: 10000.00 Withdrawals: 197.42
```

А вот как выглядит отчет и график.

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%)

Отчет эксперта со снятием средств со счета

Таким образом, при торговле минимальным лотом и загрузкой депозита не более 1% за год с небольшим удалось снять примерно 200 USD.

6.5.15 Принудительная остановка тестирования: TesterStop

При необходимости, в зависимости от возникших условий, разработчик может остановить тестирование эксперта досрочно. Например, это можно сделать при достижении заданного количества убыточных сделок или уровня просадки. Для этой цели в API имеется функция *TesterStop*.

void TesterStop()

Функция отдает команду на завершение работы тестера, то есть остановка произойдет только после того, как программа вернет управление среде исполнения.

Вызов *TesterStop* считается нормальным завершением тестирования, и поэтому будет вызвана функция *OnTester* с отдачей тестеру стратегий всей накопленной торговой статистики и значения критерия оптимизации.

Существует и альтернативный штатный способ прерывания тестирования — с помощью уже известной нам функции *ExpertRemove*. После вызова *ExpertRemove* также будет получена собранная к моменту вызова торговая статистика, однако существуют и некоторые отличия.

В результате вызова *ExpertRemove* советник выгружается из памяти агента, поэтому для выполнения прохода на следующем наборе параметров понадобится время на повторную загрузку MQL-программы. При использовании *TesterStop* этого не происходит, и данный способ является более предпочтительным в плане быстродействия.

С другой стороны, вызов *ExpertRemove* взводит в MQL-программе флаг остановки *_IsStopped*, что можно стандартным образом использовать в разных частях программы для финализации ("подчистки" ресурсов). Вызов же *TesterStop* не взводит данный флаг, и потому разработчику может потребоваться ввести свою собственную глобальную переменную, обозначающую досрочную остановку, и обрабатывать её специфическим образом.

> Важно отметить, что *TesterStop* предназначена для остановки только одного прохода тестера.
>
> В MQL5 нет функции для досрочной остановки оптимизации. Поэтому, например, если ваш эксперт обнаружит, что оптимизация запущена на неправильной модели генерации тиков — а это можно обнаружить уже только после запуска оптимизации (обработчик *OnTesterInit* здесь не поможет) — то вызовы *TesterStop* или *ExpertRemove* будут прерывать новые проходы, но сами проходы будут порождаться и порождаться, генерируя массовые нулевые результаты. Мы это увидим в большом примере эксперта, в котором будет использована защита от запуска по ценам открытия.
>
> Можно было бы предположить, что вызов *ExpertRemove* в той копии эксперта, которая выполняется в терминале и фактически является менеджером оптимизации, приведет к остановке оптимизации. Но это не так. И даже закрытие графика с этим экспертом, работающим в режиме фреймов, не останавливает оптимизацию.

Предлагается самостоятельно попробовать данные функции в действии.

6.5.16 Большой пример эксперта

Для обобщения и закрепления знаний о возможностях тестера рассмотрим поэтапно большой пример эксперта, в котором сведем воедино:

- использование нескольких символов, включая синхронизацию баров;
- использование индикатора из эксперта;
- использование событий;
- самостоятельный подсчет основных статистических показателей торговли;
- расчет пользовательского критерия оптимизации R2 с поправкой на переменные лоты;
- отправку и обработку фреймов с прикладными данными (торговыми отчетами в разбивке по символам).

За техническую основу для эксперта возьмем *MultiMartingale.mq5*, но сделаем его менее рисковым за счет переключения на торговлю по мультивалютным сигналам перекупленности/перепроданности и увеличения лотов только в качестве опционального дополнения. Схема работы по торговым сигналам индикатора у нас уже обкатана на примере *BandOsMA.mq5*, но теперь в качестве сигнального индикатора выступит *UseUnityPercentPro.mq5*. Правда, сначала нам потребуется слегка его модифицировать. Новую версию назовем *UnityPercentEvent.mq5*, и суффикс "Event" здесь неспроста.

UnityPercentEvent.mq5

Напомним суть работы индикатора *Unity*. Он рассчитывает относительную силу валют или вообще тикеров, входящих в набор заданных инструментов (предполагается, что у всех инструментов есть общая валюта, через которую возможен пересчет). На каждом баре формируются отсчеты для всех валют: что-то окажется дороже, что-то дешевле, и два крайних элемента оказываются в пограничном состоянии. Далее для них можно рассматривать две противоположных по своей сути стратегии:

- дальнейший пробой (подтверждение и продолжение сильного движения в стороны);
- отскок (разворот движения к центру из-за перекупленности и перепроданности).

Для торговли любого из этих сигналов мы должны составить рабочий инструмент из двух валют (или вообще тикеров), если для данного сочетания есть что-то подходящее в Обзоре рынка. Например, если верхняя линия индикатора принадлежит EUR, а нижняя — USD, им соответствует пара EURUSD, и по стратегии на пробой мы должны её купить, а по стратегии на отскок — продать.

> В более общем случае, например, когда в корзине рабочих инструментов индикатора указаны CFD или товары с общей валютой котирования, не всегда удастся составить реальный инструмент. Для подобных случаев потребовалось бы усложнить эксперт, введя торговлю синтетиками (составными позициями), но мы не станем здесь этого делать и ограничимся рынком Forex, где обычно в наличии практически все кросс-курсы.

Таким образом, эксперт должен не только прочитать все буфера индикатора, но и выяснить названия валют, которым соответствует максимальное и минимальное значения. И здесь нас ожидает маленькое препятствие.

Дело в том, что MQL5 не позволяет прочитать названия буферов стороннего индикатора, да и вообще любые свойства линий кроме целочисленных: для установки свойств имеется полная тройка функций — *PlotIndexSetInteger*, *PlotIndexSetDouble*, *PlotIndexSetString*, а вот для чтения — только одна *PlotIndexGetInteger*. Таково ограничение платформы, по крайней мере, сейчас.

В принципе, когда MQL-программы, составленные в единый торговый комплекс, имеют одного разработчика, это не большая проблема. В частности, мы могли бы выделить часть исходного кода индикатора в заголовочный файл и подключить его не только в индикатор, но и в эксперт. Тогда в эксперте можно было бы повторить разбор входных параметров индикатора и восстановить список валют, полностью аналогичный тому, что создает индикатор. Дублирование вычислений — не очень красиво, но оно сработало бы. Однако требуется и более универсальное решение, когда у индикатора другой разработчик, и он не желает раскрывать алгоритм или предполагает его смену в будущем (тогда откомпилированные версии индикатора и эксперта станут несовместимы). Подобная "стыковка" чужих индикаторов со своим или заказываемым в сервисе фриланс экспертом — весьма распространенная практика. И потому разработчику индикатора желательно сделать его максимально "дружелюбным" для интеграции с другими программами.

Одно из возможных решений — рассылка индикатором сообщений с номерами и названиями буферов после инициализации.

Вот как это сделано в обработчике *OnInit* индикатора *UnityPercentEvent.mq5* (приведено с сокращениями, так как почти ничего не изменилось).

```
int OnInit()
{
   // находим общую валюту у всех пар
   const string common = InitSymbols();
   ...
   // настраиваем отображаемые линии в цикле по валютам
   int replaceIndex = -1;
   for(int i = 0; i <= SymbolCount; i++)
   {
      string name;
      // порядок меняем так, чтобы базовая (общая) валюта шла под индексом 0,
      // остальные зависят от порядка ввода пар пользователем
      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);
         }
      }

      // настраиваем отрисовку буферов
      PlotIndexSetString(i, PLOT_LABEL, name);
      ...
      // рассылаем индексы и названия буферов для программ, где они нужны
      EventChartCustom(0, (ushort)BarLimit, i, SymbolCount + 1, name);
   }
   ...
}
```

По сравнению с исходной версией здесь добавлена всего одна строка с вызовом *EventChartCustom*. В качестве идентификатора копии индикатора (которых потенциально может быть несколько) используется входная переменная *BarLimit*. Поскольку индикатор будет вызываться из эксперта и не отобразится пользователю, в ней достаточно указать малое положительное число, как минимум 1, но, у нас будет, например, 10.

Теперь индикатор готов для того, чтобы использовать его сигналы в сторонних экспертах. Начнем разработку эксперта *UnityMartingale.mq5*. Чтобы упростить изложение, разобьем его на 4 этапа, постепенно добавляя новые блоки. У нас получится три предварительных версии и одна окончательная.

UnityMartingaleDraft1.mq5

На первом этапе, для версии *UnityMartingaleDraft1.mq5*, возьмем за основу *MultiMartingale.mq5* и начнем его модифицировать.

Бывшую входную переменную *StartType*, которая определяла направление первой сделки в серии, переименуем в *SignalType* и будем использовать для выбора между рассмотренными стратегиями BREAKOUT и PULLBACK.

```
enum SIGNAL_TYPE
{
   BREAKOUT,
   PULLBACK
};
...
input SIGNAL_TYPE StartType = 0; // SignalType
```

Для настройки индикатора потребуется отдельная группа входных переменных.

```
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;
```

Обратите внимание, что параметр *UnitySymbols* содержит перечень инструментов кластера для построения индикатора, и как правило отличается от перечня рабочих инструментов, которыми мы хотим торговать. Торгуемые инструменты по-прежнему задаются в параметре *WorkSymbols*.

Например, по умолчанию мы передаем в индикатор набор основных валютных пар *Forex*, и потому можем указывать в качестве торговых не только основные пары, но и любые кроссы. Обычно имеет смысл ограничить данный набор инструментами с лучшими торговыми условиями (в частности, малым или умеренным спредом). Кроме того, желательно не допускать перекосов, то есть соблюдать равное количество каждой валюты во всех парах, тем самым статистически нивелировать потенциальные риски выбора неудачного направления по одной из валют.

Управление индикатором обернем в класс *UnityController*. Помимо дескриптора самого индикатора (*handle*) в полях класса хранятся:

- количество буферов индикатора *buffers*, которое будет получено из сообщений от индикатора после его инициализации;
- номер бара *bar*, с которого считываются данные (обычно текущий незавершенный — 0 или последний завершенный — 1);
- массив *data* со значениями, прочитанными из буферов индикатора на указанном баре;
- время последнего чтения *lastRead*;
- признак работы по тикам или барам *tickwise*.

Кроме того, в классе используется объект *MultiSymbolMonitor* для синхронизации баров всех задействованных символов.

```
class UnityController
{
   int handle;
   int buffers;
   const int bar;
   double data[];
   datetime lastRead;
   const bool tickwise;
   MultiSymbolMonitor sync;
   ...
```

В конструкторе, принимающем через аргументы все параметры для индикатора, создаем сам индикатор и настраиваем объект *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);
   }
   ...
```

Количество буферов устанавливается методом *attached*. Мы его вызовем по получению сообщения от индикатора.

```
   void attached(const int b)
   {
      buffers = b;
      ArrayResize(data, buffers);
   }
```

Специальный метод *isReady* возвращает *true*, когда последние бары всех символов имеют одинаковое время. Только в состоянии такой синхронизации мы получим корректные значения индикатора. Следует обратить внимание, что здесь предполагается одинаковое расписание торговых сессий всех инструментов. Если это не так, анализ синхронизации нужно будет поменять.

```
bool isReady()
{
   return sync.check(true) == 0;
}
```

Текущее время мы определяем по-разному в зависимости от режима работы индикатора: при пересчете на каждом тике (*tickwise* равно *true*) берем серверное время, а при пересчете один раз за бар — время открытия последнего бара.

```
datetime lastTime() const
{
   return tickwise ? TimeTradeServer() : iTime(_Symbol, _Period, 0);
}
```

Наличие данного метода позволит исключить чтение индикатора, если текущее время не изменилось и, соответственно, последние прочитанные данные, хранящиеся в буфере *data*, еще актуальны. А вот, собственно, как организовано чтение индикаторных буферов в методе *read*. Нам достаточно одного значения каждого буфера для бара с индексом *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;
}
```

В конце мы как раз сохраняем время чтения в переменную *lastRead*. Если она пуста или не равна новому текущему времени, обращение за данными контроллера в следующих методах вызовет чтение буферов индикатора с помощью *read*.

Основными внешними методами контроллера являются *getOuterIndices* для получения индексов максимального и минимального значений, а также оператор '[]' для чтения самих значений.

```
   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];
   }
};
```

Напомним, что в эксперте *BandOsMA.mq5* мы ввели концепцию интерфейса *TradingSignal*.

```
interface TradingSignal
{
   virtual int signal(void);
};
```

На его основе опишем реализацию сигнала с использованием индикатора *UnityPercentEvent*. Объект контроллера *UnityController* передается в конструктор. Здесь же указываются индексы валют (буферов), сигналы по которым нас интересуют. Мы сможем создать произвольный набор разных сигналов для выбранных рабочих инструментов.

```
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; // ждем синхронизации баров
      if(!controller.isNewTime()) return 0; // ждем изменения времени

      int min, max;
      if(!controller.getOuterIndices(min, max)) return 0;

      // перекупленность
      if(currency1 == max && currency2 == min) return +1;
      // перепроданность
      if(currency2 == max && currency1 == min) return -1;
      return 0;
   }
};
```

Метод *signal* возвращает 0 в неопределенной ситуации, либо +1 или -1 в состояниях перекупленности и перепроданности двух конкретных валют.

Для формализации торговых стратегий мы использовали интерфейс *TradingStrategy*.

```
interface TradingStrategy
{
   virtual bool trade(void);
};
```

В данном случае на его основе создан класс *UnityMartingale*, во многом совпадающий с *SimpleMartingale* из *MultiMartingale.mq5*. Мы покажем лишь отличия.

```
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(); // получаем сигнал контроллера
      if(s != 0)
      {
         if(settings.startType == PULLBACK) s *= -1; // обратная логика для отскока
      }
      ulong ticket = 0;
      if(position[] == NULL) // чистый старт - позиций не было и нет
      {
         if(s == +1)
         {
            ticket = openBuy(settings.lots);
         }
         else if(s == -1)
         {
            ticket = openSell(settings.lots);
         }
      }
      else
      {
         if(position[].refresh()) // позиция существует
         {
            if((position[].get(POSITION_TYPE) == POSITION_TYPE_BUY && s == -1)
            || (position[].get(POSITION_TYPE) == POSITION_TYPE_SELL && s == +1))
            {
               // сигнал в другом направлении - нужно закрыться
               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; - сохраняем позицию в кэше
               }
               else
               {
                  position[].refresh(); // контролируем возможные ошибки закрытия
               }
            }
```

```
            else
            {
               // сигнал прежний или отсутствует - "тралим"
               position[].update();
               if(trailing[]) trailing[].trail();
            }
         }
         else // позиции нет - откроем новую
         {
            if(s == 0) // отсутствие сигналов
            {
               // здесь полностью логика старого эксперта:
               // - переворот для убытка по мартингейлу
               // - продолжение начальным лотом в прибыльном направлении
               ...
            }
            else // сигнал есть
            {
               double lots;
               if(position[].get(POSITION_PROFIT) >= 0.0)
               {
                  lots = settings.lots; // начальный лот после прибыли
               }
               else // увеличиваем лот после убытка
               {
                  lots = MathFloor((position[].get(POSITION_VOLUME) * settings.factor

                  if(lotsLimit < lots)
                  {
                     lots = settings.lots;
                  }
               }

               ticket = (s == +1) ? openBuy(lots) : openSell(lots);
            }
         }
      }
   }
   ...
}
```

Торговая часть готова. Осталось рассмотреть инициализацию. На глобальном уровне описан автоуказатель на объект *UnityController*, а также массив с названиями валют. Пул торговых систем полностью аналогичен прежним наработкам.

```
AutoPtr<TradingStrategyPool> pool;
AutoPtr<UnityController> controller;

int currenciesCount;
string currencies[];
```

В обработчике *OnInit* создаем объект *UnityController* и ждем, когда индикатор пришлет распределение валют по индексам буферов.

```
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);
   // ждем сообщений от индикатора по валютам в буферах
   return INIT_SUCCEEDED;
}
```

Если во входных параметрах индикатора выбран тип цены PRICE_CLOSE и единичный период, расчет в контроллере будет производиться один раз на баре. Во всех остальных случаях сигналы будут обновляться по тикам, но не чаще раза в секунду (вспоминаем реализацию метода *lastTime* в контроллере).

Вспомогательный метод *StartUp* в целом занимается тем же делом, что старый обработчик *OnInit* в эксперте *MultiMartingale* — заполнением структуры *Settings* с настройками, их проверкой на корректность и созданием пула торговых систем *TradingStrategyPool*, состоящего из объектов класса *UnityMartingale* для разных торгуемых символов *WorkSymbols*. Правда, теперь данный процесс разделен на 2 этапа из-за того, что нам нужно дождаться информации о распределении валют по буферам. Поэтому функция *StartUp* имеет входной параметр, обозначающий вызов из *OnInit* и — позднее — из *OnChartEvent*.

При разборе исходного кода *StartUp* важно помнить, что инициализация различается для случаев, когда мы торгуем только одним инструментом, совпадающим с текущим графиком, и когда задана корзина инструментов. Первый режим активен, когда в *WorkSymbols* пустая строка. Он удобен для оптимизации эксперта по конкретному инструменту. Найдя настройки для нескольких инструментов, мы можем объединить их в строке *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)
      {
         ... // создание торговой системы на основе индикатора
      }
   }
   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)
      {
         ... // создание торговой системы на основе индикатора
      }
   }
   return true;
}
```

В *OnInit* функция *StartUp* вызывается с параметром *true*, что означает лишь проверку корректности настроек. Создание объекта торговой системы откладывается до получения сообщения от индикатора в *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)
         {
            StartUp(); // подтверждение готовности индикатора
         }
         else
         {
            Alert("Repeated initialization!");
         }
      }
   }
}
```

Здесь мы запоминаем количество валют в глобальной переменной *currenciesCount* и сохраняем их в массиве *currencies*, после чего вызываем *StartUp* уже с параметром *false* (значение по умолчанию, поэтому опущено). Сообщения поступают из очереди в том порядке, в котором они находятся в буферах индикатора. Таким образом, мы получаем соответствие между индексом и названием валюты.

При повторном вызове *StartUp* выполняется дополнительный код:

```
bool StartUp(const bool init = false)
{
   if(WorkSymbols == "") // один текущий символ
   {
      ...
      if(!init) // окончательная инициализация уже после OnInit
      {
         controller[].attached(currenciesCount);
         // разделяем _Symbol на 2 валюты из массива currencies[]
         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;
         }
         // создаем пул из единственной стратегии
         pool = new TradingStrategyPool(new UnityMartingale(settings,
            new UnitySignal(controller[], first, second)));
      }
   }
   else // корзина символов
   {
      ...
      if(!init) // окончательная инициализация уже после OnInit
      {
         controller[].attached(currenciesCount);

         const int n = ArraySize(settings);
         pool = new TradingStrategyPool(n);
         for(int i = 0; i < n; i++)
         {
            ...
            // разделяем settings[i].symbol на 2 валюты из 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
            {
               // добавляем в пул стратегию на очередном торговом символе
               pool[].push(new UnityMartingale(settings[i],
                  new UnitySignal(controller[], first, second)));
            }
         }
      }
   }
```

Вспомогательная функция *SplitSymbolToCurrencyIndices* выделяет базовую валюту и валюту прибыли переданного символа и находит для них индексы в массиве *currencies*. Таким образом, мы получаем опорные данные для генерации сигналов в объектах *UnitySignal* — в каждом будет своя пара индексов валют.

```
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;
}
```

В целом, эксперт готов.

> Нетрудно заметить, что в последних примерах экспертов у нас фигурируют классы стратегий и классы торговых сигналов. Мы специально сделали их наследниками обобщенных интерфейсов *TradingStrategy* и *TradingSignal*, чтобы впоследствии иметь возможность собирать коллекции совместимых, но различных реализаций, которые можно комбинировать при разработке будущих экспертов. Подобные унифицированные конкретные классы обычно подлежат выделению в отдельные заголовочные файлы. В наших примерах мы не сделали это ради упрощения пошаговой модификации.
>
> Но описанный подход является стандартным для ООП. В частности, как мы уже упоминали в разделе о созданииабсолютно заготовок экспертов, вместе с MetaTrader 5 поставляется "фреймворк" (*framework*) заголовочных файлов со стандартными классами торговых операций, сигнальных индикаторов и управления денежными средствами, которые используются в Мастере MQL. Другие похожие решения публикуются на сайте *mql5.com* в разделе статей и базы кодов.
>
> В своих разработках вы можете взять за основу любую из готовых иерархий классов, которая удовлетворяет вас по своим возможностям и удобству пользования.

Для полноты картины мы хотели внедрить в эксперт собственный критерий оптимизации на базе R2. Чтобы избавиться от противоречия между линейной регрессией в формуле расчета R2 и переменными лотами, которые заложены в нашу стратегию, будем вычислять коэффициент не для обычной линии баланса, а для её кумулятивных приращений, нормированных по размерам лотов в каждой сделке.

Для этого в обработчике *OnTester* сделаем выборку сделок только по типам DEAL_TYPE_BUY и DEAL_TYPE_SELL и с направлением "выход" (OUT). Для сделок запросим все свойства, формирующие финансовый результат (прибыль/убыток), то есть DEAL_PROFIT, DEAL_SWAP, DEAL_COMMISSION, DEAL_FEE, а также их объем DEAL_VOLUME.

```
#define STAT_PROPS 5 // количество запрашиваемых свойств сделки

double OnTester()
{
   HistorySelect(0, LONG_MAX);

   const ENUM_DEAL_PROPERTY_DOUBLE props[STAT_PROPS] =
   {
      DEAL_PROFIT, DEAL_SWAP, DEAL_COMMISSION, DEAL_FEE, DEAL_VOLUME
   };
   double expenses[][STAT_PROPS];
   ulong tickets[]; // нужно из-за прототипа метода 'select', но удобно для отладки

   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_E
      IS::OR_BITWISE)
      .select(props, tickets, expenses);
   ...
```

Далее в массиве *balance* накапливаем прибыли/убытки, нормированные торговыми объемами, и вычисляем для него критерий R2.

```
   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]; // нормируем объемом
      balance[i + 1] = result + balance[i];
   }
   const double r2 = RSquaredTest(balance);
   return r2 * 100;
}
```

На этом первая версия эксперта в принципе готова. Мы оставили "за скобками" проверку модели тиков с помощью *TickModel.mqh*. Предполагается, что эксперт будет тестироваться при генерации тиков в режиме OHLC M1 или лучше. При обнаружении модели "только цены открытия" эксперт пошлет в терминал специальный фрейм со статусом ошибки и выгрузит себя из тестера. К сожалению, это остановит только данный проход, но оптимизация продолжится. Поэтому копия эксперта, которая выполняется в терминале, выдает "алерт" для пользователя, чтобы он прервал оптимизацию вручную.

```
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");
         // было бы логично, если бы следующий вызов останавливал всю оптимизацию,
         // но это не так
         ExpertRemove();
      }
   }
}
```

Вы можете провести оптимизацию параметров SYMBOL SETTINGS для любого символа, и повторить её для разных символов. При этом в группах COMMON SETTINGS и UNITY SETTINGS всегда должны быть одни и те же настройки, потому что они применяются ко всем символам и экземплярам торговых систем. Например, сопровождение (*Trailing*) должно быть либо включено, либо выключено для всех оптимизаций. Также следует помнить, что входные переменные для отдельного символа (т.е. группы SYMBOL SETTINGS) имеют эффект только пока в *WorkSymbols* находится пустая строка. Поэтому на стадии оптимизаций следует держать его пустым.

Например, для диверсификации рисков можно последовательно оптимизировать эксперт на полностью независимых парах: EURUSD, AUDJPY, GBPCHF, NZDCAD или в других сочетаниях. К исходному коду подключено 3 set-файла с примерами частных настроек.

```
#property tester_set "UnityMartingale-eurusd.set"
#property tester_set "UnityMartingale-gbpchf.set"
#property tester_set "UnityMartingale-audjpy.set"
```

Для того чтобы торговать сразу по трем символам, следует "упаковать" эти настройки в общий параметр *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(
```

Такая настройка тоже приложена в отдельном файле.

```
#property tester_set "UnityMartingale-combo.set"
```

Одна из проблем с текущей версией эксперта заключается в том, что отчет тестера сообщит нам общую статистику по всем символам (точнее по всем торговым стратегиям, так как мы можем включить в пул разные классы), в то время как нам было бы интересно контролировать и оценивать каждый компонент системы отдельно.

Чтобы это сделать, необходимо научиться самостоятельно подсчитывать основные финансовые показатели торговли, по аналогии с тем, как это делает для нас тестер. Займемся этим на втором этапе развития эксперта.

UnityMartingaleDraft2.mq5

Подсчет статистики — это часто возникающая задача, поэтому выделим её в отдельный заголовочный файл *TradeReport.mqh*, где организуем исходный код в соответствующие классы.

Основной класс так и назовем — *TradeReport*. Многие показатели торговли зависят от кривых баланса и свободных средств (эквити). Поэтому в классе имеются переменные для отслеживания текущего баланса и прибыли, а также постоянно дополняемый массив с историей баланса. Историю эквити мы хранить не будем, потому что она может меняться на каждом тике, и лучше считать её прямо на ходу. А для чего нам понадобится кривая баланса, мы расскажем чуть позже.

```
class TradeReport
{
   double balance;     // текущий баланс
   double floating;    // текущая плавающая прибыль
   double data[];      // кривая баланса целиком - цены
   datetime moments[]; // и дата/время
   ...
```

Изменение и чтение полей класса производится с помощью методов, включая и конструктор, в котором баланс инициализируется свойством 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;
   }
   ...
```

Эти методы потребуются для итеративного расчета просадки по эквити (на лету). Массив баланса *data* потребуется для одномоментного расчета просадки по балансу (это будем делать уже в конце теста).

На основе колебаний кривой (не важно, баланса или эквити) по одному и тому же алгоритму должна считаться абсолютная и относительная просадка. Поэтому данный алгоритм и

необходимые для него внутренние переменные, хранящие промежуточные состояния, реализованы во вложенной структуре *DrawDown*: мы не станем её приводить полностью, а лишь представим главные методы и свойства.

```
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;
};
```

Первый метод *calcDrawdown* рассчитывает просадки, когда нам известен весь массив и это будет использовано для баланса. Второй метод *calcDrawdown* рассчитывает просадку итеративно: при каждом вызове ему сообщается очередное значение ряда, и это будет использовано для эквити.

Помимо просадки, как мы знаем, существует большое количество стандартных статистических показателей для отчетов, но мы поддержим для начала лишь некоторые из них. Для этого опишем соответствующие поля в еще одной вложенной структуре *GenericStats*. Она унаследована от структуры *DrawDown*, потому что просадка нам все равно потребуется в отчете.

```
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;
   ...
```

По названиям переменных легко догадаться, каким стандартным метрикам они соответствуют. Некоторые показатели излишни и потому опущены. Например, имея общее количество трейдов (*trades*) и количество покупок среди них (*buy_trades*), легко найти количество продаж (*trades - sell_trades*). То же самое касается дополняющих друг друга статистик по

выигрышам/проигрышам. Длительности выигрышных и проигрышных серий не подсчитываются. Желающие могут дополнить наш отчет этими показателями.

Для унификации с общей статистикой тестера имеется метод *fillByTester*, заполняющий все поля через функцию *TesterStatistics*. Позднее мы им воспользуется.

```
      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);
      }
   };
```

Но разумеется, нам нужно реализовать и свой собственный расчет для тех раздельных балансов и эквити торговых систем, которые тестер считать не умеет. Выше были представлены прототипы методов *calcDrawdown*: они в процессе свой работы как раз заполняют последнюю группу полей с префиксом "series_dd". Также в классе *TradeReport* есть метод для расчета коэффициента Шарпа. На вход он принимает ряд чисел и ставку безрискового финансирования. С полным исходным кодом можно ознакомиться в прилагаемом файле.

```
   static double calcSharpe(const double &data[], const double riskFreeRate = 0);
```

Как нетрудно догадаться, при вызове этого метода мы передадим в параметре *data* одноименный массив-член класса *TradeReport* с отсчетами баланса. Процесс же заполнения этого массива и вызов вышеупомянутых методов для конкретных показателей происходит в методе *calcStatistics* (см. ниже). Ему на вход передается объект-фильтр сделок (*filter*), начальные депозит (*start*) и время (*origin*). Предполагается, что вызывающий код настроит фильтр таким образом, чтобы под него попали только сделки интересующей нас торговой системы.

Метод возвращает заполненную структуру *GenericStats*, а кроме того заполняет два массива внутри объекта *TradeReport*: *data* и *moments* — значениями баланса и временными отсчетами изменений, соответственно. Нам это пригодится в окончательной версии эксперта.

```
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) // пополнение/снятие
      {
         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++;        // трейды считаем по сделкам выхода
            if(m.get(DEAL_TYPE) == DEAL_TYPE_SELL)
            {
               stats.buy_trades++; // закрытие сделкой в обратном направлении
            }
            if(profit >= 0)
            {
               stats.wins++;
               if(m.get(DEAL_TYPE) == DEAL_TYPE_BUY)
               {
                  stats.sell_wins++; // закрытие сделкой в обратном направлении
               }
               else
               {
```

```
                  stats.buy_wins++;
               }
            }
         }
         else if(!TU::Equal(profit, 0))
         {
            PUSH(data, balance); // комиссия на вход (если есть)
            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);     // заполняем все поля подструктуры DrawDown
      stats.recovery = stats.series_dd > DBL_EPSILON ?
         stats.net / stats.series_dd : MathExp(10000.0);
   }
   return stats;
}
};
```

Здесь видно, как мы вызываем *calcSharpe* и *calcDrawdown* для получения соответствующих показателей на массиве *data*. Остальные показатели считаются непосредственно в цикле внутри *calcStatistics*.

Вооружившись классом *TradeReport*, расширим функционал эксперта до версии *UnityMartingaleDraft2.mq5*.

Добавим в класс *UnityMartingale* новые члены.

```
class UnityMartingale: public TradingStrategy
{
protected:
   ...
   TradeReport report;
   TradeReport::DrawDown equity;
   const double deposit;
   const datetime epoch;
   ...
```

Объект *report* нам нужен для вызова *calcStatistics*, куда будет включена просадка по балансу. Объект *equity* потребовался для независимого расчета просадки по эквити. Начальный баланс и дата, а также начало расчета просадки по эквити задаются в конструкторе.

```
public:
   UnityMartingale(const Settings &state, TradingSignal *signal):
      symbol(state.symbol), deposit(AccountInfoDouble(ACCOUNT_BALANCE)),
      epoch(TimeCurrent())
   {
      ...
      equity.calcDrawdown(deposit);
      ...
   }
```

Продолжение расчета просадки по эквити делается на лету — при каждом вызове метода *trade*.

```
   virtual bool trade() override
   {
      ...
      if(MQLInfoInteger(MQL_TESTER))
      {
         if(position[])
         {
            report.resetFloatingPL();
            // после сброса нужно просуммировать все плавающие прибыли
            // для чего вызываем addFloatingPL для каждой существующей позиции,
            // но в этой стратегии есть максимум 1 позиция в каждый момент
            report.addFloatingPL(position[].get(POSITION_PROFIT)
               + position[].get(POSITION_SWAP));
            // после учета всех сумм - обновляем просадку
            equity.calcDrawdown(report.getCurrent());
         }
      }
      ...
   }
```

Но это далеко не все, что нужно для правильного расчета. Дело в том, что плавающую прибыль или убыток мы должны учитывать поверх баланса. В показанном фрагменте делается вызов только *addFloatingPL*, но в классе *TradeReport* есть еще и метод для модификации баланса — *addBalance*. Однако баланс изменяется только при закрытии позиции.

Благодаря концепции ООП, закрытие позиции у нас соответствует удалению объекта *position* класса *PositionState*. Так не можем ли мы перехватить его?

Непосредственно в классе *PositionState* для этого средств не предусмотрено, но мы можем объявить производный класс *PositionStateWithEquity* с особым конструктором и деструктором.

При создании объекта в конструктор передается не только идентификатор позиции, но и указатель на объект отчета, в который нужно будет отправить информацию.

```
class PositionStateWithEquity: public PositionState
{
   TradeReport *report;

public:
   PositionStateWithEquity(const long t, TradeReport *r):
      PositionState(t), report(r) { }
   ...
```

В деструкторе мы находим все сделки по идентификатору закрытой позиции, подсчитываем общий финансовый результат (вместе с комиссиями и прочими отчислениями) и затем вызываем *addBalance* для связанного объекта *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);
      }
   }
};
```

Осталось прояснить один момент — как мы создадим для позиций объекты класса *PositionStateWithEquity* вместо *PositionState*. Для этого достаточно поменять оператор *new* в паре мест, где он вызывается в классе *TradingStrategy*.

```
   position = MQLInfoInteger(MQL_TESTER) ?
      new PositionStateWithEquity(tickets[0], &report) : new PositionState(tickets[0]
```

Таким образом, мы разобрались со сбором данных, но осталось непосредственно сформировать отчет, то есть вызвать *calcStatistics*. Здесь не обойтись без расширения нашего интерфейса *TradingStrategy*: добавим в него метод *statement*.

```
interface TradingStrategy
{
   virtual bool trade(void);
   virtual bool statement();
};
```

Тогда в его реализации для нашей стратегии мы сможем довести работу до логического завершения.

```
class UnityMartingale: public TradingStrategy
{
   ...
   virtual bool statement() override
   {
      if(MQLInfoInteger(MQL_TESTER))
      {
         Print("Separate trade report for ", settings.symbol);
         // просадка по эквити должна уже быть посчитана на лету
         Print("Equity DD:");
         equity.print();
         
         // просадка по балансу считается в результирующем отчете
         Print("Trade Statistics (with Balance DD):");
         // настраиваем фильтр под конкретную стратегию
         DealFilter filter;
         filter.let(DEAL_SYMBOL, settings.symbol)
            .let(DEAL_MAGIC, settings.magic, IS::EQUAL_OR_ZERO);
            // нулевое "магическое" число нужно для последней сделки выхода
            // - её делает сам тестер
         HistorySelect(0, LONG_MAX);
         TradeReport::GenericStats stats =
            report.calcStatistics(filter, deposit, epoch);
         stats.print();
      }
      return false;
   }
   ...
```

Новый метод просто распечатает в журнале все рассчитанные показатели. Пробросив этот же метод через пул торговых систем *TradingStrategyPool*, запросим раздельные отчеты для всех символов из обработчика *OnTester*.

```
double OnTester()
{
   ...
   if(pool[] != NULL)
   {
      pool[].statement(); // просим все торговые системы вывести свои результаты
   }
   ...
}
```

Проверим корректность работы своего отчета. Для этого запустим эксперт в тестере по одному символу и сравним стандартный отчет с нашими расчетами. Например, для настройки *UnityMartingale-eurusd.set*, торгуя на EURUSD H1 получим за 2021 такие показатели.

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

Отчет тестера за 2021 год, EURUSD H1

В журнале наш вариант отображается как две структуры: *DrawDown* с показателями просадки по эквити и *GenericStats* с показателями просадки по балансу и прочей статистикой.

```
Separate trade report for 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 18.3
» [average_trade] [recovery] [max_profit] [max_loss] [sharpe]
»            0.19       3.20         2.00      -2.01     0.15
```

Легко убедиться, что эти числа совпадают с отчетом тестера.

Теперь запустим на том же периоде торговлю сразу по трем символам (настройка *UnityMartingale-combo.set*).

В дополнение к записям по EURUSD в журнале появятся структуры для GBPCHF и AUDJPY.

```
Separate trade report for 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  5.2
» [average_trade] [recovery] [max_profit] [max_loss] [sharpe]
»            0.02       0.09         9.10      -6.73     0.01

Separate trade report for 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 44.5
» [average_trade] [recovery] [max_profit] [max_loss] [sharpe]
»            0.27       1.01         7.58      -5.17     0.09
```

Отчет тестера в данном случае будет содержать обобщенные данные, так что благодаря своим классам мы получили недоступную ранее детализацию.

Однако смотреть на псевдо-отчет в журнале не очень удобно. Более того, хотелось бы увидеть и графическое представление хотя бы линии баланса — её внешний вид зачастую говорит больше о пригодности системы, чем сухая статистика.

Усовершенствуем эксперт, наделив его возможностью формировать наглядные отчеты в формате HTML: в конце концов, отчеты тестера также можно выгружать в HTML, сохранять и сравнивать по прошествии времени. Кроме того, подобные отчеты можно будет в перспективе передавать во фреймах в терминал прямо во время оптимизации, и пользователь получит возможность начать изучать отчеты конкретных проходов еще до завершения всего процесса.

Это будет предпоследняя версия примера *UnityMartingaleDraft3.mq5*.

UnityMartingaleDraft3.mq5

Визуализация торгового отчета включает линию баланса и таблицу со статистическими показателями. Мы не станем формировать полный отчет, аналогичный отчету тестера, а ограничимся избранными наиболее важными величинами — нам важно реализовать рабочий механизм, который можно затем кастомизировать в соответствии с личными требованиями.

Основу алгоритма оформим в виде класса *TradeReportWriter* (*TradeReportWriter.mqh*). Класс сможет хранить произвольное количество отчетов разных торговых систем: каждая в отдельном объекте *DataHolder*, который включает массивы значений и временных меток баланса (*data* и *when*, соответственно), структуру со статистикой *stats*, а также название, цвет и ширину линии для отображения.

```
class TradeReportWriter
{
protected:
   class DataHolder
   {
   public:
      double data[];                   // изменения баланса
      datetime when[];                 // временные метки баланса
      string name;                     // описание
      color clr;                       // цвет
      int width;                       // ширина линии
      TradeReport::GenericStats stats; // торговые показатели
   };
   ...
```

Под объекты класса *DataHolder* выделен массив автоуказателей *curves*. Кроме того, нам понадобятся общие границы по суммам и срокам для совмещения линий всех торговых систем в картинке — это обеспечат переменные *lower*, *upper*, *start* и *stop*.

```
   AutoPtr<DataHolder> curves[];
   double lower, upper;
   datetime start, stop;

public:
   TradeReportWriter(): lower(DBL_MAX), upper(-DBL_MAX), start(0), stop(0) { }
   ...
```

Добавить линию баланса позволяет метод *addCurve*.

```
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); // по умолчанию статистики нет
   PUSH(curves, c);
   return true;
}
```

Второй вариант метода *addCurve* добавляет не только линию баланса, но и набор финансовых показателей в структуре *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;
}
```

Наконец, самый главный метод класса — для визуализации отчета — сделан абстрактным.

```
virtual void render() = 0;
```

Это дает возможность реализовать много способов отображения отчетов, например, как с записью в файлы разных форматов, так и с отрисовкой непосредственно на графике. Мы сейчас ограничимся формированием html-файлов, так как это наиболее технологичный и широко распространенный способ.

Новый класс *HTMLReportWriter* имеет конструктор, в параметрах которого указывается название файла, а также размеры картинки с кривыми балансов. Само изображение будем генерировать в известном формате векторной графики SVG: он идеально походит в данном случае, так как представляет собой подмножество XML-языка, коим является и сам 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;
   }
   ...
```

Прежде чем обратиться к главному публичному методу render, потребуется познакомить читателя с одной технологией, которая будет подробно описана в заключительной 7-ой Части книги. Речь о ресурсах: файлах и массивах произвольных данных, подключаемых к MQL-программе для работы с мультимедиа (звук и изображения), встраивания откомпилированных индикаторов или просто как хранилище прикладной информации. Именно последним вариантом мы сейчас и воспользуемся.

Дело в том, что генерировать HTML-страницу лучше не целиком из MQL-кода, а на основе шаблона (заготовки страницы), в которую MQL-код лишь вставит значения некоторых переменных. Это известный прием в программировании, позволяющий разделить алгоритм и внешнее представление программы (или результата её работы). За счет этого мы можем раздельно экспериментировать с HTML-шаблоном и MQL-кодом, работая с каждой из составных частей в привычной среде. В частности, MetaEditor все же не очень приспособлен для редактирования веб-страниц и их просмотра, точно также как стандартный браузер ничего не "знает" о MQL5 (хотя это можно исправить).

HTML-шаблоны отчетов мы как раз и будем хранить в текстовых файлах, подключенных к исходному коду MQL5 как ресурсы. Подключение делается с помощью специальной директивы *#resource*. Например, вот какая строка есть в файле *TradeReportWriter.mqh*.

```
#resource "TradeReportPage.htm" as string ReportPageTemplate
```

Она означает, что рядом с исходным кодом должен быть файл *TradeReportPage.htm*, который станет доступен в MQL-коде в виде строки *ReportPageTemplate*. По расширению вы можете понять, что файл представляет собой веб-страницу. Приведем содержимое этого файла с сокращениями (у нас нет задачи обучить читателя веб-разработке, хотя, как видно, знания в этой области могут пригодиться и для трейдера). Отступы добавлены для наглядного представления иерархии вложенности HTML-тегов, в файле отступов нет.

```
<!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>
```

Принципы работы шаблонов выбирает разработчик. Существует большое количество уже готовых систем HTML-шаблонов, но они предоставляют много избыточных возможностей и потому слишком сложны для нашего примера. Мы разработаем свою концепцию.

Для начала обратим внимание, что в большинстве веб-страниц есть начальная часть (заголовок), есть завершающая часть ("подвал"), и между ними располагается полезная информация. Вышеприведенная заготовка отчета в этом смысле не исключение. Для обозначения полезного содержимого в ней использован символ тильды '~'. Вместо него MQL-код должен будет вставить изображение баланса и таблицу с показателями. Но наличие '~' необязательно, так как страница может представлять собой единое целое, то есть ту саму полезную среднюю часть: ведь MQL-код сможет при необходимости вставлять результат обработки одного шаблона в другой.

Чтобы завершить отступление касательно HTML-шаблонов, обратим внимание на еще один нюанс. В принципе, веб-страница состоит из тегов, выполняющих разные по сути функции. Стандартные теги HTML сообщают браузеру, что отображать. Кроме них имеются каскадные стили (CSS), которые описывают, как это отображать. Наконец, в странице может быть динамическая составляющая в виде скриптов JavaScript, которые интерактивно управляют и первым, и вторым.

Обычно эти три компонента подвергаются шаблонизации независимо, то есть, например, HTML-шаблон, строго говоря, должен содержать только HTML, но не CSS и JavaScript. Это позволяет, опять-таки, ***"развязать" содержимое, оформление и поведение*** веб-страницы, что облегчает разработку (рекомендуется придерживаться такого же подхода и в MQL5!).

> Однако в нашем примере мы внесли в шаблон все компоненты. В частности, в приведенном шаблоне мы видим тег <style> со стилями CSS и тег <script> с некоторыми функциями JavaScript, которые опущены. Это сделано для упрощения примера, с акцентом на возможности MQL5, а не веб-разработки.

Имея шаблон веб-страницы в переменной *ReportPageTemplate*, подключенной как ресурс, мы можем написать метод *render*.

```
   virtual void render() override
   {
      string headerAndFooter[2];
      StringSplit(ReportPageTemplate, '~', headerAndFooter);
      FileWriteString(handle, headerAndFooter[0]);
      renderContent();
      FileWriteString(handle, headerAndFooter[1]);
   }
   ...
```

Он фактически разделяет страницу на верхнюю и нижнюю половины по символу '~', выводит их как есть, а между ними вызывает вспомогательный метод *renderContent*.

Мы уже описали, из чего будет состоять отчет: общая картинка с кривыми баланса и таблицы с показателями торговых систем, поэтому реализация *renderContent* закономерна.

```
private:
   void renderContent()
   {
      renderSVG();
      renderTables();
   }
```

Генерация картинки внутри *renderSVG* основана на еще одном файле шаблона *TradeReportSVG.htm*, который связывается со строковой переменной *SVGBoxTemplate*:

```
#resource "TradeReportSVG.htm" as string SVGBoxTemplate
```

Содержимое этого шаблона — последнее, которое мы здесь приведем. В исходные коды остальных шаблонов желающие могут заглянуть сами.

```
<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>
```

В коде метода *renderSVG* мы увидим знакомый прием с разделением содержимого на два блока "до" и "после" тильды, однако здесь встречается кое-что новое.

```
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]);
}
```

В начальной части страницы, в строке *headerAndFooter[0]* мы ищем подстроки особого вида "%WIDTH%" и "%HEIGHT%", и заменяем их на требуемые ширину и высоту картинки. Именно по такому принципу в наших шаблонах действует подстановка значений. Например, в данном шаблоне действительно встречаются данные подстроки в теге *rect*:

```
<rect x="0" y="0" width="%WIDTH%" height="%HEIGHT%" style="fill:none; stroke-width:1;
```

Таким образом, если построение отчета заказано размером 600 на 400, строка преобразуется в следующую:

```
<rect x="0" y="0" width="600" height="400" style="fill:none; stroke-width:1; stroke:
```

Это выведет в браузере черную рамку указанных размеров толщиной 1 пиксель.

Генерацией тегов для рисования конкретных линий баланса занимается метод *renderCurve*, в который передаются все необходимые массивы и прочие настройки (название, цвет, толщина). Мы оставим данный метод и прочие узкоспециализированные методы (*renderTables*, *renderTable*) для самостоятельного изучения.

Вернемся к основному модулю эксперта *UnityMartingaleDraft3.mq5*. Зададим размеры изображения графиков балансов и подключим *TradeReportWriter.mqh*.

```
#define MINIWIDTH  400
#define MINIHEIGHT 200

#include <MQL5Book/TradeReportWriter.mqh>
```

Для того чтобы "подружить" стратегии с построителем отчетов потребуется модифицировать метод *statement* в интерфейсе *TradingStrategy*: передадим параметром указатель на объект *TradeReportWriter*, который вызывающий код сможет создать и настроить.

```
interface TradingStrategy
{
   virtual bool trade(void);
   virtual bool statement(TradeReportWriter *writer = NULL);
};
```

В конкретной реализации этого метода в классе нашей стратегии *UnityMartingale* добавим несколько строк.

```
class UnityMartingale: public TradingStrategy
{
   ...
   TradeReport report;
   ...
   virtual bool statement(TradeReportWriter *writer = NULL) override
   {
      if(MQLInfoInteger(MQL_TESTER))
      {
         ...
         // это все уже было
         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);
         ...
         // это добавляем
         if(CheckPointer(writer) != POINTER_INVALID)
         {
            double data[];               // значения баланса
            datetime time[];             // время точек баланса для синхронизации кри
            report.getCurve(data, time); // заполняем массивы и передаем на запись в
            return writer.addCurve(stats, data, time, settings.symbol);
         }
         return true;
      }
      return false;
   }
```

Все сводится к тому, чтобы получить массив баланса и структуру с показателями из объекта *report* (класса *TradeReport*) и передать в объект *TradeReportWriter*, вызвав *addCurve*.

Разумеется, в пуле торговых стратегий обеспечена передача одного и того же объекта *TradeReportWriter* во все стратегии для генерации совмещенного отчета.

```
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;
   }
```

Наконец, наибольшей модификации подвергся обработчик *OnTester*. Для генерации HTML-отчета торговых стратегий было бы достаточно следующих строк.

```
double OnTester()
{
   ...
   const static string tempfile = "temp.html";
   HTMLReportWriter writer(tempfile, MINIWIDTH, MINIHEIGHT);
   if(pool[] != NULL)
   {
      pool[].statement(&writer); // просим стратегии сообщить свои результаты
   }
   writer.render(); // записываем полученные данные в файл
   writer.close();
}
```

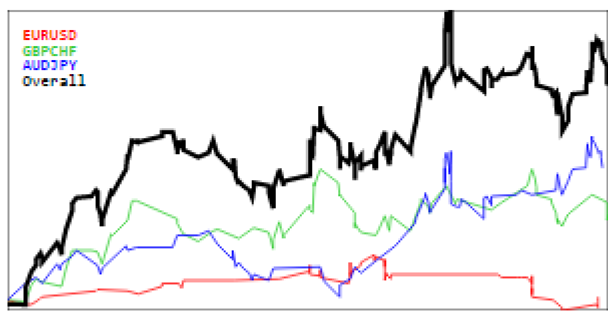
Однако, для наглядности и удобства пользователя было бы здорово добавить в отчет и общую кривую баланса, а также таблицу с общими показателями. Их имеет смысл выводить только когда в настройках эксперта указано несколько символов, потому что в противном случае отчет одной стратегии совпадает с общим в файле.

Это потребовало чуть больше кода.

```
double OnTester()
{
   ...
   // это уже было
   DealFilter filter;
   // настраиваем фильтр и заполняем по нему массив сделок tickets
   ...
   const int n = ArraySize(tickets);

   // это добавляем
   const bool singleSymbol = WorkSymbols == "";
   double curve[];    // кривая общего баланса
   datetime stamps[]; // дата и время точек общего баланса

   if(!singleSymbol) // общий баланс выводим, только если несколько символов/стратеги
   {
      ArrayResize(curve, n + 1);
      ArrayResize(stamps, n + 1);
      curve[0] = TesterStatistics(STAT_INITIAL_DEPOSIT);

      // MQL5 не позволяет узнать время начала теста,
      // это можно было бы выяснить из первой сделки,
      // но она вне условий фильтра конкретной системы,
      // поэтому просто договоримся пропускать в расчетах время 0
      stamps[0] = 0;
   }

   for(int i = 0; i < n; ++i) // цикл по сделкам
   {
      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) // отправляем статистику тестера и общую кривую в отчет
   {
      TradeReport::GenericStats stats;
      stats.fillByTester();
      writer.addCurve(stats, curve, stamps, "Overall", clrBlack, 3);
   }
   ...
}
```

Посмотрим, что у нас получилось. Если запустить эксперт с настройками *UnityMartingale-combo.set*, мы получим в папке одного из агентов *MQL5/Files* файл *temp.html*. Вот как он выглядит в браузере.

Trade Report



EURUSD

Profit>	45.09	1.03	<Profit Factor
Loss>	-43.99	0.01	<Sharpe
Net>	1.10	0.07	<Recovery
Deposit>	10000.00	144	<Deals

Trades	Wins	Losses	Total
Buys	16	18	34
Sells	17	21	38
Total	33	39	72

DD	Absolute	Relative	Deposit
Money	**16.83**	16.83	-0.00
Percent	0.17%	**0.17%**	-0.00%

Max. Profit	Average	Max. Loss
2.00	0.02	-2.00

GBPCHF

Profit>	131.26	1.25	<Profit Factor
Loss>	-105.30	0.09	<Sharpe
Net>	25.96	1.32	<Recovery
Deposit>	10000.00	150	<Deals

Trades	Wins	Losses	Total
Buys	17	18	35
Sells	18	22	40
Total	35	40	75

DD	Absolute	Relative	Deposit
Money	**19.74**	19.74	-0.00
Percent	0.20%	**0.20%**	-0.00%

Max. Profit	Average	Max. Loss
8.78	0.35	-5.55

AUDJPY

Profit>	145.96	1.38	<Profit Factor
Loss>	-106.06	0.13	<Sharpe
Net>	39.90	1.98	<Recovery
Deposit>	10000.00	184	<Deals

Trades	Wins	Losses	Total
Buys	30	19	49
Sells	21	22	43
Total	51	41	92

DD	Absolute	Relative	Deposit
Money	**20.19**	20.19	-0.00
Percent	0.20%	**0.20%**	-0.00%

Max. Profit	Average	Max. Loss
7.03	0.43	-4.90

Overall

Profit>	322.31	1.26	<Profit Factor
Loss>	-255.35	0.09	<Sharpe
Net>	66.96	1.65	<Recovery
Deposit>	10000.00	478	<Deals

Trades	Wins	Losses	Total
Buys	63	55	118
Sells	56	65	121
Total	119	120	239

DD	Absolute	Relative	Deposit
Money	**40.68**	40.68	-0.00
Percent	0.40%	**0.40%**	-0.00%

Max. Profit	Average	Max. Loss
8.78	0.28	-5.55

HTML-отчет для эксперта с несколькими торговыми стратегиями/символами

Теперь, когда мы умеем формировать отчеты об одном тестовом проходе, мы можем отправлять их во время оптимизации в терминал, отбирать на лету лучшие и представлять их пользователю еще до завершения всего процесса. Все отчеты будем складывать в отдельную папку внутри

MQL5/Files терминала. Папка получит имя, содержащее символ и таймфрейм из настроек тестера, а также название эксперта.

UnityMartingale.mq5

Как мы знаем, для отправки файла в терминал достаточно вызвать функцию *FrameAdd*. Файл мы уже сформировали в рамках предыдущей версии.

```
double OnTester()
{
   ...
   if(MQLInfoInteger(MQL_OPTIMIZATION))
   {
      FrameAdd(tempfile, 0, r2 * 100, tempfile);
   }
}
```

В приемной копии эксперта выполним необходимую подготовку. Опишем структуру *Pass* с основными параметрами каждого прохода оптимизации.

```
struct Pass
{
   ulong id;          // номер прохода
   double value;      // значение критерия оптимизации
   string parameters; // оптимизированные параметры в виде списка 'имя=значение'
   string preset;     // текст для генерации set-файла (со всеми параметрами)
};
```

В строке *parameters* пары "имя=значение" соединяются символом '&' — это пригодится для взаимодействия веб-страниц отчетов в дальнейшем (символ '&' — стандарт для объединения параметров в веб-адресах). Формат set-файлов мы не описывали, но исходный код далее для формирования строки *preset* позволяет изучить данный вопрос, так сказать, изнутри, на практике.

По мере поступления фреймов мы будем записывать улучшения по критерию оптимизации в массив *TopPasses*. Текущий лучший проход всегда будет в массиве последним и кроме того доступен в переменной *BestPass*.

```
Pass TopPasses[];     // стек постоянно улучшающихся проходов (последний - лучший)
Pass BestPass;        // текущий лучший проход
string ReportPath;    // выделенная папка для всех html-файлов данной оптимизации
```

В обработчике *OnTesterInit* сформируем имя папки.

```
void OnTesterInit()
{
   BestPass.value = -DBL_MAX;
   ReportPath = _Symbol + "-" + PeriodToString(_Period) + "-"
      + MQLInfoString(MQL_PROGRAM_NAME) + "/";
}
```

В обработчике *OnTesterPass* будем последовательно отбирать только те фреймы, в которых показатель улучшился, выяснять для них значения оптимизируемых и прочих параметров и складывать всю эту информацию в массив структур *Pass*.

```
void OnTesterPass()
{
   ulong   pass;
   string  name;
   long    id;
   double  value;
   uchar   data[];

   // входные параметры для прохода, соответствующего текущему фрейму
   string  params[];
   uint    count;

   while(FrameNext(pass, name, id, value, data))
   {
      // собираем проходы с улучшением статистики
      if(value > BestPass.value && FrameInputs(pass, params, count))
      {
         BestPass.preset = "";
         BestPass.parameters = "";
         // получаем оптимизируемые и прочие параметры для формирования set-файла
         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);
         // записываем фрейм с отчетом в html-файл
```

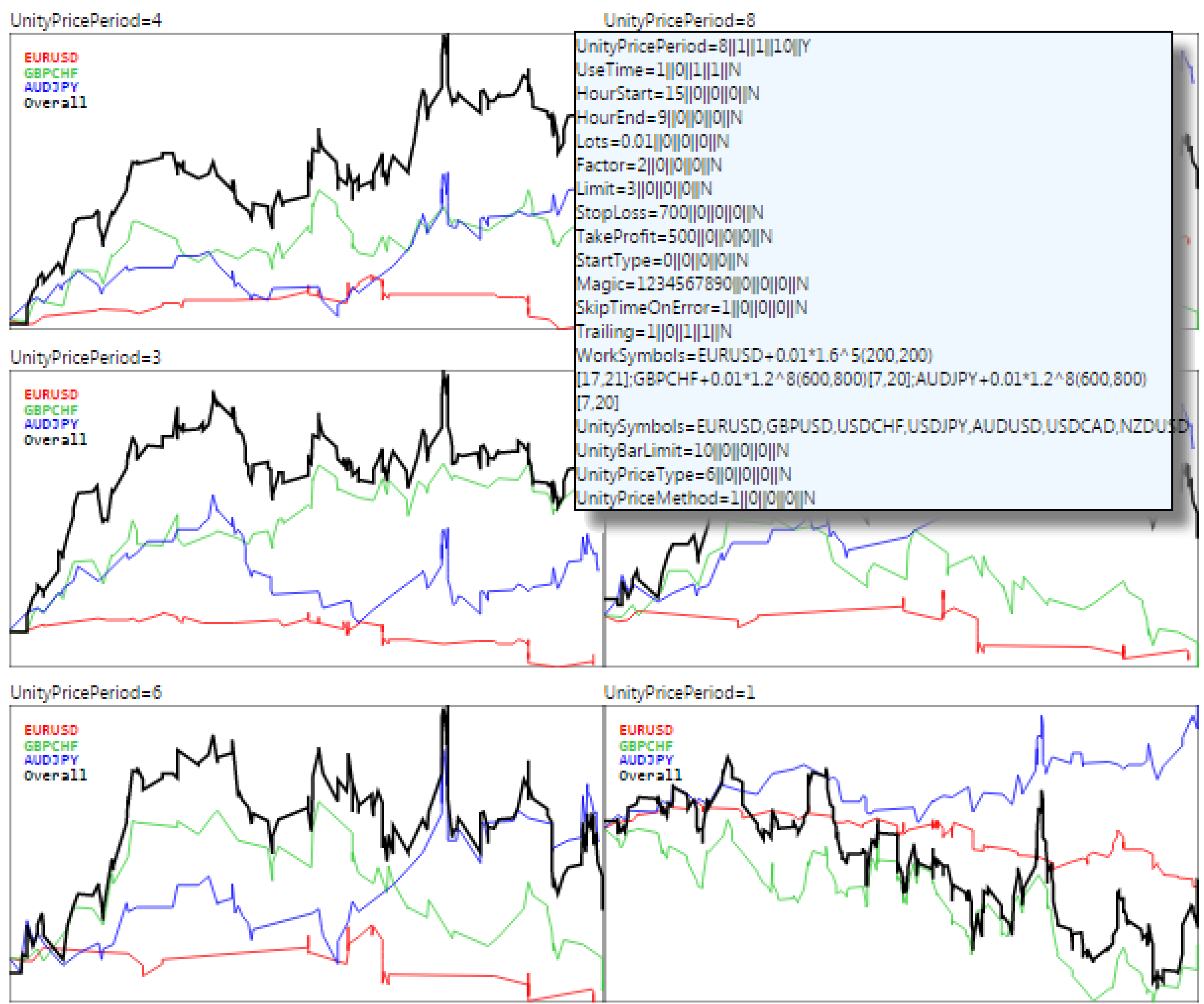
```
         const string text = CharArrayToString(data);
         int handle = FileOpen(StringFormat(ReportPath + "%06.3f-%lld.htm", value, pa
            FILE_WRITE | FILE_TXT | FILE_ANSI);
         FileWriteString(handle, text);
         FileClose(handle);
      }
   }
}
```

Полученные отчеты с улучшениями сохраняются в файлах с именами, включающими значение критерия оптимизации и номер прохода.

А теперь самое интересное. В обработчике *OnTesterDeinit* мы можем сформировать общий html-файл (*overall.htm*), позволяющий увидеть все отчеты сразу (или, скажем, 100 лучших). Здесь используется тот же принцип с шаблонами, что мы рассмотрели раньше.

```
#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]);
   // читаем не более 100 лучших записей из 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);
}
```

На следующем изображении показано, как выглядит обзорная веб-страница после выполнения оптимизации *UnityMartingale.mq5* по параметру *UnityPricePeriod* в мультивалютном режиме.



Обзорная веб-страница с торговыми отчетами лучших проходов оптимизации

Для каждого отчета мы отображаем лишь верхнюю часть, куда попадает график балансов. Эта часть — наиболее удобная для оценки беглым взглядом.

Над каждым графиком выводятся списки оптимизируемых параметров ("имя=значение&имя=значение..."). По щелчку мыши на такой строке открывается блок с текстом для set-файла всех настроек данного прохода. Если щелкнуть мышью внутри блока, его содержимое будет скопировано в буфер обмена. Его можно сохранить в текстовом редакторе и тем самым получить готовый set-файл.

Щелчок мыши по графику вызовет переход на страницу конкретного отчета, вместе с таблицами показателей (приводилась выше).

В завершении раздела коснемся еще одного вопроса. Ранее мы обещали продемонстрировать эффект от функции *TesterHideIndicators*. В эксперте *UnityMartingale.mq5* сейчас используется индикатор *UnityPercentEvent.mq5*, и после любого теста индикатор выводится на открывающийся график. Предположим, что мы хотим скрыть от пользователя механизм работы эксперта и откуда он берет сигналы. Тогда можно вызвать функцию *TesterHideIndicators* (с параметром *true*) в обработчике *OnInit*, перед созданием объекта *UnityController*, в котором и происходит получение дескриптора через *iCustom*.

```
int OnInit()
{
   ...
   TesterHideIndicators(true);
   ...
   controller = new UnityController(UnitySymbols, barwise,
      UnityBarLimit, UnityPriceType, UnityPriceMethod, UnityPricePeriod);
   return INIT_SUCCEEDED;
}
```

Такая версия эксперта уже не будет выводить индикатор на график. Однако скрыт он не очень хорошо. Если заглянуть в журнал тестера, мы там увидим среди множества полезной информации строки о загружаемых программах: сначала сообщение о загрузке самого эксперта, а чуть погодя — о загрузке индикатора.

```
...
expert file added: Experts\MQL5Book\p6\UnityMartingale.ex5.
...
program file added: \Indicators\MQL5Book\p6\UnityPercentEvent.ex5.
...
```

Таким образом, дотошный пользователь может узнать имя индикатора. Исключить такую возможность позволяет механизм ресурсов, о котором мы уже вскользь говорили в контексте заготовок веб-страниц. Оказывается, что откомпилированный индикатор также можно встроить в MQL-программу (в эксперт или в другой индикатор) как ресурс. И подобные программы-ресурсы уже не упоминаются в журнале тестера. Подробно мы изучим ресурсы в 7-й Части книги, а сейчас покажем связанные с ними строки в окончательной версии нашего эксперта.

Прежде всего, опишем ресурс с индикатором директивой *#resource*. Фактически она просто содержит путь к откомпилированному файлу индикатора (очевидно, он уже должен быть откомпилирован заранее), причем здесь обязательно использовать удвоенные обратные косые черты в качестве разделителей — прямые одинарные черты в путях ресурсов, увы, не поддерживаются.

```
#resource "\\Indicators\\MQL5Book\\p6\\UnityPercentEvent.ex5"
```

Затем в строках с вызовом *iCustom* заменим прежний оператор:

```
   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);
      ...
```

на точно такой же, но со ссылкой на ресурс (обратите внимание на синтаксис с ведущей парой двоеточий '::' — это нужно для различения обычных путей в файловой системе и путей внутри ресурсов).

```
   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);
      ...
```

Теперь откомпилированную версию советника можно поставлять пользователям саму по себе, без отдельного индикатора, так как тот спрятан внутри советника. На работе это никак не сказывается, но с учетом вызова *TesterHideIndicators*, внутреннее устройство скрыто. Следует помнить, что если затем индикатор будет обновлен, потребуется перекомпилировать и советник.

6.5.17 Математические вычисления

Тестер в терминале MetaTrader 5 можно использовать не только для проверки торговых стратегий, но и для математических расчётов. Для этого необходимо выбрать соответствующий режим в настройках тестера, в выпадающем списке Моделирование. Это тот же список, где мы выбираем способ генерации тиков, но в данном случае тестер не станет генерировать ни тики, ни котировки, ни даже подключать торговое окружение (торговый счет и символы).

Выбор между полным перебором параметров и генетическим алгоритмом делайте в зависимости от размера пространства поиска. Критерий оптимизации должен быть выбран пользовательский (Custom max).

Прочие поля ввода в настройках тестера (такие как диапазон дат, задержки) не важны и потому автоматически отключаются.

В режиме "Математические вычисления" каждый прогон агента тестирования производится с вызовом только трех функций: *OnInit*, *OnTester*, *OnDeinit*.

Типичная математическая задача для решения в тестере MetaTrader 5 — поиск экстремума от функции многих переменных. Для ее решения необходимо объявить параметры функции в виде input-переменных и разместить блок вычисления её значений в *OnTester*.

Значение функции для конкретного набора входных переменных возвращаем как выходное значение *OnTester*. Не следует использовать при вычислениях какие-либо встроенные функции, кроме математических.

Необходимо помнить, что при оптимизации всегда ищется максимум значения функции *OnTester*. Поэтому при необходимости, для поиска минимума следует возвращать обратные или умноженные на -1 величины.

Чтобы разобраться, как это работает, возьмем для примера относительно простую функцию двух переменных с одним максимумом. Опишем её в алгоритме эксперта *MathCalc.mq5*.

Обычно предполагается, что мы не знаем представление функции в аналитическом виде, иначе можно было бы рассчитать её экстремумы. Но сейчас возьмем известную формулу, чтобы убедиться в правильности ответа.

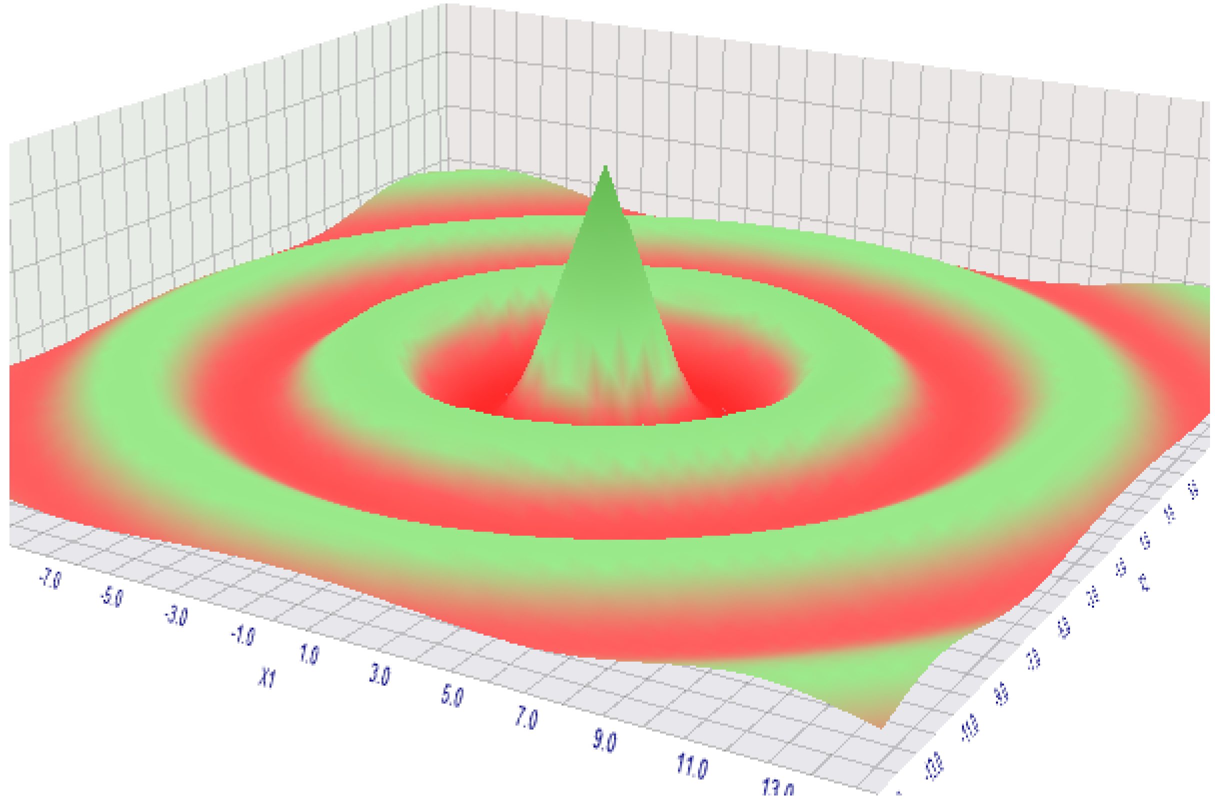
```
input double X1;
input double X2;

double OnTester()
{
   const double r = 1 + sqrt(X1 * X1 + X2 * X2);
   return sin(r) / r;
}
```

К эксперту прилагается файл с параметрами для оптимизации *MathCalc.set*: аргументы X1 и X2 перебираются в диапазонах [-15, +15] с шагом 0.5.

Запустим оптимизацию и увидим решение в таблице оптимизации. Лучший проход дает правильный результат:

```
   X1=0.0
   X2=0.0
OnTester result 0.8414709848078965
```

На графике оптимизации можно включить режим 3D и рассмотреть форму поверхности в наглядном виде.



Результат оптимизации (максимизации) функции в режиме математических вычислений

Вместе с тем, применение тестера в режиме математических вычислений не ограничивается "сухими" научными исследованиями. На его основе, в частности, можно организовать оптимизацию торговых систем с использованием альтернативных широко известных методов оптимизации, таких как метод "роя частиц" или "имитации отжига". Разумеется, для этого потребуется самостоятельно выгрузить историю котировок или тиков в файлы и подключить их к тестируемому эксперту, а также эмулировать совершение сделок, учет позиций и средств. Эта рутинная работа может оказаться привлекательной из-за того, что вам доступна произвольная

настройка процесса оптимизации (в отличие от встроенного "черного ящика" с генетическим алгоритмом) и контроль ресурсов (в первую очередь, оперативной памяти).

6.5.18 Отладка и профилирование

Тестер MetaTrader 5 пригодится не только для проверки прибыльности торговых стратегий, но и для отладки MQL-программ. Выявление ошибок, прежде всего, связано с возможностью воспроизвести проблемную ситуацию. Если бы мы могли запускать MQL-программы только в режиме онлайн, отладка и анализ выполнения исходного кода требовали бы нереально много усилий. Однако тестер позволяет "прогонять" программы на произвольных участках истории, менять настройки счета и торговых символов.

Напомним, что в редакторе MetaEditor есть 2 команды в меню *Отладка*:

- *Начать/Продолжить на реальных данных* (F5);
- *Начать/Продолжить на исторических данных* (Ctrl-F5).

В обоих случаях программа оперативно перекомпилируется специальным способом с дополнительной отладочной информацией в ex5-файле и затем запускается непосредственно в терминале (первый вариант) или в тестере (второй вариант).

При отладке в тестере можно использовать как быстрый (фоновый) режим, так и визуальный. Данная настройка находится в диалоге *Настройка*, на вкладке *Отладка/Профилирование*: включите или выключите флаг *Использовать визуальный режим для отладки на истории*. Непосредственно среду и настройки отлаживаемой программы можно брать из тестера (как они были заданы для этой программы последний раз) или в том же диалоге в полях ввода под флагом *Использовать указанные настройки* (чтобы они заработали, флаг нужно включить).

Вы можете заранее поставить точки остановки (F9) на операторах, в районе которых предположительно что-то начинает работать не так. Тестер приостановит процесс по достижении указанного места в исходном коде.

Обратите внимание, что в тестере количество баров истории, подгружаемых при старте, зависит от многих факторов (таймфрейма, номера дня внутри года и пр.) и может существенно варьироваться. При необходимости отодвигайте начальное время теста в прошлое.

Помимо явных ошибок, которые приводят к остановке программы или явному неверному функционированию, существует класс незаметных ошибок, которые негативно сказываются на быстродействии. Как правило, они не столь очевидны, но превращаются в проблемы по мере увеличения объемов обрабатываемых данных, например, на торговых счетах с очень длинной историей или на графиках с большим количеством объектов разметки.

Для поиска "узких мест" в плане быстродействия отладчик предоставляет механизм профилирования исходного кода. Его также можно выполнять онлайн или в тестере, и последнее особенно ценно, так как позволяет существенно сжать время. Соответствующие команды также доступны в меню отладки.

- *Начать профилирование на реальных данных*;
- *Начать профилирование на исторических данных*;

Для профилирования программа также предварительно перекомпилируется с особыми установками, поэтому не забывайте после завершения отладки или профилирования вновь

откомпилировать программу в обычном режиме (особенно если её планируется отправить клиенту или загрузить в MQL5 Маркет).

В результате профилирования вы получите в MetaEditor време́нную статистику исполнения вашего кода в разбивке по строкам и функциям (методам). В итоге станет ясно, что именно тормозит программу, и следующим этапом разработки обычно выступает *рефакторинг* исходного кода, то есть его переписывание с использованием улучшенных алгоритмов, структур данных или иных принципов конструктивной организации модулей (составных частей). Да, к сожалению, в программировании значительная часть времени тратится именно на переписывание уже существующего кода, поиск и исправление ошибок.

Сама программа может при необходимости выяснить свой режим работы и адаптировать поведение под среду (например, в тестере не пытаться загрузить данные из сети Интернет, так как эта возможность отключена, а считывать их из заранее подготовленного файла).

На стадии компиляции отладочную и рабочую версию программы можно формировать по-разному за счет макроопределений препроцессора _DEBUG и _RELEASE.

На стадии выполнения программы её режимы можно отличить с помощью опций функции *MQLInfoInteger*.

В следующей таблице сведены все доступные сочетания, влияющие на особенности среды исполнения.

Среда исполнения \ флаги	MQL_DEBUG	MQL_PROFILER	Штатный(release)
Онлайн	+	+	+
Тестер (MQL_TESTER)	+	+	+
Тестер (MQL_TESTER+MQL_VISUAL_MODE)	+	-	+

Профилирование в тестере возможно только без визуального режима, поэтому операции с графиками и объектами замеряйте онлайн.

Отладка не допускается в процессе оптимизации, включая и специальные обработчики *OnTesterInit*, *OnTesterDeinit*, *OnTesterPass*. При необходимости, для проверки их работоспособности предусмотрите вызов их кода по другим условиям.

6.5.19 Ограничения работы функций в тестере

При эксплуатации тестера следует учитывать некоторые ограничения, накладываемые на встроенные функции. Часть функций MQL5 API не выполняется в тестере стратегий никогда, а часть работает лишь в одиночных проходах, но не во время оптимизации.

Так для увеличения быстродействия при оптимизации советников функции *Comment*, *Print* и *PrintFormat* не выполняются.

Исключением является использование этих функций внутри обработчика *OnInit*, что сделано для облегчения поиска возможных причин ошибок инициализации.

Функции, обеспечивающие взаимодействие с "внешним миром" в тестере стратегий не выполняются. К ним относятся *MessageBox*, *PlaySound*, *SendFTP*, *SendMail*, *SendNotification*, *WebRequest* и функции работы с сокетами.

Кроме того, не имеют эффекта и многие функции по работе с графиками и объектами. В частности, вы не сможете поменять символ или период текущего графика с помощью вызова *ChartSetSymbolPeriod*, перечислить все индикаторы (включая подчиненные) с помощью *ChartIndicatorGet*, работать с шаблонами *ChartSaveTemplate* и так далее.

В тестере, даже в визуальном режиме, не генерируются интерактивные события графика, объектов, клавиатуры и мыши для обработчика *OnChartEvent*.

Часть 7. Расширенные средства MQL5

В данной части книги мы познакомимся с дополнительными возможностями MQL5 API по самым разным направлениям, которые могут потребоваться при разработке программ для среды MetaTrader 5. Некоторые из них носят прикладной трейдерский характер, например, пользовательские финансовые инструменты, встроенный экономический календарь. Другие представляют собой универсальные технологии, способные пригодиться везде: сетевые функции, базы данных, криптография.

Кроме того, мы рассмотрим расширение MQL-программ с помощью ресурсов — файлов произвольного типа, которые могут встраиваться в код и содержать в себе мультимедиа, "тяжелые" настройки из внешних программ (например, готовые модели машинного обучения или конфигурации нейронных сетей) или другие MQL-программы (индикаторы) в откомпилированном виде.

Пара глав будет посвящена модульной разработке MQL-программ. В этом контексте мы рассмотрим специальный тип программ — библиотеки, которые позволяют в закрытом виде поставлять готовые наборы специфических API для подключения к другим MQL-программам, но при этом не могут использоваться сами по себе. Также мы изучим возможности по организации процесса разработки программных комплексов и объединения логически взаимосвязанных программ в проекты.

Наконец, будет представлена интеграция с другими программными средами, в частности, Python.

В книгу не вошли некоторые узкоспециализированные темы, которые могут быть интересны продвинутым пользователям: аппаратные возможности для параллельных вычислений с помощью OpenCL, а также 2D- и 3D-графика на основе DirectX. С этими технологиями предлагается ознакомиться по документации и статьям на сайте mql5.com.

Программирование на MQL5 для трейдеров — исходные коды из книги: Часть 7

Примеры из книги также доступны в публичном проекте \MQL5\Shared Projects\MQL5Book

7.1 Ресурсы

Для работы MQL-программ может потребоваться множество вспомогательных ресурсов, которые представляют собой массивы прикладных данных или файлы различных типов: изображения, звуки, шрифты. Среда разработки MQL-программ позволяет включить все подобные ресурсы в исполняемый файл на этапе компиляции. Это исключает необходимость их параллельного переноса и установки вместе с основной программой и делает из неё законченный самодостаточный продукт, удобный для конечного пользователя.

В данной главе мы изучим способы описания ресурсов разного типа и встроенные функции для последующих операций с подключенными ресурсами.

Особое место среди ресурсов занимают графические ресурсы с растровыми изображениями, то есть массивами точек (пикселей) широко известного формата BMP — их MQL5 API позволяет создавать, редактировать и отображать на графике динамически.

Ранее мы уже знакомились с графическими объектами и, в частности, объектами типов OBJ_BITMAP и OBJ_BITMAP_LABEL, удобными для конструирования пользовательских интерфейсов. Для них существует свойство OBJPROP_BMPFILE, задающее изображение в виде файла или ресурса. И если до сих пор мы приводили только примеры с файлами, то теперь научимся работать и с картинками-ресурсами.

7.1.1 Описание ресурсов с помощью директивы #resource

Для включения файла ресурса в откомпилированную версию программы следует использовать в исходном коде директиву *#resource*. Директива имеет разные формы в зависимости от типа файла. В любом случае в директиве присутствует основная часть: ключевое слово *#resource* и далее константная строка.

```
#resource "путь_имя_файла"
```

Команда *#resource* предписывает компилятору включить в генерируемую исполняемую программу (в двоичном формате *ex5*) файл с указанным именем и, при необходимости, местом его размещения (в момент компиляции). Путь является опциональным: если в строке указано только имя файла, он ищется в каталоге рядом с компилируемым исходным кодом. При наличии в строке пути применяются правила его разбора, описанные ниже.

Компилятор ищет ресурс по указанному пути в следующей последовательности:

- Если в начале пути стоит символ-разделитель обратная косая черта '\\' (она должна быть задвоена, поскольку одиночный обратный слэш является управляющим символом: в частности, '\' используется для переводов строк '\r', '\n' и табуляций '\t'), то ресурс ищется, начиная с папки MQL5 внутри каталога данных терминала.
- Если обратной косой черты нет, то ресурс ищется относительно расположения исходного файла, в котором этот ресурс прописан.

Обратите внимание: в константных строках с путями ресурсов обязательно использовать удвоенные обратные косые черты в качестве разделителей — прямые одинарные черты здесь не поддерживаются, в отличие от путей в файловой системе.

Например:

```
#resource "\\Images\\euro.bmp" // euro.bmp находится в /MQL5/Images/
#resource "picture.bmp"        // picture.bmp находится в том же каталоге,
                               // где и исходный файл (mq5 или mqh)
#resource "Resource\\map.bmp"  // map.bmp находится в подпапке Resource того каталога
                               // где и исходный файл (mq5 или mqh)
```

Если ресурс декларируется с относительным путем в заголовочном mqh-файле, путь рассматривается от этого mqh-файла, а не от mq5-файла компилируемой программы.

В пути ресурса недопустимо использовать подстроки "..\\" и ":\\".

С помощью нескольких директив можно, например, поместить непосредственно в ex5-файл все необходимые картинки и звуки. Тогда для запуска такой программы в другом терминале не потребуется передавать их отдельно. Программные способы обращения к ресурсам из MQL5 мы рассмотрим в следующих разделах.

Длина константной строки "путь_имя_файла" не должна превышать 63 символа.

Размер файла ресурса не может быть больше 128 Mb.

Файлы ресурсов перед включением в исполняемый файл автоматически сжимаются.

После того как ресурс объявлен директивой *#resource*, его можно использовать в любой части программы. Именем ресурса становится указанная в директиве константная строка без косой черты в начале (если есть), причем перед содержимым строки следует добавлять специальный признак ресурса — два двоеточия "::".

Ниже приведены примеры ресурсов и их имена в комментариях.

```
#resource "\\Images\\euro.bmp"          // имя ресурса - ::Images\\euro.bmp
#resource "picture.bmp"                 // имя ресурса - ::picture.bmp
#resource "Resource\\map.bmp"           // имя ресурса - ::Resource\\map.bmp
#resource "\\Files\\Pictures\\good.bmp" // имя ресурса - ::Files\\Pictures\\good.bmp
#resource "\\Files\\demo.wav";          // имя ресурса - ::Files\\demo.wav"
#resource "\\Sounds\\thrill.wav";       // имя ресурса - ::Sounds\\thrill.wav"
```

Далее в MQL-коде можно ссылаться на эти ресурсы следующим образом (здесь приведены только уже известные нам функции *ObjectSetString* и *PlaySound*, но есть и другие варианты использования, такие как *ResourceReadImage*, которые будут описаны в следующих разделах).

```
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");
```

Необходимо отметить, что при установке объектам OBJ_BITMAP и OBJ_BITMAP_LABEL изображения из ресурса, значение свойства OBJPROP_BMPFILE уже нельзя менять вручную (в диалоге свойств объекта).

Обратите внимание, что wav-файлы задаются по умолчанию для функции *PlaySound* относительно папки *Sounds* (или её вложенных папок), расположенной в каталоге данных терминала. В то же время, ресурсы (включая и звуковые), если они описываются с начальной косой чертой в пути, ищутся внутри каталога MQL5. Поэтому в примере выше строка "\\Sounds\\thrill.wav" обозначает файл *MQL5/Sounds/thrill.wav*, но не *Sounds/thrill.wav* относительно каталога данных (там действительно есть каталог *Sounds* со стандартными звуками терминала).

Рассмотренный выше простой синтаксис директивы *#resource* позволяет описать только ресурсы-картинки (формата BMP) и ресурсы-звуки (формата WAV). Попытка описать как ресурс файл другого типа приведет к ошибке "неизвестный ресурс" ("unknown resource type").

В результате обработки директивы *#resource* файлы фактически встраиваются в исполняемую двоичную программу и становятся доступны в ней по имени ресурса. Причем следует обратить внимание на особое свойство подобных ресурсов — их публичную доступность и из других программ (подробнее об этом — в следующем разделе).

MQL5 поддерживает и другой способ встраивания файла в программу — в виде ресурсной переменной. Этот способ использует расширенный синтаксис директивы *#resource* и позволяет

подключать не только файлы типов BMP или WAV, но и других, например, текст или массив структур.

Практический пример подключения ресурсов мы разберем через пару разделов.

7.1.2 Разделяемое использование ресурсов разных MQL-программ

Имя ресурса — уникально во всем терминале. Позже мы научимся создавать ресурсы не на стадии компиляции (директивой *#resource*), а динамически с помощью функции *ResourceCreate*, но в любом случае ресурс объявляется в контексте создающей его программы, так что уникальность полного имени обеспечивается автоматически, за счет привязки к файловой системе (пути и имени конкретного *ex5*-файла).

Любая MQL-программа может не только сама содержать ресурсы и использовать их, но обращаться к ресурсам другой откомпилированной программы (ex5-файла). Это возможно при условии, когда использующая ресурс программа знает путь размещения и название другой программы, содержащей требуемый ресурс, а также имя этого ресурса.

Таким образом, терминал обеспечивает важное свойство ресурсов — их разделяемое использование: ресурсы из одного ex5-файла можно задействовать во многих других программах.

Для того чтобы использовать ресурс из стороннего ex5-файла, его нужно указать в виде "путь_имя_файла.ex5::имя_ресурса". Например, пусть в скрипте *DrawingScript.mq5* указан ресурс-картинка в файле *triangle.bmp*:

```
#resource "\\Files\\triangle.bmp"
```

Тогда его имя для использования в самом скрипте будет выглядеть как "::Files\\triangle.bmp".

Чтобы использовать этот же ресурс из другой программы, например, эксперта, нужно перед именем ресурса добавить путь к ex5-файлу скрипта относительно папки MQL5 каталога данных терминала и имя самого скрипта (в откомпилированном виде *DrawingScript.ex5*). Пусть скрипт лежит в стандартной папке *MQL5/Scripts/*, тогда обращение к картинке следует выполнять с помощью строки "\\Scripts\\DrawingScript.ex5::Files\\triangle.bmp". Расширение ".ex5" опционально.

Если при обращении к ресурсу другого ex5-файла путь к этому файлу не указан, то такой файл ищется в той же папке, где находится обратившаяся за ресурсом программа. Например, если предположить, что тот же советник находится в стандартной папке *MQL5/Experts/* и делает запрос ресурса без указания пути (например, так — "DrawingScript.ex5::Files\\triangle.bmp"), то это приведет к поиску *DrawingScript.ex5* в папке *MQL5/Experts/*.

За счет разделяемого использования ресурсов их динамическое создание и обновление можно применять для обмена данными между MQL-программами, причем это происходит прямо в памяти, и потому является хорошей альтернативой для файлов или глобальных переменных.

Обратите внимание, что для загрузки ресурса из MQL-программы не требуется её запускать: для чтения ресурсов достаточно самого наличия ex5-файла с ресурсами.

Важным исключением, когда разделение ресурса перестает работать, является его описание в виде ресурсной переменной.

7.1.3 Ресурсные переменные

Директива *#resource* имеет специальную форму, с помощью которой внешние файлы можно объявлять в виде ресурсных переменных и обращаться к ним внутри программы, как к обычным переменным соответствующего типа. Формат объявления таков:

```
#resource "путь_имя_файла" as тип_ресурсной_переменной имя_ресурсной_переменной
```

Вот несколько примеров объявлений:

```
#resource "data.bin" as int Data[]          // массив типа int с данными из файла da
#resource "rates.dat" as MqlRates Rates[]   // массив структур MqlRates из файла rat
#resource "data.txt" as string Message      // строка с содержимым файла data.txt
#resource "image.bmp" as bitmap Bitmap1[]   // одномерный массив с пикселями изображ
                                            // из файла image.bmp
#resource "image.bmp" as bitmap Bitmap2[][] // двумерный массив с тем же изображение
```

Дадим несколько пояснений. Ресурсные переменные являются константами (их нельзя модифицировать в коде MQL5). Для редактирования, например, изображений перед выводом на экран, следует создавать копии ресурсных переменных-массивов.

Для текстовых файлов (ресурсов типа *string*) производится автоматическое определение кодировки по наличию BOM-заголовка. Если BOM отсутствует, то кодировка определяется по содержимому файла. Поддерживаются файлы в кодировках ANSI, UTF-8 и UTF-16. При чтении данных из файлов все строки переводятся в Unicode.

Использование ресурсных строковых переменных может существенно облегчить написание программ, которые основаны не только на чистом MQL5, но и дополнительных технологиях. Например, вы можете написать код OpenCL (который поддерживается в MQL5 в качестве расширения) в отдельном файле, а затем включить его в виде строки в ресурсы MQL-программы. В примере большого эксперта мы уже применяли строки-ресурсы для подключения HTML-шаблонов.

Для изображений введен специальный тип *bitmap*, имеющий несколько особенностей.

Тип *bitmap* описывает одну точку или пиксель изображения и имеет представление 4-х байтового беззнакового целого (*uint*). Внутри пикселя содержатся 4 байта, которые соответствуют компонентам цвета в формате ARGB или XRGB (одна буква — один байт), где R — красный (Red), G — зеленый (Green), B — синий (Blue), A — прозрачность (альфа-канал), X — байт игнорируется (без прозрачности). Прозрачность может использоваться для различных эффектов при наложении изображений на график и друг на друга.

Определение форматов ARGB и XRGB мы изучим в разделе про динамическое создание графических ресурсов (см. *ResourceCreate*). Например, для ARGB число в шестнадцатеричном представлении 0xFFFF0000 задает полностью непрозрачный пиксель (старший байт равен 0xFF) красного цвета (следующий байт также равен 0xFF), а следующие байты для зеленой и синей компоненты равны нулю.

Важно отметить, что кодирование цвета пикселей отличается от байтового представления типа *color*. Напомним, что значение типа *color* можно записать в шестнадцатеричном виде так: 0x00BBGGRR, где BB, GG, RR — это соответственно синяя, зеленая и красная компоненты (в каждом байте значение 255 дает максимальную интенсивность компоненты). При аналогичной записи пикселя налицо обратный порядок байтов: 0xAARRGGBB. Полная прозрачность

получается, когда старший байт (здесь обозначен AA) равен 0, а значение 255 — сплошной цвет. Для перевода цвета *color* в ARGB формат имеется функция *ColorToARGB*.

Файлы формата BMP могут иметь различные способы кодирования (если вы их создаете или редактируете в каком-либо редакторе, уточните этот вопрос в документации этой программы). Ресурсы MQL5 поддерживают не все существующие способы кодирования. Проверить, поддерживается ли конкретный файл, можно с помощью функции *ResourceCreate*. Указание файла неподдерживаемого формата BMP в директиве приведет к ошибке компиляции.

При загрузке файла с 24-битным кодированием цвета для всех пикселей компонента альфа-канала устанавливается в значение 255 (непрозрачный).

При загрузке файла с 32-битным кодированием цвета без альфа-канала, также подразумевает отсутствие прозрачности, то есть для всех пикселей изображения компонента альфа-канала устанавливается в значение 255.

При загрузке файла с 32-битным кодированием цвета с альфа-каналом никаких манипуляций с пикселями не происходит.

Изображения можно описывать как одномерными, так и двумерными массивами. Это влияет только на способ адресации, но объем занимаемой памяти будет одинаковым. В обоих случаях размеры массива автоматически устанавливаются на основе данных из bmp-файла. Размер одномерного массива будет равен произведению высоты на ширину картинки *(height * width)*, а двумерный массив получит раздельные измерения *[height][width]*: первый индекс — номер строки, второй — точка в строке.

> Внимание! При объявлении ресурса в привязке к ресурсной переменной обращаться к ресурсу можно только через эту переменную, а стандартный способ чтения через имя "::имя_ресурса" (или в более общем случае "путь_имя_файла.ex5::имя_ресурса") больше не работает. Это также означает, что подобные ресурсы нельзя использовать, как разделяемые, из других программ.

Рассмотрим в качестве примера два индикатора: оба являются безбуферными. Данный тип MQL-программ выбран только из соображений удобства (потому что их можно бесконфликтно наложить на график в дополнение к другим индикаторам (эксперт потребовал бы график без другого эксперта), а кроме того они остаются на графике и доступны для последующего изменения настроек, в отличие от скриптов).

Индикатор *BmpOwner.mq5* содержит описание трех ресурсов:

- изображение "search1.bmp" с простой директивой *#resource* и потому доступное из других программ;
- изображение "search2.bmp" в виде ресурсной переменной-массива типа *bitmap* и потому недоступное извне;
- текстовый файл "message.txt" как ресурсная строка для вывода предупреждения пользователю;

Обе картинки никак не используются внутри данного индикатора. Строка с предупреждением требуется в функции *OnInit* для вызова *Alert*, поскольку индикатор не предназначен для самостоятельного применения, а только выступает в качестве поставщика ресурса-картинки.

> Если ресурсная переменная не используется в исходном коде, компилятор может вообще не включить ресурс в двоичный код программы, но это не касается изображений.

```
#resource "search1.bmp"
#resource "search2.bmp" as bitmap image[]
#resource "message.txt" as string Message
```

Все три файла расположены в том же каталоге, где и исходный код индикатора — *MQL5/Indicators/MQL5Book/p7/*.

Если пользователь попробует запустить индикатор, тот выводит предупреждение и сразу же прекращает работу. Предупреждение содержится в ресурсной переменной-строке Message.

```
int OnInit()
{
   Alert(Message); // эквивалент следующей строки кода
   // Alert("This indicator is not intended to run, it holds a bitmap resource");

   // удаляем индикатор явно, т.к. иначе он остается "висеть" на графике неинициализи
   ChartIndicatorDelete(0, 0, MQLInfoString(MQL_PROGRAM_NAME));
   return INIT_FAILED;
}
```

Во втором индикаторе *BmpUser.mq5*, мы попытаемся использовать внешние ресурсы, заданные во входных переменных *ResourceOff* и *ResourceOn*, для отображения в объекте OBJ_BITMAP_LABEL.

```
input string ResourceOff = "BmpOwner.ex5::search1.bmp";
input string ResourceOn = "BmpOwner.ex5::search2.bmp";
```

По умолчанию состояние объекта отключенное/отжатое ("Off") и для него берется картинка из предыдущего индикатора — "BmpOwner.ex5::search1.bmp". Этот путь и имя ресурса аналогичны полной записи "\\Indicators\\MQL5Book\\p7\\BmpOwner.ex5::search1.bmp". Краткая форма здесь допустима с учетом того, что индикаторы располагаются рядом друг с другом. Если вы впоследствии откроете диалог свойств объекта, то увидите там в полях *Bitmap file (On/Off)* именно полную запись.

Для нажатого состояния в *ResourceOn* предлагается прочитать ресурс "BmpOwner.ex5::search2.bmp" (посмотрим, что из этого выйдет).

В других входных переменных можно выбрать угол графика, относительно которого задается позиционирование картинки, и величина отступов по горизонтали и вертикали.

```
input int X = 25;
input int Y = 25;
input ENUM_BASE_CORNER Corner = CORNER_RIGHT_LOWER;
```

Непосредственно создание объекта OBJ_BITMAP_LABEL и установка его свойств, включая и имя ресурса в качестве картинки для OBJPROP_BMPFILE, выполняется в *OnInit*.

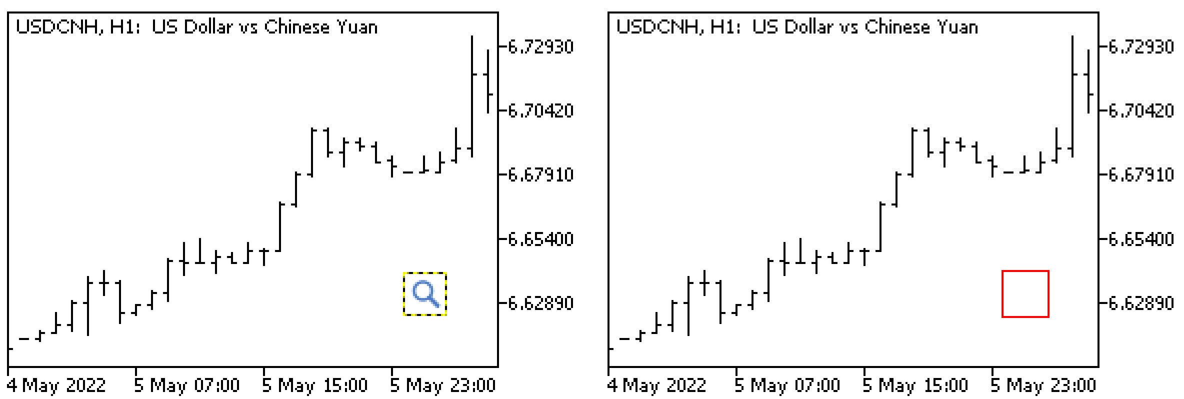
```
const string Prefix = "BMP_";
const ENUM_ANCHOR_POINT Anchors[] =
{
   ANCHOR_LEFT_UPPER,
   ANCHOR_LEFT_LOWER,
   ANCHOR_RIGHT_LOWER,
   ANCHOR_RIGHT_UPPER
};

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]);
}
```

Напомним, что при указании картинок в OBJPROP_BMPFILE нажатое состояние обозначается модификатором 0, а отжатое (по умолчанию) — модификатором 1, что несколько неожиданно.

Обработчик *OnDeinit* удаляет объект при выгрузке индикатора.

```
void OnDeinit(const int)
{
   ObjectsDeleteAll(0, Prefix);
}
```

Откомпилируем оба индикатора и запустим *BmpUser.ex5* с параметрами по умолчанию. На графике должно появиться изображение графического файла *search1.bmp* (см. слева).



Нормальное (слева) и проблемное (справа) отображение графических ресурсов в объекте на графике

Если щелкнуть мышью на картинке, то есть переключить ей в нажатое состояние, программа попытается обратиться к ресурсу "BmpOwner.ex5::search2.bmp" (который недоступен из-за того, что к нему привязан ресурсный массив *bitmap*). В результате мы увидим красный квадрат, обозначающий пустой объект без картинки (см. выше, справа). Аналогичная ситуация будет всегда, если во входном параметре указывать путь или имя с заведомо несуществующим или

неразделяемым ресурсом. Вы можете создать собственную программу, описать в ней ресурс, ссылающийся на какой-либо существующий bmp-файл, и затем указать во входных параметрах индикатора *BmpUser* — индикатор сможет вывести картинку на график.

7.1.4 Подключение пользовательских индикаторов как ресурсов

Для работы MQL-программ может потребоваться один или несколько пользовательских индикаторов. Все они могут быть включены в исполняемый ex5-файл как ресурсы, что упрощает распространение и установку.

Директива *#resource* с описанием вложенного индикатора имеет следующий формат:

```
#resource "путь_имя_индикатора.ex5"
```

Правила задания и поиска указанного файла те же самые, что и для всех ресурсов в целом.

Мы уже использовали данную возможность в большом примера эксперта, в финальной версии *UnityMartingale.mq5*.

```
#resource "\\Indicators\\MQL5Book\\p6\\UnityPercentEvent.ex5"
```

Далее этот ресурс передавался в функцию *iCustom* вместо имени индикатора: "::Indicators\\MQL5Book\\p6\\UnityPercentEvent.ex5".

Случай, когда пользовательский индикатор создает в функции *OnInit* одну или несколько копий себя, требует отдельного рассмотрения (если само это техническое решение кажется странным, мы приведем практический пример после вводных примеров).

Как мы знаем, для использования ресурса из MQL-программы его необходимо указывать в виде: "путь_имя_файла.ex5::имя_ресурса". Например, если индикатор *EmbeddedIndicator.ex5* включается в качестве ресурса в другой индикатор *MainIndicator.mq5* (а точнее, в его двоичный образ *MainIndicator.ex5*), то имя, указываемое при вызове самого себя через *iCustom*, уже не может быть кратким, без пути, а путь должен включать расположение "родительского" индикатора внутри папки MQL5. В противном случае система не сможет найти вложенный индикатор.

Действительно, в обычных обстоятельствах индикатор может вызвать самого себя, например, с помощью оператора *iCustom(_Symbol, _Period, myself,...)*, где *myself* — это строка, равная либо *MQLInfoString(MQL_PROGRAM_NAME)*, либо названию, которое было предварительно назначено в коде свойству INDICATOR_SHORTNAME. Но когда индикатор находится внутри другой MQL-программы как ресурс, имя уже не ссылается на соответствующий файл — ведь файл, который послужил прообразом для ресурса, остался на том компьютере, где производилась компиляция, а на компьютере пользователя есть только файл *MainIndicator.ex5*. Здесь потребуется некоторый анализ программного окружения при запуске программы.

Рассмотрим это на практике.

Создадим для начала индикатор *NonEmbeddedIndicator.mq5*. Важно отметить, что он расположен в папке *MQL5/Indicators/MQL5Book/p7/SubFolder/*, то есть в подпапке *SubFolder* относительно папки *p7*, выделенной для всех индикаторов данной Части книги. Это сделано намеренно, чтобы эмулировать ситуацию, когда откомпилированный файл отсутствует на компьютере пользователя. Как это работает (а точнее — демонстрирует проблему), мы сейчас увидим.

Индикатор имеет единственный входной параметр *Reference*, назначение которого — подсчет количества копий самого себя: при первом создании в параметре будет 0, и индикатор создаст

свою копию со значением параметра 1. Вторая копия, "увидев" значение 1, уже не станет создавать еще одну копию (иначе мы быстро исчерпали бы ресурсы без пограничного условия остановки размножения).

```
input int Reference = 0;
```

Для дескриптора индикатора-копии зарезервирована переменная *handle*.

```
int handle = 0;
```

В обработчике *OnInit* мы для наглядности прежде всего выводим имя и путь MQL-программы.

```
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);
   ...
```

Далее идет код, подходящий для самозапуска обособленного индикатора (существующего в виде привычного файла *NonEmbeddedIndicator.ex5*).

```
   if(Reference == 0)
   {
      handle = iCustom(_Symbol, _Period, name, 1);
      if(handle == INVALID_HANDLE)
      {
         return INIT_FAILED;
      }
   }
   Print("Success");
   return INIT_SUCCEEDED;
}
```

Такой индикатор мы могли бы успешно разместить на графике и получили бы в журнале записи такого рода (актуальные пути файловой системы будут у вас свои):

```
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
```

Таким образом, копия запустилась успешно просто по имени "NonEmbeddedIndicator".

Оставим пока данный индикатор и создадим второй — *FaultyIndicator.mq5*, в который подключим первый индикатор как ресурс (обратите внимание на указание подпапки *SubFolder* в относительном пути ресурса — это нужно, поскольку индикатор *FaultyIndicator.mq5* находится в папке на уровень выше: *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;
}
```

Если попытаться запустить откомпилированный *FaultyIndicator.ex5*, возникнет ошибка:

```
0
Name: NonEmbeddedIndicator
Full path: C:\Program Files\MT5East\MQL5\Indicators\MQL5Book\p7\FaultyIndicator.ex5 »
»  ::SubFolder\NonEmbeddedIndicator.ex5
cannot load custom indicator 'NonEmbeddedIndicator' [4802]
```

Во время запуска копии вложенного индикатора он ищется в папке основного индикатора, в котором описан ресурс. Но там файла *NonEmbeddedIndicator.ex5* нет, поскольку требуемый ресурс находится внутри *FaultyIndicator.ex5*.

Чтобы решить проблему модифицируем *NonEmbeddedIndicator.mq5*. Прежде всего, дадим ему другое, более правильное имя *EmbeddedIndicator.mq5*. В исходном коде нам потребуется добавить вспомогательную функцию *GetMQL5Path*, которая из общего пути запускаемой MQL-программы умеет вычленить относительную часть внутри папки MQL5 (в этой части будет находиться и название ресурса, если индикатор запускается из ресурса).

```
// 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;
}
```

С учетом новой функции изменим вызов *iCustom* в обработчике *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;
}
```

Убедимся, что данная правка не сломала запуск индикатора самого по себе. Наложение на график приводит к появлению в журнале ожидаемых строк:

```
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
```

Здесь добавился отладочный вывод относительного пути, который получила функция *GetMQL5Path*. Именно эта строка теперь используется в *iCustom*, и она работает в данном режиме — копия создалась.

Теперь встроим этот индикатор как ресурс в другой индикатор в папке *MQL5Book/p7* с именем *MainIndicator.mq5*. Примечательно, что *MainIndicator.mq5* полностью идентичен *FaultyIndicator.mq5* за исключением лишь подключаемого ресурса.

```
// MainIndicator.mq5
#resource "SubFolder\\EmbeddedIndicator.ex5"
...
int OnInit()
{
   handle = iCustom(_Symbol, _Period, "::SubFolder\\EmbeddedIndicator.ex5");
   ...
}
```

Откомпилируем и запустим его. В журнале появятся записи с новым относительным путем, включающим вложенный ресурс.

```
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
```

Как мы видим, на этот раз вложенный индикатор успешно создал копию самого себя, так как использовал квалифицированное название с относительным путем и именем ресурса "\\Indicators\\MQL5Book\\p7\\MainIndicator.ex5::SubFolder\\EmbeddedIndicator.ex5".

Во время многократных экспериментов с запуском данного индикатора обратите внимание, что вложенные копии не сразу выгружаются с графика после удаления главного индикатора. Поэтому повторные запуски следует производить, только дождавшись выгрузки: в противном случае еще выполняющиеся копии будут использованы повторно, и в журнале не появятся вышеприведенные строки инициализации. Для контролирования выгрузки в код добавлена распечатка значения *Reference* в обработчике *OnDeinit*.

Мы обещали показать, что создание индикатором своей копии не является чем-то экстраординарным. В качестве прикладной демонстрации данного приема возьмем индикатор *DeltaPrice.mq5*, который рассчитывает разницу приращений цен заданного порядка. Порядок 0 означает отсутствие дифференцирования (только для проверки исходного временного ряда), 1 — однократное дифференцирование, 2 — двукратное и так далее.

Порядок задается во входном параметре *Differencing*.

```
input int Differencing = 1;
```

Разностный ряд будет отображаться в единственном буфере в подокне.

```
#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[];
```

В обработчике *OnInit* мы не только настраиваем буфер, но и создаем тот же индикатор, передавая во входном параметре уменьшенную на 1 величину.

```
#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;
}
```

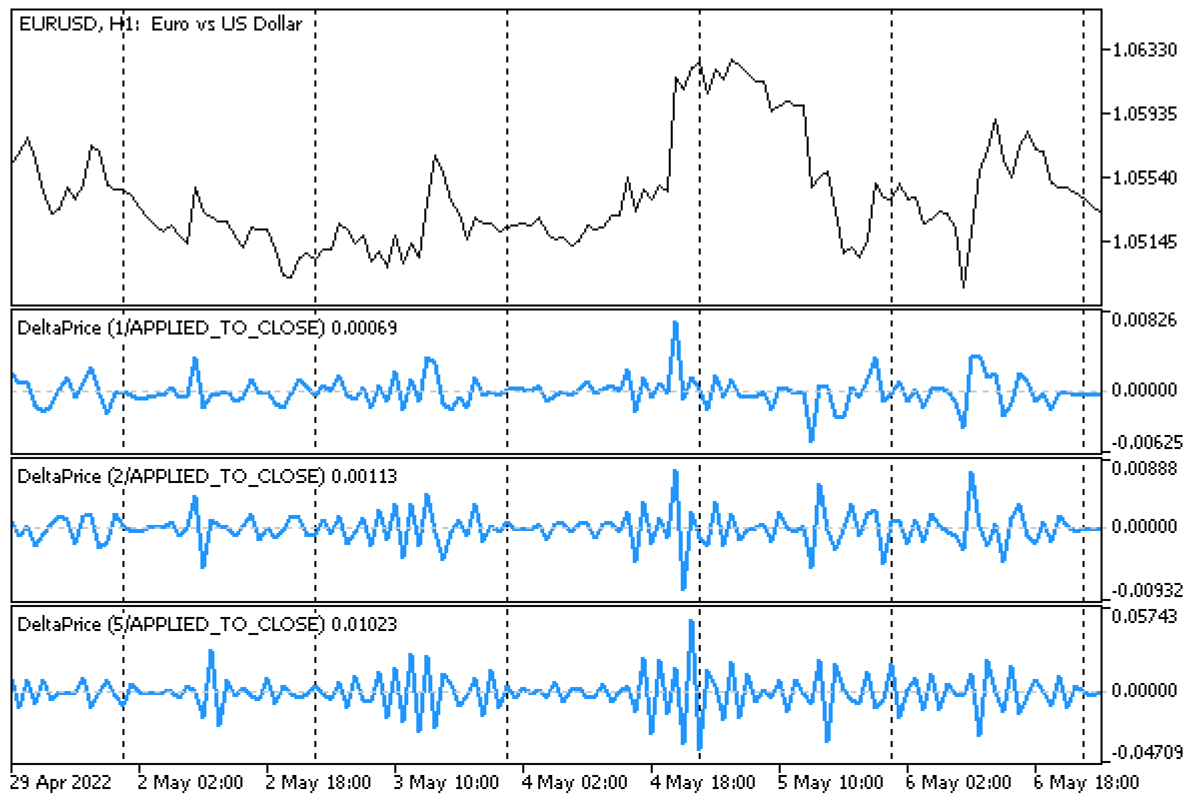
Чтобы избежать потенциальных проблем со встраиванием индикатора в качестве ресурса, мы используем уже проверенную функцию *GetMQL5Path*.

В функции *OnCalculate* выполняем операцию вычитания соседних значений временного ряда. Когда *Differencing* равно 1, операндами выступают элементы массива *price*. При большем значении *Differencing*, мы читаем буфер копии индикатора, созданной для предыдущего порядка.

```
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;
}
```

Исходный тип дифференцируемой цены задается в диалоге настроек индикатора в выпадающем списке *Применить к*. По умолчанию это цена *Close*.

Вот как на графике выглядят несколько копий индикатора с разными порядками дифференцирования.



Разница цен Close различных порядков

7.1.5 Динамическое создание ресурсов: ResourceCreate

Директивы *#resource* встраивают ресурсы в программу на этапе компиляции и потому их можно назвать статическими. Однако часто возникает необходимость генерировать ресурсы (создавать полностью новые или модифицировать имеющиеся) на стадии выполнения программы. Для этих целей MQL5 предоставляет функцию *ResourceCreate*. Созданные с помощью неё ресурсы будем называть динамическими.

Функция имеет две формы: первая позволяет загружать из файлов картинки и звуки, вторая предназначена для создания растровых изображений на основе подготовленного в памяти массива пикселей.

bool ResourceCreate(const string resource, const string filepath)

Функция загружает ресурс под именем *resource* из файла, расположенного по пути *filepath*. Если путь начинается с обратной косой черты '\' (в константных строках его следует задваивать: "\\path\\name.ext"), то файл ищется по этому пути относительно папки MQL5 в каталоге данных терминала (например, "\\Files\\CustomSounds\\Hello.wav" ссылается на *MQL5/Files/CustomSounds/Hello.wav*). Если обратной косой черты нет, то ресурс ищется начиная с папки, в которой расположен исполняемый файл, из которого вызывается функция.

Путь может указывать на статический ресурс, "зашитый" в стороннюю или в текущую MQL-программу. Например, некий скрипт способен создать ресурс на основе картинки из индикатора *BmpOwner.mq5*, рассмотренного в разделе о Ресурсных переменных.

```
ResourceCreate("::MyImage", "\\Indicators\\MQL5Book\\p7\\BmpOwner.ex5::search1.bmp");
```

Имя ресурса в параметре *resource* может содержать начальное двойное двоеточие (хотя это не обязательно, т.к. если его нет, префикс "::" добавится к имени автоматически) — за счет этого обеспечивается унификация использования одной строки как для объявления ресурса в вызове *ResourceCreate*, так и последующее обращение к нему (например, при установке свойства OBJPROP_BMPFILE).

Разумеется, приведенный выше оператор по созданию динамического ресурса избыточен, если мы просто хотим загрузить сторонний ресурс-картинку в свой объект на графике, поскольку достаточно напрямую присвоить свойству OBJPROP_BMPFILE строку "\\Indicators\\MQL5Book\\p7\\BmpOwner.ex5::search1.bmp". Однако если требуется отредактировать изображение, динамический ресурс незаменим. Далее мы покажем пример в разделе Чтение и модификация данных ресурса.

Динамические ресурсы являются публично доступными из других MQL-программ по полному имени, которое включает путь и название создавшей ресурс программы. Например, если предыдущий вызов *ResourceCreate* делал скрипт *MQL5/Scripts/MyExample.ex5*, то другая MQL-программа может обратиться к тому же ресурсу по полной ссылке "\\Scripts\\MyExample.ex5::MyImage", а любой другой скрипт в той же папке — используя краткую запись "MyExample.ex5::MyImage" (здесь относительный путь просто вырожден). Правила записи полных (от корневой папки MQL5) и относительных путей приводились выше.

Функция *ResourceCreate* возвращает логический признак успеха (*true*) или ошибки (*false*) в результате выполнения. Код ошибки, как обычно, можно узнать в переменной _LastError. В частности, вероятно получение следующих ошибок:

- ERR_RESOURCE_NAME_DUPLICATED (4015) — совпадение имени динамического и статического ресурсов;
- ERR_RESOURCE_NOT_FOUND (4016) — заданный ресурс/файл из параметра *filepath* не найден;
- ERR_RESOURCE_UNSUPPOTED_TYPE (4017) — неподдерживаемый тип ресурса или размер более 2 Gb;
- ERR_RESOURCE_NAME_IS_TOO_LONG (4018) — имя ресурса превышает 63 символа.

Все это касается не только первой формы функции, но и второй.

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)

Параметр *resource* по-прежнему означает имя нового ресурса, а содержимое изображения задается остальными параметрами.

Массив *data* может быть одномерным (*data[]*) или двумерным (*data[][]*): в нем передаются точки (пиксели) растра. Параметры *img_width* и *img_height* устанавливают размеры отображаемого изображения (в пикселях). Эти размеры могут быть меньше физического размера картинки в массиве *data*, за счет чего достигается эффект кадрирования — вывода лишь части исходной картинки. Параметры *data_xoffset* и *data_yoffset* как раз определяют координату левого верхнего угла "кадра".

Параметр *data_width* означает полную ширину исходного изображения (в массиве *data*). Значение 0 подразумевает, что эта ширина совпадает с *img_width*. Параметр *data_width* имеет смысл только при указании одномерного массива в параметре *data*, поскольку для двумерного

массива известны его размеры по обеим размерностям (при этом параметр *data_width* игнорируется и принимается равным второй размерности массива *data[][]*).

В наиболее распространенном случае, когда требуется вывести изображение полностью ("как есть"), применяйте синтаксис:

```
ResourceCreate(name, data, width, height, 0, 0, 0, ...);
```

Например, если в программе имеется статический ресурс, описанный как двумерный массив *bitmap*:

```
#resource "static.bmp" as bitmap data[][]
```

то создание на его основе динамического ресурса возможно таким образом:

```
ResourceCreate("dynamic", data, ArrayRange(data, 1), ArrayRange(data, 0), 0, 0, 0, ..
```

Создание динамического ресурса на основе статического востребовано не только при необходимости прямого редактирования, но и для управления способом обработки цветов при отображении ресурса. Этот режим выбирается с помощью последнего параметра функции — *color_format*. Для него применяется перечисление ENUM_COLOR_FORMAT.

Идентификатор	Описание
COLOR_FORMAT_XRGB_NOALPHA	Компонента альфа-канала (прозрачность) игнорируется
COLOR_FORMAT_ARGB_RAW	Компоненты цвета не обрабатываются терминалом
COLOR_FORMAT_ARGB_NORMALIZE	Компоненты цвета обрабатываются терминалом (см. ниже)

В режиме COLOR_FORMAT_XRGB_NOALPHA изображение выводится без эффектов: каждая точка отображается сплошным цветом (это наиболее быстрый способ отрисовки). Два других режима отображают пиксели с учетом прозрачности в старшем байте каждой точки, но имеют разный эффект. В случае COLOR_FORMAT_ARGB_NORMALIZE терминал выполняет следующие преобразования цветовых компонент каждой точки при подготовке растра в момент вызова *ResourceCreate*:

```
R = R * A / 255
G = G * A / 255
B = B * A / 255
A = A
```

Статические ресурсы-картинки в директивах *#resource* подключаются именно с помощью COLOR_FORMAT_ARGB_NORMALIZE.

В динамическом ресурсе размер массива ограничен значением INT_MAX байт (2147483647, 2 Gb), что существенно превышает предел, налагаемый компилятором при обработке статической директивы *#resource*: напомним, там размер файла не может превышать 128 Mb.

Если второй вариант функции вызывается для создания ресурса с одним и тем же именем, но с меняющимися другими параметрами (содержимым массива пикселей, шириной, высотой или сдвигом), то новый ресурс не пересоздается, а просто обновляется существующий. Изменять ресурс таким образом может только программа-хозяин ресурса (создавшая его изначально).

> Если при создании динамических ресурсов из разных копий программы, выполняющихся на разных графиках, требуется собственный ресурс в каждой копии, следует добавлять *ChartID* в имя ресурса.

Для демонстрации динамического создания изображений в различных цветовых схемах предлагаем разобрать скрипт *ARGBbitmap.mq5*.

К нему статически подключено изображение "argb.bmp".

```
#resource "argb.bmp" as bitmap Data[][]
```

Способ форматирования цвета пользователь выбирает параметром *ColorFormat*.

```
input ENUM_COLOR_FORMAT ColorFormat = COLOR_FORMAT_XRGB_NOALPHA;
```

Имена объекта, в котором будет отображаться изображение, и динамического ресурса описаны переменными *BitmapObject* и *ResName*.

```
const string BitmapObject = "BitmapObject";
const string ResName = "::image";
```

А вот и главная функция скрипта.

```
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()  // ждем команды пользователя на завершение демо
   && TerminalInfoInteger(TERMINAL_KEYSTATE_ESCAPE) == start)
   {
      Sleep(1000);
   }

   Comment("");
   ObjectDelete(0, BitmapObject);
   ResourceFree(ResName);
}
```

Скрипт создает новый ресурс в заданном цветовом режиме и назначает его свойству OBJPROP_BMPFILE объекта типа OBJ_BITMAP_LABEL. Далее скрипт ожидает явной остановки скрипта пользователем или нажатия клавиши *Esc*, после чего удаляет объект (вызвав *ObjectDelete*) и ресурс с помощью функции *ResourceFree*. Обратите внимание, что удаление объекта не означает автоматическое удаление ресурса. Именно поэтому нужна функция *ResourceFree*, о которой мы поговорим в следующем разделе.

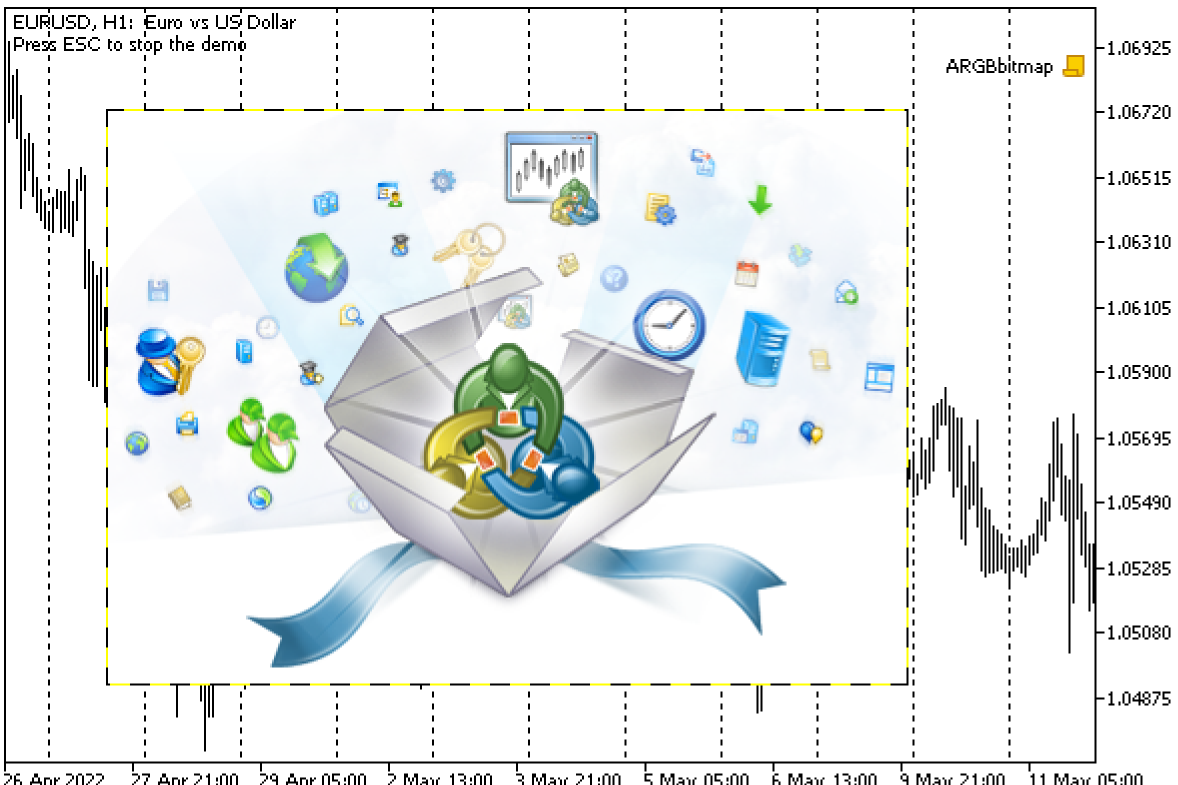
Если не вызвать *ResourceFree*, динамические ресурсы остаются в памяти терминала даже после завершения работы MQL-программы — вплоть до закрытия терминала. Это позволяет использовать их как хранилища или средство для обмена информацией между MQL-программами.

Здесь уместно отметить, что динамический ресурс, созданный с помощью второй формы *ResourceCreate*, не обязан нести в себе изображение. Массив *data* может содержать произвольные данные, если мы не используем их для визуализации. При этом важно задавать схему COLOR_FORMAT_XRGB_NOALPHA. Такой пример мы еще покажем.

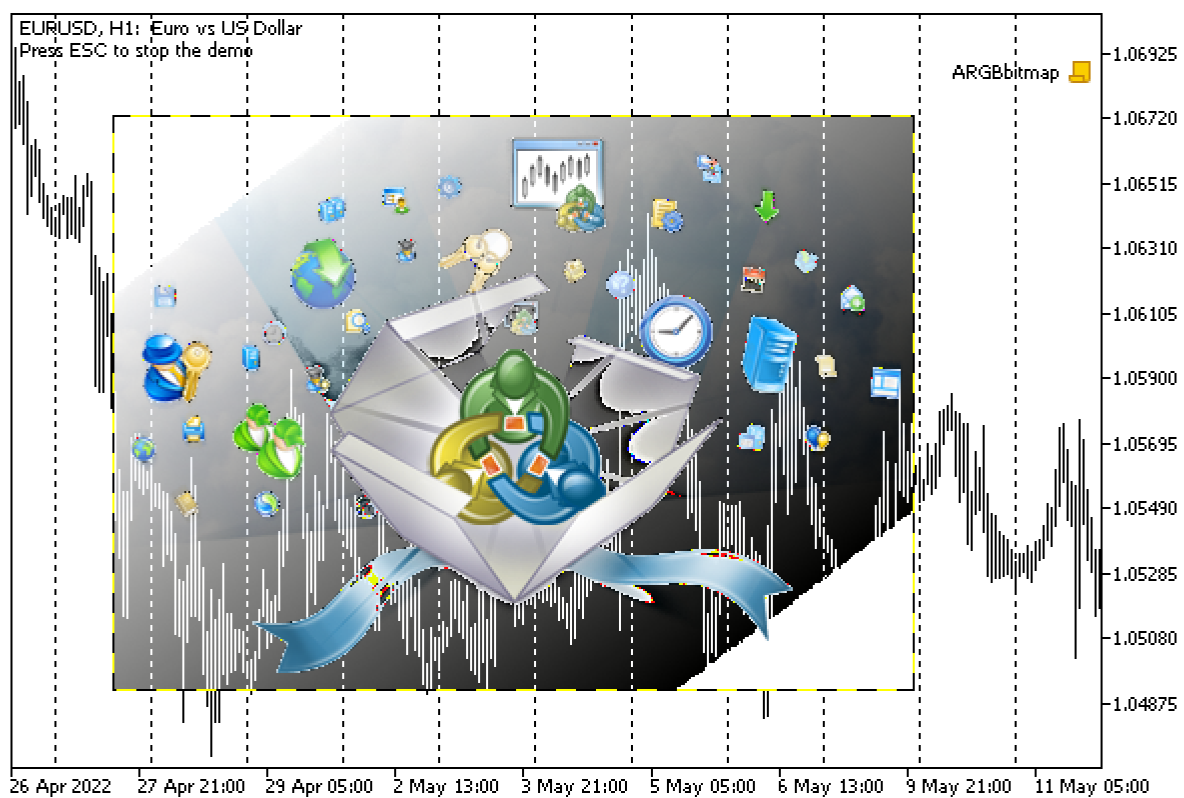
А пока проверим, как работает скрипт *ARGBbitmap.mq5*.

Вышеупомянутая картинка "argb.bmp" содержит информацию о прозрачности: верхний левый угол имеет совершенно прозрачный фон, а по диагонали по направлению к правому нижнем углу прозрачность постепенно исчезает.

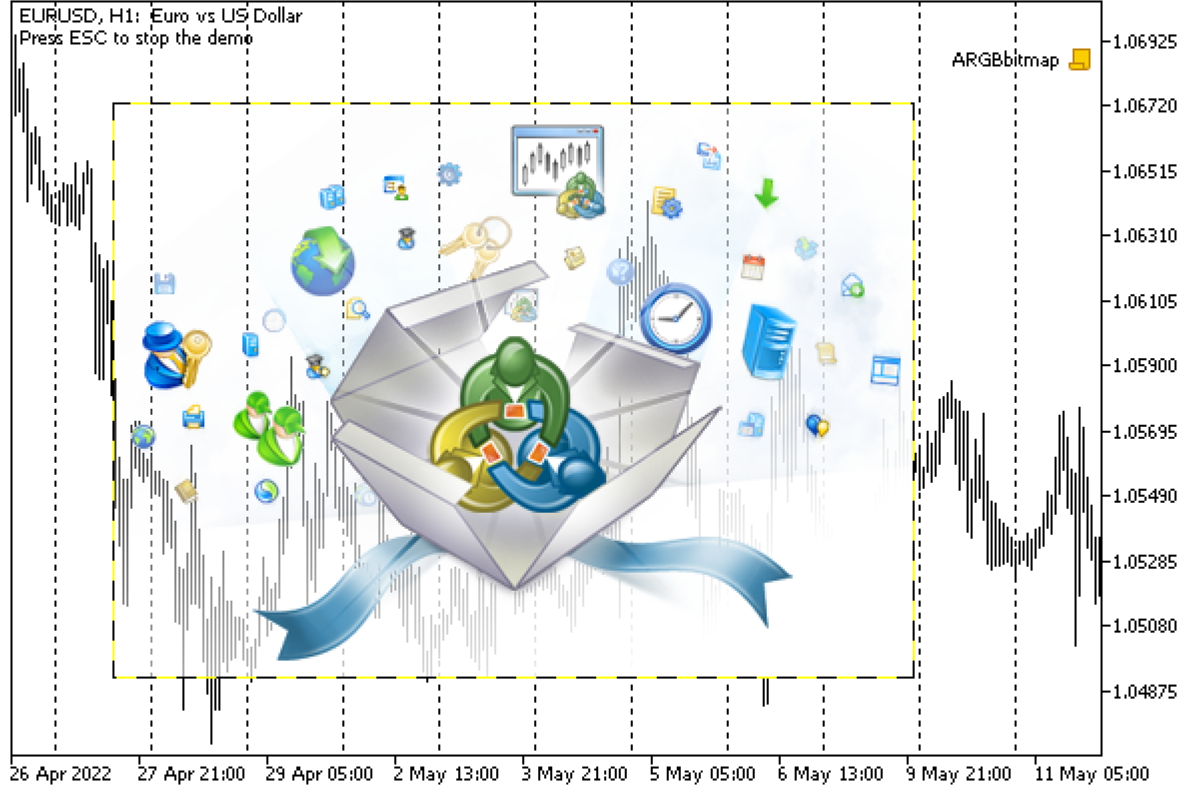
На следующих изображениях показаны результаты запуска скрипта в трех разных режимах.



Вывод изображения в цветовом формате COLOR_FORMAT_XRGB_NOALPHA



Вывод изображения в цветовом формате COLOR_FORMAT_ARGB_RAW



Вывод изображения в цветовом формате COLOR_FORMAT_ARGB_NORMALIZE

7.1.6 Удаление динамических ресурсов: ResourceFree

Функция *ResourceFree* удаляет ранее созданный динамический ресурс и освобождает занятую им память. Если не вызвать функцию *ResourceFree*, динамический ресурс останется в памяти вплоть до конца текущего сеанса работы терминала. Это можно использовать как удобную возможность хранения данных, но при штатной работе с изображениями рекомендуется освобождать их по факту исчезновения необходимости в них.

Графические объекты, привязанные к удаляемому ресурсу, будут отображаться правильно и после его удаления. Но вновь создаваемые графические объекты (OBJ_BITMAP и OBJ_BITMAP_LABEL) уже не смогут использовать удалённый ресурс.

bool ResourceFree(const string resource)

Имя ресурса задается в параметре *resource* и должно начинаться с "::".

Функция возвращает признак успеха (*true*) или ошибки (*false*).

Функция удаляет только динамические ресурсы, созданные данной MQL-программой, но не "чужие".

В предыдущем разделе мы видели пример скрипта *ARGBbitmap.mq5*, который вызывал *ResourceFree* по завершению своей работы.

7.1.7 Чтение и модификация данных ресурса: ResourceReadImage

Функция *ResourceReadImage* позволяет прочитать данные ресурса, созданного функцией *ResourceCreate* или встроенного в исполняемый файл во время компиляции согласно директиве *#resource*. Несмотря на суффикс "Image" в названии, функция работает с любыми массивами данных, в том числе пользовательского назначения (см. пример *Reservoir.mq5* далее).

bool ResourceReadImage(const string resource, uint &data[], uint &width, uint &height)

В параметре *resource* указывается имя ресурса. Для доступа к собственным ресурсам достаточно краткого вида "::resource_name". Для чтения ресурса из другого скомпилированного файла, необходимо имя в полном виде с указанием пути согласно правилам разрешения путей, описанных в разделе про ресурсы. В частности, путь, начинающийся с обратной косой черты, означает путь от корневой папки MQL5 (так "\\path\\filename.ex5::resource_name" ищется в файле */MQL5/path/filename.ex5* под именем "resource_name"), а путь без этого начального символа — относительно папки, в которой размещена выполняющаяся программа.

В приемный массив *data* будет записана внутренняя информация ресурса, а в параметры *width* и *height* — соответственно, ширина и высота, то есть размер массива (*width*height*) в опосредованном виде. Раздельно *width* и *height* имеют значение только в том случае, если в ресурсе хранится картинка. Массив должен быть динамическим или фиксированного, но достаточного размера — в противном случае получим ошибку SMALL_ARRAY (5052).

Если на основании массива *data* в дальнейшем необходимо создать графический ресурс, то в исходном ресурсе следует использовать формат цвета COLOR_FORMAT_ARGB_NORMALIZE или COLOR_FORMAT_XRGB_NOALPHA. Если массив *data* содержит произвольные прикладные данные, используйте COLOR_FORMAT_XRGB_NOALPHA.

В качестве первого примера рассмотрим скрипт *ResourceReadImage.mq5*. Он демонстрирует сразу несколько аспектов работы с графическими ресурсами:

- создание ресурса-картинки из внешнего файла;
- чтение и модификацию данных этой картинки в другом динамически создаваемом ресурсе;
- сохранность созданных ресурсов в памяти терминала между запусками скрипта;
- использование ресурсов в объектах на графике;
- удаление объекта и ресурсов.

Под модификацией изображения в данном конкретном случае понимается инвертирование всех цветов (как наиболее наглядное).

Все перечисленные приемы работы выполняются в три этапа: каждый этап выполняется за один запуск скрипта. Текущий этап скрипт определяет за счета анализа имеющихся ресурсов и объекта:

1. При отсутствии требуемых графических ресурсов скрипт создаст их (одно оригинальное изображение и одно инвертированное).
2. При наличии ресурсов, но отсутствии графического объекта скрипт создаст объект с двумя изображениями с первого шага для состояний включено/выключено (их можно будет переключать по клику мыши).
3. При наличии объекта скрипт удалит объект и ресурсы.

Главная функция скрипта начинается с определения имен для ресурсов и объекта на графике.

```
void OnStart()
{
   const static string resource = "::Images\\pseudo.bmp";
   const static string inverted = resource + "_inv";
   const static string object = "object";
   ...
```

Обратите внимание, что мы выбрали имя для исходного ресурса, которое напоминает размещение *bmp*-файла в стандартной папке *Images*, однако такого файла нет. Это подчеркивает виртуальную сущность ресурсов и позволяет делать подмены согласно техническим требованиям или в целях затруднения реверсинжиниринга ваших программ.

Следующий вызов *ResourceReadImage* используется для того, чтобы проверить, существует ли уже ресурс. В начальном состоянии (при первом запуске) мы получим отрицательный результат (*false*) и начнем первый этап: создаем исходный ресурс из файла "\\Images\\dollar.bmp", а затем инвертируем его в новом ресурсе с суффиксом "_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")); // попробуйте "argb.bmp
      ResourceCreateInverted(resource, inverted);
   }
   ...
```

Исходный код вспомогательной функции *ResourceCreateInverted* будет представлен ниже.

Если ресурс нашелся (второй запуск), скрипт проверяет наличие объекта и при необходимости создает его, включая установку свойств с ресурсами-картинками в функции *ShowBitmap* (см. ниже).

```
      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);
         }
         ...
```

Если же и ресурсы, и объект уже есть на графике, значит мы на заключительном этапе и должны удалить все ресурсы.

```
         else
         {
            Print("Cleanup state: Removing object and resources");
            PRTF(ObjectDelete(0, object));
            PRTF(ResourceFree(resource));
            PRTF(ResourceFree(inverted));
         }
      }
   }
```

Функция *ResourceCreateInverted* использует вызов *ResourceReadImage* для получения массива пикселей, а затем инвертирует в них цвет с помощью оператора '^' (XOR) и операнда со всеми единичными битами в компонентах цвета.

```
   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));
   }
```

Новый массив *data* передается в *ResourceCreate* для создания второго изображения.

Функция *ShowBitmap* привычным образом создает графический объект (в правом нижнем углу графика) и устанавливает его свойства для включенного и выключенного состояния в, соответственно, оригинальное и инвертированное изображение.

```
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);
}
```

Поскольку только что созданный объект по умолчанию находится в выключенном состоянии, мы сначала увидим инвертированное изображение и сможем его переключить в оригинальное по щелчку мыши. Но напомним, что наш скрипт выполняет действия по шагам, а потому прежде чем изображение появится на графике, скрипт нужно запустить дважды. На всех этапах в журнал выводится текущее состояние и выполняемые действия (вместе с признаком успеха или ошибки).

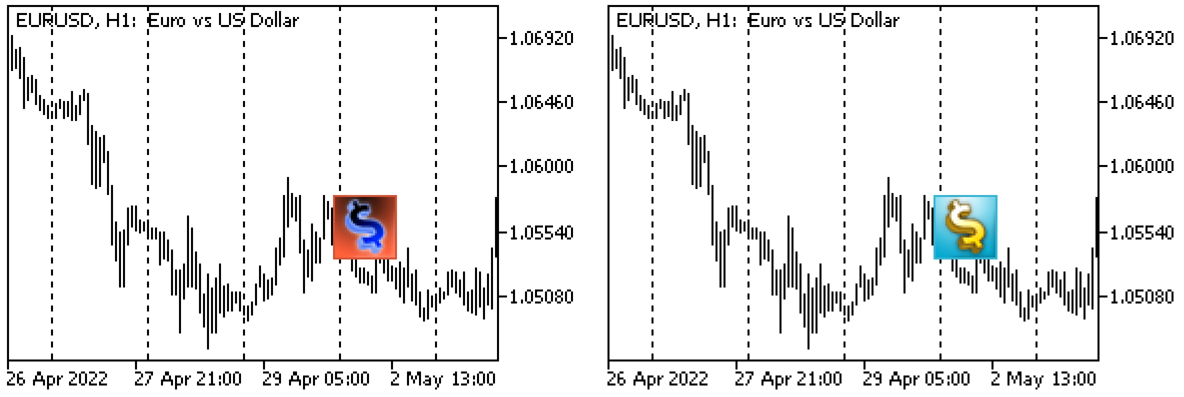
После первого запуска в журнале появятся такие записи:

```
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
```

Они говорят, что ресурсы не были обнаружены и потому скрипт их создал. После второго запуска в журнале будет сказано об обнаружении ресурсов (которые остались в памяти с предыдущего запуска скрипта), но объекта еще нет, и скрипт создаст его на основе ресурсов.

```
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
```

Мы увидим на графике объект и изображение. Переключение состояний доступно по клику мыши (события об изменении состояния мы здесь не обрабатываем).



Инвертированное и исходное изображения в объекте на графике

Наконец, в ходе третьего запуска скрипт обнаружит объект и удалит все свои наработки.

```
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
```

Далее можно повторять цикл.

Вторым примером раздела рассмотрим применение ресурсов для хранения произвольных прикладных данных — своего рода буфер обмена внутри терминала (таких буферов по идее может быть любое количество, т.к. каждый из них — это отдельный именованный ресурс). В силу универсальности задачи создадим класс *Reservoir* с основным функционалом (в файле *Reservoir.mqh*), а уже на его основе напишем демонстрационный скрипт (*Reservoir.mq5*).

Прежде чем углубляться непосредственно в Reservoir, представим вспомогательное объединение *ByteOverlay*, которое нам не раз потребуется. Объединение позволит конвертировать любой простой встроенный тип (включая простые структуры) в массив байтов и обратно. Напомним, что "простыми" мы называем все встроенные числовые типы, дату и время, перечисления, цвет, логические флаги. А вот объекты и динамические массивы уже не являются простыми и нашим новым хранилищем не будут поддерживаться (в силу технических ограничений платформы). Строки тоже не считаются простыми, но для них мы сделаем некое исключение — обработаем их особым образом.

```
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));
   }
};
```

Как мы знаем, ресурсы строятся на базе массивов типа *uint*, поэтому опишем такой массив (*storage*) в классе *Reservoir*. Туда мы будем складывать все данные, подлежащие последующей записи в ресурс. Текущая позиция в массиве, куда пишутся или откуда считываются данные, хранится в поле *offset*.

```
class Reservoir
{
   uint storage[];
   int offset;
public:
   Reservoir(): offset(0) { }
   ...
```

Положить массив данных произвольного типа в *storage* можно с помощью шаблонного метода *packArray*. В первой его половине мы преобразуем переданный массив в массив байтов с помощью *ByteOverlay*.

```
   template<typename T>
   int packArray(const T &data[])
   {
      const int bytesize = ArraySize(data) * sizeof(T); // TODO: проверить переполнен
      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));
      }
      ...
```

Во второй половине преобразуем массив байтов в последовательность значений *uint*, которые записываем в *storage* по смещению *offset*. Количество требуемых элементов *uint* определяется с учетом того, есть ли остаток от деления размера данных в байтах на размер *uint*: при необходимости добавляем один дополнительный элемент.

```
      const int size = bytesize / sizeof(uint) + (bool)(bytesize % sizeof(uint));
      ArrayResize(storage, offset + size + 1);
      storage[offset] = bytesize;       // размер данных пишем перед данными
      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;
   }
```

Перед самими данными мы пишем размер данных в байтах: это минимальный из возможных протоколов для проверки ошибок при восстановлении данных. В перспективе можно было бы записывать в *storage* также и *typename(T)* данных.

Метод возвращает текущую позицию в хранилище после записи.

На основе *packArray* легко реализовать метод для сохранения строк:

```
int packString(const string text)
{
   uchar data[];
   StringToCharArray(text, data, 0, -1, CP_UTF8);
   return packArray(data);
}
```

Есть также и возможность сохранить отдельное число:

```
template<typename T>
int packNumber(const T number)
{
   T array[1] = {number};
   return packArray(array);
}
```

Метод для восстановления массива произвольного типа T из хранилища типа *uint* "проигрывает" все операции в обратную сторону. При обнаружении нестыковок в читаемом типе и количестве данных с хранилищем метод возвращает 0 (признак ошибки). В штатном режиме возвращается текущая позиции в массиве *storage* (она всегда больше 0, если что-то успешно прочитано).

```
template<typename T>
int unpackArray(T &output[])
{
   if(offset >= ArraySize(storage)) return 0; // выход за границы массива
   const int bytesize = (int)storage[offset];
   if(bytesize % sizeof(T) != 0) return 0;     // неподходящий тип данных
   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;
}
```

Распаковка строк и чисел производится через вызов *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;
}
```

Простые вспомогательные методы позволяют узнать размер хранилища и текущую позицию в нем, а также очистить его.

```
int size() const
{
   return ArraySize(storage);
}

int cursor() const
{
   return offset;
}

void clear()
{
   ArrayFree(storage);
   offset = 0;
}
```

Теперь мы подходим к самому интересному: взаимодействию с ресурсами.

Имея заполненный массив *storage* с прикладными данными, легко "переместить" его в заданный ресурс.

```
bool submit(const string resource)
{
   return ResourceCreate(resource, storage, ArraySize(storage), 1,
      0, 0, 0, COLOR_FORMAT_XRGB_NOALPHA);
}
```

Также просто можно прочитать данные из ресурса во внутренний массив *storage*.

```
bool acquire(const string resource)
{
   uint width, height;
   if(ResourceReadImage(resource, storage, width, height))
   {
      return true;
   }
   return false;
}
```

Покажем в скрипте *Reservoir.mq5*, как этим пользоваться.

В первой половине *OnStart* опишем имя для ресурса-хранилища и объект класса *Reservoir*, а затем последовательно "упакуем" в этот объект строку, структуру *MqlTick* и число *double*. Структура "обернута" в массив из одного элемента, чтобы продемонстрировать в явном виде метод *packArray*, а кроме того нам затем потребуется сравнить восстановленные данные с исходными, а MQL5 не предоставляет оператор '==' для структур, в связи с чем удобнее будет пользоваться функцией *ArrayCompare*.

```
#include <MQL5Book/Reservoir.mqh>
#include <MQL5Book/PRTF.mqh>

void OnStart()
{
   const string resource = "::reservoir";

   Reservoir res1;
   string message = "message1";     // строка для записи в ресурс
   PRTF(res1.packString(message));

   MqlTick tick1[1];                // добавим простую структуру
   SymbolInfoTick(_Symbol, tick1[0]);
   PRTF(res1.packArray(tick1));
   PRTF(res1.packNumber(DBL_MAX));  // вещественное число
   ...
```

Когда все необходимые данные "упакованы" в объект, запишем их в ресурс и очистим объект.

```
   res1.submit(resource);           // создаем ресурс с данными хранилища
   res1.clear();                    // очистка объекта, но не ресурса
```

Во второй половине *OnStart* выполним обратные операции чтения данных из ресурса.

```
   string reply;                    // новая переменная под сообщение
   MqlTick tick2[1];                // новая структура для тика
   double result;                   // новая переменная под число

   PRTF(res1.acquire(resource));    // подключаем объект к заданному ресурсу
   PRTF(res1.unpackString(reply));  // читаем строку
   PRTF(res1.unpackArray(tick2));   // читаем простую структуру
   PRTF(res1.unpackNumber(result)); // читаем число

   // выводим и сравниваем данные поэлементно
   PRTF(reply);
   PRTF(ArrayCompare(tick1, tick2));
   ArrayPrint(tick2);
   PRTF(result == DBL_MAX);

   // убеждаемся, что хранилище прочитано до конца
   PRTF(res1.size());
   PRTF(res1.cursor());
   ...
```

В конце очищаем ресурс, поскольку это тест. В практических задачах MQL-программа, скорее всего, оставит созданный ресурс в памяти, чтобы его могли прочитать другие программы. Напомним, что в иерархии имен ресурсы объявлены вложенными в ту программу, которая их создала. Поэтому для доступа из других программ нужно задать имя ресурса вместе с названием программы и опционально путем (если программа-создатель и программа-читатель находятся в разных папках). Например, для чтения только что созданного ресурса извне подойдет полный путь "\\Scripts\\MQL5Book\\p7\\Reservoir.ex5::reservoir".

```
   PrintFormat("Cleaning up local storage '%s'", resource);
   ResourceFree(resource);
}
```

Поскольку все основные вызовы методов контролируются макросом PRTF, при запуске скрипта мы увидим в журнале подробный "отчет" о ходе действий.

```
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
[0] 2022.05.19 23:09:32 1.05867 1.05873 0.0000        0 1653001772050       6        0
result==DBL_MAX=true / ok
res1.size()=23 / ok
res1.cursor()=23 / ok
Cleaning up local storage '::reservoir'
```

Данные были успешно скопированы в ресурс, а затем восстановлены оттуда.

Программы могут использовать этот подход для обмена объемными данными, которые не помещаются в пользовательские сообщения (события CHARTEVENT_CUSTOM+) — достаточно отсылать в строковом параметре *sparam* имя ресурса для чтения. Для обратной передачи данных следует создать с ними свой собственный ресурс и отправить ответное сообщение.

7.1.8 Сохранение изображений в файл: ResourceSave

MQL5 API позволяет записать ресурс в файл формата BMP с помощью функции *ResourceSave*. В данный момент среда поддерживает только ресурсы-изображения.

bool ResourceSave(const string resource, const string filename)

В параметрах *resource* и *filename* указываются, соответственно, имя ресурса и файла. Имя ресурса должно начинаться с "::". Имя файла может содержать путь относительно папки *MQL5/Files*. При необходимости функция создаст все промежуточные подкаталоги. Если указанный файл существует, он будет перезаписан.

Функция возвращает *true* в случае успеха.

Для проверки работы данной функции желательно создать оригинальное изображение. И у нас есть для этого подходящий "материал".

В рамках изучения ООП, в главе Классы и интерфейсы, мы начали серию примеров про графические фигуры: от самой первой версии *Shapes1.mq5* в разделе про Определение класса и до последней *Shapes6.mq5* в разделе про Вложенные типы. Само рисование тогда не было нам доступно, вплоть до главы про графические объекты, где мы смогли реализовать визуализацию в скрипте *ObjectShapesDraw.mq5*. Теперь, после изучения графических ресурсов, настало время очередного "апгрейда".

В новой версии скрипта *ResourceShapesDraw.mq5* фигуры будут по-настоящему рисоваться. Чтобы было проще анализировать изменения по сравнению с прежней версией, мы оставим тот же набор фигур: прямоугольник, квадрат, овал, круг и треугольник. Это сделано для примера, а не потому что в рисовании нас что-то ограничивает: наоборот — существует потенциал по расширению набора фигур, визуальных эффектов и нанесению надписей. Некоторые из этих возможностей мы рассмотрим в этом примере, некоторые — в следующих, но продемонстрировать все многообразие применений в рамках книги просто невозможно.

После того как фигуры будут сгенерированы и отрисованы, мы сохраним получившийся ресурс в файл.

Напомним, что основой иерархии классов фигур является *Shape*, в котором был метод *draw*.

```
class Shape
{
public:
   ...
   virtual void draw() = 0;
   ...
}
```

В производных классах он был реализован на основе графических объектов, с вызовами *ObjectCreate* и последующей настройкой объектов *ObjectSet*-функциями. Общим холстом такого рисунка был непосредственно график.

Сейчас нам потребуется закрашивать пиксели в некотором общем ресурсе в соответствии с формой конкретной фигуры. Общий ресурс и методы модификации пикселей в нем желательно выделить в отдельный класс или лучше интерфейс.

Абстрактная сущность позволит не делать увязки со способом создания и настройкой ресурса. В частности, наша последующая реализация поместит ресурс в объект OBJ_BITMAP_LABEL (как мы уже делали в этой главе), а для кого-то может быть достаточно генерировать изображения в памяти и сохранять на диск, не выводя на график (так многие трейдеры любят периодически фиксировать состояния чартов).

Назовем интерфейс *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)
};
```

Здесь представлены лишь три самых базовых метода для рисования. Нам этого хватит.

Метод *point* хотя и является публичным (что дает возможность поставить отдельную точку), но в некотором смысле — низкоуровневый, так как через него будут реализованы все остальные. Именно поэтому в нем координаты сделаны вещественными, а содержимое пикселя — готовым значением типа *uint*. Это позволит при необходимости подключить различные алгоритмы сглаживания, чтобы фигуры не выглядели ступенчатыми из-за пикселизации. Здесь мы не будем касаться этого вопроса.

С учетом интерфейса метод *Shape::draw* превращается в такой:

```
virtual void draw(Drawing *drawing) = 0;
```

Тогда в классе *Rectangle* очень просто поручить отрисовку прямоугольника новому интерфейсу.

```
class Rectangle : public Shape
{
protected:
   int dx, dy; // размер (ширина, высота)
   ...
public:
   void draw(Drawing *drawing) override
   {
      // x, y - точка привязки (центр) в Shape
      drawing.rect(x - dx / 2, y - dy / 2, x + dx / 2, y + dy / 2, backgroundColor);
   }
};
```

Для рисования эллипса придется потрудиться побольше.

```
class Ellipse : public Shape
{
protected:
   int dx, dy; // большой и малый радиусы
   ...
public:
   void draw(Drawing *drawing) override
   {
      // (x, y) - центр
      const int hh = dy * dy;
      const int ww = dx * dx;
      const int hhww = hh * ww;
      int x0 = dx;
      int step = 0;

      // главный горизонтальный диаметр
      drawing.line(x - dx, y, x + dx, y, backgroundColor);

      // горизонтальные линии в верхней и нижней половине, симметрично уменьшающиеся
      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);
      }
   }
};
```

Наконец, для треугольника рисование выполнено следующим образом.

```
class Triangle: public Shape
{
protected:
   int dx;  // один размер, т.к. треугольники равносторонние
   ...
public:
   virtual void draw(Drawing *drawing) override
   {
      // (x, y) - центр
      // 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)
      // высота по теореме Пифагора: 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);

      // главная вертикальная линия (высота треугольника)
      const int base = y + (int)(R - H);
      drawing.line(x, y + (int)R, x, base, backgroundColor);

      // вертикальные линии влево и вправо меньшего размера, симметричные
      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);
      }
   }
};
```

Обратимся теперь к классу *MyDrawing* — наследнику интерфейса *Drawing*. Именно *MyDrawing* должен, руководствуясь вызовами интерфейсных методов в фигурах, обеспечить отображение в растровой картинке некоего ресурса. Поэтому внутри класса, прежде всего, описаны переменные для названий графического объекта (*object*) и ресурса (*sheet*), а также массив *data* типа *uint* для хранения картинки. Кроме того мы перенесли сюда массив фигур *shapes*, который ранее просто был объявлен в обработчике *OnStart*. Поскольку за рисование всех фигур отвечает *MyDrawing*, то и управлять их набором лучше здесь.

```
class MyDrawing: public Drawing
{
   const string object;      // объект с bitmap-ом
   const string sheet;       // ресурс
   uint data[];              // пиксели
   int width, height;        // размеры
   AutoPtr<Shape> shapes[]; // фигуры
   const uint bg;            // цвет фона
   ...
```

В конструкторе создаем графический объект на размер всего графика и выделяем память под массив *data*. Холст заполняется нулями (означает "черную прозрачность") или другим

значением, переданным в параметре *background*, после чего на его основе создается ресурс. По умолчанию имя ресурса начинается с буквы 'D' и включает идентификатор текущего графика, но можно задать другое.

```
  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);
     }
```

Вызывающий код может узнать имя ресурса с помощью метода *resource*.

```
     string resource() const
     {
        return sheet;
     }
```

В деструкторе ресурс и объект удаляются.

```
     ~MyDrawing()
     {
        ResourceFree(sheet);
        ObjectDelete(0, object);
     }
```

Для заполнения массива фигур предусмотрен метод *push*.

```
     Shape *push(Shape *shape)
     {
        shapes[EXPAND(shapes)] = shape;
        return shape;
     }
```

Для их рисования определен метод *draw*, который просто в цикле вызывает метод *draw* каждой фигуры, а затем обновляет ресурс и график.

```
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();
}
```

Осталось рассмотреть самые главные методы — методы интерфейса *Drawing*, которые собственно и обеспечивают рисование.

Начнем с метода *point*, который пока приведем в упрощенном виде (усовершенствованиями займемся позднее).

```
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;
   }
}
```

На основе *point* легко сделать рисование линии. Когда координаты начальной и конечной точки совпадают по одному из измерений, мы делегируем рисование методу *rect*: ведь прямая линия — это вырожденный случай прямоугольника единичной толщины.

```
virtual void line(const int x1, const int y1, const int x2, const int y2, const cc
{
   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) // шаг по оси с наибольшим расстоянием, по 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 // или шаг по y
      {
         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);
         }
      }
   }
}
```

А вот и сам метод *rect*.

```
virtual void rect(const int x1, const int y1, const int x2, const int y2, const cc
{
   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);
      }
   }
}
```

Осталось модифицировать обработчик *OnStart*, и скрипт будет готов.

В начале мы настраиваем график (скрываем все элементы). В принципе, это не обязательно: оставлено для сопоставления со скриптом-прототипом.

```
void OnStart()
{
   ChartSetInteger(0, CHART_SHOW, false);
   ...
```

Далее описываем объект класса *MyDrawing*, генерируем предопределенное количество случайных фигур (здесь всё осталось без изменений, включая генератор *addRandomShape* и макрос FIGURES, равный 21-у), рисуем их в ресурсе и выводим в объекте на графике.

```
   MyDrawing raster;
   
   for(int i = 0; i < FIGURES; ++i)
   {
      raster.push(addRandomShape());
   }
   
   raster.draw(); // выводим начальное состояние
   ...
```

Напомним, что в примере *ObjectShapesDraw.mq5* мы затем начинали бесконечный цикл, в котором хаотично двигали фигуры. Повторим этот прием и здесь. Но для него потребуется дополнить класс *MyDrawing* — он должен этим заняться, раз массив фигур хранится у него внутри. Напишем простой метод *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);
      }
   }
   ...
};
```

Тогда в *OnStart* мы можем задействовать новый метод в цикле, пока пользователь не остановит анимацию.

```
void OnStart()
{
   ...
   while(!IsStopped())
   {
      Sleep(250);
      raster.shake();
      raster.draw();
   }
   ...
}
```

На этом функционал прежнего примера фактически повторен. Но нам нужно добавить сохранение картинки в файл. Поэтому добавим входной параметр *SaveImage*.

```
input bool SaveImage = false;
```

Когда он будет установлен в true, проверим функцию *ResourceSave* в деле.

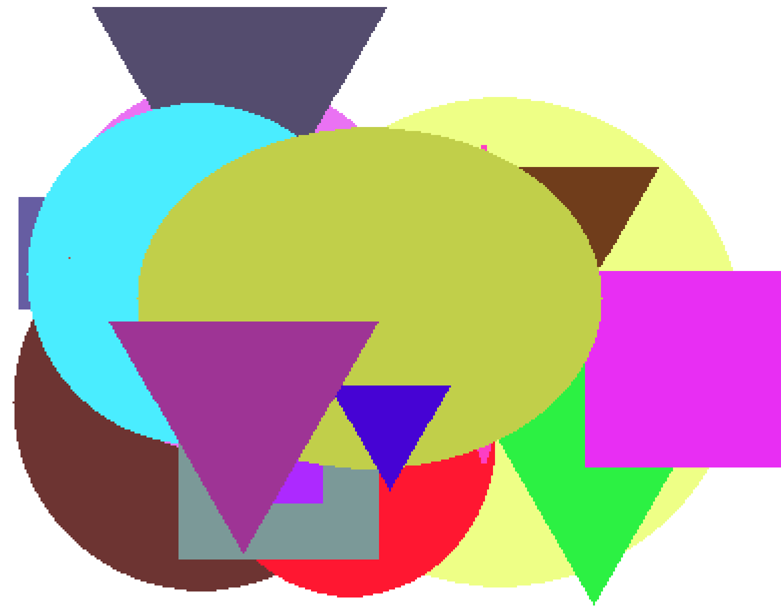
```
void OnStart()
{
   ...
   if(SaveImage)
   {
      const string filename = "temp.bmp";
      if(ResourceSave(raster.resource(), filename))
      {
         Print("Bitmap image saved: ", filename);
      }
      else
      {
         Print("Can't save image ", filename, ", ", E2S(_LastError));
      }
   }
}
```

Кроме того, раз уж речь зашла о входных переменных, позволим пользователю выбирать фон и передадим полученное значение в конструктор *MyDrawing*.

```
input color BackgroundColor = clrNONE;
void OnStart()
{
   ...
   MyDrawing raster(BackgroundColor != clrNONE ? ColorToARGB(BackgroundColor) : 0);
   ...
}
```

Итак, все готово для первого испытания.

Если запустить скрипт *ResourceShapesDraw.mq5*, на графике сформируется изображение вроде следующего.



Растровое изображение ресурса с набором случайных фигур

При сравнении данного изображения с тем, что мы видели в примере *ObjectShapesDraw.mq5*, обнаруживается, что наш новый способ визуализации несколько отличается от того, каким терминал выводит объекты. Хотя формы фигур и их цвет в отдельности не вызывают вопросов, места наложения фигур обозначаются по-разному.

Наш скрипт закрашивает фигуры указанным цветом, накладывая их друг на друга в порядке следования в массиве. Более поздние фигуры перекрывают более ранние. Терминал же применяет в местах перекрытия некое смешение цветов (инверсию).

Оба способа имеют право на существование, здесь нет ошибок. Однако интересно, нельзя ли добиться при рисовании аналогичного эффекта?

Благодаря тому, что мы полностью управляем процессом рисования, к нему можно применить любые эффекты, не только такой, как в терминале.

Давайте реализуем, в дополнение к исходному, простому способу рисования, еще несколько режимов. Все они сведены в перечисление COLOR_EFFECT.

```
enum COLOR_EFFECT
{
   PLAIN,         // простое рисование с перекрытием (по умолчанию)
   COMPLEMENT,    // рисование дополнительным цветом (как в терминале)
   BLENDING_XOR,  // смешивание цветов с помощью XOR '^'
   DIMMING_SUM,   // "затемнение" цветов с помощью '+'
   LIGHTEN_OR,    // "осветление" цветов с помощью '|'
};
```

Добавим входную переменную для выбора режима.

```
input COLOR_EFFECT ColorEffect = PLAIN;
```

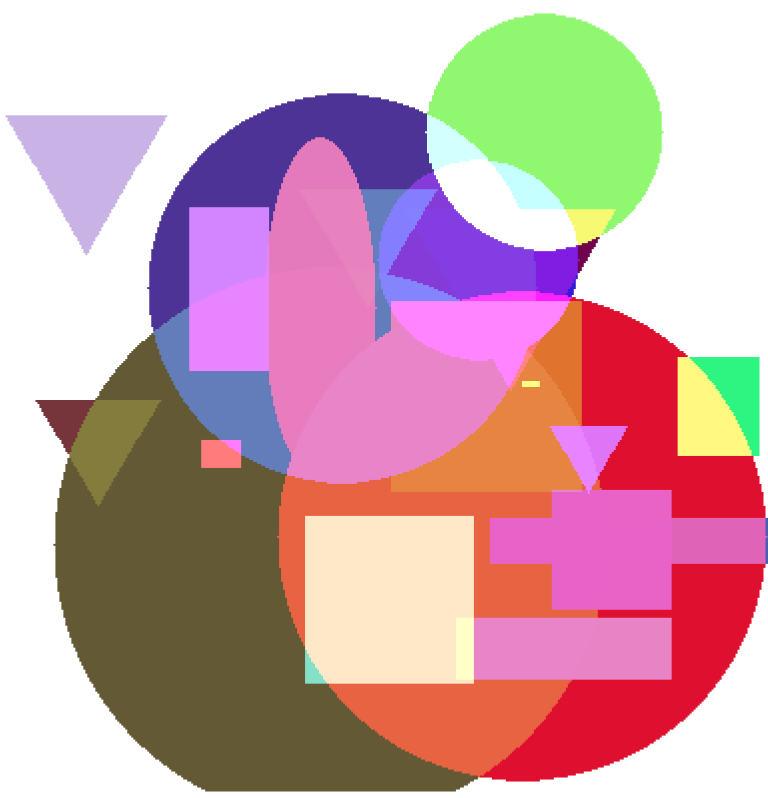
Поддержим режимы в классе *MyDrawing*. Для начала опишем соответствующие поле и метод.

```
class MyDrawing: public Drawing
{
   ...
   COLOR_EFFECT xormode;
   ...
public:
   void setColorEffect(const COLOR_EFFECT x)
   {
      xormode = x;
   }
   ...
```

Затем усовершенствуем метод *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])); // смешивание с дополнительным
            break;
         case BLENDING_XOR:
            data[index] = (pixel & 0xFF000000) | (pixel ^ data[index]); // прямое сме
            break;
         case DIMMING_SUM:
            data[index] =  (pixel + data[index]); // "затемнение" (SUM)
            break;
         case LIGHTEN_OR:
            data[index] =  (pixel & 0xFF000000) | (pixel | data[index]); // "осветлен
            break;
         case PLAIN:
         default:
            data[index] = pixel;
         }
      }
   }
```

Вы можете попробовать запускать скрипт в различных режимах и сравнить результаты. Не забывайте про возможность настройки фона. Вот, например, как выглядит осветление.



Изображение фигур с осветляющим смешиванием цветов

Чтобы наглядно увидеть разницу в эффектах, вы можете отключить рандомизацию цветов и движение фигур. Стандартный способ перекрытия объектов соответствует константе COMPLEMENT.

В качестве финального эксперимента, включите опцию *SaveImage*. В обработчике *OnStart* при генерации имени файла с изображением мы теперь используем название текущего режима. Мы должны получить в файле копию изображения на графике.

```
...
if(SaveImage)
{
   const string filename = EnumToString(ColorEffect) + ".bmp";
   if(ResourceSave(raster.resource(), filename))
   ...
```

Для более изощренных графических построений нашего интерфейса *Drawing* будет, скорее всего, недостаточно. Поэтому вы можете использовать готовые классы для рисования, поставляемые с MetaTrader 5 или доступные в базе кодов на mql5.com. В частности, загляните в файл *MQL5/Include/Canvas/Canvas.mqh*.

7.1.9 Шрифты и вывод текста в графические ресурсы

Помимо отрисовки отдельных пикселей в массиве графического ресурса нам доступны встроенные функции для вывода текста. Функции позволяют изменять текущий шрифт и его характеристики (*TextSetFont*), получать размеры прямоугольника, в который может быть вписана заданная строка (*TextGetSize*), а также непосредственно вставлять надпись в генерируемое изображение (*TextOut*).

bool TextSetFont(const string name, int size, uint flags, int orientation = 0)

Функция устанавливает шрифт и его характеристики для последующего рисования текста в буфере изображения с помощью функции *TextOut* (см. далее). Параметр *name* может содержать имя встроенного шрифта Windows или файл ttf-шрифта (TrueType Font), подключенный директивой ресурс (если имя начинается с "::").

Размер (*size*) может задаваться в пунктах (типографская единица измерения) или пикселях (точках экрана). Положительные значения означают, что единицей измерения является пиксель, отрицательные — измеряются в десятых долях пункта. Высота в пикселях будет по-разному выглядеть у пользователей в зависимости от технических возможностей и настроек их мониторов. Высота в пунктах будет примерно ("на глаз") одинаковой у всех.

> Типографский пункт — это физическая мера длины, равная традиционно 1/72-ой части дюйма. Следовательно, 1 пункт равен 0.352778 миллиметра. Пиксель на экране является виртуальной мерой длины. Его физический размер зависит от аппаратной разрешающей способности экрана. Например, при плотности экрана 96 DPI (точек на дюйм) 1 пиксель займет 0.264583 миллиметра, или 0.75 пункта. Однако большинство современных дисплеев имеют гораздо большие значения DPI и, соответственно, более мелкие пиксели. Из-за этого в операционных системах, в том числе и Windows, давно имеются настройки для увеличения видимого масштаба интерфейсных элементов. Таким образом, при указании размера в пунктах (отрицательные значения) размер текста в пикселях будет зависеть от дисплея и настроек масштаба в операционной системе (например, "стандартный" 100%, "средний" 125% или "крупный" 150%).
>
> Увеличение масштаба приводит к тому, что размер отображаемых пикселей искусственно увеличивается системой. Это эквивалентно уменьшению размера экрана в пикселях и для получения того же физического размера система применяет так называемый эффективный DPI. Если масштабирование включено, то именно эффективный DPI сообщается программам, в том числе терминалу и затем MQL-программам. Узнать DPI экрана при необходимости можно из свойства TERMINAL_SCREEN_DPI (см. Характеристики экрана). Но на самом деле, задавая размер шрифта в пунктах, мы избавлены от необходимости пересчитывать его размер в зависимости от DPI, так как система сделает это за нас.

По умолчанию используется шрифт Arial и размер -120 (12 pt). Элементы управления, в частности, встроенные в объекты на графиках также оперируют размерами шрифтов в пунктах. Например, если в MQL-программе требуется нарисовать текст такого же размера, как текст в объекте OBJ_LABEL, в котором установлен размер 10 пунктов, следует использовать параметр *size*, равный -100.

В параметре *flags* задается комбинация флагов, описывающих стиль шрифт. Комбинация составляется как битовая маска, с помощью оператора побитового ИЛИ ('|'). Флаги делятся на две группы: флаги стиля и флаги жирности.

В следующей таблице приведены флаги стиля. Их можно смешивать.

Флаг	Описание
FONT_ITALIC	Курсив
FONT_UNDERLINE	Подчёркивание
FONT_STRIKEOUT	Перечёркивание

Флаги жирности имеют соответствующие им относительные весовые коэффициенты (приведены для возможности сравнить ожидаемые эффекты).

Флаг	Описание
FW_DONTCARE	0 (будет применено системное значение по умолчанию)
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

В комбинации флагов используйте только одно из этих значений.

Параметр *orientation* задает угол наклона текста по отношению к горизонтали, в десятых долях градуса. Например, 0 означает обычный вывод текста, а orientation = 450 приведет к наклону в 45 градусов (против часовой стрелки).

Обратите внимание, что установки, сделанные одним вызовом *TextSetFont*, будут влиять на все последующие вызовы *TextOut*, пока (или если) не будут изменены.

Функция возвращает *true* в случае успеха или *false* при возникновении проблем (например, если не найден шрифт).

Пример использования этой, а также двух остальных функций мы рассмотрим после описания всех их.

bool TextGetSize(const string text, uint &width, uint &height)

Функция возвращает ширину и высоту строки при текущих настройках шрифта (это может быть шрифт по умолчанию или заданный в предыдущем вызове *TextSetFont*).

В параметре *text* передается строка, для которой требуется получить длину и ширину в пикселях. Значения размеров записываются функцией по ссылкам в параметрах *width* и *height*.

Следует отметить, что поворот (наклон) выводимого текста, заданный параметром *orientation* при вызове *TextSetFont*, никак не влияет на оценку размеров. Иными словами, если текст предполагается повернуть на 45 градусов, то вычислить минимальный квадрат, в который текст может быть вписан, должна сама MQL-программа. Функция *TextGetSize* рассчитывает размер текста в стандартном (горизонтальном) положении.

bool TextOut(const string text, int x, int y, uint anchor, uint &data[], uint width, uint height, uint color, ENUM_COLOR_FORMAT color_format)

Функция рисует текст в графическом буфере по указанным координатам и с учетом цвета, формата и предыдущих настроек (шрифта, стиля и ориентации).

Текст передается в параметре *text* и должен представлять собой одну строку.

Координаты *x* и *y*, заданные в пикселях, определяют точку в графическом буфере, где выводится текст. Какое именно место генерируемой надписи окажется в точке (x,y), зависит от способа привязки в параметре *anchor* (см. далее).

Буфер представлен массивом *data*, и хотя массив одномерный, в нем хранится двумерный "холст" размерами *width* на *height* точек. Этот массив может быть получен из функции *ResourceReadImage* или распределен MQL-программой. После завершения всех операций редактирования, включая вывод текста, следует создать новый ресурс на основе этого буфера или применить его к уже существующему ресурсу — в обоих случаях подразумевается вызов *ResourceCreate*.

Цвет текста и способ обработки цвета задаются параметрами *color* и *color_format* (см. ENUM_COLOR_FORMAT). Обратите внимание, что для цвета используется тип *uint*, то есть для передачи цвета (*color*) следует конвертировать его с помощью *ColorToARGB*.

Способ привязки, задаваемый параметром anchor, является комбинацией двух флагов расположения текста: по вертикали и по горизонтали.

Флаги расположения текста по горизонтали:

- TA_LEFT – привязка к левой стороне ограничивающего прямоугольника
- TA_CENTER – привязка к середине между левой и правой стороной прямоугольника
- TA_RIGHT – привязка к правой стороне ограничивающего прямоугольника

Флаги расположения текста по вертикали:

- TA_TOP – привязка к верхней стороне ограничивающего прямоугольника
- TA_VCENTER – привязка к середине между верхней и нижней стороной прямоугольника
- TA_BOTTOM – привязка к нижней стороне ограничивающего прямоугольника

Итого существует 9 допустимых сочетаний флагов для описания способа привязки.

Положение выводимого текста относительно точки привязки

Здесь в центре картинки утрированно большим размером нанесена точка в генерируемом изображении с координатами (x,y). В зависимости от флагов, текст появляется относительных этой точки в указанных позициях (содержимое текста соответствует примененному способу привязки).

Для простоты восприятия все надписи сделаны в стандартном горизонтальном положении, но учтите, что к любой из них мог также примениться угол (*orientation*), и тогда соответствующая надпись была бы повернута вокруг точки. На данном изображении повернута только надпись, отцентрированная по обоим измерениям.

Не следует путать данные флаги с выравниванием текста. Ограничивающий прямоугольник всегда подогнан под размеры текста, и его позиция относительно точки привязки, в некотором смысле, противоположна названиям флагов.

Рассмотрим несколько примеров с использованием трех функций.

Для начала проверим самые простые возможности — установку жирности и начертания шрифтов. Скрипт *ResourceText.mq5* позволяет выбрать во входных переменных название шрифта, его размер, а также цвета фона и текста. Надписи будут демонстрироваться на графике заданное количество секунд.

```
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)
```

Название каждой градации жирности будет выводиться в текст надписи, поэтому для упрощения процесса (за счет использования *EnumToString*) объявлено перечисление 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; // количество различных весов
```

Флаги начертания собраны в массиве *rendering* и из него выбираются случайные сочетания.

```
   const uint rendering[] =
   {
      FONT_ITALIC,
      FONT_UNDERLINE,
      FONT_STRIKEOUT
   };
   const int nr = sizeof(rendering) / sizeof(uint);
```

Для получения случайного числа в диапазоне имеется вспомогательная функция *Random*.

```
int Random(const int limit)
{
   return rand() % limit;
}
```

В главной функции скрипта мы выясняем размер графика и создаем объект OBJ_BITMAP_LABEL, перекрывающий все пространство.

```
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);

   // объект для ресурса с картинкой на все окно
   ObjectCreate(0, name, OBJ_BITMAP_LABEL, 0, 0, 0);
   ObjectSetInteger(0, name, OBJPROP_XSIZE, w);
   ObjectSetInteger(0, name, OBJPROP_YSIZE, h);
   ...
```

Далее мы выделяем память под буфер для картинки, закрашиваем его заданным фоновым цветом (или оставляем прозрачным, по умолчанию), создаем на основе буфера ресурс и привязываем его к объекту.

```
   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);
   ...
```

На всякий случай обратите внимание, что мы можем задавать свойство OBJPROP_BMPFILE без модификатора (0 или 1) в вызове *ObjectSetString*, если не предполагается переключать объект между двумя состояниями.

Все веса шрифтов перечислены в массиве *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);
```

В цикле по порядку устанавливаем очередную градацию жирности для каждой строки с помощью *TextSetFont*, предварительно выбирая случайный стиль. Описание шрифта, включающее его название и жирность рисуется в буфере с помощью *TextOut*.

```
const int step = h / (nw + 2);
int cursor = 0;    // координата Y текущей "строки текста"

for(int weight = 0; weight < nw; ++weight)
{
   // применяем случайный стиль
   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);

   // генерируем описание шрифта
   const string text = Font + EnumToString((ENUM_FONT_WEIGHTS)weights[weight]);

   // рисуем текст на отдельной "строке"
   cursor += step;
   TextOut(text, w / 2, cursor, TA_CENTER | TA_TOP, data, w, h,
      ColorToARGB(Color), COLOR_FORMAT_ARGB_RAW);
}
...
```

Наконец, мы обновляем ресурс и график.

```
ResourceCreate(name, data, w, h, 0, 0, w, COLOR_FORMAT_ARGB_RAW);
ChartRedraw();
...
```

Пользователь может остановить демонстрацию заранее.

```
    const uint timeout = GetTickCount() + Seconds * 1000;
    while(!IsStopped() && GetTickCount() < timeout)
    {
       Sleep(1000);
    }
```

В завершении скрипт удаляет ресурс и объект.

```
    ObjectDelete(0, name);
    ResourceFree("::" + name);
 }
```

Результат работы скрипта приведен на следующем изображении.

Рисование текста разной жирности и стилей

Во втором примере *ResourceFont.mq5* усложним задачу, подключив пользовательский шрифт в виде ресурса, а также задействуем поворот текста с шагом 90 градусов.

Файл шрифта находится рядом со скриптом.

```
#resource "a_LCDNova3DCmObl.ttf"
```

Сообщение можно поменять во входном параметре.

```
input string Message = "Hello world!";    // Message
```

На этот раз объект OBJ_BITMAP_LABEL не будет занимать все окно и потому центрируется по горизонтали и вертикали.

```
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);

   // объект для ресурса с картинкой
   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);
   ...
```

Буфер сначала выделяется минимального размера, просто чтобы выполнить создание ресурса. Позднее мы его расширим под габариты надписи: для них зарезервированы переменные *width* и *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);
   ...
```

В цикле с отсчетом времени теста нам потребуется менять ориентацию надписи, для чего заведена переменная *angle* (в ней будут прокручиваться градусы). Ориентация будет меняться раз в секунду, отсчет ведется в переменной *remain*.

```
   const uint timeout = GetTickCount() + Seconds * 1000;
   int angle = 0;
   int remain = 10;
   ...
```

В цикле мы постоянно меняем поворот текста, а в сам текст выводим обратный счетчик секунд. Для каждой новой надписи вычисляется её размер с помощью *TextGetSize*, на основе чего перераспределяется буфер.

```
   while(!IsStopped() && GetTickCount() < timeout)
   {
      // применяем новый угол
      TextSetFont("::a_LCDNova3DCmObl.ttf", -240, 0, angle * 10);

      // формируем текст
      const string text = Message + " (" + (string)remain-- + ")";

      // получаем размеры текста, распределяем массив
      TextGetSize(text, width, height);
      ArrayResize(data, width * height);
      ArrayInitialize(data, 0);            // прозрачность

      // при вертикальной ориентации меняем местами размеры
      if((bool)(angle / 90 & 1))
      {
         const uint t = width;
         width = height;
         height = t;
      }

      // подстраиваем размеры объекта
      ObjectSetInteger(0, name, OBJPROP_XSIZE, width);
      ObjectSetInteger(0, name, OBJPROP_YSIZE, height);

      // рисуем текст
      TextOut(text, width / 2, height / 2, TA_CENTER | TA_VCENTER, data, width, heigh
         ColorToARGB(clrBlue), COLOR_FORMAT_ARGB_RAW);

      // обновляем ресурс и график
      ResourceCreate(name, data, width, height, 0, 0, width, COLOR_FORMAT_ARGB_RAW);
      ChartRedraw();

      // меняем угол
      angle += 90;

      Sleep(100);
   }
   ...
```

Обратите внимание, что при вертикальной ориентации текста необходимо поменять местами размерности. В более общем случае с поворотом текста на произвольный угол потребовалось больше математики, чтобы получить размеры буфера, способного уместить всю надпись.

В конце также удаляем объект и ресурс.

```
   ObjectDelete(0, name);
   ResourceFree("::" + name);
}
```

Один из моментов выполнения скрипта представлен на следующем скриншоте.

Надпись с пользовательским шрифтом

В качестве последнего примера разберем скрипт *ResourceTextAnchOrientation.mq5*, демонстрирующий различные повороты и точки привязки текста.

Скрипт генерирует заданное количество надписей (*ExampleCount*), используя указанный шрифт.

```
input string Font = "Arial";             // Font Name
input int    Size = -150;                // Size
input int    ExampleCount = 11;          // Number of examples
```

Точки привязки и повороты выбираются случайным образом.

Для указания названий точек привязки в надписях объявлено перечисление ENUM_TEXT_ANCHOR со всеми допустимыми вариантами, так что для любого случайно выбранного элемента достаточно будет вызвать *EnumToString*.

```
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,
};
```

В обработчике *OnStart* первым делом объявлен массив этих новых констант.

```
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);
   ...
```

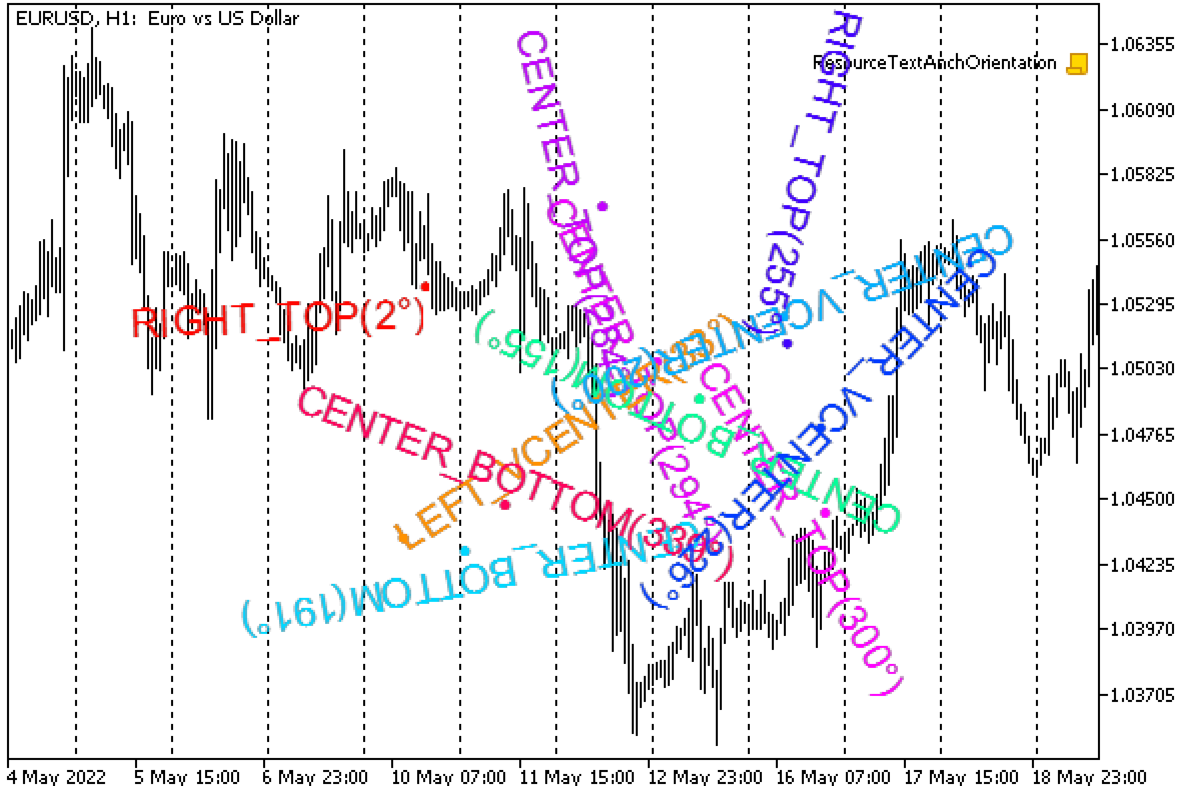
Начальное создание объекта и ресурса аналогично примеру *ResourceText.mq5*, так что опустим их здесь. Самое интересное происходит в цикле.

```
   for(int i = 0; i < ExampleCount; ++i)
   {
      // применяем случайный угол
      const int angle = Random(360);
      TextSetFont(Font, Size, 0, angle * 10);

      // берем случайные координаты и точку привязки
      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);

      // рисуем кружок непосредственно в том месте изображения,
      // куда попадает точка привязки
      TextOut(ShortToString(0x2022), x, y, TA_CENTER | TA_VCENTER, data, w, h,
         ColorToARGB(clr), COLOR_FORMAT_ARGB_NORMALIZE);

      // формируем текст с описанием типа привязки и угла
      const string text =  EnumToString(anchor) +
         "(" + (string)angle + CharToString(0xB0) + ")";

      // рисуем текст
      TextOut(text, x, y, anchor, data, w, h,
         ColorToARGB(clr), COLOR_FORMAT_ARGB_NORMALIZE);
   }
   ...
```

Остается лишь обновить картинку и график, а затем ждать команды пользователя и освободить ресурсы.

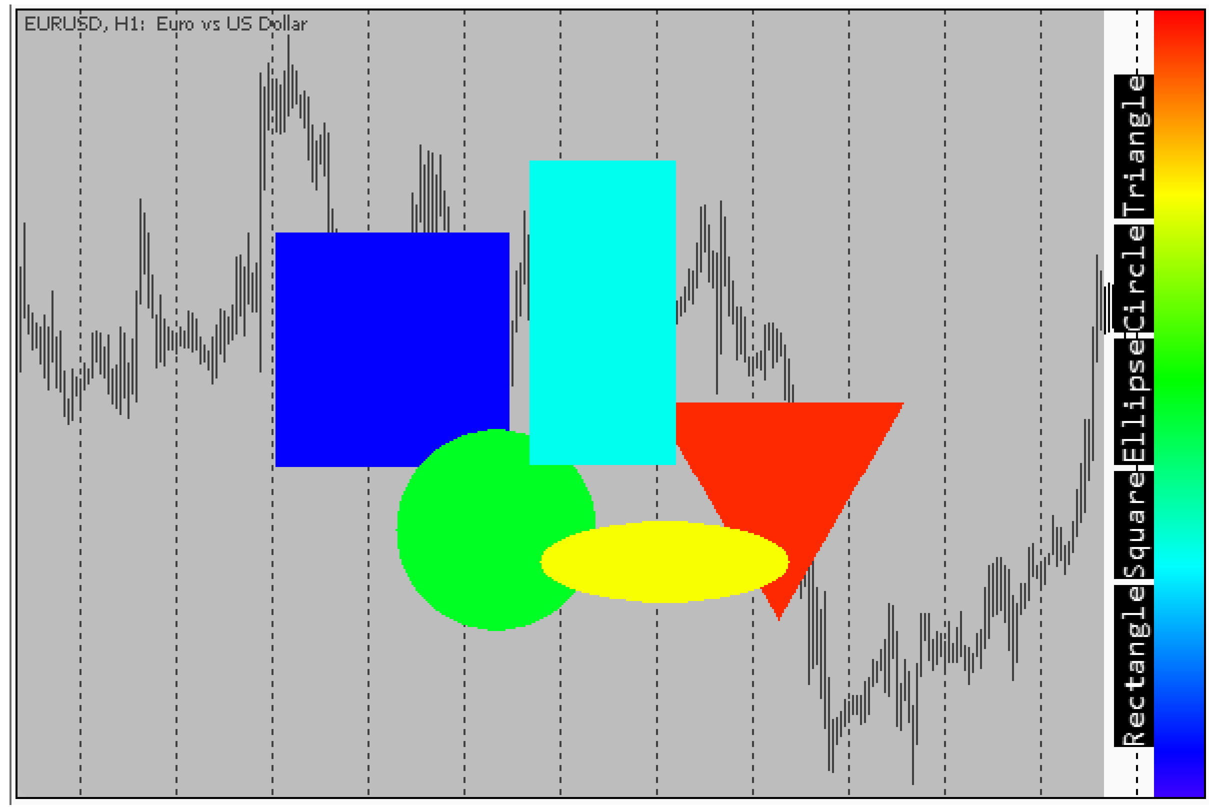
```
      ResourceCreate(name, data, w, h, 0, 0, w, COLOR_FORMAT_ARGB_NORMALIZE);
      ChartRedraw();

      const uint timeout = GetTickCount() + Seconds * 1000;
      while(!IsStopped() && GetTickCount() < timeout)
      {
         Sleep(1000);
      }

      ObjectDelete(0, name);
      ResourceFree("::" + name);
   }
```

Вот что получилось в результате.



Вывод текста со случайными координатами, точками привязки и углами

Дополнительно, для самостоятельного изучения к книге прилагается игрушечный графический редактор *SimpleDrawing.mq5*. Он выполнен как безбуферный индикатор и использует в своей работе рассмотренные ранее классы фигур (см. пример *ResourceShapesDraw.mq5*). Они практически без изменений вынесены в заголовочный файл *ShapesDrawing.mqh*. Но если ранее фигуры генерировал случайным образом скрипт, то теперь их может выбирать и наносить на график пользователь. Для этой цели реализован интерфейс с цветовой палитрой и панелью кнопок, по количеству зарегистрированных классов фигур. Весь интерфейс поддерживается классом *SimpleDrawing* (*SimpleDrawing.mqh*).



Простой графический редактор

Панель и палитра могут располагаться вдоль любой границы графика, демонстрируя возможность поворота надписей.

Выбор следующей фигуры для рисования выполняется нажатием кнопки в панели: кнопка "залипает" во включенном состоянии, причем её фоновый цвет обозначает выбранный цвет рисования. Чтобы изменить цвет, щелкните в любом месте палитры.

Когда один из типов фигур выбран в панели (одна из кнопок "активна"), щелчок мышью в рабочей области рисования (остальная часть графика, обозначенная затемнением) выводит в этом месте фигуру предопределенного размера. Кнопка при этом "отключается". В таком состоянии, когда все кнопки неактивны, можно двигать мышью фигуры по рабочей области. Если держать нажатой клавишу *Ctrl*, вместо перемещения производится изменение размера фигуры. "Горячая точка" находится в центре каждой фигуры (размер чувствительной области задан макросом в исходном коде и, вероятно, потребует увеличения для дисплеев с очень высоким DPI).

Обратите внимание, что редактор включает идентификатор графика (*ChartID*) в имена создаваемых ресурсов. Это позволяет запускать редактор параллельно на нескольких графиках.

7.1.10 Прикладное применение графических ресурсов в трейдинге

Чтобы у вас не создалось впечатление, что ресурсы подходят только для украшений, покажем, как на их основе можно разработать инструмент, полезный для трейдеров. Заодно ликвидируем еще одно упущение: до сих пор мы использовали ресурсы только внутри объектов OBJ_BITMAP_LABEL, которые позиционируются в экранных координатах. Однако графические ресурсы могут быть встроены и в объекты OBJ_BITMAP с привязкой к координатам котировок: ценам и времени.

Ранее в книге был описан индикатор *IndDeltaVolume.mq5*, рассчитывающий дельту объемов (тиковых или реальных) для каждого бара. Кроме такого представления дельты объемов существует и еще одно, не менее популярное у пользователей, — так называемый профиль рынка. Это распределение объемов в разрезе ценовых уровней. Подобную гистограмму можно строить для всего окна целиком, на заданную глубину (например, внутри дня) или для отдельно взятого бара.

Именно последний вариант мы и реализуем в виде нового индикатора *DeltaVolumeProfile.mq5*. Основные технические тонкости запроса истории тиков мы уже рассмотрели в рамках вышеупомянутого индикатора, поэтому сконцентрируемся теперь, в основном, на графической составляющей.

Флаг *ShowSplittedDelta* во входной переменной будет управлять тем, как отображать объемы: в разбивке на покупки/продажи или в свернутом виде.

```
input bool ShowSplittedDelta = true;
```

В индикаторе не будет буферов. Рассчитывать и показывать гистограмму для конкретного бара он будет по запросу пользователя, а конкретно - по щелчку мыши на этом баре. Таким образом, на первый план выходит обработчик *OnChartEvent*. В нем мы получаем экранные координаты, пересчитываем их в цену и время, и вызываем некую вспомогательную функцию *RequestData*, запускающую расчет.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strir
{
   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));
      }
   }
   ...
}
```

Для её наполнения нам потребуется класс *DeltaVolumeProfile*, который построен по подобию класса *CalcDeltaVolume* из *IndDeltaVolume.mq5*.

В новом классе описаны переменные, учитывающие способ подсчета объемов (*tickType*), тип цены, по которой строится график (*barType*), режим из входной переменной *ShowSplittedDelta* (будет помещен в переменную-член *delta*), а также префикс для генерируемых объектов на графике.

```
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: удалить ресурсы
   }
   ...
};
  
static const string DeltaVolumeProfile::prefix = "DVP";
  
DeltaVolumeProfile deltas(TickType, ShowSplittedDelta);
```

Напомним, что *TickType* можно менять на значение TRADE_TICKS только для торговых инструментов, по которым доступны реальные объемы. По умолчанию включен режим INFO_TICKS, работоспособный на всех инструментах.

Запрос тиков для конкретного бара делает метод *createProfileBar*.

```
int createProfileBar(const int i)
{
   MqlTick ticks[];
   const datetime time = iTime(_Symbol, _Period, i);
   // prev и next - временные границы бара
   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;
}
```

Непосредственный анализ тиков и подсчет объемов выполняется в защищенном методе *calcProfile*. В нем мы прежде всего узнаем диапазон цен бара и его размер в пикселях.

```
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));
   ...
```

Руководствуясь этой информацией, создаем объект OBJ_BITMAP, выделяем массив для изображения и создаем ресурс. Фон всей картинки — пустой (прозрачный). Каждый объект привязывается верхней средней точкой к цене *High* своего бара и имеет ширину одного бара.

```
   uint data[];
   ArrayResize(data, w * h);
   ArrayInitialize(data, 0);
   ResourceCreate(name + (string)ChartID(), data, w, h, 0, 0, w, COLOR_FORMAT_ARGE
   
   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);
   ...
```

Далее следует подсчет объемов в тиках переданного массива. Количество ценовых уровней равно высоте бара в пикселях (*h*). Обычно оно меньше, чем диапазон цены в пунктах, и потому пиксели выступают, своего рода, корзинами для подсчета статистики. Если на мелком таймфрейме диапазон пунктов окажется меньше размера в пикселях, гистограмма получится визуально разреженной. Объемы покупок и продаж аккумулируются отдельно в массивах *plus* и *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]); // вернет Bid или Last
   const int index = (int)((high - p1) / range * (h - 1));
   if(tickType == TRADE_TICKS)
   {
      // если доступны реальные объемы, можем учитывать их
      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)
   {
      // если реальных объемов нет,
      // движение цены вверх/вниз является оценкой типа объема
      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;
      }
   }
   ...
```

Для нормирования гистограммы попутно находим максимальное значение.

```
         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]);
         }
      }
      ...
```

Наконец, полученная статистика выводится в графический буфер *data* и отправляется в ресурс. Объемы покупки выводятся синим цветом, продажи — красным, а если включен нетто-режим, то сумма — сиреневым.

```
      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
   }
```

Теперь мы можем вернуться к функции *RequestData*: её задача — вызвать метод *createProfileBar* и обработать ошибки (если они возникли).

```
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); // запрос на обновление графика
      EventChartCustom(0, TRY_AGAIN, b, count + 1, NULL);
   }
   Comment("");
}
```

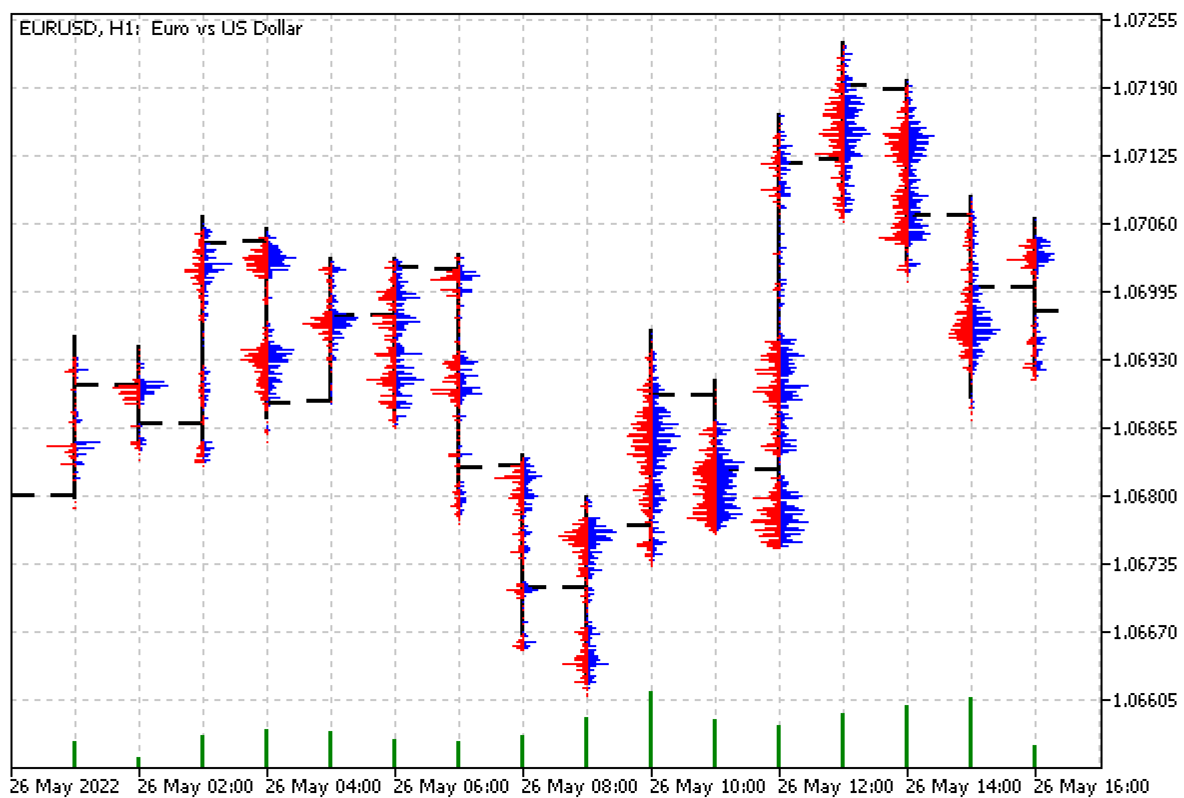
Единственная стратегия обработки ошибок заключается в том, чтобы попробовать запросить тики еще раз, потому что они могли не успеть загрузиться. Для этой цели функция посылает графику пользовательское сообщение TRY_AGAIN и сама же обрабатывает его.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strir
{
   ...
   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
      }
   }
}
```

Мы повторяем этот процесс не более 5 раз, потому что история тиков может иметь ограниченную глубину, и нагружать компьютер зря не имеет смысла.

Дополнительно класс *DeltaVolumeProfile* снабжен механизмом для обработки сообщения CHARTEVENT_CHART_CHANGE, чтобы перерисовать уже имеющиеся объекты в случае изменения размеров или масштаба графика. Подробности можно выяснить в исходном коде.

Результат работы индикатора показан на следующем изображении.



Отображение побаровых гистограмм раздельных объемов в графических ресурсах

Учтите, что сразу после нанесения индикатора гистограммы не отображаются: вам следует щелкнуть на баре, чтобы для него рассчиталась гистограмма.

7.2 Пользовательские символы

Одна из интересных технических особенностей MetaTrader 5 заключается в поддержке пользовательских финансовых инструментов. Это символы, которые определены не на сервере брокером, а самим трейдером — непосредственно в терминале.

Пользовательские символы разрешено добавлять в список *Обзора рынка* наравне со стандартными символами и использовать чарты с ними привычным образом.

Наиболее простой способ создания пользовательского инструмента — указать его расчетную формулу в соответствующем свойстве. В интерфейсе терминала для этого нужно вызвать контекстное меню в окне *Обзора рынка*, выполнить команду *Символы*, и в иерархии символов, находясь в ветви *Custom*, нажать кнопку *Создать символ*. В результате откроется диалог настройки свойств нового символа. Там же в подобные инструменты можно импортировать внешнюю историю тиков (закладка *Тики*) или котировок (закладка *Бары*) из файлов. Подробно об этом рассказано в документации MetaTrader 5.

Однако наиболее полный контроль над пользовательскими символами предоставляет MQL5 API.

Для пользовательских символов полностью поддерживается группа функций по работе с Финансовыми инструментами и обзором рынка. В частности, их можно перечислить из программы с помощью стандартных функций, таких как *SymbolsTotal*, *SymbolName*, *SymbolInfo*-функций. Мы уже вкратце касались этой возможности в разделе Свойства пользовательских

символов, и приводили пример. Отличительной чертой пользовательского символа является взведенный флаг (свойство) SYMBOL_CUSTOM.

С помощью встроенных функций можно выполнять склейки фьючерсов, генерировать случайные временные ряды с заданными характеристиками, эмулировать ренко, равнодиапазонные ("рейндж"-бары), равнообъемные (эквиобъемные) и другие нестандартные виды графиков (например, секундные таймфреймы). Также, в отличие от импорта статических файлов, программно-управляемые пользовательские символы способны в реальном времени формироваться из данных веб-сервисов, таких как криптовалютные биржи. Об интеграции MQL-программ с Интернет разговор еще впереди, но об этой возможности нельзя не упомянуть.

Пользовательский символ легко задействовать для проверки стратегий в тестере или как дополнительный способ технического анализа. Но есть в данной технологии и свои ограничения.

Из-за того, что пользовательские символы определены в терминале, а не на сервере, ими невозможно торговать онлайн. В частности, если вы создадите график ренко, торговые стратегии на его основе нужно будет адаптировать тем или иным образом, чтобы торговые сигналы и торговые операции фактически были разведены по разным символам: искусственному пользовательскому и реальному брокерскому. Мы рассмотрим пару способов решения проблемы.

Кроме того, учитывая такую специфику платформы, что длительность всех баров одного таймфрейма одинакова, любая эмуляция разнопериодных баров (ренко, эквиобъемных и т.д.) основывается, как правило, на меньшем из доступных таймфреймов M1 и не обеспечивает полную временную синхронизацию с реальностью. Иными словами, тики, принадлежащие такому бару, вынуждены иметь искусственное время внутри 60 секунд, даже если "кирпич" ренко или бар заданного объема требовал в реальности гораздо большего времени на свое формирование. В противном случае, если бы мы проставляли тикам реальное время, они сформировали бы следующие бары M1, нарушая правила ренко или эквиобъемности. Более того, возможны ситуации, когда "кирпич" ренко или другой искусственный бар должен быть создан с меньшим промежутком времени, чем 1 минута от предыдущего бара (например, при повышенной быстрой волатильности). В таких случаях в котировках пользовательского инструмента потребуется менять время исторических баров (сдвигать их влево "задним числом") или проставлять новым барам будущее время (что крайне нежелательно). Данную проблему решить в общем виде в рамках технологии пользовательских символов нельзя.

7.2.1 Создание и удаление пользовательских символов

Первые две функции, которые потребуются для работы с пользовательскими символами, — это *CustomSymbolCreate* и *CustomSymbolDelete*.

bool CustomSymbolCreate(const string name, const string path = "", const string origin = NULL)

Функция создает пользовательский символ с указанным именем (*name*) в указанной группе (*path*) и, при необходимости, со свойствами образцового символа — его имя можно задать в параметре *origin*.

Параметр *name* должен быть простым идентификатором, без иерархии. При необходимости один или несколько требуемых уровней групп (вложенных папок) следует указывать в параметре *path*, причем символом разделителем является обратная наклонная черта '\' (прямая черта здесь не поддерживается, в отличие от файловой системы). Напомним, что обратную черту нужно задваивать в строках-литералах ("\\").

По умолчанию, если строка *path* пуста ("" или NULL), символ создается непосредственно в папке *Custom* — она выделена в общей иерархии символов для пользовательских символов. Если путь заполнен, он создается внутри папки *Custom* на всю глубину (если соответствующих папок еще не было).

Имя символа, также как и название группы любого уровня, может содержать латинские буквы и цифры, без знаков препинания, пробелов и спецсимволов. Дополнительно допускаются только символы '.', '_', '&' и '#'.

Имя должно быть уникальным во всей иерархии символов, независимо от того, в какой группе предполагается создание символа. Если символ с таким именем уже существует, функция вернёт *false* и установит код ошибки *_LastError* в 5300 (ERR_NOT_CUSTOM_SYMBOL) или 5304 (ERR_CUSTOM_SYMBOL_EXIST).

Следует обратить внимание, что если последний (или даже единственный) элемент иерархии в строке *path* точно совпадает с именем *name* (с учетом регистра), то он трактуется именно как имя символа, входящее в состав пути, а не как папка. Например, если имя и путь содержат, соответственно, строки "Example" и "MQL5Book\\Example", то будет создан символ "Example" в папке "Custom\\MQL5Book\\". Вместе с тем, если изменить имя на "example", получим символ "example" в папке "Custom\\MQL5Book\\Example".

Эта особенность имеет еще одно следствие. Свойство SYMBOL_PATH возвращает путь вместе с именем символа на конце. Поэтому если перенести его значение без изменений из некоего образцового символа во вновь создаваемый, получим следующий эффект: будет создана папка с именем старого символа, внутри которой и появится новый символ. Таким образом, если необходимо создать пользовательский символ в той же группе, где и исходный символ, то необходимо отрезать имя исходного символа от строки, полученной из свойства SYMBOL_PATH.

Побочный эффект копирования свойства SYMBOL_PATH мы продемонстрируем в примере следующего раздела. Но этот эффект может использоваться и как положительный. В частности, создавая несколько своих символов на базе одного исходного, копирование SYMBOL_PATH обеспечит размещение всех новых символов в папке с именем оригинала, то есть сгруппирует символы по их прообразу.

Свойство SYMBOL_PATH у пользовательских символов всегда начинается с папки "Custom\\" (этот префикс добавляется автоматически).

Длина имени ограничена 31-м знаком. При превышении лимита *CustomSymbolCreate* вернёт *false* и установит код ошибки 5302 (ERR_CUSTOM_SYMBOL_NAME_LONG).

Максимальная длина параметра path 127 знаков с учетом "Custom\\", разделителей групп "\\" и имени символа, если оно указано в конце.

Параметр *origin* позволяет при желании задать имя символа, из которого будут скопированы свойства создаваемого пользовательского символа. После создания пользовательского символа можно изменить любое его свойство на нужное значение соответствующими функциями (см. *CustomSymbolSet*-функции).

Если в качестве параметра *origin* задан несуществующий символ, то пользовательский символ будет создан "пустым", как если бы параметр *origin* не был указан. При этом будет взведена ошибка 4301 (ERR_MARKET_UNKNOWN_SYMBOL).

В новом символе, созданным "пустым", все свойства установлены в значения по умолчанию. Например, размер контракта равен 100000, количество разрядов в цене — 4, расчет маржи — по правилам Forex, а построение графиков — по ценам *Bid*.

При указании *origin*, из этого символа в новый символ переносятся только настройки, но не котировки или тики — их следует генерировать отдельно, о чем речь пойдет в последующих разделах.

Создание символа не означает его автоматическое добавление в *Обзор рынка* — это нужно сделать явным образом (вручную или программно). Без котировок окно графика будет пустым.

bool CustomSymbolDelete(const string name)

Функция удаляет пользовательский символ с указанным именем. Удаляются не только настройки, но и все данные по символу (котировки и тики). Правда, история удаляется не сразу, а с некоторой задержкой, что может послужить источником проблем, если предполагается воссоздать символ с таким же именем (мы коснемся этого момента в примере в разделе Добавление, замена и удаление котировок).

Удалить можно только пользовательский символ. Кроме того нельзя удалить символ, выбранный в *Обзор рынка* или для которого открыт график. Напомним, что символ может быть выбран и неявным образом, без отображения в видимом списке (в таких случаях свойство SYMBOL_VISIBLE равно *false*, а свойство SYMBOL_SELECT — *true*). Такой символ требуется предварительно "скрыть" вызовом *SymbolSelect("name", false)* перед попыткой удаления: в противном случае получим ошибку CUSTOM_SYMBOL_SELECTED (5306).

Если в результате удаления символа остается пустая папка (или иерархия папок), она также удаляется.

Для примера создадим простой скрипт *CustomSymbolCreateDelete.mq5*. Во входных параметрах можно указать название, путь и образцовый символ.

```
input string CustomSymbol = "Dummy";          // Custom Symbol Name
input string CustomPath = "MQL5Book\\Part7"; // Custom Symbol Folder
input string Origin;
```

В обработчике *OnStart* проверим, имеется ли уже символ с заданным именем. Если нет, то после подтверждения пользователя, создадим такой символ. Если символ уже есть, и он является пользовательским, удалим его с разрешения пользователя (это упростит процесс очистки после завершения эксперимента).

```
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");
      }
   }
}
```

Два последовательных запуска с параметрами по умолчанию должны привести к появлению следующих записей в журнале.

```
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
```

Между запусками вы можете открыть диалог символов в терминале и убедиться, что соответствующий пользовательский символ появился в иерархии символов.

7.2.2 Свойства пользовательских символов

Пользовательские символы обладают теми же свойствами, что и символы, предоставленные брокером. Чтение их свойств выполняется стандартными функциями, рассмотренными в главе про финансовые инструменты.

Для установки свойств пользовательских символов предназначена группа *CustomSymbolSet*-функций, по одной функции для значений основополагающих типов (целочисленных, вещественных, строковых).

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)

Все функции устанавливают для пользовательского символа с именем *name* значение свойства *property* в значение *value*. Все существующие свойства сгруппированы в перечисления ENUM_SYMBOL_INFO_INTEGER, ENUM_SYMBOL_INFO_DOUBLE, ENUM_SYMBOL_INFO_STRING, которые были поэлементно рассмотрены в разделах вышеупомянутой главы.

Функции возвращают признак успеха (*true*) или ошибки (*false*). Одна из вероятных проблем для ошибок заключается в том, что не все свойства разрешено менять. При попытке установить "read-only" свойство получим ошибку CUSTOM_SYMBOL_PROPERTY_WRONG (5307). Если пытаться записать в свойство недопустимое значение, получим ошибку CUSTOM_SYMBOL_PARAMETER_ERROR (5308).

Следует учесть, что минутная и тиковая история пользовательского символа полностью удаляется, если в спецификации символа изменить любое из следующих свойств:

- SYMBOL_CHART_MODE — тип цены, который используется для построения баров (*Bid* или *Last*)
- SYMBOL_DIGITS — количество знаков после запятой в значениях цены
- SYMBOL_POINT — значение одного пункта
- SYMBOL_TRADE_TICK_SIZE — значение одного тика, минимально допустимое изменение цены
- SYMBOL_TRADE_TICK_VALUE — стоимость изменения цены в один тик (см. также SYMBOL_TRADE_TICK_VALUE_PROFIT, SYMBOL_TRADE_TICK_VALUE_LOSS)
- SYMBOL_FORMULA — формула для расчета цены

Если пользовательский символ рассчитывается по формуле, то после удаления его истории терминал автоматически попытается создать новую историю с использованием обновленных свойств. Однако для символов, генерируемых программно, сама MQL-программа должна позаботиться о пересчете.

Редактирование отдельных свойств наиболее востребовано для модификации созданных ранее по образцу пользовательских символов (после указания третьего параметра *origin* в функции *CustomSymbolCreate*).

В других случаях массовое изменение свойств может вызывать неочевидные эффекты. Дело в том, что свойства внутренне связаны между собой и изменение одного из них может требовать определенного состояния других свойств, чтобы операция завершилась успешно. Более того, установка одних свойств приводит к автоматическим изменениям других.

Наиболее простой пример: после установки свойства SYMBOL_DIGITS вы обнаружите, что изменилось также и свойство SYMBOL_POINT. Менее очевидный случай: присваивание SYMBOL_CURRENCY_MARGIN или SYMBOL_CURRENCY_PROFIT не имеет эффекта для символов Forex, так как система считает, что имена валют занимают, соответственно первые 3 и последующие 3 буквы в названии ("XXXYYY[suffix]"). Ирония здесь заключается в том, что

непосредственно после создания "пустого" символа, он по умолчанию считается символом Forex, и потому для него нельзя задать эти свойства, не изменив предварительно рынок.

При копировании или установке свойств символа следует учитывать, что платформа подразумевает некоторые нюансы. В частности, свойство SYMBOL_TRADE_CALC_MODE имеет по умолчанию значение 0 (сразу после создания символа, но до установки какого-либо свойства), а 0 в перечислении ENUM_SYMBOL_CALC_MODE соответствует элементу SYMBOL_CALC_MODE_FOREX. Вместе с тем, для форексных символов подразумеваются особые правила именования в виде XXXYYY (где XXX и YYY — коды валют) плюс опциональный суффикс. Поэтому, если не изменить заранее SYMBOL_TRADE_CALC_MODE на другой требуемый режим, подстроки задаваемого названия символа (первая и вторая тройка символов) автоматически попадут в свойства базовой валюты (SYMBOL_CURRENCY_BASE) и валюты прибыли (SYMBOL_CURRENCY_PROFIT). Например, если задать имя "Dummy", оно разрежется на 2 псевдо-валюты "Dum" и "my".

Еще один нюанс заключается в том, что прежде чем устанавливать значение SYMBOL_POINT с точностью N знаков после запятой, нужно обеспечить SYMBOL_DIGITS не менее N.

К книге прилагается скрипт *CustomSymbolProperties.mq5*, который позволяет поэкспериментировать с созданием копий символа текущего графика и изучить на практике возникающие эффекты. В частности, можно выбрать название символа, его путь и направление обхода (установки) всех поддерживаемых свойств — прямое или обратное с точки зрения нумерации свойств в языке. В скрипте используется специальный класс *CustomSymbolMonitor*, являющийся оберткой вышеупомянутых встроенных функций: мы опишем его *позднее*.

7.2.3 Установка маржинальных коэффициентов

Ранее мы изучали функцию *SymbolInfoMarginRate*, которая возвращает маржинальные коэффициенты по символу, установленные брокером. Для пользовательского символа мы вольны сами задавать эти коэффициенты с помощью функции *CustomSymbolSetMarginRate*.

bool CustomSymbolSetMarginRate(const string name, ENUM_ORDER_TYPE orderType, double initial, double maintenance)

Функция устанавливает коэффициенты взимания маржи в зависимости от типа и направления ордера (согласно значению *orderType* из перечисления ENUM_ORDER_TYPE). Коэффициенты для расчета начальной и поддерживающей маржи (залога за каждый лот открываемой или существующей позиции) передаются, соответственно, в параметрах *initial* и *maintenance*.

Напомним, что окончательные суммы залога определяются исходя из нескольких свойств символа (SYMBOL_TRADE_CALC_MODE, SYMBOL_MARGIN_INITIAL, SYMBOL_MARGIN_MAINTENANCE и других), описанных в разделе Маржинальные требования, поэтому их также следует при необходимости установить у пользовательского символа.

Функция вернет признак успеха (*true*) или ошибки (*false*).

С помощью данной функции и связанных с расчетом маржи свойств вы можете эмулировать в тестере торговые условия недоступных по тем или иным причинам серверов и отлаживать в них свои MQL-программы.

7.2.4 Настройка котировочных и торговых сессий

Две функции API позволяют устанавливать котировочные и торговые сессии пользовательского инструмента. Эти два понятия мы рассматривали в разделе Расписания торговых и котировочных сессий.

```
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 устанавливает время начала и окончания указанной по номеру (*sessionIndex*) котировочной сессии для конкретного дня недели (*dayOfWeek*). *CustomSymbolSetSessionTrade* делает то же самое для торговых сессий.

Нумерация сессий начинается с 0.

Добавлять сессии можно только последовательно, то есть сессию с индексом 1 можно добавить только в том случае, если уже существует сессия с индексом 0. При нарушении этого правила новая сессия не создастся, а функция вернет значение *false*.

Значения дат в параметрах *from* и *to* измеряются в секундах, *from* должно быть меньше *to*. Диапазон ограничен двумя сутками — от 0 (00 часов 00 минут 00 секунд) до 172800 (23 часа 59 минут 59 секунд на следующий день). Переход через сутки потребовался для того, чтобы можно было указать сессии, которые начинаются до полуночи, а заканчиваются — после. Данная ситуация часто встречается, когда биржа расположена на другом конце света относительно серверов брокера (дилера).

Если для сессии *sessionIndex* переданы нулевые параметры начала и конца (*from = 0* и *to = 0*), то она удаляется, а нумерация следующих сессий (если они есть) сдвигается вниз.

Торговые сессии не могут выходить за рамки котировочных.

Например, мы можем создать копию инструмента для другого часового пояса, сдвинув в нем внутрисуточное время котировок и расписание сессий для отладки робота в различных условиях, как у любых экзотических брокеров.

7.2.5 Добавление, замена и удаление котировок

Наполнение пользовательского символа котировками осуществляется двумя встроенными функциями: *CustomRatesUpdate* и *CustomRatesReplace*. Обе ожидают на входе, помимо названия символа, массив структур *MqlRates* для таймфрейма M1 (более старшие таймфреймы достраиваются из M1 автоматически). *CustomRatesReplace* имеет дополнительную пару параметров (*from* и *to*), задающих временной диапазон, которым ограничивается редактирование истории.

```
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 добавляет в историю отсутствующие бары и заменяет существующие совпадающие бары данными из массива.

CustomRatesReplace полностью заменяет историю в указанном временном интервале данными из массива.

Разница между функциями обусловлена разными сценариями предполагаемого применения. Более детально отличия перечислены в следующей таблице.

CustomRatesUpdate	CustomRatesReplace
Применяет к истории элементы передаваемого массива MqlRates, вне зависимости от их временных меток	Применяет только те элементы передаваемого массива MqlRates, которые попадают в указанный диапазон
Оставляет нетронутыми в истории те бары M1, которые там уже были до вызова функции и не совпадают по времени с барами в массиве	Оставляет нетронутой всю историю вне диапазона
Изменяет существующие бары истории на бары в массиве при совпадении временных меток	Полностью удаляет существующие бары истории в указанном диапазоне
Вставляет элементы из массива как "новые" бары при отсутствии совпадений с прежними барами	Вставляет в указанный диапазон истории бары из массива, которые попадают в этот диапазон

Данные в массиве *rates* должны быть корректными четверками цен OHLC, а времена открытия баров — не содержать секунд.

Интервал в пределах *from* и *to* задается включительно: *from* равно времени первого бара, подлежащего обработке, а *to* — времени последнего.

На следующей схеме данные правила продемонстрированы более наглядно. Каждая уникальная временная метка для бара обозначена своей латинской буквой. Имеющиеся бары в истории показаны заглавными, а бары в массиве — строчными буквами. Символ '-' — это пропуск в истории или в массиве для соответствующего времени.

```
История                       ABC-EFGHIJKLMN-PQRST------    Б
Массив                        -------hijk--nopqrstuvwxyz    А
Результат CustomRatesUpdate   ABC-EFGhijkLMnopqrstuvwxyz    Р
Результат CustomRatesReplace  ABC-E--hijk--nopqrstuvw---    Ы
                                   ^                ^
                                   |from          to|   ВРЕМЯ
```

Опциональный параметр *count* задает количество элементов массива *rates*, которые должны использоваться (остальные будут проигнорированы). Это позволяет обрабатывать переданный массив частично. Значение по умолчанию WHOLE_ARRAY означает весь массив.

Удалить историю котировок пользовательского символа — полностью или частично — позволяет функция *CustomRatesDelete*.

int CustomRatesDelete(const string symbol, datetime from, datetime to)

Здесь параметры *from* и *to* также задают временной диапазон удаляемых баров. Чтобы охватить всю историю, укажите 0 и LONG_MAX.

Все три функции возвращают количество обработанных баров: обновленных или удаленных. В случае ошибки результат равен -1.

Следует отметить, что котировки пользовательского символа можно формировать не только путем добавления готовых баров, но и массивов тиков или даже последовательности отдельных тиков — функции для этого будут представлены в следующем разделе. При добавлении тиков терминал сам рассчитает на их основе бары. Отличие между этими способами заключается в том, что история кастом-тиков позволяет тестировать MQL-программы в режиме "реальных" тиков, в то время как история только баров заставит либо ограничиться режимами OHLC M1 или цен открытия, либо положиться на эмуляцию тиков, реализованную тестером.

Кроме того, добавление тиков по одному позволяет имитировать на графике пользовательского символа стандартные события *OnTick* и *OnCalculate*, что "оживляет" график по подобию инструментов, доступных онлайн, и запускает соответствующие функции-обработчики в MQL-программах, если они нанесены на график. Но об этом мы поговорим в следующем разделе.

В качестве примера использования новых функций рассмотрим скрипт *CustomSymbolRandomRates.mq5*. Он предназначен для генерации случайных котировок по принципу "случайного блуждания" или зашумления существующих котировок. Последние можно применять для оценки устойчивости эксперта.

Для проверки правильности формирования котировок также поддержим режим, в котором создается полная копия исходного инструмента, на чарте которого и был запущен скрипт.

Все режимы собраны в перечисление RANDOMIZATION.

```
enum RANDOMIZATION
{
   ORIGINAL,
   RANDOM_WALK,
   FUZZY_WEAK,
   FUZZY_STRONG,
};
```

Зашумление котировок реализуем с двумя степенями интенсивности: слабой (weak) и сильной (strong).

Во входных параметрах можно выбрать, помимо режима, папку в иерархии символов, диапазон дат и число для инициализации случайного генератора (чтобы иметь возможность воспроизводить результаты).

```
input string CustomPath = "MQL5Book\\Part7";     // Custom Symbol Folder
input RANDOMIZATION RandomFactor = RANDOM_WALK;
input datetime _From;                            // From (умолчание: 120 дней назад)
input datetime _To;                              // To (умолчание: текущее время)
input uint RandomSeed = 0;
```

По умолчанию, когда даты не указаны, скрипт генерирует котировки для последних 120 дней. Значение 0 в параметре *RandomSeed* означает случайную инициализацию.

Название символа создается на основе символа текущего чарта и выбранных настроек.

```
const string CustomSymbol = _Symbol + "." + EnumToString(RandomFactor)
   + (RandomSeed ? "_" + (string)RandomSeed : "");
```

В начале *OnStart* произведем подготовку и проверку данных.

```
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);
   ...
```

Поскольку скрипт потребуется, скорее всего, запускать несколько раз, предусмотрим возможность удалять созданный ранее пользовательский символ, с предварительным запросом подтверждения у пользователя.

```
   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); // ждем пока изменения вступят в силу (навскидку)
            PRTF(CustomRatesDelete(CustomSymbol, 0, LONG_MAX));
            PRTF(SymbolSelect(CustomSymbol, false));
            PRTF(CustomSymbolDelete(CustomSymbol));
         }
      }
   }
   ...
```

Вспомогательная функция *CloseChartsForSymbol* здесь не приводится (желающие могут заглянуть в прилагаемый исходный код): её суть заключается в просмотре списка открытых графиков и закрытии тех из них, где рабочим символом является удаляемый пользовательский символ (без этого удаление не сработает).

Более важно обратить внимание на вызов *CustomRatesDelete* с указанием полного диапазона дат. Если его не сделать, на диске на некоторое время останутся данные прежнего пользовательского символа в базе истории (папка *bases/Custom/history/<имя-символа>*). Иными словами, вызов *CustomSymbolDelete*, который показан выше последней строкой, недостаточен для того, чтобы фактически вычистить пользовательский символ из терминала.

Если пользователь решит тут же создать символ с тем же именем заново (а у нас в коде далее такая возможность предусмотрена), то старые котировки могут подмешаться в новые.

Далее, опять же с запросом подтверждения пользователем, запускается процесс генерации котировок: за него отвечает функция *GenerateQuotes* (см. далее).

```
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);
      }
   }
}
```

В случае успеха вновь созданный символ выбирается в *Обзор рынка* и для него открывается график. Попутно здесь демонстрируется установка пары свойств: SYMBOL_DIGITS и SYMBOL_DESCRIPTION.

В функции *GenerateQuotes* для всех режимов кроме RANDOM_WALK требуется запросить котировки оригинального символа.

```
int GenerateQuotes()
{
   MqlRates rates[];
   MqlRates zero = {};
   datetime start;      // время текущего бара
   double price;        // последняя цена закрытия

   if(RandomFactor != RANDOM_WALK)
   {
      if(PRTF(CopyRates(_Symbol, PERIOD_M1, From, To, rates)) <= 0)
      {
         return 0; // ошибка
      }
      if(RandomFactor == ORIGINAL)
      {
         return PRTF(CustomRatesReplace(CustomSymbol, From, To, rates));
      }
      ...
```

Важно напомнить, что на *CopyRates* влияет ограничение на количество баров на графике, которое стоит в настройках терминала.

В случае режима ORIGINAL мы просто пересылаем полученный массив *rates* в функцию *CustomRatesReplace*. Для режимов зашумления устанавливаем специально выделенные переменные *price* и *start* в начальные значения цены и времени из первого бара.

```
      price = rates[0].open;
      start = rates[0].time;
   }
   ...
```

В режиме случайного блуждания котировки не нужны, поэтом мы просто выделяем массив *rates* под будущие случайные бары M1.

```
   else
   {
      ArrayResize(rates, (int)((To - From) / 60) + 1);
      price = 1.0;
      start = From;
   }
   ...
```

Далее в цикле по массиву *rates* производится добавление случайных значений либо в зашумляемые цены исходного символа, либо "как есть". В режиме RANDOM_WALK мы сами отвечаем за увеличение времени в переменной *start*. В остальных режимах время уже есть в исходных котировках.

```
   const int size = ArraySize(rates);

   double hlc[3]; // будущие High Low Close (в неизвестном порядке)
   for(int i = 0; i < size; ++i)
   {
      if(RandomFactor == RANDOM_WALK)
      {
         rates[i] = zero;              // обнуление структуры
         rates[i].time = start += 60; // плюс минута к прошлому бару
         rates[i].open = price;        // начинаем с прошлой цены
         hlc[0] = RandomWalk(price);
         hlc[1] = RandomWalk(price);
         hlc[2] = RandomWalk(price);
      }
      else
      {
         double delta = 0;
         if(i > 0)
         {
            delta = rates[i].open - price; // кумулятивная коррекция
         }
         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;
   }
   ...
```

На основе цены закрытия прошлого бара генерируются 3 случайных значения (с помощью функции *RandomWalk*). Максимальное и минимальное из них становятся, соответственно, ценами *High* и *Low* нового бара. Среднее значение — это цена *Close*.

По завершении цикла остается только передать массив в *CustomRatesReplace*.

```
   return PRTF(CustomRatesReplace(CustomSymbol, From, To, rates));
}
```

В функции *RandomWalk* сделана попытка сымитировать распределение с широкими хвостами, что свойственно настоящим котировкам.

```
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;
}
```

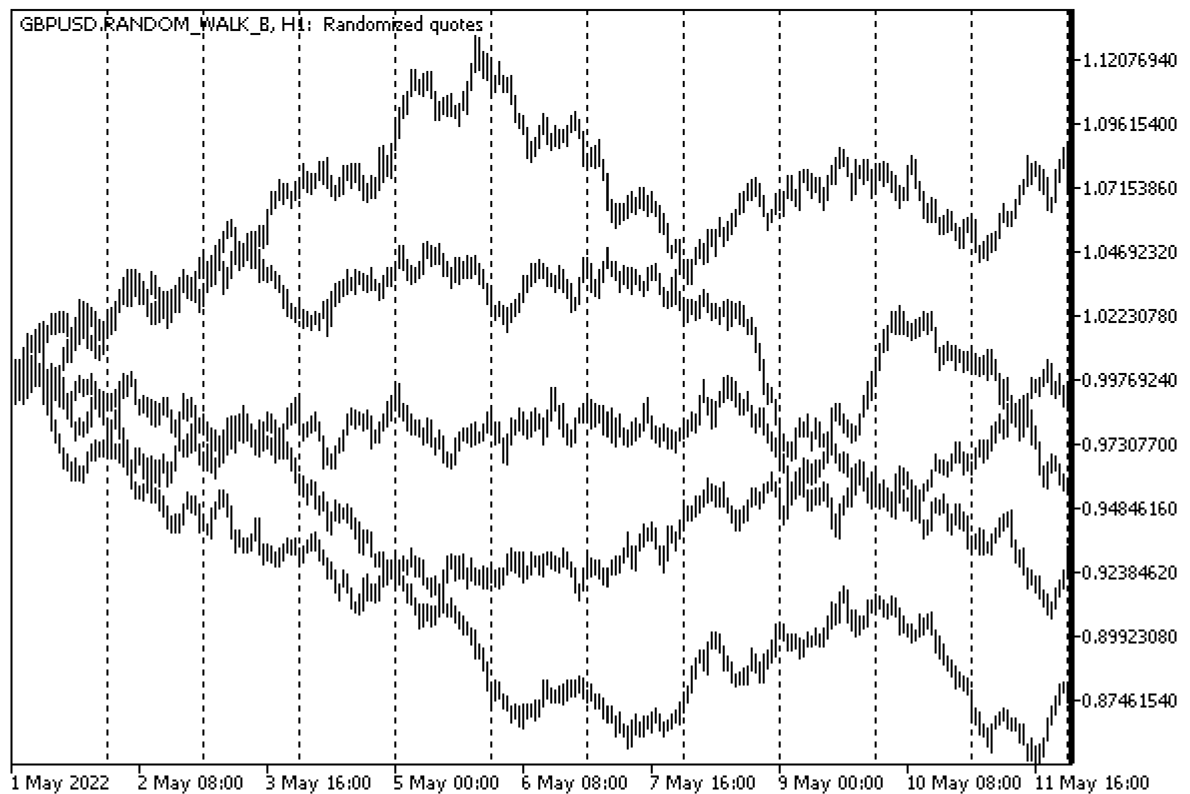
Коэффициенты разброса случайных величин зависят от режима. Например, слабое зашумление добавляет (или отнимает) максимум 1 сотую процента, а сильное — 5 сотых процента от цены.

В процессе работы скрипт выводит подробный лог вроде этого:

```
Create new custom symbol 'GBPUSD.RANDOM_WALK'?
CustomSymbolCreate(CustomSymbol,CustomPath,_Symbol)=true / ok
CustomRatesReplace(CustomSymbol,From,To,rates)=171416 / ok
Bars M1 generated: 171416
```

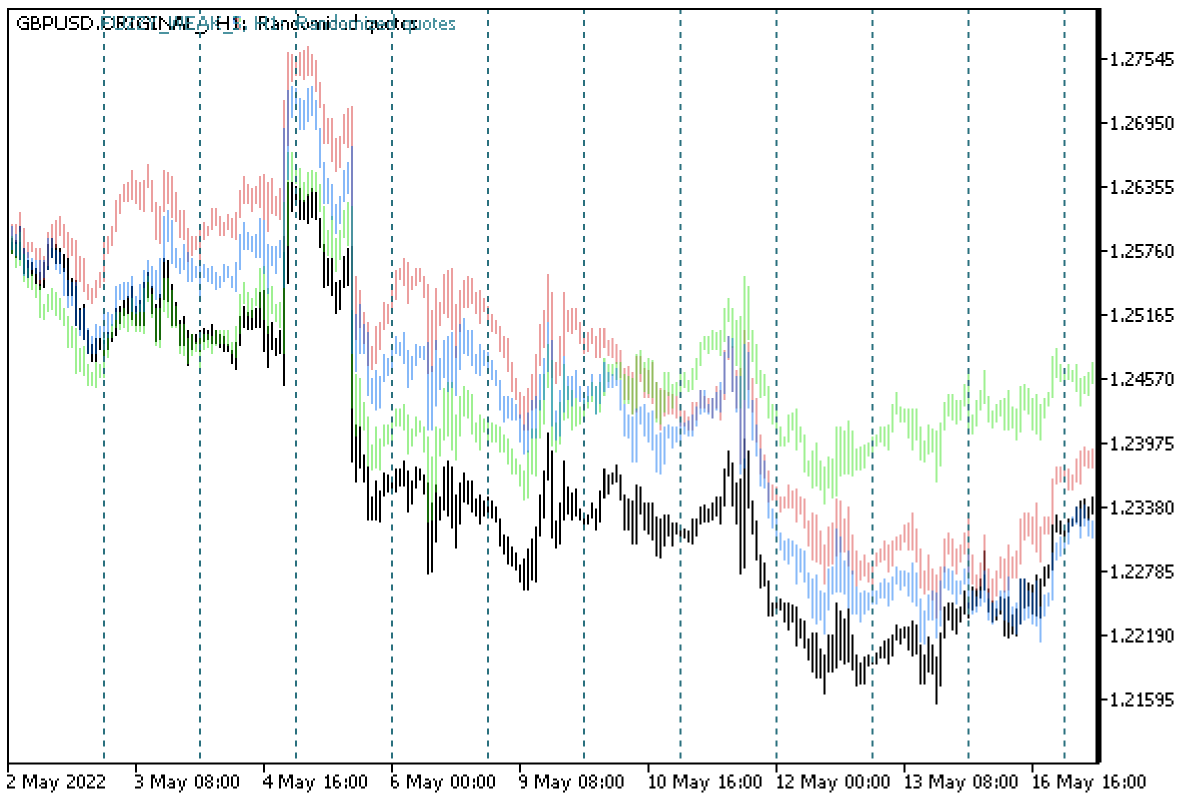
Давайте посмотрим, что получилось в результате.

На следующем изображении показано несколько реализаций случайного блуждания (визуальное наложение выполнено в графическом редакторе, в реальности каждый пользовательский символ открывается, как обычно, в отдельном окне).



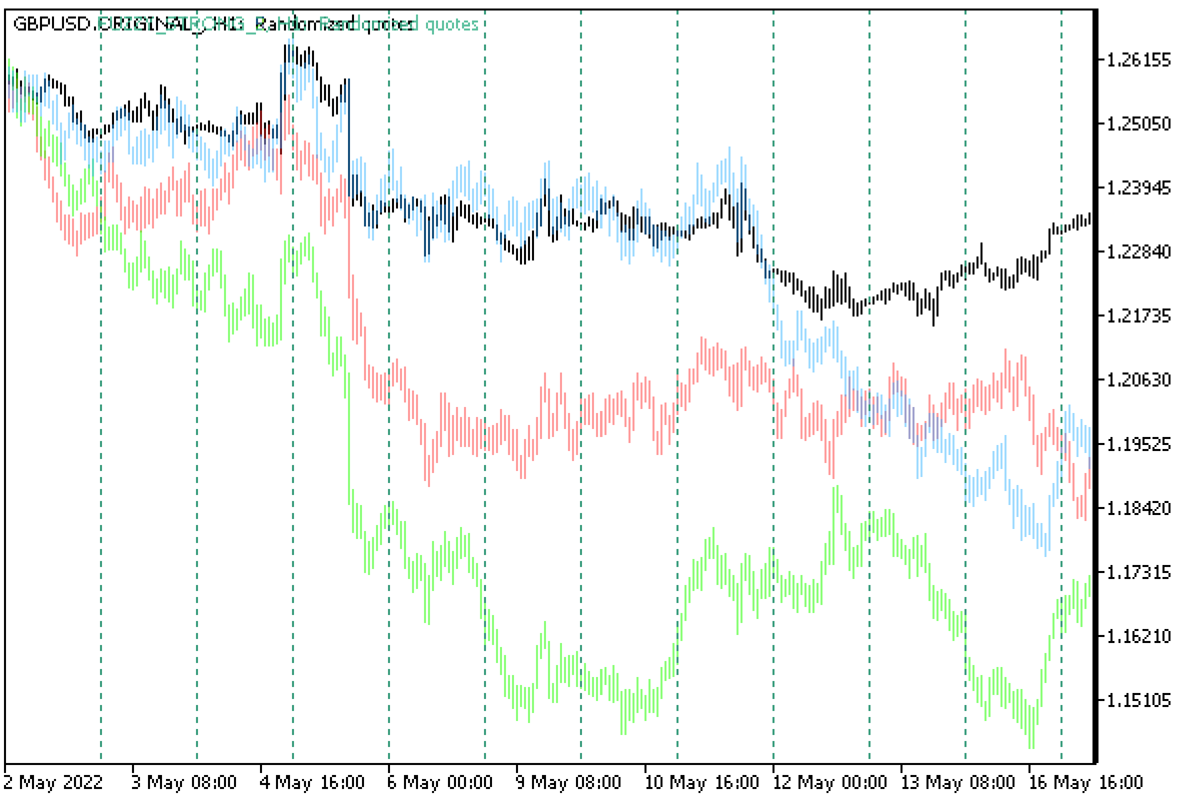
Варианты котировок пользовательских символов со случайным блужданием

А вот как выглядят зашумленные котировки GBPUSD (черным цветом оригинал, цветные с шумом). Сначала в слабом варианте.



Котировки GBPUSD со слабым зашумлением

А затем — в сильном.



Котировки GBPUSD с сильным зашумлением

Налицо б*о*льшие расхождения, но с сохранением локальных особенностей.

7.2.6 Добавление, замена и удаление тиков

MQL5 API позволяет формировать историю пользовательского символа не только на уровне баров, но и тиков. Таким образом, можно добиться большего реализма при тестировании и оптимизации экспертов, а также эмулировать в реальном времени обновление графиков пользовательских инструментов, транслируя на них свои тики. Совокупность переданных системе тиков автоматически учитывается при формировании баров. Иными словами, нет необходимости вызывать функции из предыдущего раздела, оперирующие структурами *MqlRates*, если более детальная информация об изменениях цен за тот же период предоставлена в виде тиков, а именно массивов структур *MqlTick*. Единственное преимущество, которое дают побаровые котировки *MqlRates*, — быстродействие и экономия памяти, когда это в приоритете.

Для добавления тиков существует 2 функции *CustomTicksAdd* и *CustomTicksReplace*. Первая производит добавление интерактивных тиков, которые поступают в окно *Обзора рынка* (и оттуда они автоматически переносятся терминалом в базу тиков) и генерируют соответствующие события в MQL-программах. Вторая — записывает тики напрямую в базу тиков.

int CustomTicksAdd(const string symbol, const MqlTick &ticks[], uint count = WHOLE_ARRAY)

Функция *CustomTicksAdd* добавляет в ценовую историю пользовательского инструмента под именем *symbol* данные из массива *ticks*. По умолчанию, если параметр *count* равен WHOLE_ARRAY, добавляется весь массив, но при необходимости можно указать меньшее количество и загрузить лишь часть тиков.

При этом важно, что пользовательский символ должен быть уже выбран в окне *Обзора рынка* в момент вызова функции. Для символов, не выбранных в *Обзор рынка*, необходимо использовать функцию *CustomTicksReplace* (см. далее).

Массив тиковых данных должен быть упорядочен по времени в порядке возрастания, то есть требуется, чтобы выполнялось условия *ticks[i].time_msc <= ticks[j].time_msc* для всех *i < j*.

Функция возвращает количество добавленных тиков либо -1 в случае ошибки.

Функция *CustomTicksAdd* транслирует тики на график так же, как если бы они приходили от сервера брокера. Обычно функция применяется для одного или нескольких тиков. В этом случае они "проигрываются" в окне *Обзора рынка* и из него сохраняются в базе тиков.

Однако при большом объеме данных, передаваемых за один вызов, функция меняет свое поведение для экономии ресурсов. Если передается более 256 тиков, они делятся на две части. Первая часть (большая) сразу напрямую записывается в базу тиков (как это делает *CustomTicksReplace*). Вторая часть, состоящая из последних (наиболее актуальных) 128 тиков, передается в окно *Обзор рынка* и после этого сохраняется терминалом в базе.

Структура *MqlTick* имеет два поля со значением времени: *time* (время тика в секундах) и *time_msc* (время тика в миллисекундах). Оба значения ведут отсчет от 01 января 1970 года. Заполненное (ненулевое) поле *time_msc* имеет приоритет перед *time*. При этом *time* заполняется в секундах в результате пересчета по формуле *time_msc / 1000*. Если поле *time_msc* равно нулю, используется время из поля *time*, причем поле *time_msc* в свою очередь получает значение в миллисекундах из формулы *time * 1000*. Если оба поля равны нулю, в тик проставляется текущее время сервера (с точностью до миллисекунд).

Из двух полей, описывающих объем, *volume_real* имеет высший приоритет по сравнению с *volume*.

В зависимости от того, какие другие поля заполнены в конкретном элементе массива (структуре *MqlTick*), система устанавливает для сохраняемого тика флаги в поле *flags*:

- ticks[i].bid — TICK_FLAG_BID (тик изменил цену Bid)
- ticks[i].ask — TICK_FLAG_ASK (тик изменил цену Ask)
- ticks[i].last — TICK_FLAG_LAST (тик изменил цену последней сделки)
- ticks[i].volume или ticks[i].volume_real — TICK_FLAG_VOLUME (тик изменил объем)

Если значение какого-то поля меньше или равно нулю, соответствующий ему флаг не записываются в поле *flags*.

Флаги TICK_FLAG_BUY и TICK_FLAG_SELL в историю пользовательского инструмента не добавляются.

Функция *CustomTicksReplace* полностью заменяет ценовую историю пользовательского инструмента в указанном временном интервале данными из передаваемого массива.

```
int CustomTicksReplace(const string symbol, long from_msc, long to_msc,
  const MqlTick &ticks[], uint count = WHOLE_ARRAY)
```

Интервал задается параметрами *from_msc* и *to_msc*, в миллисекундах с 01.01.1970. Оба значения входят в интервал.

Массив *ticks* должен быть упорядочен в хронологическом порядке прихода тиков, что соответствует возрастанию, а точнее, неубыванию времени, так как в потоке тиков нередко подряд идут тики с одним и тем же временем с точностью до миллисекунды.

Параметр *count* позволяет обрабатывать не весь массив, а лишь его часть.

Замена тиков производится последовательно день за днём до времени указанного в *to_msc* либо до возникновения ошибки в очередности тиков. Сначала обрабатывается первый день из указанного диапазона, затем следующий, и так далее. Как только обнаружится несоответствие времени тика порядку возрастания (неубывания), то процесс замены тиков прекращается на текущем дне. При этом тики за предыдущие дни будут успешно заменены, а текущий день (на момент неправильного тика) и все оставшиеся дни в указанном интервале останутся без изменения. Функция вернет -1, причем код ошибки в *_LastError* равен 0 ("нет ошибки").

Если в массиве *ticks* отсутствуют данные за какой-то период внутри общего интервала от *from_msc* до *to_msc* (включительно), то после выполнения функции в истории пользовательского инструмента образуется "дыра", соответствующая пропущенным данным.

Если в базе тиков в указанном интервале времени данные отсутствуют, то *CustomTicksReplace* просто добавит в нее тики из массива *ticks*.

Удалить все тики в указанном временном интервале позволяет функция *CustomTicksDelete*.

```
int CustomTicksDelete(const string symbol, long from_msc, long to_msc)
```

Имя редактируемого пользовательского инструмента задается в параметре *symbol*, а очищаемый интервал — параметрами *from_msc* и *to_msc* (включительно), в миллисекундах.

Функция возвращает количество удаленных тиков либо -1 в случае ошибки.

> Внимание! Удаление тиков с помощью *CustomTicksDelete* приводит к автоматическому удалению соответствующих баров! Однако вызов *CustomRatesDelete*, то есть удаление баров, не удаляет тики!

Для освоения материала на практике решим с помощью новых функций несколько прикладных задач.

Для начала коснемся такой интересной задачи, как создание пользовательского инструмента на основе реального символа, но с прореживанием тиков. Это позволит добиться ускорения тестирования и оптимизации, а также снизит потребление ресурсов (в первую очередь, оперативной памяти) по сравнению с режимом по реальным тикам, оставив при этом приемлемое, близкое к идеальному, качество процесса.

Ускорение тестирования и оптимизации

Трейдеры часто задаются вопросом, каким образом можно ускорить оптимизацию и тестирование экспертов. Среди возможных решений есть очевидные, для которых достаточно изменить настройки (когда это допустимо), а есть более трудоемкие, которые требуют адаптации эксперта или тестовой среды.

Среди первого класса решений можно отметить:

· уменьшение пространства оптимизации за счет исключения некоторых параметров или уменьшения их шага;
· уменьшение срока оптимизации;
· переход на режим моделирования тиков более низкого качества (например, от реальных к OHLC M1);
· включение опции расчета прибыли в пунктах, вместо денег;
· апгрейд компьютера;
· использование MQL Cloud или дополнительных компьютеров локальной сети.

Среди второго класса решений, связанных с разработкой, упомянем:

· профилировку кода, на основе которой можно ликвидировать "узкие" места в коде;
· по возможности, использовать экономный расчет индикаторов — без директивы *#property tester_everytick_calculate*;
· перенос алгоритмов индикаторов (если они используются) непосредственно в код советника: вызовы индикаторов налагают определенные накладные расходы;
· исключение работы с графикой и объектами;
· кэширование расчетов, если возможно;
· уменьшение количества одновременно открытых позиций и выставленных ордеров (их обсчет на каждом тике может стать заметным при большом числе);
· полная виртуализация расчетов, ордеров, сделок и позиций: встроенный механизм денежного учета в силу своей универсальности, поддержки мультивалютности и прочих особенностей имеет свои накладные расходы, которые можно исключить, выполняя аналогичные действия в коде на MQL5 (хотя этот вариант наиболее трудоемок);

Прореживание тиков относится к промежуточному классу решений: оно требует программного создания пользовательского символа, но зато не затрагивает исходный код эксперта.

Пользовательский символ с прореженными тиками будет создаваться скриптом *CustomSymbolFilterTicks.mq5*. Исходным инструментом будет выступать рабочий символ графика,

на котором запускается скрипт. Во входных параметрах можно указать папку для пользовательского символа и начальную дату обработки истории. По умолчанию, если дата не задана, делается расчет для последних 120 дней.

```
input string CustomPath = "MQL5Book\\Part7"; // Custom Symbol Folder
input datetime _Start;                        // Start (default: 120 days back)
```

Название символа формируется из имени исходного инструмента и суффикса ".TckFltr". Позднее мы добавим к нему обозначение метода прореживания тиков.

```
string CustomSymbol = _Symbol + ".TckFltr";
const uint DailySeconds = 60 * 60 * 24;
datetime Start = _Start == 0 ? TimeCurrent() - DailySeconds * 120 : _Start;
```

Для удобства пользователя, в обработчике *OnStart* предусмотрена возможность удалить предыдущую копию символа, если она уже существует.

```
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;
      }
   }
```

Далее с согласия пользователя создается символ. Заполнение истории тиковыми данными производится во вспомогательной функции *GenerateTickData*. В случае успеха, скрипт добавляет новый символ в *Обзор рынка* и открывает график.

```
      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);
            }
         }
      }
   }
```

Функция *GenerateTickData* обрабатывает тики в цикле порциями, по суткам. Тики за сутки запрашиваются с помощью вызова *CopyTicksRange*. Далее их надо тем или иным способом проредить, что делегировано классу *TickFilter*, который мы покажем ниже. Наконец, массив тиков добавляется в историю пользовательского символа с помощью *CustomTicksReplace*.

```
bool GenerateTickData()
{
   bool result = true;
   datetime from = Start / DailySeconds * DailySeconds; // округляем до начала суток
   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;
}
```

На всех этапах проводится контроль ошибок и подсчет обработанных тиков. В конце выводим в журнал количество исходных тиков и оставшихся, а также коэффициент "сжатия".

Теперь обратимся непосредственно к методике прореживания тиков. Очевидно, что подходов может быть много, и каждый лучше или хуже подойдет к конкретной торговой стратегии. Мы предложим 3 базовых варианта, объединенных в классе *TickFilter* (*TickFilter.mqh*). Также, для полноты картины, там поддержан и режим копирования тиков без прореживания.

Таким образом, в классе реализованы следующие режимы:

- без прореживания;
- пропуск последовательностей тиков с монотонным изменением цены без разворота (а-ля "зиг-заг");
- пропуск колебаний цен в пределах спреда;
- запись только тиков с фрактальной конфигурацией, когда цена *Bid* или *Ask* представляет собой экстремум между двумя соседними тиками.

Данные режимы описаны в виде элементов перечисления FILTER_MODE.

```
class TickFilter
{
public:
   enum FILTER_MODE
   {
      NONE,
      SEQUENCE,
      FLUTTER,
      FRACTALS,
   };
   ...
```

Каждый из режимов реализован отдельным статическим методом, принимающим на вход массив тиков, который необходимо проредить. Редактирование массива производится по месту (без выделения нового выходного массива).

```
   static int filterBySequences(MqlTick &data[]);
   static int filterBySpreadFlutter(MqlTick &data[]);
   static int filterByFractals(MqlTick &data[]);
```

Все методы возвращают количество оставшихся тиков (уменьшенный размер массива).

Для унификации выполнения процедуры в разных режимах предусмотрен метод *filter*. Для режима NONE массив *data* просто остается без изменений.

```
   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);
   }
```

Например, вот как реализована фильтрация по монотонным последовательностям тиков в методе *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);
}
```

А вот как выглядит прореживание по фракталам.

```
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);
}
```

Давайте последовательно создадим пользовательский символ для EURUSD в нескольких режимах прореживания и сравним их показатели, то есть степень "сжатия", насколько ускорится тестирование и как изменятся торговые показатели эксперта.

Например, прореживание последовательностей тиков дает такие результаты (для полуторагодичной истории на MQ Demo).

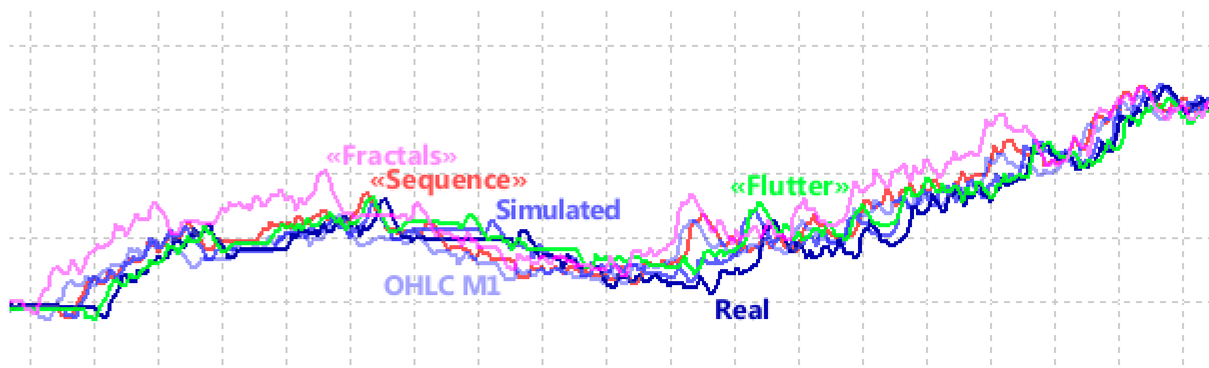
```
Create new custom symbol 'EURUSD.TckFltr-SE'?
Fixing SYMBOL_TRADE_TICK_VALUE: 0.0 <<< 1.0
true  SYMBOL_TRADE_TICK_VALUE 1.0 -> SUCCESS (0)
Fixing SYMBOL_TRADE_TICK_SIZE: 0.0 <<< 1e-05
true  SYMBOL_TRADE_TICK_SIZE 1e-05 -> SUCCESS (0)
Number of found discrepancies: 2
Fixed
Done ticks - read: 31553509, written: 16927376, ratio: 53.6%
```

Для режимов сглаживания колебаний и по фракталам показатели другие:

```
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%
```

Для практических торговых экспериментов на основе прореженных тиков нам потребуется эксперт. Возьмем адаптированную версию *BandOsMAticks.mq5*, в которой по сравнению с оригиналом включена торговля на каждом тике (в методе *SimpleStrategy::trade* отключена строка *if(lastBar == iTime(_Symbol, _Period, 0)) return false;*), а значения сигнальных индикаторов берутся с баров 0 и 1 (раньше были только завершенные бары 1 и 2).

Запустим эксперт на диапазоне дат с начала 2021 года по 1 июня 2022. Настройки прилагаются в файле *MQL5/Presets/MQL5Book/BandOsMAticks.set*. Общее поведение кривой баланса во всех режимах достаточно схожее.



Совмещенные графики балансов тестов в разных режимах по тикам

Смещение эквивалентных экстремумов разных кривых по горизонтали вызвано тем, что стандартный график отчета использует для горизонтальной координаты не время, а количество трейдов, которое, разумеется, отличается из-за точности срабатывания торговых сигналов по разным базам тиков.

Различия в показателях приведены в следующей таблице (N — количество трейдов, $ — прибыль, PF — профит-фактор, RF — фактор восстановления, DD — просадка):

Режим	Тики	Время mm:ss.msec	Память	N	$	PF	RF	DD
Реальные	31002919	02:45.251	835 Mb	962	166.24	1.32	2.88	54.99
Эмуляция	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
Sequence	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

Будем считать тест по реальным тикам наиболее достоверным и оценивать остальные по степени близости к нему. Очевидно, что наибольшую скорость и меньшие затраты ресурсов за счет существенной потери точности показал режим OHLC M1 (режим по ценам открытия не рассматривался). У него сверх-оптимистичные финансовые результаты.

Среди трех режимов с искусственно прореженными тиками наиболее близким к реальному по комплексу показателей является Sequence. По времени он в 2 раза быстрее реального, а по памяти в 1.5 раза экономичнее. Режим Flutter, судя по всему, лучше сохраняет оригинальное количество сделок. Наиболее быстрый и наименее требовательный к памяти режим по фракталам, конечно, проигрывает OHLC M1, но зато не завышает торговые оценки.

Следует иметь в виду, что алгоритмы прореживания тиков могут по-разному подходить или, наоборот, давать плохие результаты с различными торговыми стратегиями, финансовыми инструментами и даже тиковой историей конкретного брокера. Проводите исследования с вашими экспертами и в вашей рабочей среде.

В рамках второго примера работы с пользовательскими символами рассмотрим интересную возможность, которую предоставляет трансляция тиков с помощью *CustomTicksAdd*.

Как известно, многие трейдеры любят применять в своей практике торговые панели — программы с интерактивными элементами управления для выполнения произвольных торговых действий вручную. Отрабатывать навыки работы с ними приходится в основном в режиме онлайн, потому что тестер налагает некоторые ограничения. Прежде всего, в тестере не поддерживаются события на графике и нанесенных на него объектах. Из-за этого элементы управления перестают функционировать. Также в тестере нельзя наносить произвольные объекты для графической разметки.

Попробуем решить эти проблемы собственными силами.

Мы можем генерировать пользовательский символ по историческим тикам в замедленном режиме. Тогда график такого символа станет представлять собой аналог визуального тестера.

Данный подход имеет несколько преимуществ:

- стандартное поведение всех событий чарта;
- интерактивное нанесение и настройка индикаторов;
- интерактивное нанесение и настройка объектов;
- переключение таймфрейма на лету;
- тест на истории вплоть до текущего времени, включая сегодняшний день (стандартный тестер не позволяет тестировать сегодня).

По поводу последнего пункта отметим, что разработчики MetaTrader 5 намеренно запретили проверку торговли на последнем (текущем) дне, хотя она бывает нужна для оперативного поиска ошибок (в коде или в торговой стратегии).

Также потенциально интересна модификация цен на лету (увеличение спреда, например).

На основе графика подобного кастом-символа мы сможем позднее реализовать эмулятор ручной торговли на исторических данных.

Генератором символа будет неторгующий эксперт *CustomTester.mq5*. В его входных параметрах предусмотрим указание размещения нового кастом-символа в иерархии символов, начальную дату в прошлом для трансляции тиков (и построения котировок кастом-символа), а также таймфрейм для графика, который автоматически будет открыт для визуального тестирования.

```
input string CustomPath = "MQL5Book\\Part7"; // Custom Symbol Folder
input datetime _Start;                       // Start (по умолчанию: отступ на 120 дней)
input ENUM_TIMEFRAMES Timeframe = PERIOD_H1;
```

Имя нового символа конструируется из имени символа текущего графика и суффикса ".Tester".

```
string CustomSymbol = _Symbol + ".Tester";
```

Если начальная дата в параметрах не задана, эксперт сделает отступ назад на 120 дней от текущей даты.

```
const uint DailySeconds = 60 * 60 * 24;
datetime Start = _Start == 0 ? TimeCurrent() - DailySeconds * 120 : _Start;
```

Тики будут считываться из истории реальных тиков рабочего символа пакетами сразу за целый день. Указатель на считываемый день хранится в переменной Cursor.

```
bool FirstCopy = true;
// дополнительно 1 день назад, потому что иначе график не сразу обновится
datetime Cursor = (Start / DailySeconds - 1) * DailySeconds; // округляем по границе сутс
```

Подлежащие воспроизведению тики одних суток будут запрашиваться в массив *Ticks*, откуда мелкими порциями размером *Step* транслироваться на график кастом-символа.

```
MqlTick Ticks[];       // тики для "текущего" дня в прошлом
int Index = 0;         // позиция в тиках внутри дня
int Step = 32;         // перемотка вперед по 32 тика за раз (по умолчанию)
int StepRestore = 0;   // запоминаем скорость на время паузы
long Chart = 0;        // созданный график кастом-символа
bool InitDone = false; // признак завершенной инициализации
```

Для воспроизведения тиков с постоянной скоростью запустим таймер в *OnInit*.

```
void OnInit()
{
   EventSetMillisecondTimer(100);
}

void OnTimer()
{
   if(!GenerateData())
   {
      EventKillTimer();
   }
}
```

Генерацию тиков поручим функции *GenerateData*. Сразу после запуска, когда флаг *InitDone* сброшен, попытаемся создать новый символ или очистить прежние котировки и тики, если кастом-символ уже существует.

```
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)
```

На данном этапе мы кое-что опустим в месте (A) и вернемся к данному моменту позднее.

После создания символа, выбираем его в *Обзор рынка* и открываем для него чарт.

```
 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;
   }
   ...
```

Здесь тоже пропущена пара строк (B), связанных с будущими усовершенствованиями, но пока они не требуются.

Если символ уже создан, запускаем трансляцию тиков пакетами *Step* тиков, но не более 256. Это ограничение связано с особенностью функции *CustomTicksAdd*.

```
      else
      {
         for(int i = 0; i <= (Step - 1) / 256; ++i)
         if(Step > 0 && !GenerateTicks())
         {
            return false;
         }
      }
      return true;
   }
```

Вспомогательная функция *GenerateTicks* транслирует тики порциями по *Step* тиков (но не более 256), считывая их из суточного массива *Ticks* по смещению *Index*. Когда массив пуст или мы прочитали его до конца, запрашиваем тики следующего дня путем вызова *FillTickBuffer*.

```
   bool GenerateTicks()
   {
      if(Index >= ArraySize(Ticks)) // суточный массив пуст или прочитан до конца
      {
         if(!FillTickBuffer()) return false; // заполняем массив тиками за сутки
      }
      
      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); // на случай ERR_CUSTOM_TICKS_WRONG_ORDER (5310)
         ExpertRemove();
      }
      Comment("Speed: ", (string)Step, " / ", STR_TIME_MSC(array[n - 1].time_msc));
      Index += Step; // перемещаемся на Step тиков вперед
      return true;
   }
```

Функция *FillTickBuffer* использует в своей работе *CopyTicksRange*.

```
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: этот предварительный вызов нужен только, чтобы вывести график
         // из состояния "Ожидания Обновления"
         PRTF(CustomTicksReplace(CustomSymbol, Cursor * 1000L,
            (Cursor + DailySeconds) * 1000L - 1, Ticks));
         FirstCopy = false;
         r = 0;
      }
      Cursor += DailySeconds;
   }
   while(r == 0 && Cursor < TimeCurrent()); // пропускаем неторговые дни
   Index = 0;
   return r > 0;
}
```

При остановке эксперта будем также закрывать зависимый график (чтобы он не дублировался при следующем запуске).

```
void OnDeinit(const int)
{
   if(Chart != 0)
   {
      ChartClose(Chart);
   }
   Comment("");
}
```

На этом эксперт можно было бы считать завершенным, однако существует одна проблема. Дело в том, что по тем или иным причинам свойства кастом-символа не копируются один в один из исходного рабочего символа, по крайней мере, в текущей реализации MQL5 API. Это касается даже очень важных свойств, таких как SYMBOL_TRADE_TICK_VALUE, SYMBOL_TRADE_TICK_SIZE. Если мы выведем на печать значения этих свойств сразу после вызова *CustomSymbolCreate(CustomSymbol, CustomPath, _Symbol)*, то увидим там нули.

Чтобы организовать проверку свойств, их сравнение и при необходимости исправление был написан специальный класс *CustomSymbolMonitor* (*CustomSymbolMonitor.mqh*), производный от *SymbolMonitor*. С его внутренним устройством предлагается разобраться самостоятельно, а здесь мы лишь приведем публичный интерфейс.

Конструкторы позволяют создать монитор пользовательского символа с указанием образцового рабочего символа (по имени в строке или из объекта *SymbolMonitor*), который служит источником настроек.

```
class CustomSymbolMonitor: public SymbolMonitor
{
public:
   CustomSymbolMonitor(); // образец - _Symbol
   CustomSymbolMonitor(const string s, const SymbolMonitor *m = NULL);
   CustomSymbolMonitor(const string s, const string other);

   // установить/заменить символ-образец
   void inherit(const SymbolMonitor &m);

   // скопировать все свойства из символа-образца в прямом или обратном порядке
   bool setAll(const bool reverseOrder = true, const int limit = UCHAR_MAX);

   // сверить все свойства с образцом, вернуть количество исправлений
   int verifyAll(const int limit = UCHAR_MAX);

   // сверить указанные свойства с образцом, вернуть количество исправлений
   int verify(const int &properties[]);

   // скопировать заданные свойства из образца, вернуть true если все они применились
   bool set(const int &properties[]);

   // скопировать конкретное свойство из образца, вернуть true если оно применилось
   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);
   }
};
```

Поскольку кастом-символы, в отличие от стандартных символов, позволяют задавать свои свойства, в классе добавлена тройка *set*-методов. Они, в частности, используются для пакетного переноса свойств символа-образца и проверки успешности этих действий в других методах класса.

Теперь мы можем вернуться к генератору кастом-символа и фрагменту его исходного кода, обозначенного ранее комментарием (A).

```
      // (A) проверяем важные свойства и устанавливаем их в "ручном" режиме
      SymbolMonitor sm; // _Symbol
      CustomSymbolMonitor csm(CustomSymbol, &sm);
      int props[] = {SYMBOL_TRADE_TICK_VALUE, SYMBOL_TRADE_TICK_SIZE};
      const int d1 = csm.verify(props); // проверяем и пытаемся исправить
      if(d1)
      {
         Print("Number of found descrepancies: ", d1); // число исправлений
         if(csm.verify(props)) // проверяем еще раз
         {
            Alert("Custom symbol can not be created, internal error!");
            return false; // без успешных правок использовать символ нельзя
         }
         Print("Fixed");
      }
```

Сейчас эксперт *CustomTester.mq5* уже можно запускать и наблюдать, как в автоматически открытом графике динамически формируются котировки, а в окне *Обзора рынка* пробрасываются тики из истории.

Однако это делается с постоянной скоростью 32 тика за 0.1 секунды. Желательно на лету менять скорость воспроизведения по желанию пользователя, как в большую, так и в меньшую сторону. Такое управление можно организовать, например, с клавиатуры.

Следовательно, требуется добавить обработчик *OnChartEvent*. Как мы знаем, для события CHARTEVENT_KEYDOWN в программу поступает код нажатой клавиши в параметре *lparam*, и мы передаем его в функцию *CheckKeys* (см. ниже). Некий фрагмент (C), тесно связанный с (B), пока пришлось отложить — мы к ним скоро вернемся.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strir
{
   ... // (C)
   if(id == CHARTEVENT_KEYDOWN) // эти события поступают только пока график активен!
   {
      CheckKeys(lparam);
   }
}
```

В функции *CheckKeys* обрабатываем клавиши "стрелка вверх" и "стрелка вниз" для увеличения и уменьшения скорости воспроизведения. Кроме того, клавиша "пауза" позволяет и совсем приостановить процесс "тестирования" (трансляции тиков). Повторное нажатие "паузы" возобновляет работу с прежней скоростью.

```
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);
      }
   }
}
```

Новый код можно проверить в действии, предварительно убедившись, что активным является график, на котором работает эксперт. Напомним, что события клавиатуры поступают только в активное окно. В этом кроется еще одна проблема нашего тестера.

Поскольку пользователь должен выполнять торговые действия на графике кастом-символа, окно с генератором практически всегда будет находиться в фоновом режиме. Переключаться на окно генератора, чтобы на время остановить поток тиков, а потом возобновить — не практично. Поэтому требуется неким образом организовать интерактивное управление с клавиатуры непосредственно из окна кастом-символа.

Для этой цели подойдет специальный индикатор, который мы можем автоматически добавить в открываемое окно кастом-символа. Индикатор будет перехватывать события клавиатуры в своем окне (окне с кастом-символом) и посылать их в окно генератора.

Исходный код индикатора прилагается в файле *KeyboardSpy.mq5*. Разумеется, индикатор не имеет диаграмм. Пара входных параметров предназначена для получения идентификатора графика *HostID*, куда следует отправлять сообщения, и кода пользовательского события *EventID*, в которое будут "упаковываться" интерактивные события.

```
#property indicator_chart_window
#property indicator_plots 0

input long HostID;
input ushort EventID;
```

Основная "работа" выполняется в обработчике *OnChartEvent* — она элементарна.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strir
{
   if(id == CHARTEVENT_KEYDOWN)
   {
      EventChartCustom(HostID, EventID, lparam,
         // здесь всегда 0, когда внутри iCustom
         (double)(ushort)TerminalInfoInteger(TERMINAL_KEYSTATE_CONTROL),
         sparam);
   }
}
```

Обратите внимание, что все выбранные нами "горячие клавиши" являются простыми, то есть не используют сочетаний с клавишами состояния клавиатуры, такими как *Ctrl* или *Shift*. Это сделано вынужденно, потому что внутри индикаторов, созданных программно (в частности, через *iCustom*), состояние клавиатуры не считывается. Иными словами, вызов *TerminalInfoInteger(TERMINAL_KEYSTATE_XYZ)* всегда возвращает 0. В вышеприведенном обработчике мы добавили его просто для демонстрации, чтобы вы могли при желании убедиться в данном ограничении, выведя поступающие параметры на "принимающей стороне".

Однако одиночные нажатия "стрелок" и "паузы" будут передаваться в "родительский" чарт нормально, и нам этого достаточно. Дело осталось за малым: интегрировать индикатор с экспертом.

В пропущенном ранее фрагменте (B), во время инициализации генератора, создадим индикатор и добавим его на график кастом-символа.

```
#define EVENT_KEY 0xDED // пользовательское событие
      ...
      // (B)
      const int handle = iCustom(CustomSymbol, Timeframe, "MQL5Book/p7/KeyboardSpy",
         ChartID(), EVENT_KEY);
      ChartIndicatorAdd(Chart, 0, handle);
```

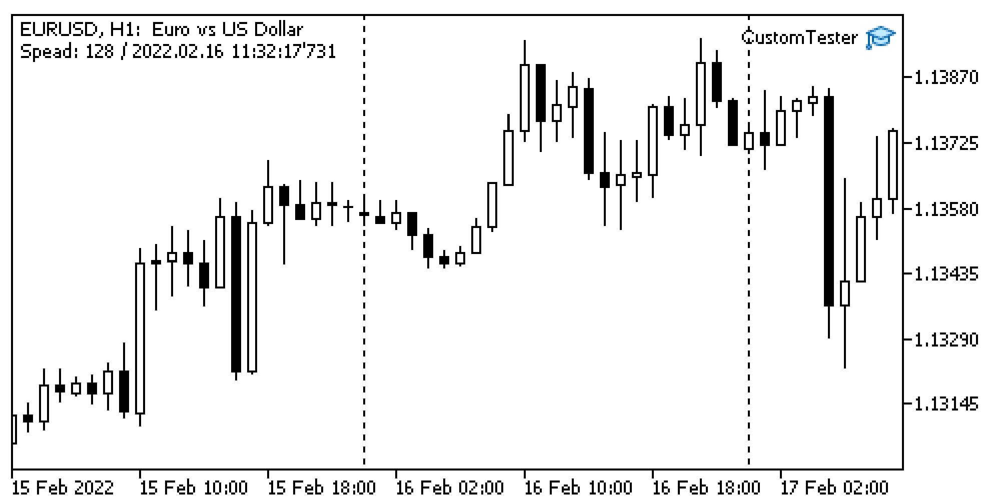
Далее во фрагменте (C) обеспечим прием пользовательских сообщений из индикатора и их передачу в уже известную функцию *CheckKeys*.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strin
{
   // (C)
   if(id == CHARTEVENT_CUSTOM + EVENT_KEY) // уведомления с зависимого чарта, когда с
   {
      CheckKeys(lparam); // "дистанционная" обработка нажатий клавиш
   }
   else if(id == CHARTEVENT_KEYDOWN) // эти события поступают только пока график акти
   {
      CheckKeys(lparam); // стандартная обработка
   }
}
```

Таким образом, управлять скоростью воспроизведения теперь можно как на графике с экспертом, так и на генерируемом им графике кастом-символа.

С новым инструментарием вы можете попробовать интерактивную работу с графиком, "живущим прошлой жизнью". На график выводится комментарий с текущей скоростью воспроизведения или метка паузы.

На графике с экспертом в комментарий выводится время "текущих" транслируемых тиков.



Эксперт, воспроизводящий историю тиков (и котировок) реального символа

В этом окне пользователю в принципе нечего делать (если только удалить эксперт и прекратить генерацию кастом-символа). Сам процесс трансляции тиков здесь не виден. Более того, поскольку эксперт автоматически открывает чарт кастом-символа (где и обновляются исторические котировки), то именно тот становится активным. А для получения

вышеприведенного скриншота нам потребовалось специально ненадолго переключиться на исходный график.

Поэтому вернемся на график кастом-символа. То, как он плавно и поступательно обновляется в прошлом времени, — это уже здорово, но на нем нельзя проводить торговые эксперименты. Например, если набросить на него привычную торговую панель, её элементы управления хоть и будут формально работать, не приведут к сделкам — из-за отсутствия кастом-символа на сервере, получим ошибки. Как мы знаем, эта особенность проявляется у любых программ, которые специально не адаптированы для кастом-символов. Покажем пример того, как можно виртуализировать торговлю кастом-символом.

Вместо торговой панели (в целях упрощения примера, но без потери общности) возьмем за основу максимально простой эксперт *CustomOrderSend.mq5*, который умеет выполнять несколько торговых действий по нажатию клавиш:

- 'B' — покупка по рынку;
- 'S' — продажа по рынку;
- 'U' — установка лимитного ордера на покупку;
- 'L' — установка лимитного ордера на продажу;
- 'C' — закрыть все позиции;
- 'D' — удалить все ордера;
- 'R' — вывести в журнал торговый отчет.

Во входных параметрах эксперта зададим объем одной сделки (по умолчанию, минимальный лот) и расстояние до уровней стоп-лосс и тейк-профит в пунктах.

```
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
```

Если *Distance2SLTP* оставлен равным нулю, защитные уровни в рыночных ордерах на проставляются, а отложенные ордера не формируются. Когда *Distance2SLTP* имеет ненулевое значение, именно оно используется как расстояние от текущей цены при установке отложенного ордера (либо вверх, либо вниз, в зависимости от команды).

С учетом ранее представленных классов из *MqlTradeSync.mqh*, вышеописанная логика преобразуется в следующий исходный код.

```
#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 strir
{
   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:
            // тут что-то должно быть...
            break;
         }
      }
   }
```

Как мы видим, здесь используются как стандартные функции торгового API, так и методы *MqlTradeRequestSync*, которые, опосредованно, также в конечном счете вызывают множество встроенных функций. Нам необходимо "заставить" этот эксперт торговать кастом-символом.

Наиболее простая, хотя и трудоемкая идея заключатся в том, чтобы подменить все стандартные функции на собственные аналоги, которые вели бы подсчет ордеров, сделок, позиций и финансовых показателей в неких структурах. Разумеется, такое возможно только в случаях, когда мы имеем исходный код эксперта, который следует адаптировать.

Экспериментальная реализация подхода продемонстрирована в прилагаемом файле *CustomTrade.mqh*. С полным кодом можно ознакомиться самостоятельно, а в рамках книги мы перечислим лишь основные моменты.

Прежде всего отметим, что многие расчеты сделаны в упрощенном виде, многие режимы не поддерживаются, и не делается полная проверка данных на корректность. Используйте исходный код, как отправную точку для собственных разработок.

Весь код обернут в пространство имен *CustomTrade* для исключения конфликтов.

Сущности ордер, сделка и позиция формализованы в виде соответствующих классов *CustomOrder*, *CustomDeal*, *CustomPosition*. Все они являются наследниками класса *MonitorInterface<I,D,S>::TradeState*. Напомним, что в этом классе уже автоматически поддержано формирование массивов целочисленных, вещественных и строковых свойств для каждого типа объектов и его специфических троек перечислений. Например, *CustomOrder* выглядит так:

```
class CustomOrder: public MonitorInterface<ENUM_ORDER_PROPERTY_INTEGER,
   ENUM_ORDER_PROPERTY_DOUBLE,ENUM_ORDER_PROPERTY_STRING>::TradeState
{
   static long ticket; // счетчик ордеров и постащик тикетов
   static int done;    // счетчик исполненных (исторических) ордеров
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: пока нет отложенного исполнения
         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;
   }
};
```

Обратите внимание, что в виртуальном окружении старого "текущего" времени нельзя использовать функцию *TimeCurrent* и вместо неё берется последнее известное время кастом-символа *SymbolInfoInteger(symbol, SYMBOL_TIME)*.

В процессе виртуальной торговли текущие объекты и их история накапливается в массивах соответствующих классов.

```
AutoPtr<CustomOrder> orders[];
CustomOrder *selectedOrders[];
CustomOrder *selectedOrder = NULL;
AutoPtr<CustomDeal> deals[];
CustomDeal *selectedDeals[];
CustomDeal *selectedDeal = NULL;
AutoPtr<CustomPosition> positions[];
CustomPosition *selectedPosition = NULL;
```

Метафора выделения ордеров, сделок и позиций потребовалась для имитации аналогичного подхода во встроенных функциях. Для них в пространстве имен *CustomTrade* созданы "двойники", которые подменяют оригиналы с помощью директив макро-подстановки.

```
#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
```

Вот, например, как реализована функция *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;
}
```

Как нетрудно заметить, все функции этой группы имеют префикс "MT5", чтобы было сразу понятно их "двойное" назначение и легко было отличить от функций второй группы.

Вторая группа функций в пространстве *CustomTrade* выполняет утилитарные действия: проверяет и обновляет состояния ордеров, сделок и позиций, создает новые и удаляет старые объекты в соответствии с обстановкой. В частности, среди них есть функции *CheckPositions* и *CheckOrders*, которые можно вызывать по таймеру или в ответ на действия пользователя. Но это можно и не делать, если использовать пару других функций, предназначенных для отображения текущего и исторического состояния виртуального торгового счета:

- *string ReportTradeState()* — возвращает многострочный текст со списком открытых позиций и выставленных ордеров;
- *void PrintTradeHistory()* — выводит в журнал историю ордеров и сделок.

Эти функции самостоятельно вызывают *CheckPositions* и *CheckOrders*, чтобы предоставить вам актуальную информацию.

Кроме того имеется функция для визуализации позиций и действующих ордеров на графике в виде объектов: *DisplayTrades*.

Заголовочный файл *CustomTrade.mqh* следует подключать в эксперт до прочих заголовков, чтобы подмена макросов имела эффект во всех последующих строках исходных кодов.

```
#include <MQL5Book/CustomTrade.mqh>
#include <MQL5Book/MqlTradeSync.mqh>
```

Этого достаточно, чтобы приведенный ранее алгоритм *CustomOrderSend.mq5* без каких-либо изменений стал "торговать" в виртуальном окружении на основе текущего кастом-символа (для которого не нужен сервер или стандартный тестер).

Для оперативного отображения состояния запустим секундный таймер и периодически будем менять комментарий, а также отображать графические объекты.

```
int OnInit()
{
   EventSetTimer(1);
   return INIT_SUCCEEDED;
}

void OnTimer()
{
   Comment(CustomTrade::ReportTradeState());
   CustomTrade::DisplayTrades();
}
```

Для построения отчета по нажатию 'R' дополним обработчик *OnChartEvent*.

```
void OnChartEvent(const int id, const long &lparam, const double &dparam, const strir
{
   if(id == CHARTEVENT_KEYDOWN)
   {
      switch((int)lparam)
      {
      ...
      case KEY_R:
         CustomTrade::PrintTradeHistory();
         break;
      }
   }
}
```

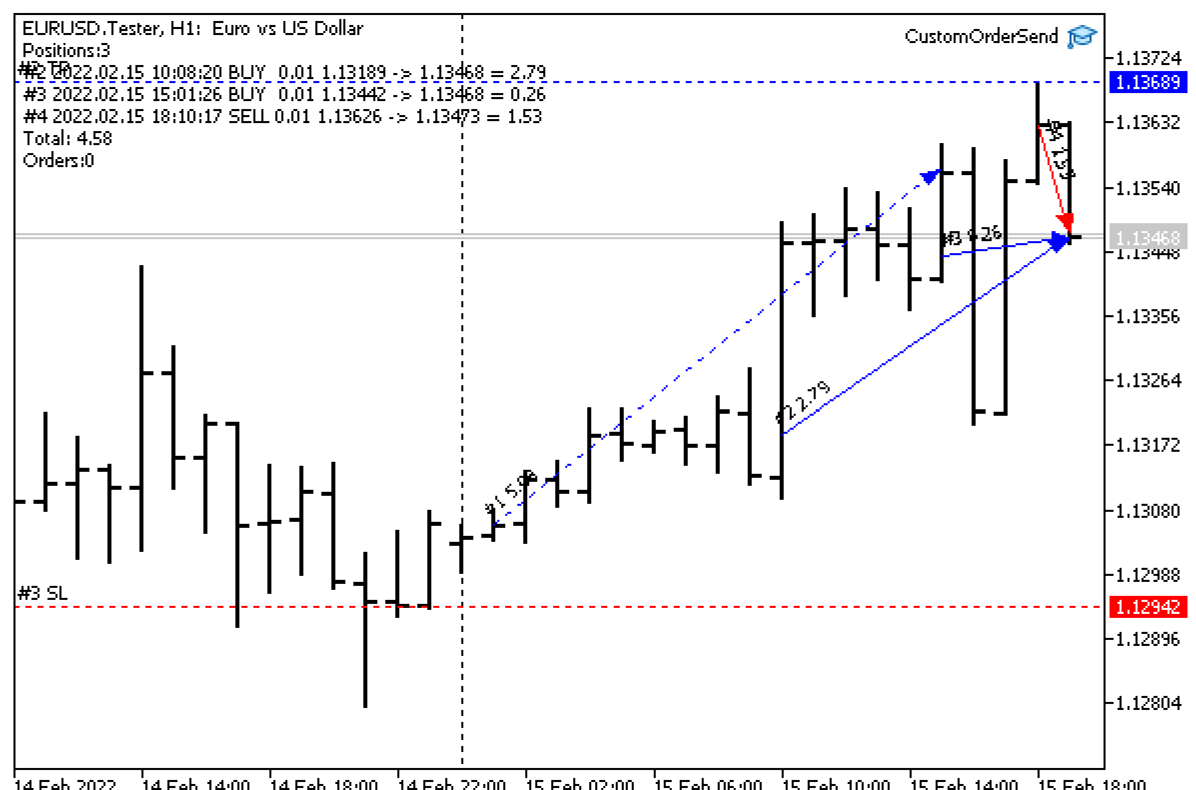
Наконец, все готово для проверки нового программного комплекса в действии.

Запустим генератор кастом-символа *CustomTester.mq5* на EURUSD. На открывшемся графике "EURUSD.Tester" запустим *CustomOrderSend.mq5* и начнем "торговать". Ниже показано изображение процесса тестирования.

Виртуальная торговля на графике пользовательского символа

Здесь видны 2 открытых длинных позиции (с защитными уровнями) и отложенный лимитный ордер на продажу.

Спустя некоторое время одна из позиций закрывается (обозначена ниже пунктирной синей линией со стрелкой), а отложенный ордер на продажу срабатывает (красная линия со стрелкой), в результате чего получаем следующую картину.



Виртуальная торговля на графике пользовательского символа

После закрытия всех позиций (части — по тейк-профиту, а остальных — по команде пользователя), был заказан отчет нажатием '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
```

В круглых скобках — идентификаторы позиций, в квадратных скобках — тикеты ордеров для соответствующих сделок (тикеты обоих типов предваряются "решёткой" '#').

Здесь не учитываются свопы и комиссии. Их расчет можно добавить.

Еще один пример работы с тиками пользовательских символов мы рассмотрим в разделе об особенностях реальной торговли с пользовательскими символами. Речь пойдет о создании эквиобъемных графиков.

7.2.7 Трансляция изменений стакана заявок

При необходимости, MQL-программа может генерировать для пользовательского символа стакан заявок с помощью функции *CustomBookAdd*. Это, в частности, может быть полезно для инструментов с внешних бирж, таких как криптовалютные.

int CustomBookAdd(const string symbol, const MqlBookInfo &books[], uint count = WHOLE_ARRAY)

Функция транслирует подписавшимся MQL-программам состояние стакана цен для пользовательского инструмента *symbol*, используя данные из массива *books*. Массив описывает полное состояние стакана, то есть все заявки на покупку и продажу. Переданное состояние полностью заменяет предыдущее и становится доступно через функцию *MarketBookGet*.

Параметр *count* позволяет задать количество элементов массива *books*, которое должно быть передано в функцию. По умолчанию используется весь массив.

Функция возвращает признак успеха (*true*) или ошибки (*false*).

Чтобы получить генерируемые функцией *CustomBookAdd* стаканы, заинтересованная в них MQL-программа должна, как обычно, подписаться на них с помощью *MarketBookAdd*.

При вбросе стакана цены *Bid* и *Ask* инструмента не обновляются: для этой цели следует отдельно вбрасывать тики при помощи *CustomTicksAdd*.

Передаваемые данные проверяются на корректность: цены и объемы должны быть больше нуля, для каждого элемента должны быть указаны тип, цена и объем (поля *volume* и/или *volume_real*). Если хотя бы один элемент стакана описан неверно, функция вернет ошибку.

Также проверяется параметр "Глубина стакана" (SYMBOL_TICKS_BOOKDEPTH) пользовательского инструмента. Если количество уровней на продажу или покупку в передаваемом стакане превышает это значение, лишние уровни отбрасываются.

Объем с повышенной точностью *volume_real* имеет больший приоритет по сравнению с обычным *volume*. Если для элемента стакана указаны оба значения, будет использовано *volume_real*.

> Внимание! В текущей реализации *CustomBookAdd* автоматически блокирует пользовательский символ, как будто на него имеется подписка, выполненная с помощью *MarketBookAdd*, но события *OnBookEvent* при этом не поступают (в принципе, генерирующая стаканы программа может на них подписаться, вызвав *MarketBookAdd* явным образом и контролировать то, что будут получать другие программы). Удалить эту блокировку можно с помощью вызова *MarketBookRelease*.
>
> Это может потребоваться в связи с тем, что символы, для которых имеются подписки на стакан, нельзя скрыть из *Обзора рынка* никакими средствами (пока все явные или неявные подписки не будут отменены из программ, а также не будет закрыто окно стакана). И как следствие, такие символы нельзя удалить.

В качестве примера создадим неторгующий эксперт *PseudoMarketBook.mq5*, который будет генерировать из ближайшей истории тиков псевдо-состояния стакана. Это может быть полезно для символов, для которых стакан не транслируется, в частности, для Forex. При желании можно

использовать такие пользовательские символы для формальной отладки собственных торговых алгоритмов, использующих стакан.

Среди входных параметров укажем максимальную глубину стакана.

```
input uint CustomBookDepth = 20;
```

Название кастом-символа будем формировать за счет добавления суффикса ".Pseudo" к названию текущего символа графика.

```
string CustomSymbol = _Symbol + ".Pseudo";
```

В обработчике *OnInit* создадим пользовательский символ и установим для него формулу, равную названию исходного символа. Таким образом, мы получим автоматически обновляемую терминалом копию исходного символа, и нам не потребуется утруждать себя копированием котировок или тиков.

```
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 + "\"");
      }
   }
   ...
```

Если пользовательский символ уже существует, эксперт может предложить пользователю удалить его, и на этом завершить работу (предварительно пользователю следует закрыть все графики с этим символом).

```
   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;
      }
   }
   ...
```

Особенностью данного символа является установка свойства SYMBOL_TICKS_BOOKDEPTH, а также чтение размера контракта SYMBOL_TRADE_CONTRACT_SIZE — он потребуется при генерации объемов.

```
   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;
}
```

Запуск алгоритма производится в обработчике *OnTick*. Здесь мы вызываем некую функцию *GenerateMarketBook*, которую еще предстоит написать. Она заполнит передаваемый по ссылке массив структур *MqlBookInfo*, и мы отправим его на пользовательский символ с помощью *CustomBookAdd*.

```
void OnTick()
{
   MqlBookInfo book[];
   if(GenerateMarketBook(2000, book))
   {
      ResetLastError();
      if(!CustomBookAdd(CustomSymbol, book))
      {
         Print("Can't add market books, ", E2S(_LastError));
         ExpertRemove();
      }
   }
}
```

Функция *GenerateMarketBook* анализирует последние *count* тиков и на их основе эмулирует возможное состояние стакана, руководствуясь гипотезами:

- то, что было куплено, скорее всего, будет продано;
- то, что было продано, скорее всего, будет куплено.

Разделение тиков на те, что соответствуют покупкам и продажам, в общем случае (при отсутствии биржевых флагов) можно оценить по движению самой цены:

- движение цены *Ask* вверх трактуется, как покупка;
- движение цены *Bid* вниз трактуется, как продажа.

В результате получим следующий алгоритм.

```
bool GenerateMarketBook(const int count, MqlBookInfo &book[])
{
   MqlTick tick; // центр стакана
   if(!SymbolInfoTick(_Symbol, tick)) return false;

   double buys[];  // объемы покупок по ценовым уровням
   double sells[]; // объемы продажи по ценовым уровням

   MqlTick ticks[];
   CopyTicks(_Symbol, ticks, COPY_TICKS_ALL, 0, count); // запрашиваем историю тиков
   for(int i = 1; i < ArraySize(ticks); ++i)
   {
      // считаем, что ask вверх подтолкнули покупки
      int k = (int)MathRound((tick.ask - ticks[i].ask) / _Point);
      if(ticks[i].ask > ticks[i - 1].ask)
      {
         // уже купили, вероятно будут фиксировать прибыль продажей
         if(k <= 0)
         {
            Place(sells, -k, contract / sqrt(sqrt(ArraySize(ticks) - i)));
         }
      }

      // считаем, что bid вниз сдвинули продажи
      k = (int)MathRound((tick.bid - ticks[i].bid) / _Point);
      if(ticks[i].bid < ticks[i - 1].bid)
      {
         // уже продали, вероятно будут фиксировать прибыль покупкой
         if(k >= 0)
         {
            Place(buys, k, contract / sqrt(sqrt(ArraySize(ticks) - i)));
         }
      }
   }
   ...
```

Вспомогательная функция *Place* заполняет массивы *buys* и *sells*, аккумулируя в них объемы по ценовым уровням. Мы покажем её ниже. Индексы в массивах определяются как расстояние в пунктах от текущих лучших цен (*Bid* или *Ask*). Размер объема обратно пропорционален возрасту тика, то есть более отдаленные в прошлое тики оказывают меньшее влияние.

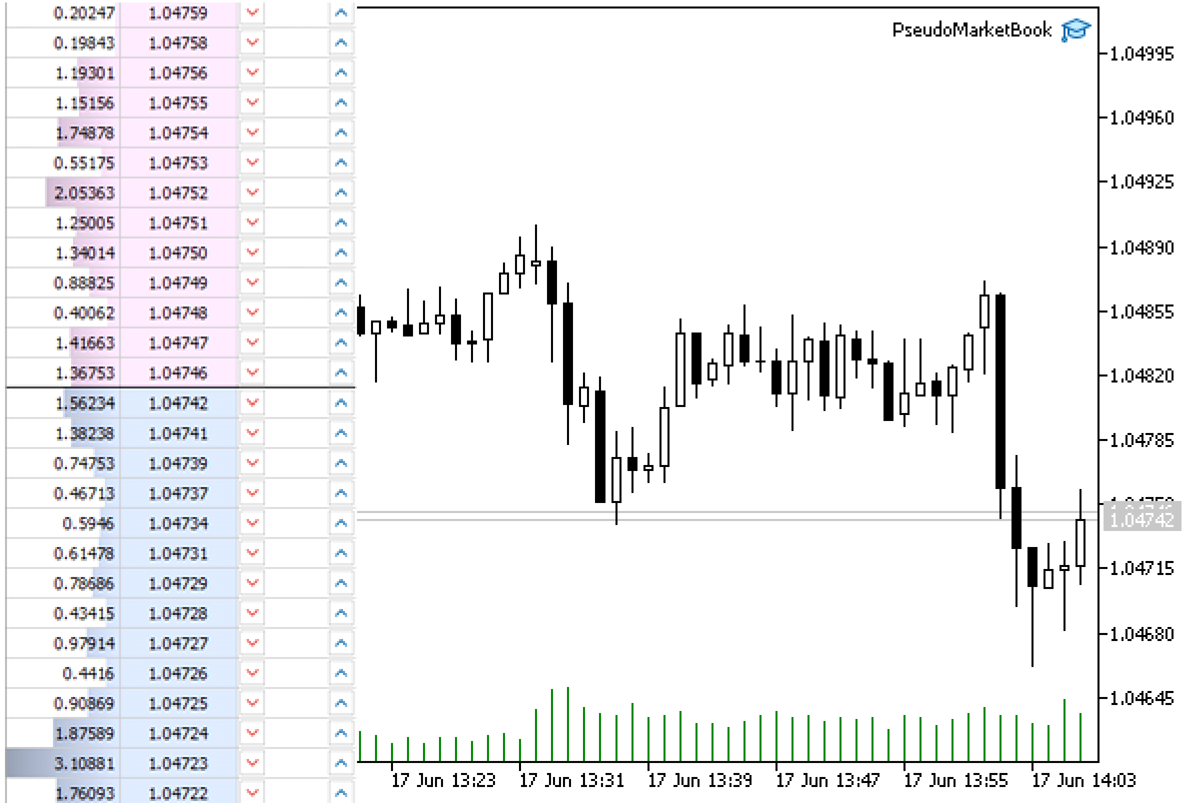
После того как массивы заполнены, на их основе формируется массив структур *MqlBookInfo*.

```
      for(int i = 0, k = 0; i < ArraySize(sells) && k < depth; ++i) // верхняя половина
      {
         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) // нижняя половина ст
      {
         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;
   }
```

Функция *Place* довольно проста.

```
   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;
   }
```

На следующем скриншоте показан график EURUSD с работающим на нем экспертом *PseudoMarketBook.mq5* и получившийся вариант стакана.



Синтетический стакан заявок пользовательского символа на основе EURUSD

7.2.8 Особенности торговли с пользовательскими символами

Пользовательский символ известен только клиентскому терминалу и отсутствует на торговом сервере. Поэтому, если пользовательский символ строится на основе некоторого реального символа, то любой советник, размещенный на графике такого пользовательского символа, должен формировать торговые приказы для исходного символа.

В качестве простейшего решения данной задачи можно размещать советник на чарте исходного символа, но получать сигналы (например, с индикаторов) с кастом-символа.

Другой напрашивающийся подход заключается в подмене имен символов при выполнении торговых операций.

Чтобы проверить оба подхода нам потребуется кастом-символ и торговый эксперт.

В качестве интересного практичного примера кастом-символов возьмем эквиобъемные графики нескольких разновидностей.

Эквиобъемный (равнообъемный) график — это график из баров, построенных по принципу равенства заключенного в них объема. На обычном графике каждый новый бар формируется с заданной периодичностью, совпадающей с размером таймфрейма. На эквиобъемном графике каждый бар считается сформированным, когда сумма тиков или реальных объемов достигает предустановленного значения. В этот момент программа начинает подсчет суммы для следующего бара. Разумеется, в процессе подсчета объемов производится контроль движений цены, и мы получаем на графике привычные четверки цен: *Open*, *High*, *Low*, *Close*.

Схожим образом строятся и равнодиапазонные бары: в них новый бар открывается, когда цена пройдет заданное количество пунктов в любом направлении.

Таким образом, в эксперте *EqualVolumeBars.mq5* мы поддержим три режима, то есть, фактически, три типа графика:

- EqualTickVolumes — эквиобъемные бары по тикам;
- EqualRealVolumes — эквиобъемные бары по реальным объемам (если они транслируются);
- RangeBars — равнодиапазонные бары.

Они выбираются с помощью входного параметра *WorkMode*.

Размер бара и глубина истории для расчета указываются в параметрах *TicksInBar* и *StartDate*.

```
input int TicksInBar = 1000;
input datetime StartDate = 0;
```

В зависимости от режима, кастом-символ получит суффикс "_Eqv", "_Qrv" или "_Rng", соответственно, с добавлением размера бара.

Хотя горизонтальная ось на эквиобъемном/равнодиапазонном графике по-прежнему обозначает хронологию, временные отметки каждого бара носят произвольный характер и зависят от волатильности (количества или размера сделок) в каждый промежуток времени. В связи с этим таймфрейм графика кастом-символа следует выбирать равным минимальному M1.

> Ограничением платформы является то, что все бары имеют равную номинальную длительность, но в случае наших "искусственных" графиков следует помнить, что настоящая длительность у каждого бара своя и может существенно превышать 1 минуту или наоборот быть меньше. Так при достаточно небольшом заданном объеме для одного бара может сложиться ситуация, что новые бары формируются гораздо чаще, чем раз в минуту, и тогда виртуальное время баров кастом-символа будет убегать вперед от реального времени, в будущее. Чтобы такого не происходило, следует увеличить объем бара (параметр *TicksInBar*) или сдвигать старые бары влево.

Инициализация и другие вспомогательные задачи по управлению пользовательскими символами (в частности, сброс уже имеющейся истории, открытие чарта с новым символом) выполняются схожим образом, как и в других примерах, и мы их опустим. Обратимся к специфике прикладного характера.

Считывать историю реальных тиков мы будем с помощью встроенных функций *CopyTicks*/*CopyTicksRange*: первая — для подкачки истории пакетами по 10000 тиков, вторая — для запроса новых тиков с момента предыдущей обработки. Весь этот функционал упакован в класс *TicksBuffer* (полный исходный код прилагается).

```
class TicksBuffer
{
private:
   MqlTick array[]; // внутренний массив тиков
   int tick;        // инкрементируемый индекс очередного тика для чтения
public:
   bool fill(ulong &cursor, const bool history = false);
   bool read(MqlTick &t);
};
```

Публичный метод *fill* предназначен для заполнения внутреннего массива очередной порцией тиков, начиная со времени *cursor* (в миллисекундах). При этом время в *cursor* при каждом вызове сдвигается вперед на основе времени последнего прочитанного в буфер тика (обратите внимание, что параметр передается по ссылке).

Параметр *history* определяет, будет ли использоваться *CopyTicks* или *CopyTicksRange*. Как правило, в онлайне мы будем считывать один или несколько новых тиков из обработчика *OnTick*.

Метод *read* возвращает один тик из внутреннего массива и сдвигает внутренний указатель (*tick*) на следующий тик. Если при чтении достигнут конец массива, метод вернет *false*, что означает, что пора вызвать метод *fill*.

С помощью данных методов алгоритм обхода истории тиков реализуется следующим образом (данный код опосредованно вызывается из *OnInit* через таймер).

```
ulong cursor = StartDate * 1000;
TicksBuffer tb;

while(tb.fill(cursor, true) && !IsStopped())
{
   MqlTick t;
   while(tb.read(t))
   {
      HandleTick(t, true);
   }
}
```

В задействованной здесь функции *HandleTick* требуется учесть свойства тика *t* в неких глобальных переменных, в которых контролируется количество тиков, суммарный торговый объем (реальный, если есть), а также дистанция движения цены. В зависимости от режима работы, эти переменные должны по-разному анализироваться на условие формирования нового бара. Так если в эквиобъемном режиме количество тиков превысило *TicksInBar*, мы должны начать новый бар, сбросив счетчик в 1. При этом время нового бара берется как округленное до минуты время тика.

В этой группе глобальных переменных предусмотрено хранение виртуального времени последнего ("текущего") бара на кастом-символе (*now_time*), его цен OHLC и объемов.

```
datetime now_time;
double now_close, now_open, now_low, now_high;
long now_volume, now_real;
```

Переменные постоянно обновляются как в процессе чтения истории, так и впоследствии, когда эксперт начинает обрабатывать онлайн-тики в реальном времени (к этому мы вернемся чуть позже).

В несколько упрощенном виде алгоритм внутри *HandleTick* выглядит так:

```
void HandleTick(const MqlTick &t, const bool history = false)
{
   now_volume++;                // подсчет количества тиков
   now_real += (long)t.volume; // суммирование реальных объемов
   
   if(!IsNewBar()) // продолжаем текущий бар
   {
      if(t.bid < now_low) now_low = t.bid;   // отслеживаем колебания цен вниз
      if(t.bid > now_high) now_high = t.bid; // и вверх
      now_close = t.bid;                     // обновляем цену закрытия
      
      if(!history)
      {
         // обновляем текущий бар, если находимся не в истории
         WriteToChart(now_time, now_open, now_low, now_high, now_close,
            now_volume - !history, now_real);
      }
   }
   else // новый бар
   {
      do
      {
         // сохраняем закрытый бар со всеми атрибутами
         WriteToChart(now_time, now_open, now_low, now_high, now_close,
            WorkMode == EqualTickVolumes ? TicksInBar : now_volume,
            WorkMode == EqualRealVolumes ? TicksInBar : now_real);

         // округляем время до минуты для нового бара
         datetime time = t.time / 60 * 60;

         // предотвращаем бары со старым или одинаковым временем
         // если ушли в "будущее", должны просто взять следующий отсчет M1
         if(time <= now_time) time = now_time + 60;

         // начинаем новый бар с текущей цены
         now_time = time;
         now_open = t.bid;
         now_low = t.bid;
         now_high = t.bid;
         now_close = t.bid;
         now_volume = 1;             // первый тик в новом баре
         if(WorkMode == EqualRealVolumes) now_real -= TicksInBar;
         now_real += (long)t.volume; // начальный реальный объем в новом баре

         // сохраняем новый бар 0
         WriteToChart(now_time, now_open, now_low, now_high, now_close,
            now_volume - !history, now_real);
      }
      while(IsNewBar() && WorkMode == EqualRealVolumes);
   }
}
```

Параметр *history* определяет, происходит ли расчет на истории или уже в реальном времени (на приходящих онлайн-тиках). На истории достаточно формировать каждый бар единожды, в то время как в онлайне текущий бар обновляется с каждым тиком. Это позволяет ускорить обработку истории.

Вспомогательная функция *IsNewBar* возвращает *true*, когда выполняется условие для закрытия очередного бара согласно режиму.

```
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;
}
```

Функция *WriteToChart* создает бар с заданными характеристиками, вызывая *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);
   }
}
```

Вышеприведенный цикл чтения и обработки тиков выполняется при первичном обращении к истории, после создания или при полном пересчете уже существующего пользовательского символа. Когда же дело доходит до новых тиков, в функции *OnTick* используется аналогичный код, но уже без флагов "историчности".

```
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);
}
```

Функция *RefreshWindow* пробрасывает тик в *Обзор рынка* для пользовательского символа.

Обратите внимание, что проброс тика увеличивает счетчик тиков в баре на 1, в связи с чем при записи счетчика тиков в 0-й бар мы ранее вычитали единицу (см. выражение *now_volume - !history* при вызове *WriteToChart*).

Генерация тиков важна, поскольку вызывает на графиках пользовательских инструментов событие *OnTick*, что потенциально позволяет экспертам, размещенным на таких чартах, торговать. Однако данная технология требует кое-каких дополнительных ухищрений, которые мы рассмотрим позднее.

```
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);
   }
}
```

Подчеркнем, что время генерируемого кастом-тика всегда ставится равным метке текущего бара, поскольку мы не можем оставить время реального тика: если оно ушло вперед более, чем на 1 минуту, и мы пошлем такой тик в *Обзор рынка*, терминал создаст следующий бар M1, что нарушит нашу "эквиобъемную" структуру, ведь у нас бары формируются не по времени, а по наполнению объемом (и мы сами контролируем данный процесс).

В принципе, мы могли бы добавлять к каждому тику одну миллисекунду, но у нас нет гарантии, что в баре не потребуется сохранить более 60000 тиков (например, если пользователь закажет график с некоторым размахом цены, который непредсказуем с точки зрения того, сколько тиков потребуется для такого движения).

В режимах по объемам теоретически возможна интерполяция секундной и миллисекундной составляющей времени тика по линейным формулам:

- EqualTickVolumes — (now_volume - 1) * 60000 / TicksInBar;
- EqualRealVolumes — (now_real - 1) * 60000 / TicksInBar;

Однако это — не более чем средство идентификации тиков, а не попытка приближения времени "искусственных" тиков к времени реальных. И речь здесь не только, и не столько об утрате неравномерности реального потока тиков, которая сама по себе уже будет приводить к отличиям в цене между оригинальным символом и генерируемом на его основе кастом-символом.

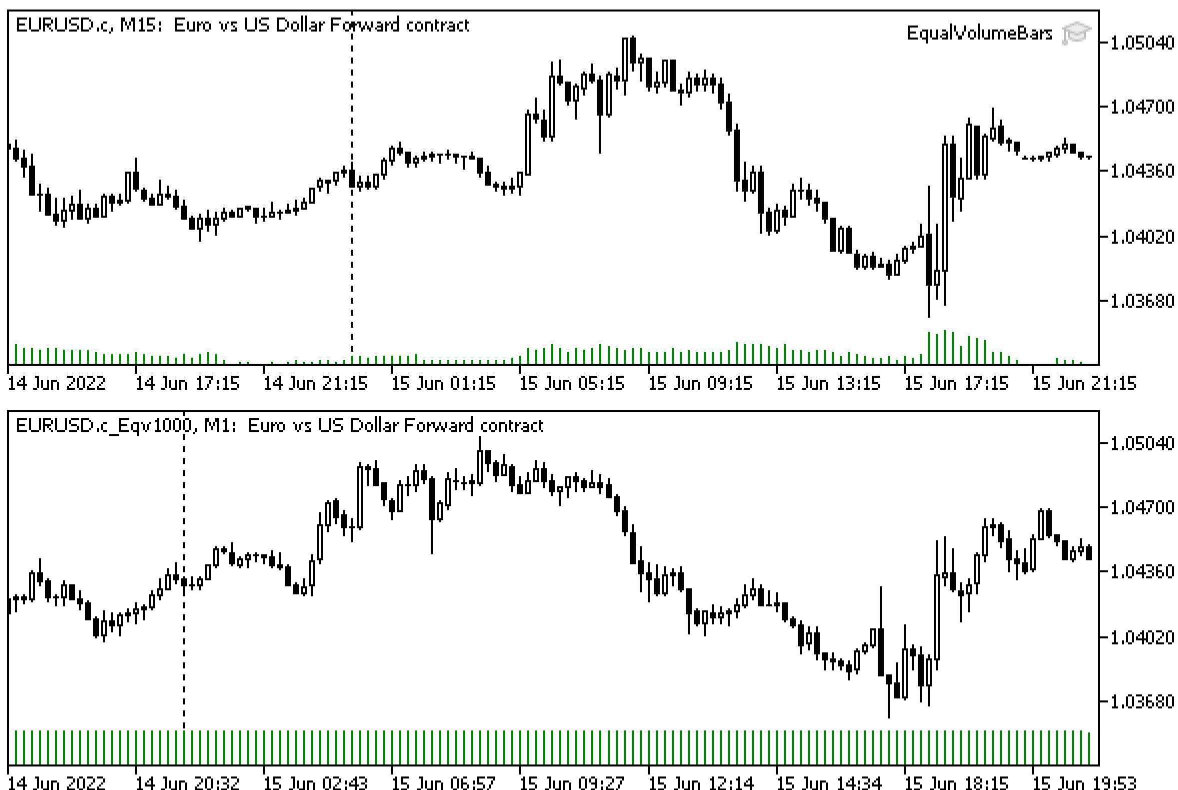
Основная проблема заключается в необходимости округления времени тиков по границе бара M1 и их "упаковки" в пределах одной минуты (см. далее врезку про специальные виды графиков). Например, очередной тик с реальным временем 12:37:05'123 становится 1001-м по счету тиком и должен сформировать новый эквиобъемный бар. Однако бар M1 может быть помечен временем только с точностью до минуты, то есть 12:37. В результате, реальная цена инструмента на 12:37 не будет совпадать с ценой в тике, который дал цену *Open* для эквиобъемного бара 12:37. Кроме того, если следующие 1000 тиков растянутся на несколько минут, мы все равно будем вынуждены "сжимать" их время, чтобы не достичь метки 12:38.

Проблема носит системный характер из-за квантования времени при эмуляции специальных графиков стандартным графиком таймфрейма M1. На них эту проблему нельзя полностью исключить. Зато при генерации кастом-символов с тиками в непрерывном времени (например, с синтетическими котировками или по потоковым данным из внешних сервисов) такой проблемы не возникает.

Важно отметить, что проброс тиков делается в данной версии генератора только онлайн, в то время как на истории кастом-тики не генерируются! Это сделано в целях ускорения создания котировок. Если вам требуется формировать историю тиков, невзирая на более медленный процесс, эксперт *EqualVolumeBars.mq5* следует адаптировать: избавиться от функции

WriteToChart, и всю генерацию выполнять с помощью *CustomTicksReplace*/*CustomTicksAdd*. При этом следует помнить, что оригинальное время тиков должно подменяться на другое — в пределах минутного бара, чтобы не нарушить структуру формируемого эквиобъемного графика.

Посмотрим, как работает *EqualVolumeBars.mq5*. Вот рабочий график EURUSD M15, на котором размещен эксперт, и созданный им эквиобъемный график, где на каждый бар отведена 1000 тиков.



Эквиобъемный график EURUSD с 1000 тиков на бар, сгенерированный экспертом EqualVolumeBars

Заметьте, что тиковые объемы на всех барах равны, за исключением последнего, который еще только формируется (подсчет тиков продолжается).

В журнал выводится статистика.

```
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
```

Проверим другой режим работы — равнодиапазонный. Ниже представлен график, на котором размах каждого бара составляет 250 пунктов.

Равнодиапазонный график EURUSD с барами размахом 250 пунктов, сгенерированный EqualVolumeBars

Для биржевых инструментов эксперт позволяет использовать режим реальных объемов, например, так:

Исходный и эквиобъемный график Ethereum с реальным объемом 10000 на бар

Таймфрейм рабочего символа при размещении эксперта-генератора не важен, поскольку для расчетов всегда используется тиковая история.

Вместе с тем, таймфрейм графика пользовательского символа должен быть равен M1 (как наименьший доступный в терминале). Таким образом, время баров, как правило, максимально близко (насколько возможно) соответствует моментам их формирования. Однако при сильных движениях на рынке, когда количество тиков или размер объемов формирует несколько баров в минуту, время баров будет убегать вперед от реального. Когда рынок успокоится, ситуация с временными отметками эквиобъемных баров нормализуется. На потоке онлайн-цен это не сказывается, так что, вероятно, не особенно критично, поскольку весь смысл использования равнообъемных или равнодиапазонных баров заключается в отвязке от абсолютного времени.

К сожалению, имя исходного символа и созданного на его основе пользовательского никак нельзя связать средствами самой платформы. Было бы удобно иметь среди свойств пользовательского символа строковое поле "origin" (источник), в которое мы могли бы записать имя реального рабочего инструмента. По умолчанию оно было бы пустым, но если его заполнить, то платформа могла бы автоматически и прозрачно для пользователя подменять символ во всех торговых приказах и запросах истории. В принципе, среди свойств пользовательских символов есть подходящее по смыслу поле SYMBOL_BASIS, но поскольку мы не можем гарантировать, что произвольные генераторы пользовательских символов (любые MQL-программы), будут корректно заполнять его или использовать именно по такому назначению, закладываться на его использование нельзя.

Поскольку данный механизм отсутствует в платформе, нам потребуется реализовать его самостоятельно. А задавать соответствие имен исходного и пользовательского символов придется с помощью параметров.

Для решения задачи разработан класс *CustomOrder* (см. прилагаемый файл *CustomOrder.mqh*). Он содержит методы-обертки для всех функций MQL API, связанных с отправкой торговых приказов и запросом истории, в которых имеется строковый параметр с именем инструмента. В этих методах производится подмена пользовательского символа на текущий рабочий или обратно. Прочие функции API не требуют "перехвата". Ниже представлен фрагмент.

```
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);
   }
   ...
```

Обратите внимание, что основной рабочий метод *replaceRequest* делает не только подмену символа, но и текущих цен *Ask* и *Bid*. Это связано с тем, что многие кастом-инструменты, такие как наш эквиобъемный график, имеют виртуальное время, отличное от времени настоящего символа-прототипа. Поэтому эмулируемые тестером цены кастом-инструмента рассинхронизированы по времени с соответствующими ценами реального инструмента.

Данный артефакт происходит только в тестере. При торговле онлайн график кастом-символа будет обновляться (по ценам) синхронно с реальным, хотя метки баров будут отличаться (один "искусственный" бар M1 имеет реальную длительность больше или меньше минуты, и момент его отсчета не кратен минуте). Таким образом, данное преобразование цены является, скорее, предосторожностью, чтобы не получать реквоты в тестере. Однако в тестере нам обычно и не нужно делать подмену символа, так как тестер умеет торговать кастом-символом (в отличие от сервера брокера). Далее мы просто ради интереса сравним результаты тестов, запущенных как с подменой, так и без подмены символа.

Для минимизации правок клиентского исходного кода предусмотрены глобальные функции и макросы следующего вида (для всех методов *CustomOrder*):

```
bool CustomOrderSend(const MqlTradeRequest &request, MqlTradeResult &result)
{
  return CustomOrder::OrderSend((MqlTradeRequest)request, result);
}

#define OrderSend CustomOrderSend
```

Они позволяют автоматически перенаправлять все вызовы стандартных функций API на методы класса *CustomOrder* — для этого достаточно включить *CustomOrder.mqh* в эксперт и задать рабочий символ, например, в параметре *WorkSymbol*:

```
#include <CustomOrder.mqh>
#include <Expert/Expert.mqh>
...
input string WorkSymbol = "";

int OnInit()
{
  if(WorkSymbol != "")
  {
    CustomOrder::setReplacementSymbol(WorkSymbol);

    // инициируем открытие вкладки графика рабочего символа (в визуальном режиме те
    MqlRates rates[1];
    CopyRates(WorkSymbol, PERIOD_CURRENT, 0, 1, rates);
  }
  ...
}
```

Важно, чтобы директива *#include <CustomOrder.mqh>* шла самой первой, перед другими. Таким образом она оказывает эффект на все исходные коды, в том числе и на подключаемые стандартные библиотеки из поставки MetaTrader 5. Если подстановочный символ не задан, подключенный *CustomOrder.mqh* не оказывает никакого эффекта на эксперт и "прозрачно" передает управление стандартным функциям API.

Теперь у нас все готово для проверки идеи торговли на кастом-символе, включая и сам кастом-символ.

Модифицируем указанным выше способом уже знакомый нам эксперт *BandOsMaPro*, переименовав в *BandOsMaCustom.mq5*. Давайте протестируем его на эквиобъемном графике EURUSD с размером бара 1000 тиков, полученном с помощью *EqualVolumeBars.mq5*.

Режим оптимизации или тестирования — по ценам OHLC M1 (более точные методы не имеют смысла, потому что мы не генерировали тики, а также потому что данная версия эксперта торгует по ценам сформированных баров). Диапазон дат — 2021 год и первая половина 2022 года. Файл с настройками прилагается — *BandOsMACustom.set*.

В настройках тестера следует не забыть выбрать кастом-символ EURUSD_Eqv1000 и таймфрейм M1, поскольку именно на нем эмулируются эквиобъемные бары.

Когда параметр *WorkSymbol* пуст, эксперт торгует кастом-символом. Вот его результаты:

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

Отчет тестера при торговле на эквиобъемного графике EURUSD_Eqv1000

Если параметр *WorkSymbol* равен EURUSD, эксперт торгует парой EURUSD, несмотря на то, что работает на графике EURUSD_Eqv1000. Результаты отличаются, но не сильно.

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

Отчет тестера при торговле EURUSD с эквиобъемного графика EURUSD_Eqv1000

Но, как уже было сказано в начале раздела, для экспертов, которые торгуют по сигналам индикаторов, существует более простой способ поддержать кастом-символы. Для этого достаточно создавать индикаторы на кастом-символе, а сам эксперт размещать на графике рабочего символа.

Мы можем легко реализовать такой вариант. Назовем его *BandOsMACustomSignal.mq5*.

Заголовочный файл *CustomOrder.mqh* больше не понадобится, а вместо входного параметра *WorkSymbol* добавим два новых:

```
input string SignalSymbol = "";
input ENUM_TIMEFRAMES SignalTimeframe = PERIOD_M1;
```

Их потребуется передать в конструктор класса *BandOsMaSignal*, который заведует индикаторами. Ранее в нем везде использовались *_Symbol* и *_Period*.

```
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;
   }
}
```

Так как символ и таймфрейм для сигналов теперь могут отличаться от символа и периода графика, мы расширили интерфейс *TradingSignal* методами для их чтения. Передача актуальных значений в конструктор производится в *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;
}
```

В классе *SimpleStrategy* метод *trade* теперь проверяет наступление нового бара не по текущему графику, а согласно свойствам сигнала.

```
   virtual bool trade() override
   {
      // ищем сигнал один раз на открытии бара нужного символа и таймфрейма
      if(lastBar == iTime(command[].symbol(), command[].timeframe(), 0)) return false

      int s = command[].signal(); // получаем сигнал
      ...
   }
```

Для сравнительного эксперимента с теми же настройками эксперт *BandOsMACustomSignal.mq5* следует запускать на EURUSD (можно M1 или другой таймфрейм), а в параметре *SignalSymbol* указать EURUSD_Eqv1000. *SignalTimeframe* следует оставить равным по умолчанию PERIOD_M1. В результате мы получим похожий отчет.

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

Отчет тестера при торговле на графике EURUSD по сигналам с эквиобъемного символа EURUSD_Eqv1000

Количество баров и тиков здесь другое, потому что тестируемым инструментом выбран EURUSD, а не пользовательский EURUSD_Eqv1000.

Все три результата тестов слегка отличаются. Это происходит из-за "упаковки" котировок в минутные бары и незначительной рассинхронизации движения цен оригинального и пользовательского инструмента. Какой из результатов точнее? Это, скорее всего, зависит от конкретной торговой системы и особенностей её реализации. В случае нашего эксперта *BandOsMa* с контролем открытия баров наиболее реалистичные показатели должны быть у варианта с прямой торговлей на EURUSD_Eqv1000. Но в принципе, почти всегда выполняется эмпирическое правило, что из нескольких альтернативных проверок наиболее достоверной является наименее прибыльная.

Итак, мы разобрали пару приемов по адаптации экспертов для торговли на пользовательских символах, имеющих прототип среди рабочих символов брокера. Однако такая ситуация не является обязательной. Во многих случаях кастом-символы генерируются на основе данных из внешних систем, таких как криптобиржи. Торговля на них должна выполняться с использованием их публичного API с помощью сетевых функций MQL5.

Эмуляция специальных видов графиков с помощью пользовательских символов

Многие трейдеры применяют на практике специальные виды графиков, в которых непрерывное реальное время исключено из рассмотрения. Сюда относятся не только эквиобъемные и равнодиапазонные бары, но также Ренко, Point-And-Figure (PAF), Kagi, и другие. Пользовательские символы позволяют эмулировать эти виды графиков в MetaTrader 5 с помощью графиков таймфрейма M1, но к ним следует относиться с осторожностью, когда речь заходит о тестировании торговых систем, а не техническом анализе.

У специальных видов графиков фактическое время открытия бара (с точностью до миллисекунды) практически всегда не совпадает ровно с минутой, которой будет маркирован бар M1. Таким образом, цена открытия кастом-бара отличается от цены открытия бара M1 стандартного символа.

Тем более будут отличаться и прочие цены OHLC, потому что реальная длительность формирования бара M1 на специальном графике не равна одной минуте. Например, 1000 тиков для эквиобъемного графика могут накапливаться в течение 5 минут.

Цена закрытия кастом-бара также не соответствует реальному времени закрытия, потому что кастом-бар — это, технически, бар M1, т.е. он имеет номинальную длительность 1 минута.

Особую осторожность следует проявлять для таких видов графиков как классический Ренко или PAF. Дело в том, что в них разворотные бары имеют цену открытия с гепом от закрытия предыдущего бара. Таким образом, цена открытия становится предиктором будущего ценового движения.

Анализ таких графиков предполагается проводить по сформированным барам, то есть их характеристическая цена — цена закрытия, однако тестер при побаровой работе предоставляет для текущего (последнего) бара только цену открытия (режима по ценам закрытия нет). Даже если брать сигналы индикаторов с закрытых баров (обычно, с 1-го), сделки в любом случае совершаются по текущей цене 0-го бара. И даже если обратиться к тиковым режимам, тестер всегда генерирует тики по обычным правилам, руководствуясь опорными точками на основе конфигурации каждого бара. Тестер не учитывает

особенностей строения и поведения специальных графиков, которые мы пытаемся визуально эмулировать барами M1.

Торговля в тестере по таким символам в любом режиме (по ценам открытия, M1 OHLC или по тикам) сказывается на точности результатов — они слишком оптимистичны и могут служить источником псевдо-граалей. В связи с этим насущно необходимо проверять торговую систему не на отдельном графике Ренко или PAF, а в связке с исполнением приказов на реальном символе.

Еще одним полем для применения кастом-символов являются секундные таймфреймы или графики тиков. В этом случае для баров и тиков также генерируется виртуальное время, отвязанное от реального. Поэтому такие графики хорошо подходят для оперативного анализа, но требуют дополнительного внимания при разработке и тестировании торговых стратегий, особенно мультисимвольных.

Альтернативой для любых пользовательских символов являет самостоятельный расчет массивов баров и тиков внутри эксперта или индикатора. Однако отладка и визуализация таких структур требует дополнительных усилий.

7.3 Экономический календарь

При разработке торговых стратегий желательно учитывать фундаментальные факторы, влияющие на рынок. В MetaTrader 5 встроен календарь экономических новостей, который доступен в интерфейсе программы в виде отдельной вкладки в инструментальной панели, а также в виде меток, опционально выводимых непосредственно на график. Включение данного календаря — в целом как сервиса — производится отдельным флажком на закладке Сообщество в диалоге настройки терминала (логиниться в сообщество при этом не обязательно).

Вполне закономерно, что в платформе для алготрейдинга к записям экономического календаря можно обращаться и программным образом из MQL5 API. В данной главе мы познакомимся с функциями и структурами данных, обеспечивающими чтение, фильтрацию и мониторинг изменений экономических событий.

Экономический календарь содержит описание, график выхода и историю значений макроэкономических индикаторов множества стран. Для каждого события известно точное время планируемого выхода, степень важности, влияние на конкретные валюты, прогнозные значения и другие атрибуты. Актуальные значения макроэкономических показателей поступают в MetaTrader 5 моментально в момент публикации.

Наличие календаря позволяет автоматически анализировать поступающие события и реагировать на них в эксперте самыми разными способами, например, торгуя в рамках стратегии на пробой или колебаний волатильности внутри коридора. С другой стороны, знание предстоящих возмущений на рынке позволяет находить в расписании спокойные часы и временно отключать те роботы, для которых сильные движения цены опасны убытками.

Все функции и структуры для работы с экономическим календарем используют в значениях типа *datetime* время торгового сервера (*TimeTradeServer*), с учетом его временной зоны и перехода на "летнее" (DST, Daylight Saving Time) или "зимнее" время. Иными словами, для корректного тестирования новостных советников, их разработчик должен самостоятельно изменить времена исторических новостей в те периоды (примерно полгода внутри каждого года), когда режим DST отличается от текущего.

Функции календаря нельзя использовать в тестере: при попытке вызвать любую из них получим ошибку FUNCTION_NOT_ALLOWED (4014). В связи с этим тестирование стратегий на основе календаря предполагает предварительное сохранение записей календаря во внешних хранилищах (например, в файлах) при запуске MQL-программы на онлайн-чарте, а затем их загрузку и чтение из MQL-программы, выполняющейся в тестере.

7.3.1 Основные понятия календаря

При работе с календарем мы будем оперировать несколькими понятиями, для формального описания которых в MQL5 определены специальные типы структур.

Прежде всего, события связаны с конкретными странами, и каждая страна описана с помощью структуры *MqlCalendarCountry*.

```
struct MqlCalendarCountry
{
   ulong  id;              // идентификатор страны по стандарту ISO 3166-1
   string name;            // текстовое имя страны (в текущей кодировке терминала)
   string code;            // двухбуквенное обозначение страны по ISO 3166-1 alpha-2
   string currency;        // международный код валюты страны
   string currency_symbol; // символ/знак валюты страны
   string url_name;        // имя страны, используемое в URL на сайте mql5.com
};
```

О том, как получить перечень доступных в календаре стран и их атрибутов в виде массива структур *MqlCalendarCountry*, мы узнаем в следующем разделе.

Пока лишь обратим внимание на поле *id*. Оно важно, потому что является ключевым для определения принадлежности календарных событий к той или иной стране. В каждой стране (или зарегистрированном объединении стран, таком как Евросоюз) существует специфический, известный на международном уровне перечень видов экономических индикаторов и информационных поводов, оказывающих влияние на рынок и потому включаемых в календарь.

Каждый вид событий задается структурой *MqlCalendarEvent*, в которой поле *country_id* однозначно связывает событие со страной. Типы используемых перечислений мы рассмотрим чуть ниже.

```
struct MqlCalendarEvent
{
   ulong                          id;          // идентификатор события
   ENUM_CALENDAR_EVENT_TYPE       type;        // тип события
   ENUM_CALENDAR_EVENT_SECTOR     sector;      // сектор, к которому относится событие
   ENUM_CALENDAR_EVENT_FREQUENCY  frequency;   // частота (периодичность) события
   ENUM_CALENDAR_EVENT_TIMEMODE   time_mode;   // режим времени события
   ulong                          country_id;  // идентификатор страны
   ENUM_CALENDAR_EVENT_UNIT       unit;        // единица измерения показателя
   ENUM_CALENDAR_EVENT_IMPORTANCE importance;  // важность события
   ENUM_CALENDAR_EVENT_MULTIPLIER multiplier;  // множитель показателя
   uint                           digits;      // количество знаков после запятой
   string                         source_url;  // URL источника публикации события
   string                         event_code;  // код события
   string                         name;        // текстовое имя события на языке терми
};
```

Важно понимать, что структура *MqlCalendarEvent* описывает именно вид события (например, публикацию индекса потребительских цен, Consumer Price Index (CPI)), но не конкретное событие, которое может происходить раз в квартал, раз в месяц или по другому расписанию. Здесь содержатся общие характеристики события — его важность, периодичность, отношение к сектору экономики, единицы измерения, название, источник информации. А фактические и прогнозные показатели будут предоставляться в записях календаря о каждом конкретном событии данного вида: эти записи хранятся как структуры *MqlCalendarValue*, о которых речь пойдет ниже по тексту. Функции для запроса поддерживаемых видов событий будут представлены в последующих разделах.

Тип события в поле *type* указывается как одно из значений перечисления ENUM_CALENDAR_EVENT_TYPE.

Идентификатор	Описание
CALENDAR_TYPE_EVENT	Событие (митинг, речь и так далее)
CALENDAR_TYPE_INDICATOR	Экономический индикатор
CALENDAR_TYPE_HOLIDAY	Праздник (выходной день)

Сектор экономики, к которому относится событие, выбирается из перечисления ENUM_CALENDAR_EVENT_SECTOR.

Идентификатор	Описание
CALENDAR_SECTOR_NONE	Сектор не задан
CALENDAR_SECTOR_MARKET	Рынок, биржа
CALENDAR_SECTOR_GDP	Валовый внутренний продукт (GDP)
CALENDAR_SECTOR_JOBS	Рынок труда
CALENDAR_SECTOR_PRICES	Цены

Идентификатор	Описание
CALENDAR_SECTOR_MONEY	Деньги
CALENDAR_SECTOR_TRADE	Торговля
CALENDAR_SECTOR_GOVERNMENT	Правительство
CALENDAR_SECTOR_BUSINESS	Бизнес
CALENDAR_SECTOR_CONSUMER	Потребление
CALENDAR_SECTOR_HOUSING	Жилье
CALENDAR_SECTOR_TAXES	Налоги
CALENDAR_SECTOR_HOLIDAYS	Праздники

Периодичность события указывается в поле *frequency* с использованием перечисления ENUM_CALENDAR_EVENT_FREQUENCY.

Идентификатор	Описание
CALENDAR_FREQUENCY_NONE	Частота публикации не задана
CALENDAR_FREQUENCY_WEEK	Раз в неделю
CALENDAR_FREQUENCY_MONTH	Раз в месяц
CALENDAR_FREQUENCY_QUARTER	Раз в квартал
CALENDAR_FREQUENCY_YEAR	Раз в год
CALENDAR_FREQUENCY_DAY	Раз в день

Продолжительность события (*time_mode*) может быть описана одним из элементов перечисления ENUM_CALENDAR_EVENT_TIMEMODE.

Идентификатор	Описание
CALENDAR_TIMEMODE_DATETIME	Известно точное время наступления события
CALENDAR_TIMEMODE_DATE	Событие занимает весь день
CALENDAR_TIMEMODE_NOTIME	Время не публикуется
CALENDAR_TIMEMODE_TENTATIVE	Предварительно известен только день, но не точное время события (время уточняется по факту)

Важность события в поле *importance* указывается с помощью перечисления ENUM_CALENDAR_EVENT_IMPORTANCE.

Идентификатор	Описание
CALENDAR_IMPORTANCE_NONE	Не задана
CALENDAR_IMPORTANCE_LOW	Низкая
CALENDAR_IMPORTANCE_MODERATE	Средняя
CALENDAR_IMPORTANCE_HIGH	Высокая

Единицы измерения, в которых даются значения события, определены в поле *unit* как элемент перечисления ENUM_CALENDAR_EVENT_UNIT.

Идентификатор	Описание
CALENDAR_UNIT_NONE	Единица измерения не задана
CALENDAR_UNIT_PERCENT	Проценты
CALENDAR_UNIT_CURRENCY	Национальная валюта
CALENDAR_UNIT_HOUR	Количество часов
CALENDAR_UNIT_JOB	Количество рабочих мест
CALENDAR_UNIT_RIG	Буровые установки
CALENDAR_UNIT_USD	Доллары США
CALENDAR_UNIT_PEOPLE	Число людей
CALENDAR_UNIT_MORTGAGE	Количество ипотечных кредитов
CALENDAR_UNIT_VOTE	Число голосов
CALENDAR_UNIT_BARREL	Количество в баррелях
CALENDAR_UNIT_CUBICFEET	Объем в кубических футах
CALENDAR_UNIT_POSITION	Чистый объем спекулятивных позиций в контрактах
CALENDAR_UNIT_BUILDING	Количество строений

В некоторых случаях значения экономического показателя требуют множителя (*multiplier*), согласно одному из элементов перечисления ENUM_CALENDAR_EVENT_MULTIPLIER.

Идентификатор	Описание
CALENDAR_MULTIPLIER_NONE	Множитель не задан
CALENDAR_MULTIPLIER_THOUSANDS	Тысячи
CALENDAR_MULTIPLIER_MILLIONS	Миллионы
CALENDAR_MULTIPLIER_BILLIONS	Миллиарды
CALENDAR_MULTIPLIER_TRILLIONS	Триллионы

Итак, мы разобрались со всеми специальными типами данных, используемыми для описания видов событий в структуре *MqlCalendarEvent*.

Отдельная запись календаря оформлена как структура *MqlCalendarValue*. Её подробное описание приведено ниже, а пока нам важно обратить внимание на такой нюанс. В *MqlCalendarValue* есть поле *event_id*, которое указывает на идентификатор разновидности события, то есть содержит один из существующих *id* в структурах *MqlCalendarEvent*.

Как мы видели выше, структура *MqlCalendarEvent* в свою очередь имеет связь с *MqlCalendarCountry* через поле *country_id*. Таким образом, внеся один раз информацию о конкретной стране или виде события в базу календаря, можно регистрировать для них произвольное количество однотипных событий. Разумеется, наполнением базы занимается поставщик информации, а не разработчики.

Подведем промежуточный итог: система хранит раздельно три внутренних таблицы:

- таблицу структур *MqlCalendarCountry* для описания стран;
- таблицу структур *MqlCalendarEvent* с описаниями видов событий;
- таблицу структур *MqlCalendarValue* с показателями конкретных событий различных видов.

За счет ссылок на идентификаторы видов событий из записей о конкретных событиях исключается дублирование информации. Например, ежемесячные публикации значений CPI лишь ссылаются на одну и ту же структуру *MqlCalendarEvent* с общими характеристиками этого вида событий. Если бы не разные таблицы, потребовалось бы в каждой записи календаря о CPI повторять одни и те же свойства. Данный подход к установлению отношений между таблицами с данными с помощью полей-идентификаторов называется *реляционным*, и мы еще вернемся к нему в главе про SQLite. Все это проиллюстрировано на следующей схеме.

Схема связей между структурами по полям с идентификаторами

Все таблицы хранятся во внутренней базе календаря, которая постоянно поддерживается в актуальном состоянии, пока терминал подключен к серверу.

Записи календаря (конкретные события) — это структуры *MqlCalendarValue*. Они также идентифицируется по собственному уникальному номеру в поле *id* (в каждой из трех таблиц свое поле *id*).

```
struct MqlCalendarValue
{
   ulong      id;                   // ID записи
   ulong      event_id;             // ID вида события
   datetime   time;                 // время и дата события
   datetime   period;               // отчетный период события
   int        revision;             // ревизия публикуемого индикатора по отношению к с
   long       actual_value;         // актуальное значение в миллионных долях или LONG_
   long       prev_value;           // предыдущее значение в миллионных долях или LONG_
   long       revised_prev_value; // пересмотренное предыдущее значение в миллионных
   long       forecast_value;       // прогнозное значение в миллионных долях или LONG_
   ENUM_CALENDAR_EVENT_IMPACT impact_type;  // потенциальное влияние на курс валюты

   // функции для проверки значений
   bool HasActualValue(void) const;      // true, если поле actual_value заполнено
   bool HasPreviousValue(void) const;    // true, если поле prev_value заполнено
   bool HasRevisedValue(void) const;     // true, если поле revised_prev_value заполне
   bool HasForecastValue(void) const;    // true, если поле forecast_value заполнено

   // функции для получение значений
   double GetActualValue(void) const;    // actual_value или nan, если значение не зад
   double GetPreviousValue(void) const; // prev_value или nan, если значение не задан
   double GetRevisedValue(void) const;  // revised_prev_value или nan, если значение
   double GetForecastValue(void) const; // forecast_value или nan, если значение не з
};
```

Для каждого события, помимо времени его публикации (*time*), предусмотрено хранение 4-х значений:

- актуального показателя (*actual_value*), который становится известен сразу после публикации новости;
- предыдущего показателя (*prev_value*), ставшего известным в прошлый выход той же новости;
- пересмотренного значения предыдущего показателя, *revised_prev_value* (если оно было изменено с момента прошлой публикации);
- прогнозного значения (*forecast_value*);

Очевидно, что могут быть заполнены не все поля. Так, актуальный показатель отсутствует (еще не известен) у будущих событий, а пересмотр прошлых значений также происходит не всегда. Кроме того, все 4 поля имеют смысл только для количественных показателей, в то время как календарь отражает также выступления регуляторов, встречи и праздники.

Пустое поле (отсутствие значения) обозначается константой LONG_MIN (-9223372036854775808). Если значение в поле задано (не равно LONG_MIN), то оно соответствует увеличенной в миллион раз реальной величине показателя, то есть для получения показателя в привычном (вещественном) виде необходимо разделить значение поля на 1000000.

Для удобства программиста в структуре определены 4 *Has*-метода для проверки заполненности поля, а также 4 *Get*-метода, возвращающих уже преобразованное в вещественное число значение соответствующего поля, причем в случае, когда оно не заполнено, метода вернет NaN (Not A Number).

Иногда для получения абсолютных величин (если они требуются для алгоритма) важно дополнительно анализировать свойство *multiplier* в структуре *MqlCalendarEvent*, так как некоторые показатели указываются в кратных единицах согласно перечислению ENUM_CALENDAR_EVENT_MULTIPLIER. Кроме того, в *MqlCalendarEvent* есть поле *digits*, задающее количество значащих цифр в получаемых значениях для последующего корректного форматирования (например, в вызове *NormalizeDouble*).

Отчетный период (за который посчитан публикуемый показатель) задается в поле *period* его первым днем. Например, если показатель считается ежемесячно, то дата '2022.05.01 00:00:00' означает месяц май. Длительность периода (например, месяц, квартал, год) определена в поле *frequency* связанной структуры *MqlCalendarEvent*: типом этого поля является специальное перечисление ENUM_CALENDAR_EVENT_FREQUENCY, описанное выше, вместе с другими перечислениями.

Особый интерес представляет поле *impact_type*, в котором после выхода новости автоматически устанавливается направление влияния на курс соответствующей валюты за счет сравнения актуального и прогнозного значений. Это влияние может быть положительным (валюта предположительно должна дорожать) или отрицательным (валюта предположительно должны дешеветь). Например, более сильное падение продаж, чем ожидалось, будет помечено, как имеющее отрицательное влияние, а более сильное уменьшение безработицы, как положительное. Но не для всех событий данная характеристика трактуется однозначно (некоторые экономические индикаторы считаются противоречивыми), а кроме того следует обращать внимание и на относительные числа изменений.

Потенциальное влияние события на курс национальной валюты указывается с помощью перечисления ENUM_CALENDAR_EVENT_IMPACT.

Идентификатор	Описание
CALENDAR_IMPACT_NA	Влияния не указано
CALENDAR_IMPACT_POSITIVE	Положительное влияние
CALENDAR_IMPACT_NEGATIVE	Отрицательное влияние

Еще одним важным понятием календаря является факт его изменения. Специальной структуры для изменения, к сожалению, не предусмотрено. Единственное свойство, которым обладает изменение, — это его уникальный идентификатор — целое число, присваиваемое системой при каждом изменении внутренней базы календаря.

Как известно, календарь постоянно модифицируется поставщиками информации — в него добавляются новые грядущие события, корректируются уже опубликованные показатели и прогнозы. Поэтому очень важно отслеживать любые правки, появление которых и дают обнаружить периодически увеличивающиеся номера изменений.

Время правки с конкретным идентификатором и её суть в MQL5 недоступны. При необходимости MQL-программы должны сами реализовать периодические запросы состояния календаря и анализ записей.

Получить информацию о странах, видах событий и конкретных записях календаря, а также их изменениях позволяет набор функций MQL5, которые мы рассмотрим в следующих разделах.

Внимание! При первом обращении к календарю (если до этого не была открыта вкладка Календарь в инструментальной панели терминала) может потребоваться несколько секунд на синхронизацию внутренней базы календаря с сервером.

7.3.2 Получение списка и описаний доступных стран

Получить полный список стран, события по которым транслируются в календаре, можно с помощью функции *CalendarCountries*.

int CalendarCountries(MqlCalendarCountry &countries[])

Функция заполняет переданный по ссылке массив *countries* структурами *MqlCalendarCountry*. Массив может быть динамическим или фиксированного достаточного размера.

В случае успеха функция возвращает количество полученных с сервера описаний стран или 0 при ошибке. Среди возможных кодов ошибок в *_LastError* могут встретиться, в частности, 5401 (ERR_CALENDAR_TIMEOUT, превышение лимита запроса по времени) или 5400 (ERR_CALENDAR_MORE_DATA, если размер фиксированного массива недостаточен для получения описаний всех стран). В последнем случае система скопирует только то, что уместилось.

Напишем простой скрипт *CalendarCountries.mq5*, который получает полный список стран и выводит его в журнал.

```
void OnStart()
{
   MqlCalendarCountry countries[];
   PRTF(CalendarCountries(countries));
   ArrayPrint(countries);
}
```

Вот пример результата.

```
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"
```

Важно отметить, что идентификатор 0 (код "WW" и псевдо-валюта "ALL") соответствует общемировым событиям (касающихся многих стран, например, встречи G7, G20), а валюта "EUR" связана с несколькими странами Евросоюза, доступными в календаре (как видно, представлена не вся зона Евро). Также и сам Евросоюз имеет обобщающий идентификатор 999.

Если вас интересует конкретная страна, можно проверить её наличие по цифровому коду согласно стандарту ISO 3166-1. В частности, в журнале выше эти коды выведены в первом столбце (поле *id*).

Получить описание одной страны по её идентификатору, заданному в параметре *id*, позволяет функция *CalendarCountryById*.

bool CalendarCountryById(const long id, MqlCalendarCountry &country)

В случае успеха функция вернет *true* и заполнит поля структуры *country*.

Если страна не найдена, получим *false*, а в *_LastError* — код ошибки 5402 (ERR_CALENDAR_NO_DATA).

Пример использования данной функции смотрите в разделе Получение записей о событиях по странам или валютам.

7.3.3 Запрос видов событий по странам и валютам

Календарь экономических событий и праздников имеет в каждой стране свою специфику. MQL-программа может запросить виды событий внутри конкретной страны, а также виды событий,

связанных с конкретной валютой. Последнее актуально в тех случаях, когда несколько стран пользуются одной валютой, как, например, большинство участников Евросоюза.

int CalendarEventByCountry(const string country, MqlCalendarEvent &events[])

Функция *CalendarEventByCountry* заполняет переданный по ссылке массив структур *MqlCalendarEvent* описаниями всех видов событий, доступных в календаре для страны, заданной двухбуквенным кодом country (по стандарту ISO 3166-1 alpha-2). Примеры таких кодов мы видели в предыдущем разделе, в журнале: EU — Евросоюз, US — США, DE — Германия, CN — Китай, и так далее.

Приемный массив может быть динамическим или фиксированного достаточного размера.

Функция возвращает количество полученных описаний и 0 в случае ошибки. В частности, если фиксированный массив не способен вместить все события, функция заполнит его уместившейся частью имеющихся данных и установит код *_LastError*, равный CALENDAR_MORE_DATA (5400). Также возможны ошибки выделения памяти (4004, ERR_NOT_ENOUGH_MEMORY) или таймаута запроса календаря с сервера (5401, ERR_CALENDAR_TIMEOUT).

Если страна с заданным кодом не существует, случится ошибка INTERNAL_ERROR (4001).

Указав NULL или пустую строку "" вместо *country*, можно получить полный перечень событий по всем странам.

Проверим работу функции с помощью простого скрипта *CalendarEventKindsByCountry.mq5*. Он имеет единственных входной параметр — код интересующей нас страны.

```
input string CountryCode = "HK";
```

Далее делается запрос видов событий вызовом *CalendarEventByCountry*, и в случае успеха полученный массивы выводится в журнал.

```
void OnStart()
{
   MqlCalendarEvent events[];
   if(PRTF(CalendarEventByCountry(CountryCode, events)))
   {
      Print("Event kinds for country: ", CountryCode);
      ArrayPrint(events);
   }
}
```

Вот пример результата (из-за того, что строки получаются длинными, они искусственно поделены на 2 блока для публикации в книге: в первый блок попали числовые поля структур *MqlCalendarEvent*, а во второй — строковые).

```
CalendarEventByCountry(CountryCode,events)=26 / ok
Event kinds for country: HK
          [id] [type] [sector] [frequency] [time_mode] [country_id] [unit] [importanc
[ 0] 344010001      1        5           2           0          344      6
[ 1] 344010002      1        5           2           0          344      1
[ 2] 344020001      1        4           2           0          344      1
[ 3] 344020002      1        2           3           0          344      1
[ 4] 344020003      1        2           3           0          344      1
[ 5] 344020004      1        6           2           0          344      1
[ 6] 344020005      1        6           2           0          344      1
[ 7] 344020006      1        6           2           0          344      2
[ 8] 344020007      1        9           2           0          344      1
[ 9] 344020008      1        3           2           0          344      1
[10] 344030001      2       12           0           1          344      0
[11] 344030002      2       12           0           1          344      0
[12] 344030003      2       12           0           1          344      0
[13] 344030004      2       12           0           1          344      0
[14] 344030005      2       12           0           1          344      0
[15] 344030006      2       12           0           1          344      0
[16] 344030007      2       12           0           1          344      0
[17] 344030008      2       12           0           1          344      0
[18] 344030009      2       12           0           1          344      0
[19] 344030010      2       12           0           1          344      0
[20] 344030011      2       12           0           1          344      0
[21] 344030012      2       12           0           1          344      0
[22] 344030013      2       12           0           1          344      0
[23] 344030014      2       12           0           1          344      0
[24] 344030015      2       12           0           1          344      0
[25] 344500001      1        8           2           0          344      0
```

Продолжение журнала (правый фрагмент).

```
     »                     [source_url]                           [event_code]
  [ 0]» "https://www.hkma.gov.hk/eng/"    "foreign-exchange-reserves"         "Foreign
  [ 1]» "https://www.hkma.gov.hk/eng/"    "hkma-m3-money-supply-yy"           "HKMA M3
  [ 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
  [ 6]» "https://www.censtatd.gov.hk/en/" "imports-mm"                        "Imports
  [ 7]» "https://www.censtatd.gov.hk/en/" "trade-balance"                     "Trade Ba
  [ 8]» "https://www.censtatd.gov.hk/en/" "retail-sales-yy"                   "Retail S
  [ 9]» "https://www.censtatd.gov.hk/en/" "unemployment-rate-3-months"        "Unemploy
  [10]» "https://publicholidays.hk/"      "new-years-day"                     "New Year
  [11]» "https://publicholidays.hk/"      "lunar-new-year"                    "Lunar Ne
  [12]» "https://publicholidays.hk/"      "ching-ming-festival"               "Ching Mi
  [13]» "https://publicholidays.hk/"      "good-friday"                       "Good Fri
  [14]» "https://publicholidays.hk/"      "easter-monday"                     "Easter M
  [15]» "https://publicholidays.hk/"      "birthday-of-buddha"                "The Birt
  [16]» "https://publicholidays.hk/"      "labor-day"                         "Labor Da
  [17]» "https://publicholidays.hk/"      "tuen-ng-festival"                  "Tuen Ng
  [18]» "https://publicholidays.hk/"      "hksar-establishment-day"           "HKSAR Es
  [19]» "https://publicholidays.hk/"      "day-following-mid-autumn-festival" "The Day
  [20]» "https://publicholidays.hk/"      "national-day"                      "National
  [21]» "https://publicholidays.hk/"      "chung-yeung-festival"              "Chung Ye
  [22]» "https://publicholidays.hk/"      "christmas-day"                     "Christma
  [23]» "https://publicholidays.hk/"      "first-weekday-after-christmas-day" "The Firs
  [24]» "https://publicholidays.hk/"      "day-following-good-friday"         "The Day
  [25]» "https://www.markiteconomics.com" "nikkei-pmi"                        "S&P Glob
```

int CalendarEventByCurrency(const string currency, MqlCalendarEvent &events[])

Функция *CalendarEventByCurrency* заполняет переданный массив *events* описаниями всех видов событий в календаре, которые связаны с указанной валютой *currency*. Трехбуквенное обозначение валют хорошо знакомо трейдерам Forex.

При указании неверного кода валюты, функция вернет 0 (нет ошибки) и пустой массив.

Указав NULL или пустую строку "" вместо *currency*, можно получить полный перечень событий календаря.

Протестируем функцию скриптом *CalendarEventKindsByCurrency.mq5*. Во входном параметре задаётся код валюты.

```
input string Currency = "CNY";
```

В обработчике *OnStart* мы запрашиваем события и выводим их в журнал.

```
void OnStart()
{
   MqlCalendarEvent events[];
   if(PRTF(CalendarEventByCurrency(Currency, events)))
   {
      Print("Event kinds for currency: ", Currency);
      ArrayPrint(events);
   }
}
```

Вот пример результата (приводится с сокращениями).

```
CalendarEventByCurrency(Currency,events)=40 / ok
Event kinds for currency: CNY
          [id] [type] [sector] [frequency] [time_mode] [country_id] [unit] [importanc
[ 0] 156010001      1        4           2           0          156      1
[ 1] 156010002      1        4           2           0          156      1
[ 2] 156010003      1        4           2           0          156      1
[ 3] 156010004      1        2           3           0          156      1
[ 4] 156010005      1        2           3           0          156      1
[ 5] 156010006      1        9           2           0          156      1
[ 6] 156010007      1        8           2           0          156      1
[ 7] 156010008      1        8           2           0          156      0
[ 8] 156010009      1        8           2           0          156      0
[ 9] 156010010      1        8           2           0          156      1
[10] 156010011      0        5           0           0          156      0
[11] 156010012      1        3           2           0          156      1
[12] 156010013      1        8           2           0          156      1
[13] 156010014      1        8           2           0          156      1
[14] 156010015      1        8           2           0          156      0
[15] 156010016      1        8           2           0          156      1
[16] 156010017      1        9           2           0          156      1
[17] 156010018      1        2           3           0          156      1
[18] 156020001      1        6           2           3          156      6
[19] 156020002      1        6           2           3          156      1
[20] 156020003      1        6           2           3          156      1
[21] 156020004      1        6           2           3          156      2
[22] 156020005      1        6           2           3          156      1
[23] 156020006      1        6           2           3          156      1
...
```

Правый фрагмент.

```
     »                        [source_url]                                 [event_code
[ 0]» "http://www.stats.gov.cn/english/"  "cpi-mm"
[ 1]» "http://www.stats.gov.cn/english/"  "cpi-yy"
[ 2]» "http://www.stats.gov.cn/english/"  "ppi-yy"
[ 3]» "http://www.stats.gov.cn/english/"  "gdp-qq"
[ 4]» "http://www.stats.gov.cn/english/"  "gdp-yy"
[ 5]» "http://www.stats.gov.cn/english/"  "retail-sales-yy"
[ 6]» "http://www.stats.gov.cn/english/"  "industrial-production-yy"
[ 7]» "http://www.stats.gov.cn/english/"  "manufacturing-pmi"
[ 8]» "http://www.stats.gov.cn/english/"  "non-manufacturing-pmi"
[ 9]» "http://www.stats.gov.cn/english/"  "fixed-asset-investment-yy"
[10]» "http://www.stats.gov.cn/english/"  "nbs-press-conference-on-economic-situation
[11]» "http://www.stats.gov.cn/english/"  "unemployment-rate"
[12]» "http://www.stats.gov.cn/english/"  "industrial-profit-yy"
[13]» "http://www.stats.gov.cn/english/"  "industrial-profit-ytd-yy"
[14]» "http://www.stats.gov.cn/english/"  "composite-pmi"
[15]» "http://www.stats.gov.cn/english/"  "industrial-production-ytd-yy"
[16]» "http://www.stats.gov.cn/english/"  "retail-sales-ytd-yy"
[17]» "http://www.stats.gov.cn/english/"  "gdp-ytd-yy"
[18]» "http://english.customs.gov.cn/"    "trade-balance-usd"
[19]» "http://english.customs.gov.cn/"    "imports-usd-yy"
[20]» "http://english.customs.gov.cn/"    "exports-usd-yy"
[21]» "http://english.customs.gov.cn/"    "trade-balance"
[22]» "http://english.customs.gov.cn/"    "imports-yy"
[23]» "http://english.customs.gov.cn/"    "exports-yy"
...
```

Внимательный читатель заметит, что идентификатор вида события содержит в себе код страны, номер источника новости и порядковый номер внутри источника (нумерация начинается с 1). То есть общий формат идентификатора вида события такой: CCCSSNNNN, где CCC — код страны, SS — источник, NNNN — номер. Например, 156020001 — первая новость из второго источника по Китаю, а 344030010 — десятая новость из третьего источника по Гонконгу. Единственное исключение — общемировые новости, для них код "страны" — не 000, а 1000.

7.3.4 Получение описания вида события по идентификатору

Реальные MQL-программы, как правило, запрашивают текущие или приближающиеся события календаря, с фильтрацией по временному диапазону, странам, валютам или другим признакам. Предназначенные для этого функции API, которые нам еще предстоит рассмотреть, возвращают структуры *MqlCalendarValue*, где вместо описания события хранится лишь его идентификатор. Поэтому для извлечения полной информации может пригодиться функция *CalendarEventById*.

bool CalendarEventById(ulong id, MqlCalendarEvent &event)

Функция *CalendarEventById* получает описание события по его идентификатору. Функция возвращает признак успеха или ошибки.

Пример использования данной функции мы приведем в следующем разделе.

7.3.5 Получение записей о событиях по странам или валютам

Конкретные события различных видов запрашиваются в календаре для заданного диапазона дат и с возможностью фильтрации по стране или валюте.

```
int CalendarValueHistory(MqlCalendarValue &values[], datetime from, datetime to = 0,
  const string country = NULL, const string currency = NULL)
```

Функция *CalendarValueHistory* заполняет передаваемый по ссылке массив *values* записями календаря во временном диапазоне от *from* до *to*. Оба параметра могут включать дату и время. Значение *from* входит в интервал, а значение *to* — нет. Иными словами, функция отбирает записи календаря (структуры *MqlCalendarValue*), в которых для свойства *time* выполняется составное условие: *from* <= *time* < *to*.

Начальное время *from* должно быть указано обязательно. Конечное время *to* является опциональным: если оно опущено или равно 0, в массив копируются все будущие события.

Время *to* должно быть больше *from*, за исключением случаев, когда оно равно 0. Особое сочетание для запроса всех имеющихся событий (и прошлых, и будущих) — когда *from* и *to* равны 0.

Если приемный массив динамический, для него будет автоматически выделена память. В случае массива с фиксированным размером, будет скопировано количество записей не больше размера массива.

Параметры *country* и *currency* позволяют задать дополнительную фильтрацию записей по стране или валюте. Параметр *country* принимает двухбуквенный код страны по стандарту ISO 3166-1 alpha-2 (например, "DE", "FR", "EU"), параметр *currency* — трехбуквенное обозначение валюты (например, "EUR", "CNY").

Значение по умолчанию NULL или пустая строка "" в любом из параметров равносильны отсутствию соответствующего фильтра.

Если указаны оба фильтра, выбираются значения только тех событий, для которых удовлетворяются одновременно оба условия — страна и валюта. Это может пригодиться, если в календаре окажутся страны с несколькими валютами, каждая из которых имеет хождение также в нескольких станах. В данный момент в календаре таких событий нет. Для получения событий стран зоны Евро достаточно указать код конкретной страны или EU, и валюта EUR будет подразумеваться.

Функция возвращает количество скопированных элементов и может установить код ошибки. В частности, если превышено время ожидания запроса с сервера, получим в *_LastError* ошибку 5401 (ERR_CALENDAR_TIMEOUT). Если же фиксированный массив не уместил всех записей, код будет равен 5400 (ERR_CALENDAR_MORE_DATA), однако массив будет заполнен. При выделении памяти под динамический массив потенциально возможна ошибка 4004 (ERR_NOT_ENOUGH_MEMORY).

> Внимание! Порядок элементов в массиве может отличаться от хронологического. Сортируйте записи по времени самостоятельно.

С помощью функции *CalendarValueHistory* мы могли бы запросить грядущие события примерно так:

```
   MqlCalendarValue values[];
   if(CalendarValueHistory(values, TimeCurrent()))
   {
      ArrayPrint(values);
   }
```

Однако с таким кодом мы получим недостаточно информативную таблицу, в которой суть событий — их названия, важность, коды валют — будут скрыты за идентификатором события в поле *MqlCalendarValue::event_id* и, опосредовано, за идентификатором страны в поле *MqlCalendarEvent::country_id*. Чтобы сделать вывод информации более дружественным следует по коду события запросить его описание, а из этого описания взять код страны и получить её атрибуты. Покажем это в примере скрипта *CalendarForDates.mq5*.

Во входных параметрах предусмотрим ввод кода страны и валюты для фильтрации. По умолчанию запрашиваются события по Евросоюзу.

```
input string CountryCode = "EU";
input string Currency = "";
```

Диапазон дат событий будем автоматически рассчитывать на некоторое время назад и вперед. Это "некоторое время" также предоставим выбирать пользователю из трех вариантов: сутки, неделя или месяц.

```
#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;
```

Определим свою структуру *MqlCalendarRecord*, производную от *MqlCalendarValue*, и добавим в неё поля для удобного представления атрибутов, которые будут заполняться по ссылкам (идентификаторам) из зависимых структур.

```
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
```

Среди добавленных полей — строки с важностью (одно из значений статического массива *importances*), названием события, страны и валюты, а также четверка значений в формате *double*. Это фактически означает дублирование информации в угоду наглядному представлению при выводе на печать. Позднее мы подготовим более совершенную "обертку" для календаря.

Для заполнения объекта потребуется параметрический конструктор, принимающий исходную структуру *MqlCalendarValue*. После того как все унаследованные поля неявным образом скопированы в новый объект оператором '=', мы вызываем специально подготовленный метод *extend*.

```
   MqlCalendarRecord() { }

   MqlCalendarRecord(const MqlCalendarValue &value)
   {
      this = value;
      extend();
   }
```

В методе *extend* следует получить описание события по его идентификатору, а на основе идентификатора страны из описания события — структуру с атрибутами страны. После этого можно заполнить первую половину добавленных полей из полученных структур *MqlCalendarEvent* и *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();
   }
```

Далее мы вызвали встроенные *Get*-методы для заполнения четверки полей типа *double* с финансовыми показателями.

Теперь мы можем использовать новую структуру в основном обработчике *OnStart*.

```
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);
   }
}
```

Здесь делается заполнение массива стандартных структур *MqlCalendarValue* с помощью вызова *CalendarValueHistory* для текущих условий, выставленных во входных параметрах. Далее все элементы переносятся в массив *MqlCalendarRecord*, причем в процессе создания объектов они расширяются дополнительной информацией. Наконец, массив событий выводится в журнал.

Записи в журнале получаются довольно длинными. Сначала приведем левую половину, которая полностью соответствует тому, что мы увидели бы при печати массива стандартных структур *MqlCalendarValue*.

```
CalendarValueHistory(values,from,to,CountryCode,Currency)=6 / ok
Near past and future calendar records (extended):
      [id] [event_id]              [time]            [period] [revision] [actual_valu
[0] 162723  999020003 2022.06.23 03:00:00 1970.01.01 00:00:00    0 -92233720368547758
[1] 162724  999020003 2022.06.24 03:00:00 1970.01.01 00:00:00    0 -92233720368547758
[2] 168518  999010034 2022.06.24 11:00:00 1970.01.01 00:00:00    0 -92233720368547758
[3] 168515  999010031 2022.06.24 13:10:00 1970.01.01 00:00:00    0 -92233720368547758
[4] 168509  999010014 2022.06.24 14:30:00 1970.01.01 00:00:00    0 -92233720368547758
[5] 161014  999520001 2022.06.24 22:30:00 2022.06.21 00:00:00    0 -92233720368547758
```

А вот вторая половина с "расшифровкой" названий, важности и значений.

```
CalendarValueHistory(values,from,to,CountryCode,Currency)=6 / ok
Near past and future calendar records (extended):
     [importance]                                              [name] [currency] [c
[0]  "High"       "EU Leaders Summit"                                  "EUR"      "E
[1]  "High"       "EU Leaders Summit"                                  "EUR"      "E
[2]  "Medium"     "ECB Supervisory Board Member McCaul Speech"         "EUR"      "E
[3]  "Medium"     "ECB Supervisory Board Member Fernandez-Bollo Speech" "EUR"      "E
[4]  "Medium"     "ECB Vice President de Guindos Speech"               "EUR"      "E
[5]  "Low"        "CFTC EUR Non-Commercial Net Positions"              "EUR"      "E
```

7.3.6 Получение записей о событиях конкретного вида

При необходимости MQL-программа имеет возможность запросить события конкретного вида: для этого достаточно заранее узнать идентификатор события, например, с помощью функций *CalendarEventByCountry* или *CalendarEventByCurrency*, которые были представлены в разделе Запрос видов событий по странам и валютам.

int CalendarValueHistoryByEvent(ulong id, MqlCalendarValue &values[], datetime from, datetime to = 0)

Функция *CalendarValueHistoryByEvent* заполняет переданный по ссылке массив записями о событиях конкретного вида, заданного идентификатором *id*. Параметры *from* и *to* позволяют ограничить диапазон дат, в которых ищутся события.

Если необязательный параметр *to* не указан, в массив будут помещены все записи календаря, начиная с времени *from* и далее, в будущее. Чтобы запросить все прошлые события, установите *from* в 0. Если оба параметра *from* и *to* равны 0, будет возвращена вся имеющая история и планируемые события. Во всех остальных случаях, когда *to* не равно 0, оно должно быть больше *from*.

Массив *values* может быть динамическим (тогда функция автоматически расширит или уменьшит его под объем данных) или фиксированного размера (тогда в массив будет скопировано только то, что уместилось).

Функция возвращает количество скопированных элементов.

В качестве примера рассмотрим скрипт *CalendarStatsByEvent.mq5*, который подсчитывает статистику (частоту встречаемости) событий разных видов для заданной страны или валюты, в заданном временном диапазоне.

Условия анализа задаются во входных переменных.

```
input string CountryOrCurrency = "EU";
input ENUM_CALENDAR_SCOPE Scope = SCOPE_YEAR;
```

В зависимости от длины строки *CountryOrCurrency*, она интерпретируется как код страны (2 символа) или валюты (3 символа).

Для сбора статистики объявим структуру, в полях которой сохраним идентификатор и название вида события, его важность и, собственно, счетчик таких событий.

```
struct CalendarEventStats
{
   static const string importances[];
   ulong id;
   string name;
   string importance;
   int count;
};

static const string CalendarEventStats::importances[] = {"None", "Low", "Medium", "Hi
```

В функции *OnStart* сначала запросим все виды событий с помощью функции *CalendarEventByCountry* или *CalendarEventByCurrency* на указанную глубину истории и в будущее, а затем в цикле по полученным описаниям событий в массиве *events* вызовем *CalendarValueHistoryByEvent* для каждого идентификатора события. В данном применении нас не интересуют содержащиеся в массиве *values* записи, достаточно лишь знать их количество.

```
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);
}
```

При успешном вызове функции заполняем структуру *CalendarEventStats* и добавляем её в массив структур *stats*. Далее мы сортируем структуру уже известным нам способом (макрос SORT_STRUCT был описан в разделе Сравнение, сортировка и поиск в массивах).

Запуск скрипта с настройками по умолчанию генерирует примерно такие записи в журнале (приведено с сокращениями).

```
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"
```

Обратите внимание, что всего было получено 82 вида событий, однако в массиве статистики у нас оказалось только 74. Дело в том, что функция *CalendarValueHistoryByEvent* возвращает *false* (неуспех) и нулевой код ошибки в *_LastError*, если событий какого-либо вида не оказалось в заданном диапазоне дат. В приведенном тесте таких теоретически существующих, но ни разу за год не встретившихся событий набралось 8.

7.3.7 Чтение записи о событии по идентификатору

После того как трейдер изучит расписание событий на ближайшее будущее, он может настраивать соответствующим образом своих роботов. В API календаря нет функций или событий ("событий" — в смысле функций обработки новой финансовой информации вроде *OnCalendar*, по аналогии с *OnTick*) для автоматического отслеживания выхода новостей. Алгоритм должен это делать сам с любой выбранной периодичностью. В частности, достаточно загодя выяснить идентификатор интересующего вас события с помощью одной из рассмотренных ранее функций (например, *CalendarValueHistoryByEvent*, *CalendarValueHistory*) и затем вызывать *CalendarValueById*, чтобы получать актуальное состояние полей в структуре *MqlCalendarValue*.

bool CalendarValueById(ulong id, MqlCalendarValue &value)

Функция заполняет переданную по ссылке структуру текущей информаций о конкретном событии.

Результат функции обозначает признак успеха (*true*) или ошибки (*false*).

Создадим простой безбуферный индикатор *CalendarRecordById.mq5*, который найдет в будущем самое близкое событие с типом "финансового индикатора" (т.е. числового показателя) и будет

по таймеру опрашивать его состояние. Когда новость опубликуют, данные изменятся (станет известно "актуальное" значение показателя), и индикатор выведет алерт.

Периодичность опроса календаря задается во входной переменной.

```
input uint TimerSeconds = 5;
```

Таймер запустим в *OnInit*.

```
void OnInit()
{
   EventSetTimer(TimerSeconds);
}
```

Для удобного вывода в журнал описания события используется структура *MqlCalendarRecord*, знакомая нам по примеру скрипта *CalendarForDates.mq5*.

Для хранения исходного состояния информации о новости опишем структуру *track*.

```
MqlCalendarValue track;
```

Когда структура пуста (и в поле *id* находится 0), программа должна запросить предстоящие события и найти среди них ближайшее с типом CALENDAR_TYPE_INDICATOR и у которого еще не известно актуальное значение.

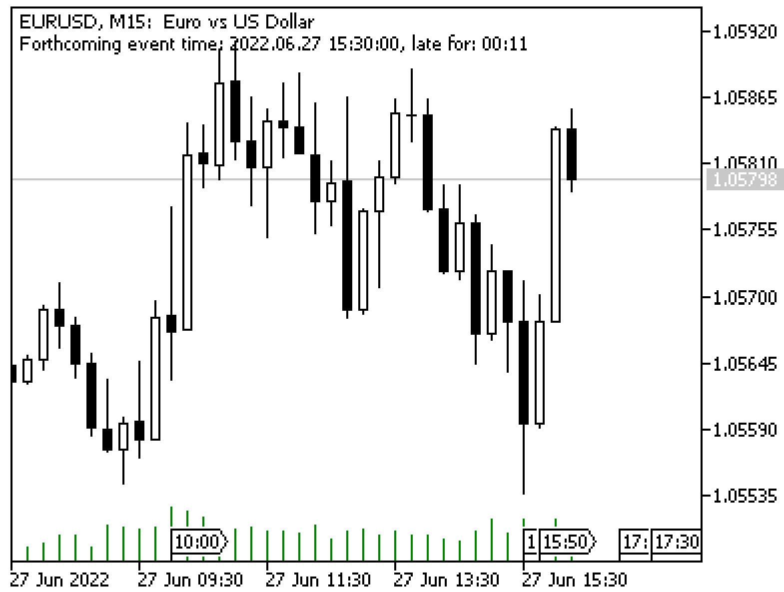
```
void OnTimer()
{
   if(!track.id)
   {
      MqlCalendarValue values[];
      if(PRTF(CalendarValueHistory(values, TimeCurrent(), TimeCurrent() + DAY_LONG *
      {
         for(int i = 0; i < ArraySize(values); ++i)
         {
            MqlCalendarEvent event;
            CalendarEventById(values[i].event_id, 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;
            }
         }
      }
   }
   ...
```

Найденное событие копируется в *track* и выводится в журнал. После этого каждый вызов *OnTimer* сводится к получению обновленной информации о событии в структуру *update* — именно она передается в *CalendarValueById*, с указанием идентификатора *track.id*. Далее производится сравнение исходной и новой структур с помощью вспомогательной функции *StructCompare* (на основе *StructToCharArray* и *ArrayCompare*, см. полный исходный код). Любое различие приводит к печати нового состояния (может быть поменялся прогноз), а если появилось актуальное значение, таймер останавливается. Чтобы запустить ожидание следующей новости, этот

индикатор нужно переинициализировать: он является демонстрационным, а для контроля обстановки по списку новостей мы позднее разработаем более практичный класс-фильтр.

```
      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)));
         }
      }
   }
```

Во время ожидания события индикатор выводит в комментарий ожидаемое время выхода новости, и сколько до неё осталось (или каково опоздание).



Комментарий об ожидании или опоздании ближайшей новости

Важно отметить, что новость может выйти как чуть раньше, так и чуть позже запланированного срока. Это создает некоторые проблемы при тестировании новостных стратегий на истории, так как время обновления записей календаря в терминале и через MQL5 API не предоставляется. Мы попробуем частично решить эту проблему в следующем разделе.

Вот фрагменты вывода в журнал, произведенного индикатором с некоторым промежутком:

```
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_typ
» -9223372036854775808       400000 -9223372036854775808                0
» [importance]                     [name] [currency] [code] [actual] [previous] [revi
» "Medium"     "Durable Goods Orders m/m" "USD"      "US"        nan    0.40000
...
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] [revi
» "Medium"     "Durable Goods Orders m/m" "USD"      "US"    0.70000    0.40000
Timer stopped
```

В обновленной новости появилось значение *actual_value*.

Чтобы не ждать во время теста слишком долго, данный индикатор желательно запускать в рабочие часы основных рынков, когда плотность выхода новостей высока.

Функция *CalendarValueById* — не единственная, и, вероятно, не самая гибкая, с помощью которой можно мониторить изменения в календаре. Пару других подходов мы рассмотрим в следующих разделах.

7.3.8 Отслеживание изменений событий по стране или валюте

Как было сказано в разделе про основные понятия календаря, платформа регистрирует все изменения событий некими внутренними средствами. Каждое состояние характеризуется идентификатором изменений (*change_id*). Среди функций MQL5 есть две, которые позволяют этот идентификатор выяснить (в произвольный момент времени) и затем запрашивать записи календаря, измененные позднее. Одна из этих функций — *CalendarValueLast*, о которой речь пойдет в этом разделе, а про вторую — *CalendarValueLastByEvent* — мы расскажем в следующем разделе.

int CalendarValueLast(ulong &change_id, MqlCalendarValue &values[],
  const string country = NULL, const string currency = NULL)

Функция *CalendarValueLast* предназначена для двух задач: получения последнего известного идентификатора изменений календаря *change_id* и заполнения массива *values* измененными записями с момента предыдущего изменения, заданного переданным идентификатором в том же *change_id*. Иными словами, параметр *change_id* работает и как входной, и как выходной. Именно поэтому он является ссылкой и требует указания переменной.

Если подать на вход функции *change_id*, равный 0, то функция заполнит переменную актуальным идентификатором, но не станет заполнять массив.

Дополнительно с помощью параметров *country* и *currency* можно установить фильтрацию записей по стране и валюте.

Функция возвращает количество скопированных элементов календаря. Поскольку в первом режиме работы (*change_id = 0*), массив не заполняется, возврат 0 не является ошибкой. Мы также можем получить 0, если с момента указанного изменения календарь более не модифицировался. Поэтому для проверки на ошибку следует анализировать *_LastError*.

Таким образом, обычный способ применения функции заключается в циклическом опросе календаря на предмет изменений.

```
ulong change = 0;
MqlCalendarValue values[];
while(!IsStopped())
{
   // передаем последний известный нам идентификатор и получаем новый, если он появил
   if(CalendarValueLast(change, values))
   {
      // анализ добавленных и измененных записей
      ArrayPrint(values);
      ...
   }
   Sleep(1000);
}
```

Это можно делать в цикле, по таймеру или по другим событиям.

Идентификаторы постоянно увеличиваются, но могут идти не по порядку, то есть перескакивать несколько значений.

> Важно отметить, что каждая запись календаря доступна всегда только в одном единственном, последнем состоянии: история изменений в MQL5 не предоставляется. Как правило, это не является проблемой, поскольку жизненный цикл каждой новости стандартный: добавление в

базу заранее за достаточно долгое время и дополнение актуальными данными в момент проведения мероприятия. Однако на практике могут случаться различные отклонения: редактирование прогноза, перенос времени, пересмотр факта. В какое именно время и что было изменено в записи — узнать через MQL5 API из истории календаря нельзя. Поэтому те торговые системы, которые принимают решения, исходя из сиюминутной обстановки, потребуют самостоятельного сохранения истории изменений и её интеграции в эксперт для прогона в тестере.

С помощью функции *CalendarValueLast* мы можем создать полезный сервис *CalendarChangeSaver.mq5*, который будет с заданной периодичностью проверять календарь на изменения и, при их наличии, сохранять в файл идентификаторы изменений вместе с текущим временем сервера. Это позволит в дальнейшем использовать информацию файла для более реалистичного тестирования экспертов на истории календаря. Разумеется, для этого потребуется организовать экспорт/импорт всей базы календаря, чем мы со временем займемся.

Предусмотрим входные переменные для задания имени файла и периода между опросами (в миллисекундах).

```
input string Filename = "calendar.chn";
input int PeriodMsc = 1000;
```

В начале обработчика *OnStart* открываем двоичный файл на запись, а точнее на дозапись (если он уже существует). Формат существующего файла здесь не проверяется — добавьте защиту сами при встраивании в реальное приложение.

```
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...");
   ...
```

Здесь следует сделать небольшое отступление.

При каждом изменении календаря в файл должна записываться как минимум пара целых 8-байтных чисел: текущее время (*datetime*) и идентификатор новости (*ulong*), однако одновременно измененных записей может быть и больше одной. Поэтому в первое число помимо даты упаковывается количество измененных записей. Для этого принимается во внимание, что даты умещаются в 0x7FFFFFFFF и, следовательно, старшие 3 байта остаются неиспользуемыми.

Именно в два старших байта (по смещению влево на 48 битов) и помещается количество идентификаторов, которые сервис запишет после соответствующей временной метки. Макрос PACK_DATETIME_COUNTER создает "расширенную" дату, а два других — DATETIME и COUNTER — востребованы впоследствии при чтении архива изменений (другой программой).

```
#define PACK_DATETIME_COUNTER(D,C) (D | (((ulong)(C)) << 48))
#define DATETIME(A) ((datetime)((A) & 0x7FFFFFFFF))
#define COUNTER(A)  ((ushort)((A) >> 48))
```

Теперь вернемся к основному коду сервиса. В цикле, который "просыпается" каждые заданные *PeriodMsc* миллисекунд мы запрашиваем изменения с помощью *CalendarValueLast*. Если изменения есть, записываем текущее серверное время и массив полученных идентификаторов в файл.

```
      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);
   }
```

Для удобного представления информации о каждой новости написана вспомогательная функция *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));
}
```

Таким образом, в журнале будет выведен не только идентификатор, но код страны, название и плановое время новости.

Предполагается, что сервис должен работать довольно продолжительное время, чтобы собрать сведения за период, достаточный для тестирования (дни, недели, месяцы). К сожалению, так же как и в случае со стаканами цен, платформа не предоставляет готовой истории стакана или правок календаря, так что их сбор ложится полностью на разработчика MQL-программ.

Посмотрим на сервис в действии. В следующем фрагменте журнала (на периоде 2022.06.28 15:30 — 16:00) обратите внимание, что некоторые новости относятся к отдаленному будущему (в них проставляются значения поля prev_value, которое является по совместительству полем *actual_value* одноименного текущего события). Однако более важно другое: реальное время выхода новостей может существенно, порой на несколько минут, отличаться от планового.

```
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)]
```

Разумеется, это важно не для всех классов торговых стратегий, а только тех, что торгуют оперативно по рынку. Для них создаваемый архив хронологии правок календаря может обеспечить более точное тестирование новостных экспертов. О том, как можно "подключить" календарь к тестеру, мы разберем в дальнейшем, а пока покажем, как осуществить чтение полученного файла.

Для целей демонстрации подготовлен скрипт *CalendarChangeReader.mq5*. На практике приведенный исходный код должен размещаться в эксперте.

Входные переменные позволяют задать имя файл для чтения и начальную дату проверки. Если сервис продолжает работать (записывать файл), необходимо скопировать файл под другим именем или в другую папку (в примере скрипта файл переименован). Если параметр *Start* оставлен пустым, чтение новостных изменений начнется с начала текущего дня.

```
input string Filename = "calendar2.chn";
input datetime Start;
```

Для хранения информации об отдельной правке описана структура *ChangeState*.

```
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);
   }
};
```

Она используется в классе *ChangeFileReader*, который выполняет основную работу по чтению файла и предоставлению в вызывающий код тех изменений, которые соответствуют конкретному моменту времени.

Дескриптор файла передается как параметр в конструктор, также как и начальное время теста. Чтение файла и заполнение структуры *ChangeState* для одной правки календаря выполняется в методе *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); // находим первую правку после 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;
   }
   ...
```

Метод *check* читает файл до тех пор, пока очередная правка не окажется в будущем. При этом все предыдущие (по временным меткам) правки с момента предыдущего вызова метода помещаются в выходной массив *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;
      }
   };
```

Вот как класс используется в *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);
   
   // читаем шаг за шагом, время now увеличиваем искусственно в этом демо
   while(!FileIsEnding(handle))
   {
      // в реальном приложении вызов reader.check можно делать на каждом тике
      ulong records[];
      if(reader.check(now, records))
      {
         Print(now);          // выводим время
         ArrayPrint(records); // массив идентификаторов изменившихся новостей
      }
      now += 60; // прибавляем по 1 минуте за раз, можно посекундно
   }
   
   FileClose(handle);
}
```

А вот результаты работы скрипта для тех же изменений календаря, которые сохранялись сервисом в контексте предыдущего фрагмента журнала.

```
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
```

Те же самые идентификаторы воспроизводятся в виртуальном времени с тем же запаздыванием, что и онлайн, правда, здесь видно округление до 1 минуты, которое получилось, потому что мы задали такой искусственный шаг в цикле. По идее, из соображений эффективности мы можем откладывать проверки вплоть до времени, хранящемся в структуре *ChangeState current*. В прилагаемом исходном коде определен метод *getState* для получения этого времени.

7.3.9 Отслеживание изменений событий по типу

MQL5 API позволяет запрашивать свежие изменения не только в целом по всему календарю или в разрезе стран или валют, но и в более узком спектре, а точнее — по конкретному виду событий.

В принципе, можно сказать, что встроенные функции обеспечивают фильтрацию событий по нескольким основным условиям: время, страна, валюта или вид события. Для остальных атрибутов, таких как важность или сектор экономики, требуется реализовать собственную фильтрацию, и мы займемся этим в дальнейшем. А пока представим функцию *CalendarValueLastByEvent*.

int CalendarValueLastByEvent(ulong id, ulong &change_id, MqlCalendarValue &values[])

Функция заполняет передаваемый по ссылке массив *values* записями о событиях конкретного вида с идентификатором *id*, которые произошли с момента *change_id*. Данный параметр *change_id* является одновременно и входным, и выходным: вызывающий код передает в нем метку прошлого состояния календаря, после которой запрашиваются изменения, а при возврате управления функция записывает в *change_id* текущую метку состояния базы календаря. Её следует использовать при следующем вызове функции.

Если в функцию передать нулевой *change_id*, то она не заполняет массив, а просто сообщает через параметр *change_id* текущее состояние базы.

Массив может быть динамическим (тогда он будет автоматически подстроен под объем данных) или фиксированного размера (если его размер окажется недостаточным, скопируется только то, что уместится).

Выходное значение функции равно количеству элементов, скопированных в массив *values*. Если изменений нет, или при вызове было указано *change_id = 0*, функция вернет 0.

Для проверки на ошибку следует анализировать встроенную переменную *_LastError*. Некоторые из возможных кодов ошибок:

- 4004 – ERR_NOT_ENOUGH_MEMORY (недостаточно памяти для выполнения запроса),
- 5401 – ERR_CALENDAR_TIMEOUT (превышено время ожидания запроса),
- 5400 – ERR_CALENDAR_MORE_DATA (размер фиксированного массива недостаточен для получения всех значений).

Мы не станем приводить отдельный пример для *CalendarValueLastByEvent*. Вместо этого обратимся к более сложной, но востребованной задаче по запросу и фильтрации записей календаря с произвольными условиями на атрибуты новостей, где будут задействованы все "календарные" функции API. Этому будет посвящен целиком следующий раздел.

7.3.10 Фильтрация событий по множеству условий

Как мы знаем из предыдущих разделов этой главы, MQL5 API позволяет запрашивать события календаря по нескольким условиям:

- странам (*CalendarValueHistory*, *CalendarValueLast*);
- валютам (*CalendarValueHistory*, *CalendarValueLast*);
- идентификаторам видов событий (*CalendarValueHistoryByEvent*, *CalendarValueLastByEvent*);
- временному диапазону (*CalendarValueHistory*, *CalendarValueHistoryByEvent*);

- изменениям с момента предыдущего опроса календаря (*CalendarValueLast*, *CalendarValueLastByEvent*);
- конкретную новость по идентификатору (*CalendarValueById*).

Это можно обобщить в виде следующей таблицы функций (из всех *CalendarValue*-функций здесь отсутствует только *CalendarValueById* для получения одного конкретного значения).

Условия	Временной диапазон	Последние изменения
Страны	CalendarValueHistory	CalendarValueLast
Валюты	CalendarValueHistory	CalendarValueLast
События	CalendarValueHistoryByEvent	CalendarValueLastByEvent

Подобный инструментарий покрывает основные, но далеко не все востребованные сценарии анализа календаря. Поэтому на практике часто требуется реализовать в MQL5 собственные механизмы фильтрации, включающие, в частности, запросы событий по:

- нескольким странам;
- нескольким валютам;
- нескольким видам событий;
- значениями произвольных свойств событий (важность, сектор экономики, отчетный период, тип, наличие прогноза, оценочное влияние на курс, подстрока в названии события, и т.д.).

Для решения данных задач создан класс *CalendarFilter* (*CalendarFilter.mqh*).

В силу специфики встроенных функций API, часть новостных атрибутов сделана более приоритетной, чем остальные. Сюда относятся страна, валюта и диапазон дат. Их можно указать в конструкторе класса, и тогда соответствующее свойство нельзя затем динамически менять в условиях фильтрации.

Это вызвано тем, что класс фильтра будет впоследствии расширяться возможностями кэширования новостей для их чтения из тестера, и начальные условия конструктора фактически определяют контекст кэширования, в пределах которого и возможна последующая фильтрации. Например, если мы при создании объекта укажем код страны "EU", то очевидно, бессмысленно запрашивать через него новости по США или Бразилии. Аналогично с диапазоном дат: его указание в конструкторе сделает невозможным получение новостей вне диапазона.

Вместе с тем, никто не мешает создать объект без начальных условий (т.к. все параметры конструктора являются опциональными), и тогда он будет способен кэшировать и фильтровать новости по всей базе календаря (по состоянию на момент сохранения).

Кроме того, поскольку страны и валюты имеют сейчас почти однозначное отображение (за исключением Европейского союза и EUR), их передача в конструктор осуществляется через единственный параметр *context*: если в нем указать строку длиной 2 символа — подразумевается код страны (или объединения стран), а если длина равна 3-м символам — подразумевается код валюты. Для кодов "EU" и "EUR", еврозона является подмножеством "EU" (в рамках стран с официальными договорами). В особых случаях, когда интерес представляют страны Евросоюза вне зоны евро, их также можно описать контекстом "EU". При необходимости, более узкие условия на новости по валютам этих стран (BGN, HUF, DKK, ISK, PLN, RON, HRK, CZK, SEK) можно

добавить в фильтр динамически, с помощью методов, которые мы представим позднее. Однако ввиду экзотичности, нет гарантий, что такие новости попадут в календарь.

Приступим к изучению класса.

```
class CalendarFilter
{
protected:
   // начальные (необязательные) условия, задаваемые в конструкторе, инварианты
   string context;    // страна или валюта
   datetime from, to; // диапазон дат
   bool fixedDates;   // если 'from'/'to' переданы в конструкторе - их нельзя менять

   // выделенные селекторы (страны/валюты/идентификаторы видов событий)
   string country[], currency[];
   ulong ids[];

   MqlCalendarValue values[]; // отфильтрованные результаты

   virtual void init()
   {
      fixedDates = from != 0 || to != 0;
      if(StringLen(context) == 3)
      {
         PUSH(currency, context);
      }
      else
      {
         // даже если context равен NULL, берем его для опроса всей базы календаря
         PUSH(country, context);
      }
   }
   ...
public:
   CalendarFilter(const string _context = NULL,
      const datetime _from = 0, const datetime _to = 0):
      context(_context), from(_from), to(_to)
   {
      init();
   }
   ...
```

Под страны и валюты выделены два массива *country* и *currency*. Если они не заполнены из *context*-а во время создания объекта, то затем MQL-программа получит возможность добавить условия на несколько стран или валют, чтобы осуществить по ним комбинированный запрос новостей.

Для хранения условий на все прочие атрибуты новостей в объекте *CalendarFilter* описан массив *selectors*, с размером 3 по второму измерению. Можно сказать, что это некая таблица, в которой каждая строка имеет 3 колонки.

```
long selectors[][3];   // [0] - свойство, [1] - значение, [2] - условие
```

По 0-му индексу будут располагаться идентификаторы свойств новостей. Поскольку атрибуты разнесены по трем таблицам базы (*MqlCalendarCountry*, *MqlCalendarEvent*, *MqlCalendarValue*) они описаны с помощью элементов обобщенного перечисления ENUM_CALENDAR_PROPERTY (*CalendarDefines.mqh*).

```
enum ENUM_CALENDAR_PROPERTY
{                                      // +/- означает поддержку фильтрации по полю
   CALENDAR_PROPERTY_COUNTRY_ID,       // -ulong
   CALENDAR_PROPERTY_COUNTRY_NAME,     // -string
   CALENDAR_PROPERTY_COUNTRY_CODE,     // +string (2 символа)
   CALENDAR_PROPERTY_COUNTRY_CURRENCY, // +string (3 символа)
   CALENDAR_PROPERTY_COUNTRY_GLYPH,    // -string (1 символ)
   CALENDAR_PROPERTY_COUNTRY_URL,      // -string

   CALENDAR_PROPERTY_EVENT_ID,         // +ulong (ID вида события)
   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+ символов или с символом подстан

   CALENDAR_PROPERTY_RECORD_ID,        // -ulong
   CALENDAR_PROPERTY_RECORD_TIME,      // +datetime
   CALENDAR_PROPERTY_RECORD_PERIOD,    // +datetime (как 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,  // +нестандарное (previous или revised если ес

   CALENDAR_PROPERTY_CHANGE_ID,        // -ulong (зарезервировано)
};
```

По индексу 1 будут храниться значения, для сравнения с ними в условиях отбора новостных записей. Например, если потребуется установить фильтр по сектору экономики, то в *selectors[i][0]* запишем CALENDAR_PROPERTY_EVENT_SECTOR, а в *selectors[i][1]* — одно из значений стандартного перечисления ENUM_CALENDAR_EVENT_SECTOR.

Наконец, последняя колонка (под 2-м индексом) зарезервирована для операции сравнения значения селектора со значением атрибута в новости: все поддерживаемые операции сведены в перечисление IS. Напомним его.

```
enum IS
{
   EQUAL,
   NOT_EQUAL,
   GREATER,
   LESS,
   OR_EQUAL,
   ...
};
```

Похожий подход нам уже встречался в *TradeFilter.mqh*. Таким образом, мы сможем компоновать условия не только на равенство значений, но и на неравенство или больше/меньше. Например, легко представить фильтр на поле CALENDAR_PROPERTY_EVENT_IMPORTANCE, которое должно быть больше (GREATER), чем CALENDAR_IMPORTANCE_LOW (это — элемент стандартного перечисления ENUM_CALENDAR_EVENT_IMPORTANCE), что означает выборку новостей средней и высокой важности.

Следующим перечислением, определенным специально для календаря, является ENUM_CALENDAR_SCOPE. Поскольку фильтрация календаря часто связана с отрезками времени, здесь перечислены наиболее востребованные из них.

```
#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
};
```

Все перечисления вынесены в отдельный заголовочный файл *CalendarDefines.mqh*.

Но вернемся к классу *CalendarFilter*. Тип массива *selectors* в нем — *long*, что подходит для хранения значений почти всех задействованных типов: перечислений, даты и времени, идентификаторов, целых чисел и даже значений экономический показателей, потому что они хранятся в календаре в виде *long*-чисел (в миллионных долях от вещественных величин). Однако что делать со строковыми свойствами?

Данная проблема решена за счет массива строк *stringCache*, в который будут добавляться все строки, упомянутые в условиях фильтров.

```
class CalendarFilter
{
protected:
   ...
   string stringCache[];  // кэш всех строк в 'selectors'
   ...
```

Тогда вместо значения строки в *selectors[i][1]* легко сохранить индекс элемента в массиве *stringCache*. Сейчас мы покажем это в деталях.

Для заполнения массива *selectors* условиями фильтров предусмотрено несколько *let*-методов, в частности, для перечислений:

```
class CalendarFilter
{
...
public:
   // здесь обрабатываются все поля типов перечислений
   template<typename E>
   CalendarFilter *let(const E e, const IS c = EQUAL)
   {
      const int n = EXPAND(selectors);
      selectors[n][0] = resolve(e); // по типу E вернуть элемент ENUM_CALENDAR_PROPER
      selectors[n][1] = e;
      selectors[n][2] = c;
      return &this;
   }
   ...
```

Для фактических значений показателей:

```
   // здесь обрабатываются поля:
   // CALENDAR_PROPERTY_RECORD_ACTUAL, CALENDAR_PROPERTY_RECORD_PREVIOUS,
   // CALENDAR_PROPERTY_RECORD_REVISED, CALENDAR_PROPERTY_RECORD_FORECAST,
   // а также CALENDAR_PROPERTY_RECORD_PERIOD (как 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;
   }
   ...
```

И для строк:

```
// здесь условия на все строковые свойства (с сокращениями)
CalendarFilter *let(const string find, const IS c = EQUAL)
{
   const int wildcard = (StringFind(find, "*") + 1) * 10;
   switch(StringLen(find) + wildcard)
   {
   case 2:
      // если начальный контекст отличен от страны, мы можем дополнить его страной
      // иначе фильтр игнорируется
      if(StringLen(context) != 2)
      {
         if(ArraySize(country) == 1 && StringLen(country[0]) == 0)
         {
            country[0] = find; // сужение "всех стран" до одной (можно добавить ещ
         }
         else
         {
            PUSH(country, find);
         }
      }
      break;
   case 3:
      // фильтр на валюту можем задать, только если в начальном контексте её не бь
      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;
}
```

В перегрузке метода для строк обратите внимание, что строки длиной 2 и 3 символа (если они без шаблонной звездочки '*', которая является заменой для произвольной последовательности знаков) попадают в массивы стран и символов соответственно, а все остальные строки —

трактуются как фрагменты названия или источника новостей, и оба эти поля задействуют *stringCache* и *selectors*.

Особым образом в классе поддержана и фильтрация по виду (идентификатору) событий.

```
protected:
   ulong ids[];           // фильтруемые виды событий
   ...
public:
   CalendarFilter *let(const ulong event)
   {
      PUSH(ids, event);
      return &this;
   }
   ...
```

Таким образом, в число приоритетных фильтров (которые обрабатываются вне массива selectors), входят не только страны, валюты, диапазон дат, но и идентификаторы видов событий. Такое конструктивное решение вызвано тем, что именно эти параметры могут передаваться в те или иные API-функции календаря как входные. Все остальные атрибуты новостей мы получаем как выходные значения полей в массивах структур (*MqlCalendarValue*, *MqlCalendarEvent*, *MqlCalendarCountry*). Именно по ним мы будет выполнять дополнительную фильтрацию, согласно правилами в массиве selectors.

Все *let*-методы возвращают указатель на объект, что позволяет нанизывать их вызовы в цепочку. Например, так:

```
CalendarFilter f;
f.let(CALENDAR_IMPORTANCE_LOW, GREATER) // важные и умеренно важные новости
  .let(CALENDAR_TIMEMODE_DATETIME) // только события с точным временем
  .let("DE").let("FR") // парочка стран, или, на выбор...
  .let("USD").let("GBP") // ...парочка валют (но оба условия сразу не сработают)
  .let(TimeCurrent() - MONTH_LONG, TimeCurrent() + WEEK_LONG) // диапазон дат "вокруг
  .let(LONG_MIN, CALENDAR_PROPERTY_RECORD_FORECAST, NOT_EQUAL) // есть прогноз
  .let("farm"); // полнотекстовой поиск по названиям новостей
```

Условия по странам и валютам можно, в принципе, комбинировать, но следует иметь в виду, что несколько значений можно задать только либо для стран, либо для валют, но не для того и другого. Один из этих двух аспектов контекста (любой из двух) в текущей реализации поддерживает только одно или ни одного значения (т.е. отсутствие фильтра по нему). Например, при выбранной валюте "EUR" можно сузить контекст поиска новостей только по Германии и Франции (коды стран "DE" и "FR") — в результате будут отброшены новости ЕЦБ и евростата, а также, в частности, по Италии и Испании. Однако указание "EUR" в данном случае является избыточным, так как в Германии и Франции нет других валют.

Поскольку класс использует встроенные функции, в которых параметры *country* и *currency* применяются к новостям с помощью операции логического И, проверяйте непротиворечивость условий фильтрации.

После того как вызывающий код настроит условия фильтрации, следует на их основе произвести выборку новостей. Этим занимается публичный метод *select* (приводится с упрощениями).

```
public:
   bool select(MqlCalendarValue &result[])
   {
      int count = 0;
      ArrayFree(result);
      if(ArraySize(ids)) // идентификаторы видов событий
      {
         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
      {
         // несколько стран или валют, выбираем за основу то, чего из них больше,
         // а из меньшего массива используется только первый элемент
         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;
}
```

В зависимости от того, какие из массивов с приоритетными атрибутами заполнены, метод вызывает разные функции API для опроса календаря:

- если заполнен массив *ids*, в цикле для всех идентификаторов вызывается *CalendarValueHistoryByEvent*;
- если заполнен массив *country*, и он больше, чем массив валют, выполняется цикл по странам с вызовом *CalendarValueHistory*;
- если заполнен массив *currency*, и он больше или равен размеру массива стран, выполняется цикл по валютам с вызовом *CalendarValueHistory*;

Каждый вызов функции заполняет временный массив структур *MqlCalendarValue temp[]*, который последовательно аккумулируется в массиве-параметре *result*. После записи в него всех подходящих новостей по основным условиям (даты, страны, валюты, идентификаторы), если они есть, в дело вступает вспомогательный метод *filter*, который прореживает массив на основе условий в *selectors*. В завершении метода *select* производится сортировка новостей в хронологическом порядке, который может быть нарушен из-за объединения результатов множественных запросов "календарных" функций. Для сортировки используется макрос SORT_STRUCT, рассмотренный в разделе Сравнение, сортировка и поиск в массивах.

Метод *filter* вызывает для каждого элемента массива новостей рабочий метод *match*, возвращающий логический признак того, подходит ли новость под условия фильтра. Если нет, элемент удаляется из массива.

```
protected:
   void filter(MqlCalendarValue &result[])
   {
      for(int i = ArraySize(result) - 1; i >= 0; --i)
      {
         if(!match(result[i]))
         {
            ArrayRemove(result, i, 1);
         }
      }
   }
   ...
```

Наконец, метод *match* анализирует наш массив *selectors* и сравнивает его с полями переданной структуры *MqlCalendarValue*. Здесь код приводится с сокращениями.

```
bool match(const MqlCalendarValue &v)
  {
     MqlCalendarEvent event;
     if(!CalendarEventById(v.event_id, event)) return false;

     // цикл по всем условиям фильтров, кроме стран, валют, дат, ID,
     // которые были уже ранее использованы при вызовах Calendar-функций
     for(int j = 0; j < ArrayRange(selectors, 0); ++j)
     {
        long field = 0;
        string text = NULL;

        // получаем значение поля из новости или её описания
        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 или revised (если есть)
           field = v.revised_prev_value != LONG_MIN ? v.revised_prev_value : v.prev_
           break;
        case CALENDAR_PROPERTY_RECORD_FORECAST:
           field = v.forecast_value;
           break;
```

```
         ...
         }

         // сравниваем значение с условием фильтра
         if(text == NULL) // числовые поля
         {
            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 // строковые поля
         {
            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;
   }
```

Методы *equal* и *greater* перенесены почти без изменений из наших предыдущих наработок с классами-фильтрами.

На этом задача фильтрации, в целом, решена, то есть MQL-программа может использовать объект *CalendarFilter* следующим образом:

```
CalendarFilter f;
f.let()... // серия вызовов метода let для настройки условий фильтрации
MqlCalendarValue records[];
if(f.select(records))
{
   ArrayPrint(records);
}
```

Но на самом деле метод *select* умеет еще кое-что важное, что мы оставили для факультатива.

Во-первых, в получаемом списке новостей желательно тем или иным образом вставить разделитель (*delimiter*) между прошлым и будущим, чтобы за него мог зацепиться глаз. В принципе, эта возможность является крайне важной для календарей, но по каким-то причинам отсутствует в пользовательском интерфейсе MetaTrader 5 и на сайте mql5.com. Наша реализация умеет вставлять между прошлым и будущим пустую структуру, которую лишь остается наглядно отобразить (чем мы займемся чуть ниже).

Во-вторых, размер результирующего массива может оказаться довольно большим (особенно на первых этапах подбора настроек), и потому метод *select* дополнительно предоставляет возможность ограничить размер массива (*limit*). Это делается за счет удаления элементов наиболее далеко отстоящих от текущего времени.

Таким образом, полный прототип метода выглядит так:

```
bool select(MqlCalendarValue &result[],
   const bool delimiter = false, const int limit = -1);
```

По умолчанию разделитель не вставляется, и массив не усекается.

Парой абзацев выше была вскользь упомянута дополнительная подзадача фильтрации — визуализация полученного массива. В классе *CalendarFilter* имеется специальный метод *format*, который превращает передаваемый массив структур *MqlCalendarValue &data[]* в массив удобочитаемых строк *string &result[]*. С телом метода можно ознакомиться в прилагаемом файле *CalendarFilter.mqh*.

```
bool format(const MqlCalendarValue &data[],
   const ENUM_CALENDAR_PROPERTY &props[], string &result[],
   const bool padding = false, const bool header = false);
```

Какие именно поля из *MqlCalendarValue* мы хотим отобразить, задается в массиве props. Напомним, что в перечислении ENUM_CALENDAR_PROPERTY имеются поля из всех трех зависимых структур календаря, так что MQL-программа может автоматически выводить не только экономические показатели из конкретной записи о событии, но и его название, характеристики, код страны или валюты — это все берет на себя метод *format*.

Каждая строка в выходном массиве *result* содержит текстовое представление значения одного из полей (число, описание, элемент перечисления). Размер массива *result* равен произведению количества структур на входе (в *data*) на количество отображаемых полей (в *props*). Опциональный параметр *header* позволяет добавить в начало выходного массива ряд с названиями полей (колонок). Параметр *padding* управляет генерацией дополнительных пробелов в тексте, чтобы таблицу было удобно выводить моноширинным шрифтом (например, в журнал).

В завершение описания класса *CalendarFilter* упомянем еще один важный публичный метод — *update*.

```
bool update(MqlCalendarValue &result[]);
```

Он по своей структуре почти полностью повторяет select, но вместо вызовов функций *CalendarValueHistoryByEvent* и *CalendarValueHistory* использует *CalendarValueLastByEvent* и *CalendarValueLast*. Назначение метода очевидно: он запрашивает календарь на наличие свежих изменений, соответствующих условиям фильтрации. Но для своей работы он требует идентификатор изменений. И такое поле действительно определено в классе: в первый раз оно заполняется внутри метода *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);
      ...
   }
```

Кое-какие нюансы класса *CalendarFilter* все еще "остались за кадром", но к некоторым из них мы обратимся в следующих разделах.

Давайте проверим фильтр в деле: сначала в простом скрипте *CalendarFilterPrint.mq5*, а затем в более практичном индикаторе *CalendarMonitor.mq5*.

Во входных параметрах скрипта можно задать контекст (код страны или валюту), временной диапазон, строку для полнотекстного поиска по названиям событий и ограничить размер результирующей таблицы с новостями.

```
input string Context; // Context (страна - 2 знака, currency - 3 знака, пусто - без ф
input ENUM_CALENDAR_SCOPE Scope = SCOPE_MONTH;
input string Text = "farm";
input int Limit = -1;
```

С учетом параметров создается глобальный объект фильтра.

```
CalendarFilter f(Context, TimeCurrent() - Scope, TimeCurrent() + Scope);
```

Затем в *OnStart* настраивается пара дополнительных постоянных условий (средняя и высокая важность событий), наличие прогноза (поле не равно LONG_MIN), а также в объект передается поисковая строка.

```
void OnStart()
{
   f.let(CALENDAR_IMPORTANCE_LOW, GREATER)
      .let(LONG_MIN, CALENDAR_PROPERTY_RECORD_FORECAST, NOT_EQUAL)
      .let(Text); // с поддержкой символа-заместителя '*'
      // NB: строки длиной 2 или 3 символа без '*' будут трактоваться
      // как код страны или валюты соответственно
```

Далее вызывается метод *select* и полученный массив структур *MqlCalendarValue* форматируется в таблицу с 9-ю колонками методом *format*.

```
 MqlCalendarValue records[];
   // применяем условия фильтра и получаем результат
   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);

      // выводим отформатированный результат
      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));
         }
      }
   }
}
```

Ячейки таблицы "склеиваются" в ряды и выводятся в журнал.

С настройками по умолчанию (то есть, по всем странам и валютам, с фрагментом "farm" в названии событий средней и высокой важности) можно получить примерно такое расписание.

```
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⊠ |
2022.06.02 15:15 |  USD | ADP Nonfarm Employment Change |      HIGH |  +128 |  -225 |
2022.06.02 15:30 |  USD |      Nonfarm Productivity q/q |  MODERATE |  -7.3 |  -7.5 |
2022.06.03 15:30 |  USD |              Nonfarm Payrolls |      HIGH |  +390 |   -19 |
2022.06.03 15:30 |  USD |      Private Nonfarm Payrolls |  MODERATE |  +333 |    +8 |
2022.06.09 08:30 |  EUR |          Nonfarm Payrolls q/q |  MODERATE |  +0.3 |  +0.3 |
               – |    – |                             – |         – |     – |     – |
2022.07.07 15:15 |  USD | ADP Nonfarm Employment Change |      HIGH |  +nan |  -263 |
2022.07.08 15:30 |  USD |              Nonfarm Payrolls |      HIGH |  +nan |  -229 |
2022.07.08 15:30 |  USD |      Private Nonfarm Payrolls |  MODERATE |  +nan |   +51 |
```

Теперь займемся индикатором *CalendarMonitor.mq5*. Его назначение: отображать пользователю на графике текущую подборку событий в соответствии с заданными фильтрами. Для визуализации таблицы будет использоваться уже знакомый нам класс табло (*Tableau.mqh*, см. раздел Расчет залога для будущего ордера). Буферов и диаграмм индикатор не имеет.

Входные параметры позволяют задать диапазон временного окна (*Scope*), а также глобальный контекст для объекта *CalendarFilter* — либо кодом валюты или страны в строке *Context* (по умолчанию пусто, т.е. без ограничений), либо с помощью логического флага *UseChartCurrencies*. Он по умолчанию включен, и именно им рекомендуется пользоваться, чтобы автоматически получать новости тех валют, которые составляют рабочий инструмент графика.

```
input string Context; // Context (country - 2 chars, currency - 3 chars, empty - all)
input ENUM_CALENDAR_SCOPE Scope = SCOPE_WEEK;
input bool UseChartCurrencies = true;
```

Дополнительные фильтры можно наложить на тип события, сектор и важность.

```
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
```

Важность задает нижнюю границу отбора, а не точное соответствие. Таким образом, установленное по умолчанию значение IMPORTANCE_MODERATE захватит не только умеренную, но и высокую важность.

Внимательные читатель обратит внимание, что здесь использованы неизвестные перечисления: ENUM_CALENDAR_EVENT_TYPE_EXT, ENUM_CALENDAR_EVENT_SECTOR_EXT, ENUM_CALENDAR_EVENT_IMPORTANCE_EXT. Они находятся в уже упоминавшемся файле *CalendarDefines.mqh* и почти один в один совпадают с аналогичными встроенными перечислениями. Единственное отличие заключается в том, что в них добавлен элемент, означающий "любое" значение. Необходимость в описании подобных перечислений возникла для упрощения ввода условий: сейчас фильтр для каждого поля настраивается с помощью выпадающего списка, где можно выбрать как одно из значений, так и отключить фильтр. Если бы не добавленный элемент перечисления, пришлось бы вводить в интерфейс для каждого поля логический флаг "включено/выключено".

Кроме того входные параметры позволяют запрашивать события по наличию в них актуальных, прогнозных и предыдущих показателей, а также с поиском строки текста (*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;
```

Объекты *CalendarFilter* и *Tableau* описаны на глобальном уровне.

```
CalendarFilter f(Context);
AutoPtr<Tableau> t;
```

Обратите внимание, что фильтр создается единожды, а таблица представлена автоуказателем и будет пересоздаваться динамически в зависимости от размера полученных данных.

Настройки фильтра делаются в *OnInit* последовательными вызовами *let*-методов согласно входным параметрам.

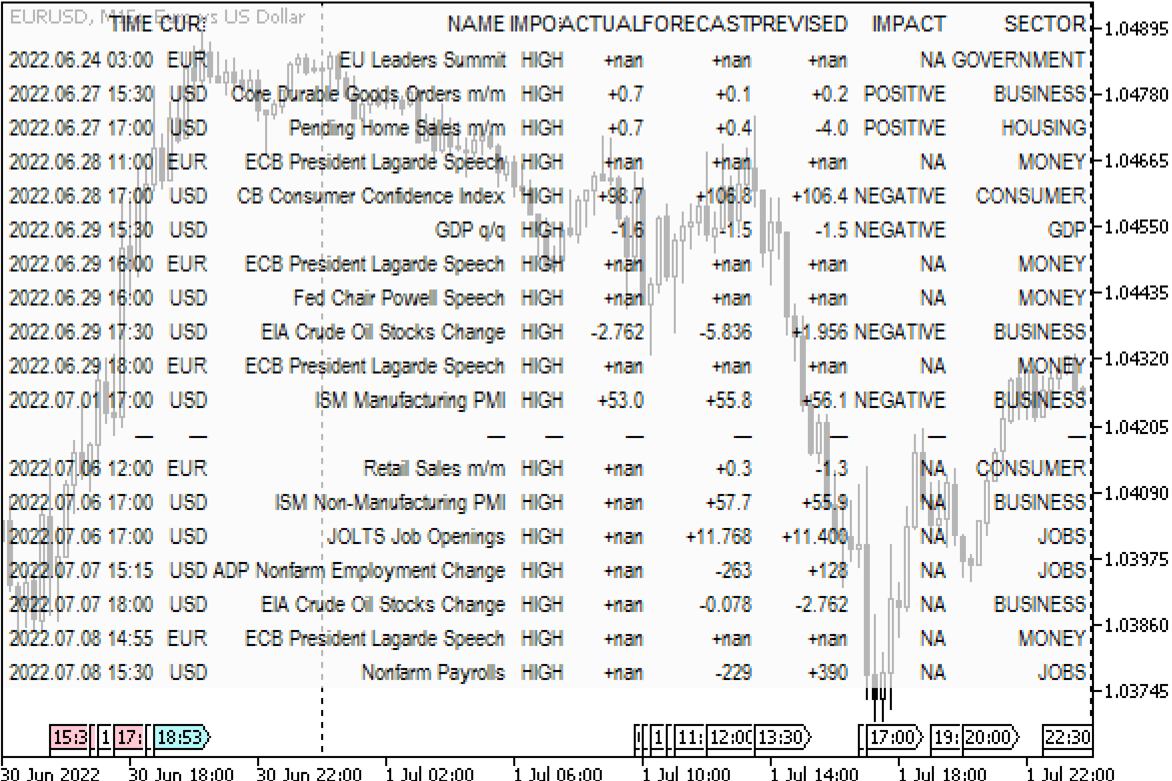
```
int OnInit()
{
   if(!f.isLoaded()) return INIT_FAILED;
   
   if(UseChartCurrencies)
   {
      const string base = SymbolInfoString(_Symbol, SYMBOL_CURRENCY_BASE);
      const string profit = SymbolInfoString(_Symbol, SYMBOL_CURRENCY_PROFIT);
      f.let(base);
      if(base != profit)
      {
         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;
}
```

В конце запускается секундный таймер. Вся работа выполняется в *OnTimer*.

```
void OnTimer()
{
   static const ENUM_CALENDAR_PROPERTY props[] = // колонки таблицы
   {
      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[];

   f.let(TimeCurrent() - Scope, TimeCurrent() + Scope); // временнОе окно сдвигаем ка

   const ulong trackID = f.getChangeID();
   if(trackID) // если состояние уже снимали, проверяем на изменения
   {
      if(f.update(records)) // запрашиваем изменения по фильтрам
      {
         // если изменения есть, уведомляем пользователя
         string result[];
         f.format(records, props, result);
         for(int i = 0; i < ArraySize(result) / p; ++i)
         {
            Alert(SubArrayCombine(result, " | ", i * p, p));
         }
         // "проваливаемся" далее на обновление таблицы
      }
      else if(trackID == f.getChangeID())
      {
         return; // календарь без изменений
      }
   }

   // запрашиваем полный набор новостей по фильтрам
   f.select(records, true, Limit);

   // выводим таблицу новостей на график
   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);
   }
```

Если запустить индикатор на графике EURUSD с настройками по умолчанию можем получить следующую картину.



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

Отфильтрованный и отформатированный набор новостей на графике

7.3.11 Перенос базы календаря в тестер

Календарь доступен для MQL-программ только в режиме онлайн, в связи с чем тестирование новостных торговых стратегий представляет некоторую сложность. Одно из решений — самостоятельное создание некоего образа календаря, то есть кэша, и последующее его использование внутри тестера. Технологии хранения кэша могут быть разными, например, файлы или встроенная база данных SQLite. В данном разделе мы покажем реализацию с применением файла.

В любом случае, при использовании кэша календаря следует помнить, что он соответствует конкретному моменту времени X. Во всех "старых" событиях (финансовых отчетах) более ранних, чем X, уже проставлены актуальные значения, а в более поздних ("будущих" относительно X) — актуальных значений нет и не будет, пока не появится новая, более свежая копия кэша. Иными словами, тестировать индикаторы и эксперты правее X не имеет смысла. А левее X следует избегать заглядывания вперед, то есть не читать актуальные показатели до времени публикации каждой конкретной новости.

> Внимание! При запросе данных календаря в терминале время всех событий сообщается с учетом текущей временной зоны сервера, включая и возможную поправку на "летнее" время (как правило, это означает увеличение меток времени на 1 час). Это синхронизирует выпуск новостей со временем котировок в режиме онлайн. Однако прошлые переводы часов (полгода, год назад и более) отображаются только в котировках, но не в событиях календаря. Вся база календаря считывается через MQL5 по текущему часовому поясу сервера. Из-за этого любой создаваемый архив календаря будет содержать корректные временные метки для тех событий, которые происходили при том же режиме DST (включенном или выключенном), что был активен в момент сохранения. Для событий в "противоположных" "полугодиях" требуется самостоятельно делать поправку на час после чтения архива. В рассматриваемых далее примерах этот нюанс опущен.

Назовем класс кэша *CalendarCache* и поместим его в файл *CalendarCache.mqh*. Нам потребуется сохранять в файле все 3 таблицы базы календаря (*MqlCalendarCountry*, *MqlCalendarEvent*, *MqlCalendarValue*). Напомним, что MQL5 предоставляет функции *FileWriteArray* и *FileReadArray* (см. Запись и чтение массивов), способные напрямую писать и считывать массивы простых структур в файлы. Однако 2 из 3 структур в нашем случае не являются простыми, поскольку в них имеются строковые поля. Поэтому нам потребуется механизм отдельного сохранения строк, похожий на тот, что мы уже использовали в классе *CalendarFilter* (там был массив строк *stringCache*, а в фильтрах указывался индекс нужной строки из этого массива).

Чтобы не перемешивать строки из разных "календарных" структур в одном "словаре" мы подготовим шаблонный класс *StringRef*: параметром-типом T будет выступать любая из *MqlCalendar*-структур. За счет этого мы получим отдельный кэш строк для стран, и отдельный кэш строк для видов событий.

```
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[];
```

Сами строки "складируются" в массиве *cache* с помощью *operator=*, и извлекаются из него с помощью *operator[]* (с фиктивным индексом, который всегда опущен). В каждом объекте хранится лишь индекс строки в массиве. Массив *cache* объявлен статическим, так что будет аккумулировать все строковые поля одной структуры T. Желающие могут поменять принцип кэширования таким образом, чтобы каждое поле структуры имело свой массив, но нам это не принципиально.

Запись массива в файл и чтение из файла выполняются парой статических методов *save* и *load*: оба принимают в качестве параметра дескриптор файла.

С учетом класса *StringRef* опишем структуры, дублирующие стандартные структуры календаря, но в которых вместо строковых полей используются объекты *StringRef*. Например, для *MqlCalendarCountry* получим *MqlCalendarCountryRef*. Стандартная и модифицированная структуры копируются друг в друга также перегруженными операторами '=' и '[]'.

```
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;
   }
};
```

Обратите внимание, что в операторах присваивания первого метода работает перегрузка '=' из *StringRef*, за счет чего все строки попадают в массив *StringRef<MqlCalendarCountry>::cache*. Во

втором методе вызовы оператора '[]' невидимым образом получают адрес строки и возвращают из *StringRef* непосредственно строку, хранящуюся по этому адресу в массиве *cache*.

Структура *MqlCalendarEventRef* определена похожим образом, но в ней всего 3 поля (*source_url*, *event_code*, *name*) требуют замены типа *string* на *StringRef<MqlCalendarEvent>*. Структура *MqlCalendarValue* подобных преобразований не требует, так как в ней нет строковых полей.

На этом подготовительные этапы завершены, и можно приступать к основному рабочему классу кэша *CalendarCache*.

Исходя из общих соображений, а также для совместимости с уже разработанным классом *CalendarFilter* опишем в кэше поля, задающие контекст (страну или валюту), диапазон дат хранимых событий, и момент генерации кэша (время X, переменная 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)
   {
      ...
   }
```

В принципе, прописывать ограничения при создании кэша из календаря не имеет особого смысла: более практичным, вероятно, является полный кэш, так как его размер не является критичным — пара десятков мегабайт на середину 2022 года (это история с 2007 года и планируемые события до 2024). Впрочем, ограничения могут пригодиться для демонстрационных программ с искусственно урезанным функционалом.

Очевидно, что в кэше следует предусмотреть массивы календарных структур для хранения всех данных.

```
   MqlCalendarValue values[];
   MqlCalendarEvent events[];
   MqlCalendarCountry countries[];
   ...
```

Изначально они заполняются из базы календаря методом *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)))
   {
      // объект не готов, t = 0
   }
   else
   {
      t = TimeTradeServer();
   }
   return (bool)t;
}
```

Признаком работоспособности кэша выступает поле *t* со временем заполнения массивов.

Объект заполненного кэша можно записать в файл с помощью метода *save*. В начале файла идет заголовок CALENDAR_CACHE_HEADER — это строка "MQL5 Calendar Cache\r\nv.1.0\r\n", позволяющая убедиться при чтении в правильности формата. Далее сохраняются переменные *context*, *from*, *to* и *t*, а также массив *values* "как есть". Перед самим массивом мы записываем его размер, чтобы восстановить его при чтении.

```
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);
   ...
```

С массивами *events* и *countries* в дело вступают наши структуры-"обертки" с суффиксом "Ref". Вспомогательный метод *store* конвертирует массив *events* в массив простых структур *erefs*, в которых строки заменены на номера в "словаре" строк *StringRef<MqlCalendarEvent>*. Такие простые структуры уже можно обычным способом записать в файл, но для их последующего чтения требуется также сохранить все строки "словаря" (вызов *StringRef<MqlCalendarEvent>::save(handle)*). Структуры стран преобразуются и сохраняются в файл аналогичным образом.

```
   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;
}
```

Упомянутый метод *store* довольно прост: в нем в цикле по элементам выполняется перегруженный оператор присваивания в структурах *MqlCalendarEventRef* или *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];
   }
}
```

Для загрузки полученного файла в объект кэша написан "зеркальный" метод *load*. Он в том же порядке читает данные из файла в переменные и массивы, попутно выполняя обратные преобразования строковых полей для видов событий и стран.

```
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; // не наш формат

   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);
   ... // здесь будет кое-что еще
}
```

Вспомогательный метод *restore* использует в цикле по элементам перегрузку оператора '[]' в структурах *MqlCalendarEventRef* или *MqlCalendarCountryRef*, чтобы по номеру строки получить саму строку и присвоить её в стандартную структуру *MqlCalendarEvent* или *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][];
   }
}
```

Уже на этом этапе мы могли бы написать на основе класса *CalendarCache* простой тестовый индикатор, запустить его на онлайн-чарте и сохранить в файл с кэшем календаря. Затем файл

можно было бы загрузить из копии индикатора в тестере и получить полный набор событий. Однако для практических разработок этого недостаточно.

Дело в том, что для быстрого доступа к данным требуется обеспечить *индексирование* — широко известный в программировании принцип, с которым мы еще встретимся в рамках главы про базы данных. По идее, мы могли бы использовать встроенный движок SQLite для хранения кэша, и тогда получили бы индексы "бесплатно", но об этом чуть позже.

Суть индексирования легко понять, если представить, как в нашем кэше эффективно реализовать аналоги стандартных функций календаря. Например, в функцию *CalendarValueById* передается идентификатор события. Прямой перебор записей в массиве *values* был бы очень затратным по времени. Поэтому требуется дополнить массив некоторой "структурой данных", которая позволила бы оптимизировать поиск. "Структура данных" взята в кавычки, потому что речь не о сущности языка программирования (*struct*), а в целом об архитектуре построения данных — она может состоять из разных частей и основываться на разных организационных принципах. Разумеется, дополнительные данные потребуют памяти, но обмен памяти на скорость — обычное явление в программировании.

Наиболее простым решением для индексирования выступает отдельный двумерный массив, отсортированный по возрастанию, так что к нему можно применить быстрый поиск с помощью функции *ArrayBsearch*. По второму измерению достаточно двух элементов: значения с индексами *[i][0]*, по которым и выполняется сортировка, содержат идентификаторы, а значения *[i][1]* — порядковые позиции в массиве структур.

Также часто применяется *хэширование* — преобразование исходных значений в некие ключи (хэши, целые числа) таким образом, что это обеспечивает минимальное количество коллизий (совпадений ключей для разных исходных данных). Фундаментальное свойство ключей — близкое к равномерному случайное распределение их значений, за счет чего они могут использоваться как индексы в заранее распределенных массивах ("корзинах"). Вычисление хэш-функции для одного элемента исходных данных — это быстрый процесс, который фактически выдает адрес самого элемента. По такому принципу, в частности, работают известные "структуры данных" хэш-карты (hashmap).

Если два исходных значения все же получают один и тот же хэш (хотя это бывает редко), они выстраиваются в список для своего ключа, и внутри списка уже будет выполняться последовательный поиск перебором. Но поскольку хэш-функции выбираются так, чтобы количество совпадений было мало, то обычно поиск достигает цели сразу после вычисления хэша.

Для демонстрации мы используем в классе *CalendarCache* оба подхода: хэширование и бинарный поиск.

В поставку MetaTrader 5 входит набор классов для создания хэш-карт (MQL5/Include/Generic/HashMap.mqh), но мы обойдемся собственной более простой реализацией, в которой останется только принцип использования хэш-функции.

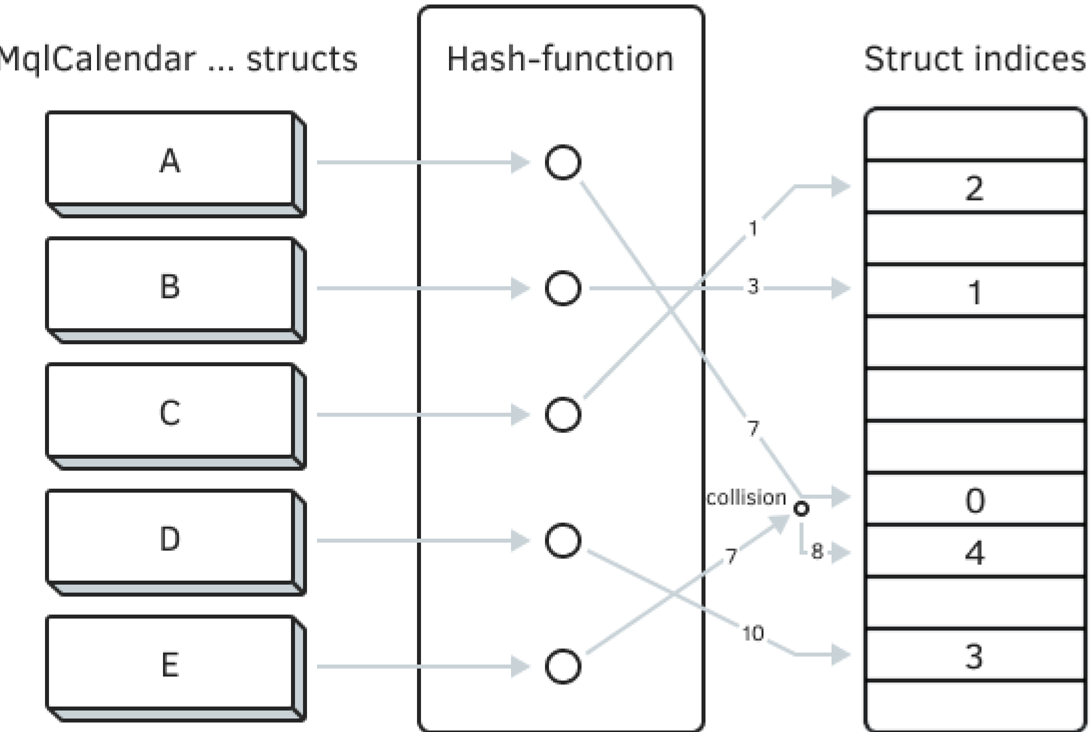


Схема индексации данных путем хэширования

Хэшировать в нашем случае достаточно только идентификаторы объектов календаря. Функция хэширования, которую мы выберем, должна будет преобразовать идентификатор в индекс внутри специального массива: в ячейку с этим индексом и будет сохранена позиция индентификатора в массиве "календарных" структур. Для стран, видов событий и конкретных новостей выделено по собственному массиву.

```
int id4country[];
int id4event[];
int id4value[];
```

В их элементах будет храниться порядковый номер записи в соответствующем массиве (*countries*, *events*, *values*).

Под каждый из массивов "переадресации" следует выделить как минимум в 2 раза больше элементов, чем количество соответствующих структур в базе (и в кэше) календаря. За счет этой избыточности мы минимизируем количество коллизий при хэшировании. Считается, что наибольшая эффективность достигается при выборе размера, равного простому числу. Поэтому в классе имеется статический метод *size2prime*, возвращающий рекомендованный размер массива хэш-"корзинок" (одного из *id4*-массивов) по количеству элементов в исходных данных.

```
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;
}
```

Весь процесс хэширования календаря описан в методе *hash*. Рассмотрим его начало на примере массива структур *countries*, а два остальных массива обрабатываются аналогично.

Итак, мы получаем рекомендованный "простой" размер индекса *id4country* из размера массива *countries*, вызвав *size2prime*. Изначально индексный массив заполняется значением -1, то есть всего его элементы свободны. Далее в цикле по странам необходимо вычислить хэш для каждого очередного идентификатора страны и найти по нему свободный индекс в массиве *id4country*. Этим занимается вспомогательный метод *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; // неудача
      }
   }
   ...
   return true; // успех
}
```

В качестве хэш-функции внутри *place* используется выражение: *(MathSwap(id) ^ 0xEFCDAB8967452301) % n*, где *id* — это наш идентификатор, а *n* — размер индексного массива. Таким образом, результат вычислений всегда приводится к валидному индексу внутри *array[]*. Принцип выбора хэш-функции — это отдельная тема, выходящая за рамки книги.

```
int place(const ulong id, const int index, int &array[])
{
   const int n = ArraySize(array);
   int p = (int)((MathSwap(id) ^ 0xEFCDAB8967452301) % n); // хэш-функция
   int attempt = 0;
   while(array[p] != -1)
   {
      if(++attempt > n / 10) // количество коллизий - не больше 1/10 от количества
      {
         return -1; // ошибка записи в индексный массив
      }
      p = (p + attempt) % n;
   }
   array[p] = index;
   return p;
}
```

Если ячейка под номером *p* в индексном массиве не занята (равна -1), мы сразу же записываем в элемент *[p]* адрес размещения "календарной" структуры. Если ячейка уже занята, пытаемся выбрать следующую по формуле *p = (p + attempt) % n*, где *attempt* — счетчик попыток (это наш закамуфлированный вариант списка элементов с совпавшим хэшем). Если количество неудачных попыток достигнет одной десятой части исходных данных, индексирование завершится ошибкой, но такое практически исключено при нашем выбранном с запасом размере индексного массива и известной природе хэшируемых данных (уникальных идентификаторов).

В результате хэширования массива структур мы получаем заполненный индексный массив (в нем остаются свободные места, но так и задумано), через который можно по идентификатору элемента календаря узнать расположение соответствующей структуры в массиве структур. Для этого имеется метод *find*, обратный по смыслу к *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); // хэш-функция
   int attempt = 0;
   while(structs[array[p]].id != id)
   {
      if(++attempt > n / 10)
      {
         return -1; // ошибка извлечения из индексного массива
      }
      p = (p + attempt) % n;
   }
   return array[p];
}
```

Покажем, как это используется на практике. Среди стандартных функций календаря есть, в частности, *CalendarCountryById* и *CalendarEventById*. Когда потребуется протестировать какую-либо MQL-программу в тестере, она не сможет напрямую обратиться к ним, но зато сможет загрузить кэш календаря в объект *CalendarCache*, и потому в нем должны быть аналогичные методы.

```
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;
}
```

Они как раз и используют метод *find* и индексные массивы *id4country* и *id4event*.

Но это не самые востребованные возможности календаря. Гораздо чаще MQL-программа с новостной стратегией нуждается в функциях *CalendarValueHistory*, *CalendarValueHistoryByEvent*, *CalendarValueLast* или *CalendarValueLastByEvent*. С помощью них обеспечивается быстрый доступ к записям календаря по времени, по стране или валюте.

Значит, класс *CalendarCache* должен предоставить аналогичные методы. И здесь мы воспользуемся вторым методом "индексирования" — через двоичный поиск в отсортированном массиве.

Для реализации вышеперечисленных методов добавим в класс еще 4 двумерных массива для установления соответствия между новостью и видом события, новостью и страной, новостью и валютой, а также новостью и временем её публикации.

```
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
```

В первом элементе каждого ряда, то есть под индексами *[i][0]* будет записываться идентификатор события, страны, валюты или время, соответственно. Во втором элементе ряда, под индексами *[i][1]* разместятся идентификаторы конкретных новостей. После однократного заполнения всех массивов они сортируются с помощью *ArraySort* по значениям *[i][0]*. Затем мы сможем искать по идентификатору, например, события *event_id* все такие новости в массиве *value2event*: функция *ArrayBsearch* вернет номер первого подходящего элемента, за которым будут следовать остальные с таким же *event_id*, пока не встретится отличный идентификатор. Порядок во второй "колонке" не определен (может быть любым).

MqlCalendarValue

id	event
...	
news01	123028
news03	123005
news05	123021
news11	123005
news17	123021
news18	123028
news22	123005
news23	123004
news29	123021
news42	123021
news49	123028
news50	123005
...	

value2event

ArraySort()

	[0]	[1]
	...	
GDP	123005	news11
	123005	news22
	123005	news03
CPI	123021	news42
	123021	news05
	123021	news29
	123021	news17
PMI	123028	news49
	123028	news18
	123028	news01
	...	

Быстрый поиск связанных структур на основе сортировки

Данная операция взаимной увязки структур разных типов осуществляется в методе *bind*. Размер каждого "связующего" массива идентичен размеру массива новостей. Проходя в цикле по всем новостям, мы пользуемся уже готовыми индексными массивами и методом *find* для быстрой адресации.

```
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;
}
```

В случае валют в качестве идентификатора берется специальное число, получаемое из строки функцией *currencyId*.

```
static ulong currencyId(const string s)
{
   union CRNC4
   {
      ushort word[4];
      ulong ul;
   } v;
   StringToShortArray(s, v.word);
   return v.ul;
}
```

Теперь мы можем, наконец, целиком представить конструктор класса *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) // метка того, что context - это имя файла
   {
      load(_context);
   }
   else
   {
      if(!update() || !hash() || !bind())
      {
         t = 0;
      }
   }
}
```

При запуске на онлайн-чарте созданный объект с параметрами по умолчанию соберет всю информацию календаря (*update*), проиндексирует её (*hash*) и свяжет между собой таблицы (*bind*). Если что-то пойдет не так на любом из этапов, признаком ошибки станет 0 в переменной *t*. В случае успеха там останется значение из функции *TimeTradeServer* (напомним, оно ставится внутри *update*). Такой готовый к работе объект можно экспортировать в файл методом *save*, описанным выше.

При запуске в тестере объект следует создавать с особым сочетанием параметров *from* и *to* (*from* > *to*), чтобы программа посчитала строку *context* именем файла и загрузила из него состояние календаря. Проще всего это сделать так:

```
CalendarCache calca("filename.cal", true);
```

Внутри метода *load* мы также вызовем *hash* и *bind*, чтобы привести объект в рабочее состояние.

```
bool load(const string filename)
{
   ... // чтение файла было показано ранее
   const bool result = hash() && bind();
   if(!result) t = 0;
   return result;
}
```

На примере функции *CalendarValueLast* покажем эквивалентную реализацию метода *calendarValueLast* (с точно таким же прототипом). В качестве идентификатора изменений кэш будет использовать текущее "серверное" время, за неимением открытого программного API для чтения таблицы изменений онлайн-календаря. Гипотетически мы могли бы воспользоваться информацией об идентификаторах изменений, сохраненных сервисом *CalendarChangeSaver.mq5*, но этот подход требует долговременного сбора статистики, прежде чем можно начать тестирование. Поэтому "серверное" время, генерируемое тестером, принято достаточно адекватной заменой.

Когда MQL-программа запросит изменения первый раз с нулевым идентификатором, просто вернем значение из *TimeTradeServer*.

```
int calendarValueLast(ulong &change, MqlCalendarValue &result[],
   const string code = NULL, const string currency = NULL)
{
   if(!change)
   {
      change = TimeTradeServer();
      return 0;
   }
   ...
```

Если идентификатор изменений уже ненулевой, продолжаем основную ветвь алгоритма.

В зависимости от содержимого параметров *code* и *currency*, находим идентификаторы страны и валюты. По умолчанию она равны 0, что означает поиск всех изменений.

```
   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;
         }
      }
   }
   ...
```

Далее, используя переданный отсчет времени *change* как начало поиска, находим все новости в *value2time* вплоть до нового, текущего значения *TimeTradeServer*. Внутри цикла с помощью метода *find* находим индекс соответствующей структуры *MqlCalendarValue* в массиве *values* и при необходимости сравниваем страну и валюту связанного вида события с желаемыми. Все новости, удовлетворяющие критериям, записываются в выходной массив *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);
   }
```

По похожему принципу реализованы методы *calendarValueHistory*, *calendarValueHistoryByEvent*, *calendarValueLastByEvent* (последний фактически делегирует всю работу рассмотренному методу *calendarValueLast*). С полным исходным кодом можно ознакомиться в прилагаемом файле *CalendarCache.mqh*.

На основе класса кэша логично создать класс-наследник *CalendarFilter*, который бы при обработке запросов обращался к кэшу, вместо календаря.

Готовое решение находится в файле *CalendarFilterCached.mqh*. Благодаря тому, что программный интерфейс кэша проектировался по кальке стандартного API, интеграция сводится лишь к пробрасыванию вызовов фильтра в объект-кэш (автоуказатель *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
   {
      // готовность определяется кэшем
      return cache[].isLoaded();
   }
```

```
};
```

Для проверки работы календаря в тестере создадим новую версию индикатора *CalendarMonitor.mq5* — *CalendarMonitorCached.mq5*.

Основные отличия заключаются в следующем.

Предполагаем, что некоторый файл кэша будет создан или уже создан под именем "xyz.cal" (в папке *MQL5/Files*) и потому подключаем его к MQL-программе директивой *tester_file*.

```
#property tester_file "xyz.cal"
```

Напомним, эта директива обеспечивает передачу кэша на любые агенты, включая распределенные (что, впрочем, более актуально для экспертов, а не индикатора). Создать файл кэша с этим (или другим именем) позволяет новая входная переменная *CalendarCacheFile*. Если пользователь изменит имя по умолчанию на что-то другое, то для работы в тестере нужно будет подправить директиву (требует перекомпиляции!), или перенести файл в общую папку терминалов (эта возможность поддержана в классе кэша, но "оставлена за кадром"), правда такой файл уже недоступен для удаленных агентов.

```
input string CalendarCacheFile = "xyz.cal";
```

Объект *CalendarFilter* теперь описан как автоуказатель, потому что в зависимости от того, где запускается индикатор, он может использовать как исходный класс *CalendarFilter*, так и производный *CalendarFilterCached*.

```
AutoPtr<CalendarFilter> fptr;
AutoPtr<CalendarCache> cache;
```

В начале *OnInit* появился новый фрагмент, отвечающий за генерацию кэша и его чтение.

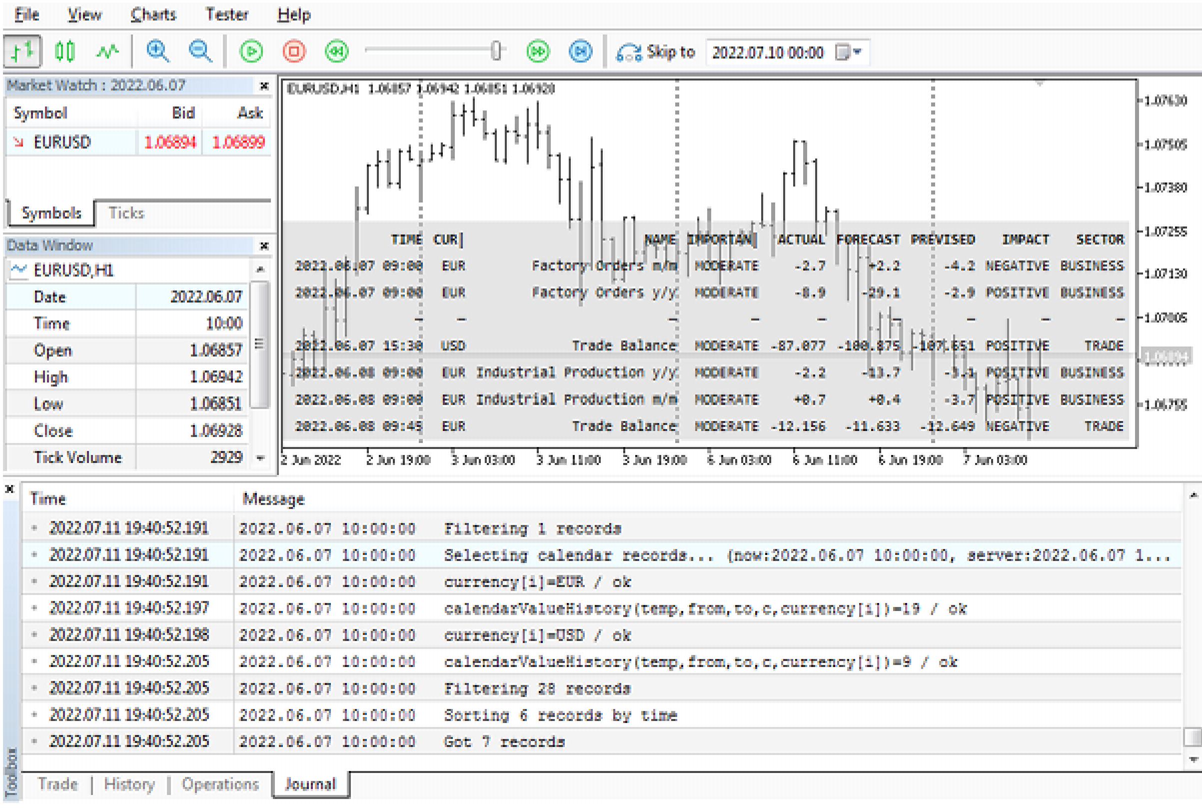
```
int OnInit()
{
   cache = new CalendarCache(CalendarCacheFile, true);
   if(cache[].isLoaded())
   {
      fptr = new CalendarFilterCached(cache[]);
   }
   else
   {
      if(MQLInfoInteger(MQL_TESTER))
      {
         Print("Can't run in the tester without calendar cache file");
         return 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[];
   ... // далее без изменений
```

Если файл кэша удалось прочитать, мы получим готовый объект *CalendarCache*, который передается в конструктор *CalendarFilterCached*. В противном случае программа проверяет, выполняется ли она в тестере или онлайн. Отсутствие кэша в тестере — это фатальный случай. На обычном же графике программа создает новый объект на основе данных встроенного календаря и сохраняет его в кэш под указанным именем. Если же имя файла сделать пустым, индикатор будет работать в точности, как исходный — напрямую с календарем.

Запустим индикатор на графике EURUSD. Пользователю будет выведено предупреждение о том, что указанный файл не найден и сделана попытка его сохранить. При условии, что календарь включен в настройках терминала, мы должны получить примерно следующие строки в журнале. Здесь приводится вариант с подробной диагностической информацией — её можно отключить, закомментировав в исходном коде директиву *#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
```

Теперь мы можем выбрать индикатор *CalendarMonitorCached.mq5* в тестере и увидеть в динамике, на истории, как меняется таблица новостей.



Новостной индикатор с кэшем календаря в тестере

Наличие кэша календаря позволяет тестировать торговые стратегии на новостях. Покажем это в следующем разделе.

7.3.12 Торговля по календарю

Существует множество новостных торговых стратегий: с рыночными ордерами или отложенными, с анализом финансовых показателей (направление ценового движения) и без (захват волатильности). Кроме того, во многие другие торговые системы полезно вставлять анти-новостной фильтр. Все такие программы затруднительно оптимизировать и отлаживать, поскольку в тестере календарь MQL5 недоступен. Однако с помощью кэша, разработанного в предыдущем разделе, мы можем исправить ситуацию.

Попробуем создать эксперт, который будет входить в рынок по новостям, в соответствии с оценкой их влияния на цену. Файл кэша "xyz.cal" был только что создан с помощью индикатора *CalendarMonitorCached.mq5*.

> Напомним, что образ календаря в кэше всегда соответствует моменту сохранения и требует осторожности при чтении: у более поздних событий актуальные показатели неизвестны, а более отдаленные события могут вообще не существовать. Регулярно обновляейте файл кэша календаря перед очередной оптимизацией или тестированием.
>
> При необходимости также учтите переводы часов на "летнее" и "зимнее" время в течение года: события из периодов с режимом DST, противоположном режиму DST в момент сохранения архива календаря, потребуется сдвинуть на 1 час назад или вперед. Избежать данных сложностей можно за счет выбора брокера без переключения DST или построение стратегии на таймфреймах больше H1.

Эксперт *CalendarTrading.mq5* будет торговать только по новостям, которые:

- относятся к рабочему символу графика;
- имеют тип финансового индикатора (то есть количественные);
- высокой важности;
- получили только что актуальное значение индикатора;

Последнее важно, поскольку для показателей, имеющих прогнозное и актуальное значения, система выставляет соответствующим образом значение поля *impact_type*: именно оно будет служить торговым сигналом (указывать направление входа в рынок).

> Точное время выхода новости, как правило, не совпадает с плановым, проставленным в поле *MqlCalendarValue::time*. Календарь не фиксирует это время, и оно недоступно в кэше. В связи с этим точность тестирования новостных стратегий может страдать. Если требуется приблизить анализ и принятие решений к онлайн-процессу, накапливайте статистику выхода новостей с помощью сервиса типа *CalendarChangeSaver.mq5* и встройте её в кэш.

По умолчанию торговля ведется минимальным лотом, с установкой уровней тейк-профит и стоп-лосс на заданном расстоянии в пунктах — все это отражено во входных параметрах.

```
input double Volume;                // Volume (0 = minimal lot)
input int Distance2SLTP = 500;      // Distance to SL/TP in points (0 = no)
input uint MultiplePositions = 25;
```

Для счетов с хеджинговым учетом разрешим одновременное существование нескольких позиций, по умолчанию 25. Это рекомендованная среда для тестирования, потому что она позволяет независимо оценить прибыльность параллельной торговли по новостям разных видов (каждая позиция создается независимо и не приводит к закрытию позиций по другим новостям). С другой стороны, ведение только одно позиции автоматически нивелирует противоречивые сигналы разных новостей.

Опционально эксперт поддерживает фильтры на идентификатор вида новости и текст для поиска по названию.

```
sinput ulong EventID;
sinput string Text;
```

Это может пригодиться для проведения последующих исследований конкретных новостей.

На глобальном уровне описаны указатели объектов аналитической обработкой новостей и сопровождения позиций.

```
AutoPtr<CalendarFilter> fptr;
AutoPtr<CalendarCache> cache;
AutoPtr<TrailingStop> trailing[];
```

Режим работы и пара валют текущего рабочего символа сохраняются в соответствующих переменных. Для упрощения примера предполагается применение на Forex (на других рынках получится торговля одной валютой — валютой котирования тикера).

```
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);
```

В обработчике *OnInit* загрузим кэш календаря и настроим фильтры согласно вышеприведенному описанию. Отсутствие кэша допускается на онлайн-графике: тогда эксперт работает в боевом режиме, напрямую с календарем. В тестере отсутствие файла кэша не даст запуститься эксперту.

```
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;

   // если задан конкретный вид события, смотрим только его
   if(EventID > 0) f.let(EventID);
   else
   {
      // иначе следим за новостями по валютам текущего символа
      f.let(Base);
      if(Base != Profit)
      {
         f.let(Profit);
      }

      // финансовые показатели, высокая важность, наличие актуального значения
      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);
   }
   // проверку фильтра новостей и торговлю по нему запускаем по секундному таймеру
   EventSetTimer(1);
```

```
   return INIT_SUCCEEDED;
}
```

В обработчике *OnTimer* запросим изменения новостей по настроенным фильтрам.

```
void OnTimer()
{
   CalendarFilter *f = fptr[];
   MqlCalendarValue records[];

   f.let(TimeTradeServer() - SCOPE_DAY, TimeTradeServer() + SCOPE_DAY);

   if(f.update(records)) // найти изменения, подпадающие под фильтры
   {
      // вывод свойств измененных новостей в журнал
      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));
      }
      ...
```

Когда подходящие изменения обнаружены, они выводятся в журнал следующим образом (фрагмент реального журнала ниже), с указанием времени, валюты, страны, названия, актуального и прогнозного значений, предыдущего значения и теоретической трактовки сигнала:

```
...
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)
...
```

На основе оценки в поле *impact_type* следует вычислить потенциальное влияние новостей на цену. Здесь важно отметить, что у нас две валюты: базовая и котирования. Когда новость имеет положительный эффект для базовой валюты, курс предположительно будет повышаться, а если отрицательный — то понижаться. Для валюты котирования все наоборот: положительный эффект должен удорожать вторую валюту в паре, что означает уменьшение курса, в то время как отрицательный — ведет к его увеличению. Это нормализованное направление движения цены вычисляется в следующем фрагменте с помощью переменной *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; // нет сигнала
      ...
```

Часто несколько новостей выходит одновременно, поэтому требуется аккумулировать оценки для них всех. Это делается в переменной *impact*. Поскольку в нашей стратегии фильтруются только новости одной, самой высокой важности, все одиночные сигналы от них просто суммируются, без весовых коэффициентов. В строковой переменной *about* подготавливается текст для комментария готовящейся сделки: там будут упомянуты идентификаторы событий, вызвавших сделку.

В случае, если робот запущен на счете с неттингом или достигнуто максимальное разрешенное количество позиций, закроем одну.

```
      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();
      }
      ...
```

Теперь можно открыть новую позицию по сигналу. В качестве "магического" номера устанавливается идентификатор события, что позволит нам позднее провести анализ финансовых показателей торговли в разрезе разных видов новостей.

```
      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) // ищем свободный слот для объекта сопровождени
            {
               trailing[i] = new TrailingStop(ticket, Distance2SLTP, Distance2SLTP /
               break;
            }
         }
      }
   }
}
```

Двигаем стоп-лоссы у всех позиций по приходу тиков.

```
void OnTick()
{
   for(int i = 0; i < ArraySize(trailing); ++i)
   {
      if(trailing[i][])
      {
         if(!trailing[i][].trail()) // позиция была закрыта
         {
            trailing[i] = NULL; // освобождаем объект и слот
         }
      }
   }
}
```

А теперь самое интересное. Благодаря тестеру появляется возможность проанализировать успешность новостной стратегии не только в целом, но и в разбивке по конкретным новостям. Соответствующий блок реализован у нас в обработчике *OnTester*. Сбор данных выполняется с помощью фильтра сделок. Получив из него массив кортежей *trades*, в котором сообщается прибыль, своп, комиссия и "магическое" число каждой сделки, мы аккумулируем результаты в трех объектах *MapArray*: они подсчитывают раздельно прибыли, убытки и количество трейдов для каждого *magic*-а.

```
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);
         }
      }
      ...
```

В результате получим таблицу, в которой построчно выводится статистика для каждого вида события: его идентификатор, страна, валюта, общая прибыль или убыток, количество трейдов (количество новостей), профит-фактор и название события.

```
      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;
}
```

Для проверки идеи запустим эксперт на периоде с начала 2021 года (по середину 2022) на паре EURUSD. Ниже приведен фрагмент журнала с распечаткой из *OnTester*.

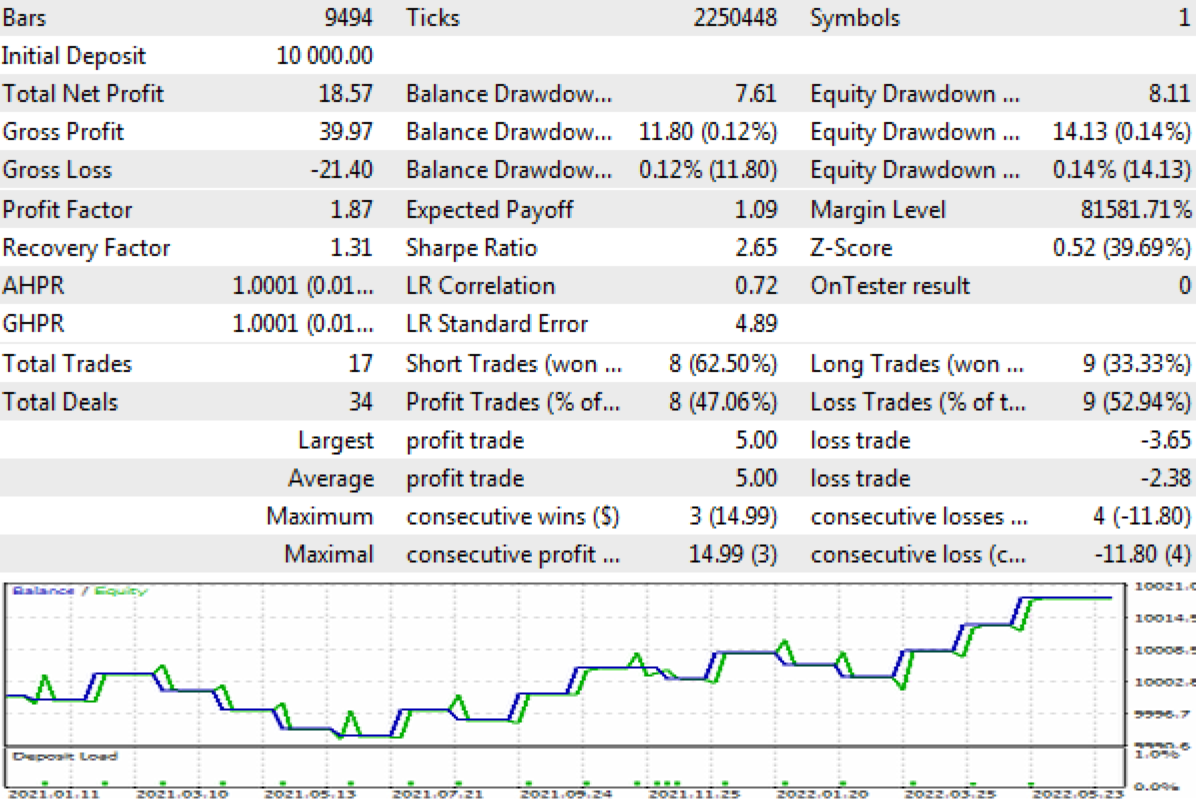
```
Trade profits by calendar events:
840040001 US USD -21.81 [17] (PF:0.53) ISM Manufacturing PMI
840190001 US USD -10.95 [17] (PF:0.69) ADP Nonfarm Employment Change
840200001 US USD -67.09 [78] (PF:0.60) EIA Crude Oil Stocks Change
999030003 EU EUR +14.13 [19] (PF:1.46) Retail Sales m/m
840040003 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
```

Результаты не очень впечатляют. Все же, торговля на новостях полна субъективизма. Во-первых, теоретические оценки влияния актуального значения новости на курс могут расходиться с эмоциональными ожиданиями толпы или дополнительным информационным фоном (остающимся

за пределами календаря и не поддающимся количественной оценке). Во-вторых, мы уже упоминали о неточности времени публикации актуального значения. В-третьих, наша стратегия реализована в самом простом виде, без анализа предварительного движения цены (когда, вероятно, была утечка, и новость "отыграна" раньше).

В целом, данный тест выявил, что любимые трейдерами Nonfarm Payrolls или отчеты по GDP не гарантируют успех, по крайней мере, с нашими настройками по умолчанию. Далее требуется, в обычном порядке, анализировать отдельные сделки, выяснять, что пошло не так, подбирать параметры и совершенствовать алгоритм, в частности добавить модуль корректировки времени по переключению DST в таймзоне сервера.

Вместе с тем, сам технический прием работает нормально, и мы можем для начала просто попытаться выбрать наиболее успешные новости. Например, возьмем новость 276030003 (Ifo Business Climate). Установив её в *EventID*, получим следующий отчет, совпадающим с нашими расчетными показателями.



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)

Отчет торговли в тестере по новостям Ifo Business Climate

Вы можете также попробовать торговлю по группе одноименных событий. В частности, чтобы реагировать только на новости по GDP (разных стран), введите в переменную *Text* строку "*GDP*". Звездочки добавлены, потому что без них строка длиной 3 символа будет трактоваться классом фильтра как валюта. Строки любой длины, отличной от 2 (код страны) или 3 (код валюты), могут задаваться как есть, например "farm", "Nonfarm", "Sales" — они будут искаться фильтром как подстроки названий, с учетом регистра.

7.4 Криптография

Алготрейдинг появился на стыке биржевой торговли и информационных технологий, позволив, с одной стороны, подключать к работе все новые и новые рынки, а с другой, расширять

функциональные возможности торговых платформ. Одно из технологических направлений, которое проникло в большинство сфер деятельности, в том числе и в арсеналы трейдеров, — это криптография или, в более общем смысле, защита информации.

В MQL5 есть пара функций для шифрования, хэширования и сжатия данных: *CryptEncode* и *CryptDecode*. Мы уже использовали их в некоторых примерах книги: скрипте *EnvSignature.mq5* (в разделе Привязка программы к свойствам среды исполнения) и сервисе *ServiceAccount.mq5* (в разделе Сервисы).

В этой главе мы познакомимся с этими функциями более подробно. Но прежде чем переходить непосредственно к их описанию, сделаем обзор методов преобразования информации: данное направление программирования очень обширно, и MQL5 поддерживает лишь часть стандартов. Вероятно, в будущем этот перечень будет расширен, а пока, если вы не обнаружите в справке требуемого метода шифрования, попробуйте найти готовую реализацию на сайте mql5.com (в разделах статей или в базе исходных кодов).

7.4.1 Обзор доступных методов преобразования информации

Защита информации может выполняться для разных целей и потому использовать разные способы. В частности, может требоваться полностью скрыть суть информации от постороннего наблюдателя или обеспечить её передачу, гарантировав неизменное состояние, но сама информация остается доступной. В первом случае речь идет о шифровании, а во втором — о цифровом отпечатке (хэше). Таким образом, шифрование и хэширование сводятся к переработке исходных данных в новое представление с использованием, при необходимости, дополнительных параметров.

И шифрование, и хэширование имеют множество разновидностей.

Наиболее общая градация делит шифрование на шифрование с открытыми ключами (асимметричное) и с закрытым ключом (симметричное).

> Асимметричная схема подразумевает наличие 2-х ключей — открытого и закрытого — у каждого участника обмена данными. Пары открытого и закрытого ключей предварительно генерируются по специальным алгоритмам. Каждый закрытый ключ известен только хозяину. Открытые ключи всех известны всем. Перед передачей зашифрованных данных потребуется обменяться тем или иным способом открытыми ключами. Далее поставщик данных применяет свой, известный только ему, закрытый ключ в купе с одним или несколькими открытыми ключами получателей данных. Те, в свою очередь, применяют свои закрытые ключи и открытый ключ отправителя для расшифровки.

Симметричная схема шифрования использует один и тот же секретный (закрытый) ключ — и для шифрования, и для дешифрования.

MQL5 поддерживает "из коробки" закрытые ключи (симметричные). Встроенными средствами MQL5 в данный момент нельзя обеспечить электронную подпись, которая использует асимметричное шифрование.

Особняком среди методов шифрования стоят примитивные алгоритмы без ключей — с их помощью добиваются условного сокрытия информации или преобразования типа информации. Сюда относятся, например, ROT13 (замена символов со сдвигом их алфавитно-цифровых кодов на 13, используется, в частности, в реестре Windows) или Base64 (перевод двоичных файлов в текст и обратно, как правило, в веб-проектах). Также востребованной задачей преобразования

данных является их сжатие. Его тоже можно рассматривать в некотором смысле как шифрование, поскольку данные становятся нечитаемыми человеком или прикладной программой.

Методов хэширования также известно очень много. Пожалуй, самым известным и простым можно назвать CRC (Cyclic Redundancy Check). В отличие от шифрования, которое позволяет восстановить исходное сообщение из зашифрованного, хэширование лишь создает отпечаток (характерный набор байтов) на основе исходной информации, — такой, что его неизменность при последующем пересчете гарантирует (с высокой вероятностью) неизменность исходной информации. Разумеется, это предполагает, что информация доступна всем участникам/пользователям соответствующей программной системы. Восстановить информацию по хэшу невозможно. Как правило, размер хэша (количество байт в нем) ограничен и стандартизован для каждого метода, так что и для строки длиной 80 символов и для файла размером 1 Мбайт мы получим хэш одного размера. Наиболее "близким" для большинства пользователей применением хэширования является хэширование паролей сайтами и программами, то есть последние хранят у себя и сверяются при логине именно с хэшем пароля, а не с паролем в исходном виде.

Следует отметить, что термин "хэш" уже встречался нам в предыдущей главе: там мы использовали хэширующую функцию для индексации структур экономического календаря. Тот простой хэш имеет очень слабую степень защиты, что, в частности, выражается в высокой вероятности коллизий (совпадения результатов для разных данных), которые мы специально обрабатывали в алгоритме. Он подходит для задач равномерного псевдо-случайного распределения данных по ограниченному количеству "корзинок". В отличие от этого, промышленные стандарты хэширования ориентированы именно на подтверждение целостности информации и применяют намного более сложные способы расчета. Но и длина хэша в этом случае — несколько десятков байтов, а не отдельное число.

Методы шифрования и хэширования информации, доступные MQL-программам, собраны в перечислении ENUM_CRYPT_METHOD.

Константа	Описание
CRYPT_BASE64	Перекодировка по стандарту Base64
CRYPT_DES	Шифрование DES с ключом 56 бит (7 байт)
CRYPT_AES128	Шифрование AES с ключом 128 бит (16 байт)
CRYPT_AES256	Шифрование AES с ключом 256 бит (32 байта)
CRYPT_HASH_MD5	Расчёт хэша MD5 (16 байт)
CRYPT_HASH_SHA1	Расчёт хэша SHA1 (20 байт)
CRYPT_HASH_SHA256	Расчёт хэша SHA256 (32 байта)
CRYPT_ARCH_ZIP	Сжатие методом "deflate"

Указанное перечисление используется в обеих функциях криптографического API — *CryptEncode* (шифрование/хэширование) и *CryptDecode* (дешифрование). Они будут рассмотрены в следующих разделах.

Методы шифрования AES и DES требуют помимо данных ключ шифрования — массив байтов предопределенной длины (она указана в таблице в скобках). Как уже говорилось, ключ должен

храниться в секрете и оставаться известным только разработчику программы или хозяину информации. Криптостойкость шифрования, то есть сложность подбора ключа компьютером злоумышленника, напрямую зависит от размера ключа: чем он больше, тем надежнее защита. Поэтому DES считается устаревшим и заменен в финансовой сфере своей усовершенствованной версией Triple DES: она заключается в последовательном троекратном применении DES с тремя разными ключами, что легко реализовать на MQL5. Существует популярная разновидность Triple DES, которая выполняет на второй итерации дешифрацию вместо шифрации с ключем номер 2, то есть как-бы восстанавливает данные в промежуточное, заведомо неправильное представление перед заключительным, третьим раундом DES. Но и Triple DES планируется вывести из промышленных стандартов после 2024 года.

Вместе с тем, криптостойкость следует соизмерять со временем жизни секрета (ключа и информации). Если требуется быстрый обмен потоком защищенных сообщений, то менее длинные регулярно обновляемые ключи обеспечат лучшую производительность.

Из методов хэширования наиболее современным является SHA256 (подмножество стандарта SHA-2). Методы SHA1 и MD5 признаны небезопасными, но по-прежнему широко используются в целях совместимости с существующими сервисами. Для методов хэширования в скобках указан размер получаемого массива байтов с цифровым отпечатком данных. Ключ для хэширования не нужен, но во многих применениях к хэшируемым данным пристыковывают так называемую "соль" — секретную компоненту, которая затрудняет воспроизводство требуемых хэшей злоумышленниками (например, при подборе пароля).

Элемент CRYPT_ARCH_ZIP обеспечивает ZIP-архивирование и передачу/прием запросов данных в сети Интернет (см. *WebRequest*).

> Несмотря на то, что название метода включает ZIP, сжатые данные не эквиваленты привычным архивам ZIP, в которых помимо контейнеров стандарта "deflate" всегда присутствуют мета-данные: специальные заголовки, список файлов и их атрибутов. На сайте mql5.com в статьях и библиотеки исходных кодов можно найти готовые реализации сжатия файлов в ZIP-архив и их извлечения оттуда. Само сжатие и извлечение производится функциями *CryptEncode*/*CryptDecode*, а все дополнительные необходимые структуры формата ZIP описаны и заполняются в MQL5-коде.

Метод Base64 предназначен для конвертации двоичных данных в текст и обратно. Двоичные данные в общем случае содержат множество непечатаемых символов и не поддерживаются средствами редактирования и ввода, такими как переменные input в диалогах свойств MQL-программ. Base64 может пригодиться, например, при работе с популярным текстовым форматом обмена объектными данными JSON.

Каждые 3 исходных байта кодируются в Base64 4-мя символами, что приводит к увеличению размера данных на треть. К книге прилагаются тестовые файлы, с которыми мы поэкспериментируем в последующих примерах, в частности, веб-страница *MQL5/Files/MQL5Book/clock10.htm* и используемый в ней файл с изображением часов *MQL5/Files/MQL5Book/clock10.png*. Уже на данном ознакомительном этапе вы можете наглядно увидеть возможности и разницу во внутреннем представлении двоичных данных и текста Base64, при сохранении идентичного внешнего вида.

Веб-страница со встроенным двоичным изображением и в формате Base64

Одно и то же изображение с циферблатом вставлено в страницу и как внешний файл *clock10.png*, и как его Base64-кодировка в теге *img* (в его атрибуте *src*: это так называемый "data URL"). Непосредственно в самом тексте веб-страницы это выглядит так (переносить длинную строку Base64 по ширине 76 символов не обязательно, но допускается стандартом и сделано здесь для публикации):

```
<img src="data:image/png;base64,
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" />
```

Скоро мы воспроизведем эту последовательность символов с помощью функции *CryptEncode*, а пока лишь отметим, что пользуясь подобным приемом, мы можем генерировать из MQL5 HTML-отчеты со встроенной графикой.

7.4.2 Шифрование, хэширование и упаковка данных: CryptEncode

За шифрование, хэширование и сжатие данных в MQL5 отвечает функция *CryptEncode*. Она преобразует данные переданного массива-источника *data* в массив-приемник *result* указанным методом.

int CryptEncode(ENUM_CRYPT_METHOD method, const uchar &data[], const uchar &key[], uchar &result[])

Для методов шифрования требуется также передать байтовый массив *key* с закрытым (секретным) ключом: его длина зависит от конкретного метода и указана в таблице методов ENUM_CRYPT_METHOD в предыдущем разделе. Если размер массива *key* будет больше, для ключа все равно будут использованы лишь первые байты в необходимом количестве.

Для хэширования или сжатия ключ не нужен, но для CRYPT_ARCH_ZIP существует один нюанс. Дело в том, что встроенная в терминал реализация алгоритма "deflate" дописывает в результирующие данные несколько байтов для контроля целостности: 2 начальных байта содержат настройки алгоритма "deflate", а 4 байта в конце — проверочную сумму Adler32. Из-за

этой особенности полученный упакованный контейнер отличается от того, который генерируют ZIP-архиваторы для каждого отдельного элемента архива (стандарт ZIP хранит аналогичный по смыслу CRC32 в своих заголовках). Поэтому для возможности создания и чтения совместимых ZIP-архивов на основе данных, упакованных функцией *CryptEncode*, MQL5 позволяет отключить собственный контроль целостности и генерацию лишних байтов с помощью специального значения в массиве *key*.

```
uchar key[] = {1, 0, 0, 0};
CryptEncode(CRYPT_ARCH_ZIP, data, key, result);
```

В принципе, подойдет любой ключ длиной не менее 4-х байтов, с ненулевым первым байтом. Полученный таким образом массив *result* можно дополнить заголовком по стандарту формата ZIP (данный вопрос выходит за рамки книги) и получить архив, доступный для других программ.

Функция возвращает количество байтов, помещенных в массив-приемник, или 0 в случае ошибки. Код ошибки, как обычно, будет сохранен в *_LastError*.

Проверить работоспособность функции предлагается скриптом *CryptEncode.mq5*. Он позволяет пользователю ввести текст (*Text*) или указать файл (*File*) для обработки. Чтобы задействовать файл, нужно очистить текст (поле *Text*).

Можно выбрать конкретный метод (*Method*) или пробежаться в цикле по всем методам сразу, чтобы наглядно увидеть и сравнить разные результаты. Для такого обзорного цикла следует оставить в параметре *Method* значение по умолчанию _CRYPT_ALL.

Кстати говоря, для введения такого функционала нам опять потребовалось расширить стандартное перечисление (в этот раз ENUM_CRYPT_METHOD), но поскольку перечисления в MQL5 нельзя наследовать как классы, здесь фактически объявлено новое перечисление ENUM_CRYPT_METHOD_EXT. Дополнительным бонусом этого стало то, что мы добавили более дружественные названия для элементов (в комментариях, с подсказками — как мы знаем, они отобразятся в диалоге настроек).

```
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;
```

По умолчанию, параметр *Text* заполнен некоторым сообщением, которое и предполагается шифровать. Вы можете заменить его на свое. Если *Text* очистить, программа обработает файл. Хотя бы один из параметров *Text* и *File* должен содержать информацию.

Поскольку для шифрования требуется ключ, два других параметра позволяют ввести его напрямую в виде текста (хотя ключ не обязан быть текстовым и может содержать любые двоичные данные, но они не поддерживаются в input-ах) или сгенерировать нужной длины, в зависимости от метода шифрования.

```
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 (длина, или из
input string CustomKey = "My top secret key is very strong";
```

Наконец, для включения режима ZIP-совместимости предусмотрена опция *DisableCRCinZIP* — она влияет только на метод CRYPT_ARCH_ZIP.

```
input bool DisableCRCinZIP = false;
```

Для упрощения проверок, когда метод требует ключа шифрования, а когда рассчитывается хэш (необратимое одностороннее преобразование), определено 2 макроса.

```
#define KEY_REQUIRED(C) ((C) == CRYPT_DES || (C) == CRYPT_AES128 || (C) == CRYPT_AES2
#define IS_HASH(C) ((C) == CRYPT_HASH_MD5 || (C) == CRYPT_HASH_SHA1 || (C) == CRYPT_H
```

В начале *OnStart* описаны требуемые переменные и массивы.

```
void OnStart()
{
   ENUM_CRYPT_METHOD method = 0;
   int methods[];           // сюда соберем все элементы ENUM_CRYPT_METHOD для цикла
   uchar key[] = {};        // пусто по умолчанию: подходит для хэширования, zip, bas
   uchar zip[], opt[] = {1, 0, 0, 0}; // "опции" для zip
   uchar data[], result[];  // исходные данные и результат
```

Согласно настройке *GenerateKey*, получаем ключ из поля *CustomKey* или просто заполняем массив *key* монотонно возрастающими целыми значениями. В реальности ключ должен представлять собой секретный, нетривиальный, произвольно выбранный блок значений.

```
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;
}
```

Обратите внимание здесь и далее на использование *ArrayResize* после *StringToCharArray*. Обязательно сокращайте массив на 1 элемент, поскольку случае функция *StringToCharArray* преобразует строку в массив байтов, включая терминальный 0, а это способно нарушить ожидаемое выполнение программы. В частности, в данном случае у нас появится лишний нулевой байт в секретном ключе, и если на принимающей стороне не используется программа с аналогичным артефактом, то там не смогут расшифровать посылку. Подобные лишние нули могут сказаться и на совместимости с протоколами обмена данными (если производится та или иная интеграция MQL-программы с "внешним миром").

Далее мы выводим в журнал "сырое" представление получившегося ключа в шестнадцатеричном формате: это делает функция *ByteArrayPrint*, уже использовавшаяся в разделе Запись и чтение файлов в упрощенном режиме, но, как и там, мы оставим её исходный код для самостоятельного изучения.

```
if(ArraySize(key))
{
   Print("Key (bytes):");
   ByteArrayPrint(key);
}
else
{
   Print("Key is not provided");
}
```

В зависимости от наличия *Text* или *File*, заполняем массив *data* либо символами текста, либо содержимым файла.

```
      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; // ошибка
         }
      }
```

Наконец, запускаем цикл по всем методам или однократно выполняем преобразование конкретным методом.

```
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); // массив с доп.опцией динамический для ArraySwap
      }
      ArraySwap(key, zip); // подменяем ключ на пустой или опцию
   }

   if(PRTF(CryptEncode(method, data, key, result)))
   {
      if(StringLen(Text))
      {
         // кодовая страница Latin (Western) для унификации отображения у всех пол
         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));
               // чтобы принять закодированные данные обратно для раскодировки
               // через строковый input, применим Base64 поверх двоичного результа
               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, "");
            }
```

```
          }
        }
      }
    }
  }
```

Когда мы конвертируем текст, мы выводим результат в журнал, но поскольку это почти всегда двоичные данные, за исключением метода CRYPT_BASE64, их отображение будет представлять собой абракадабру (по-хорошему, двоичные данные не стоит выводить в журнал, но мы это делаем для наглядности). Непечатаемые символы и символы с кодами больше 128 по-разному выводятся на компьютерах с разными языками. Поэтому, в целях унификации отображения примеров у всех читателей, мы используем при формировании строки в *CharArrayToString* явно заданную кодовую страницу (1252, Западноеропейские языки). Правда, используемые при публикации книги шрифты, скорее всего, внесут свою лепту в то, как именно будут отображаться те или иные знаки (набор глифов в шрифтах может быть ограничен).

> Важно отметить, что выбором кодовой страницы мы управляем лишь способом отображения, а байты в массиве *result* от этого не меняются (разумеется, полученную таким образом строку не следует куда-либо пересылать далее — она нужна только для визуализации — для обмена данными используйте байты самого результата).

Однако нам желательно все же предоставить пользователю какую-то возможность сохранить зашифрованный результат, чтобы затем раскодировать. Наиболее простой способ — применить к двоичным данным повторное преобразованием методом CRYPT_BASE64.

В случае кодирования файла мы просто сохраняем результат в новом файле с именем, в котором к исходному добавляется расширение по последнему слову в названии метода. Например, применив CRYPT_HASH_MD5 к файлу *Example.txt*, мы получим на выходе файл *Example.txt.MD5*, содержащий MD5-хэш исходного файла. Обратите внимание, что для метода CRYPT_ARCH_ZIP мы получим файл с расширением ZIP, но он не является стандартным ZIP-архивом (из-за отсутствия заголовков с мета-информацией и оглавлением).

Запустим скрипт с настройками по умолчанию: они соответствуют проверке в цикле всех методов для сообщения "Let's encrypt this message".

```
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+ýº'¥ 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*©ºø
€|)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=='
```

Ключ в данном случае достаточной длины для всех трех методов шифрования, а прочие методы, для которых он не нужен, просто игнорируют его. Поэтому все вызовы функции отработали успешно.

В следующем разделе мы научимся раскодировать шифровки и сможем проверить, выдаст ли функция *CryptDecode* исходное сообщение. Возьмите на заметку этот фрагмент журнала.

Включение опции *DisableCRCinZIP* приведет к сокращению результата метода CRYPT_ARCH_ZIP на несколько служебных байтов.

```
- 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=='
```

Теперь переведем эксперименты по кодированию в плоскость файлов. Для этого запустим скрипт еще раз и сотрем текст из поля *Text*. В результате программа обработает файл *MQL5Book/clock10.htm* несколько раз и создаст несколько производных файлов с разными расширениями.

```
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
```

Вы можете заглянуть внутрь всех файлов из файл-менеджера, и убедиться, что с исходным содержимым не осталось ничего общего. Многие файловые менеджеры имеют команды или плагины для расчета хэш-сумм, так что их можно сравнить с шестнадцатеричными значениями MD5, SHA1 и SHA256, выведенными в журнал.

Если мы попытаемся закодировать текст или файл, не предоставив ключ нужной длины, поручим ошибку INVALID_ARRAY(4006). Например, для текстового сообщения по умолчанию выберем в параметре *Method* AES256 (требует ключа в 32 байта), но с помощью параметра *GenerateKey* закажем ключ длиной 16 байт (или можно частично или полностью удалить текст из поля *CustomKey*, оставив *GenerateKey* по умолчанию).

```
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)
```

Также вы можете сжать один и тот же файл (как мы делали с *clock10.htm*) и скриптом методом CRYPT_ARCH_ZIP, и штатным архиватором. Если потом заглянуть с помощью утилиты просмотра двоичных файлов (которая встроена обычно в файловый менеджер) то оба результата покажут общий упакованный блок, а различия будут лишь в обрамляющих его мета-данных.

Сравнение файла, сжатого методом CRYPT_ARCH_ZIP (слева), и стандартного ZIP-архива с ним (справа)

Здесь показано, что середину и основную часть архива составляет последовательность байтов (выделенных темным), идентичных тем, что выдает функция *CryptEncode*, сжимавшая ту же веб-страницу.

Напоследок покажем, как было сгенерировано текстовое *Base64*-представление графического файла *clock10.png*. Для этого очистим поле *Text*, а в параметре *File* напишем *MQL5Book/clock10.png*. В выпадающем списке *Method* выберем *Base64*.

```
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
```

В результате был создан файл *clock10.png.BASE64*. Внутри него мы увидим ту самую строку, что вставлена в код веб-страницы, в тег *img*.

Кстати говоря, метод сжатия "deflate" является основой для графического формата PNG, поэтому мы можем использовать *CryptEncode* для сохранения битмапов ресурсов в PNG-файлы. К книге прилагается заголовочный файл *PNG.mqh* с минимальной поддержкой внутренних структур, необходимых для описания изображения: с его исходным кодом предлагается разобраться самостоятельно. С помощью *PNG.mqh* написан простой скрипт *CryptPNG.mq5*, преобразующий ресурс из поставляемого с терминалом файла "euro.bmp" в файл "my.png". Загрузка PNG-файлов не реализована.

```
#resource "\\Images\\euro.bmp"

#include <MQL5Book/PNG.mqh>

void OnStart()
{
   uchar null[];      // пустой ключ для CRYPT_ARCH_ZIP
   uchar result[];    // приемный массив
   uint data[];       // исходные пиксели
   uchar bytes[];     // исходные байты
   int width, height;
   PRTF(ResourceReadImage("::Images\\euro.bmp", data, width, height));

   ArrayResize(bytes, ArraySize(data) * 3 + width); // *3 для 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; // каждая линия предваряется байтом режима ф
      const uint c = data[i];
      // bytes[j++] = (uchar)((c >> 24) & 0xFF); // alpha, для PNG_CTYPE_TRUECOLORALF
      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); // по умолчанию PNG_CTYPE_TRUECOLOR (RGB)
   image.write(h);

   FileClose(h);
}
```

7.4.3 Дешифрование и распаковка данных: CryptDecode

Для выполнения операций дешифрования и распаковки данных в MQL5 предусмотрена функция *CryptDecode*.

Функция *CryptDecode* производит обратное преобразование данных массива *data* в приемный массив *result*, используя указанный метод.

int CryptDecode(ENUM_CRYPT_METHOD method, const uchar &data[], const uchar &key[], uchar &result[])

Обратите внимание, что получение хэш-сумм, выполняемое, в частности, функцией *CryptEncode*, является односторонним преобразованием: из хэшей нельзя восстановить исходные данные.

Функция возвращает количество байтов, помещенных в массив-приемник или 0 в случае ошибки. Код ошибки попадет в *_LastError*. Это может быть, например, INVALID_PARAMETER

(4003), если пытаться декодировать хэш (*method* равен одной из CRYPT_HASH-констант) или INVALID_ARRAY (4006), если ключ дешифрования недостаточной длины или отсутствует.

Если ключ неправильный (отличается от того, который использовался при шифровке), получим абракадабру в результате вместо закодированных исходных данных, но код ошибки при этом — нулевой. Это штатное поведение функции.

Проверим работу *CryptDecode* с помощью одноименного скрипта *CryptDecode.mq5*.

Во входных параметрах можно указать текст или файл для преобразования. Текст всегда подразумевается в кодировке *Base64*, так как все закодированные данные имеют двоичный формат и не поддерживаются в *input*-ах. Метод преобразования выбирается в списке *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;
```

Для методов шифрования необходим ключ, который можно ввести как строку в поле *CustomKey*, если *GenerateKey* содержит опцию DUMMY_KEY_CUSTOM, или сгенерировать демо-ключ требуемой длины из перечисления DUMMY_KEY_LENGTH (оно такое же, как в скрипте *CryptEncode.mq5*).

```
input DUMMY_KEY_LENGTH GenerateKey = DUMMY_KEY_CUSTOM; // GenerateKey (длина, или из
input string CustomKey = "My top secret key is very strong";
input bool DisableCRCinZIP = false;
```

В *GenerateKey* и *CustomKey* нужно выбирать те же значения, что и при запуске *CryptEncode.mq5*.

Алгоритм в *OnStart* начинается с описания требуемых массивов и получения ключа из строки или путем незатейливой генерации (только для демо, для генерации рабочего криптостойкого ключа используйте специальный софт или алгоритмы).

```
void OnStart()
{
   ENUM_CRYPT_METHOD method = 0;
   int methods[];
   uchar key[] = {};          // пустой по умолчанию ключ подходит для zip и 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");
   }
```

Далее мы считываем содержимое файла или декодируем *Base64* из поля *Text* (в зависимости от того, что заполнено), чтобы получить данные для обработки.

```
   method = (ENUM_CRYPT_METHOD)Method;
   Print("- ", EnumToString(method), ", key required: ", KEY_REQUIRED(method));
   if(StringLen(Text))
   {
      if(method != CRYPT_BASE64)
      {
         // так как все методы, кроме Base64, выдают двоичные результаты,
         // они дополнительно конвертируются в CryptEncode.mq5 с помощью Base64 в тек
         // поэтому здесь требуется восстановить двоичные данные из текстового ввода
         // перед расшифровкой
         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; // ошибка
         }
         
         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; // ошибка
      }
   }
```

Если пользователь пытается восстановить данные из хэша, выдаем предупреждение, что ничего не получится.

```
   if(IS_HASH(method))
   {
      Print("WARNING: hashes can not be used to restore data! CryptDecode will fail."
   }
```

Наконец, выполняем непосредственно расшифровку или распаковку. В случае текста полученный результат просто выводится в журнал. В случае файла мы добавляем к имени расширение ".dec" и записываем новый файл: его можно будет сравнить с исходным, который обрабатывался скриптом *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);
         }
      }
   }
```

Если запустить скрипт с настройками по умолчанию, он попытается раскодировать файл *MQL5Book/clock10.htm.BASE64*. Предполагается, что такой был создан в ходе экспериментов в предыдущем разделе, поэтому процесс должен пройти успешно.

```
- 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
```

Полученный файл *clock10.htm.BASE64.dec* полностью совпадает с исходным *clock10.htm*. То же самое должно произойти, если расшифровывать файлы с расширениями AES128, AES256, DES, при условии, что вы укажете тот же ключ, что и при шифровании.

Для наглядности проверим дешифровку текста. Так, шифрование известной фразы методом AES128 выдало нам ранее двоичный результат, превращенный для удобства в такую *Base64*-строку.

```
AQuvVCoSy1szaN8Owy8tQxl9rIrRj9hOqK7KgYYGh9E=
```

Введем её в поле *Text*, а в выпадающем списке *Method* выберем AES128. Увидим в журнале следующее.

```
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
```

Сообщение успешно расшифровано.

Если при том же входном тексте выбрать генерацию произвольного ключа (хоть и достаточной длины), вместо сообщения получится тарабарщина.

```
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
```

Аналогично поведет себя программа и если перепутать метод шифрования.

Выбирать методы "расхэширования" не имеет смысла — 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)
```

А вот попытка выполнить распаковку (CRYPT_ARCH_ZIP) того, что не является упакованным блоком "deflate", приведет к INTERNAL_ERROR (4001). Такую же ошибку можно получить, если включить опцию пропуска CRC для "архива" без неё или, наоборот, разжимать данные без опции, хотя упаковка делалась с ней.

7.5 Сетевые функции

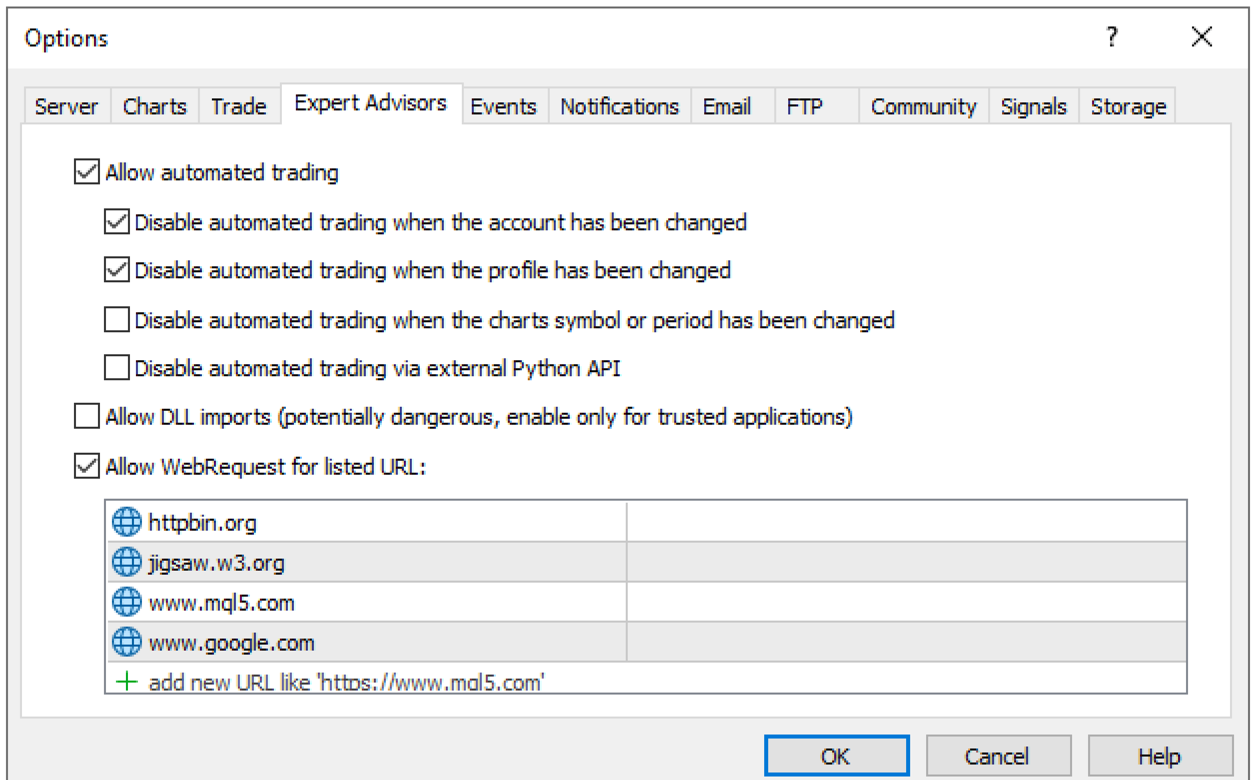
MQL-программы могут обмениваться данными с другими компьютерами в распределенной сети или серверами Интернет, используя различные протоколы. В частности, поддерживаются: веб-сайты и сервисы (HTTP/HTTPS), передача файлов (FTP), отправка электронных писем (SMTP) и push-уведомления.

Все функции данной направленности можно условно поделить на 3 группы:

- *SendFTP*, *SendMail* и *SendNotification* — это наиболее простые функции для отправки файлов, электронной почты и мобильных уведомлений;
- Функция *WebRequest* предназначена для работы с веб-ресурсами и позволяет легко отправлять HTTP-запросы (в том числе, GET и POST);
- Набор *Socket*-функций позволяет создать TCP-соединение (в том числе защищенное соединение TLS) с удаленным хостом через системные сокеты.

Последовательность, в которой перечислены группы, соответствует переходу от высокоуровневых функций, предлагающих уже готовые механизмы взаимодействия клиента и сервера, до низкоуровневых, позволяющих реализовать произвольный прикладной протокол по требованиям того или иного публичного сервиса (например, криптовалютной биржи или сервиса торговых сигналов). Разумеется, такая реализация предполагает большие усилия.

Для безопасности конечного пользователя MetaTrader 5 требует, чтобы список разрешенных web-адресов, с которыми может соединяться MQL-программа при помощи *Socket*-функций и *WebRequest*, был явно указан в диалоге настроек на закладке *Советники*. Здесь можно указывать домены, полный путь к web-страницам (не только сайт, но и остальные фрагменты URL, например, папки или номер порта) или IP-адреса. Ниже представлен скриншот с настройками для некоторых доменов из примеров этой главы.



Разрешения на доступ к сетевым ресурсам в настройках терминала

Программно редактировать этот список нельзя. При попытке обратиться к сетевому ресурсу, который отсутствует в данном списке, MQL-программа получит ошибку, и запрос будет отклонен.

Важно отметить, что все сетевые функции обеспечивают лишь клиентское соединение с тем или иным сервером, то есть организовать с помощью MQL5 сервер для ожидания и обработки входящих запросов нельзя. Для этой цели потребуется интегрировать терминал с внешней программой или интернет сервисом (например, с облачным).

7.5.1 Отправка Push-уведомлений

Как известно, терминал позволяет отсылать push-уведомления от служб компании MetaQuotes, самого терминала и MQL-программ на мобильное устройство с операционными системами iOS или Android. Для работы этой технологии используется MetaQuotes ID — уникальный идентификатор "пользователя" (см. врезку). MetaQuotes ID выделяется при установке мобильной версии терминала на гаджет пользователя, после чего ID следует указать в настройках терминала, на закладке Уведомления (через запятую можно задать несколько идентификаторов). После этого функционал по отправке push-уведомлений становится доступен для MQL-программ.

> На самом деле MetaQuotes ID идентифицирует не пользователя, а конкретную установку мобильного терминала, которых у пользователя может быть несколько, причем ID не связан по-умолчанию с регистрацией в сообществе mql5.com, хотя такую привязку можно указать на сайте. Не следует путать регистрацию пользователя в сообществе и MetaQuotes ID. Для работы с уведомлениями пользователь терминала не обязан логиниться в сообществе.

bool SendNotification(const string text)

Функция *SendNotification* посылает push-уведомления с заданным текстом во все мобильные терминалы, имеющие MetaQuotes ID из настроек терминала. Длина сообщения — не более 255 символов.

При успешной отправке уведомления из терминала функция возвращает *true*, а в случае ошибки — *false*. Возможные коды ошибок *_LastError*:

- 4515 – ERR_NOTIFICATION_SEND_FAILED — проблемы со связью;
- 4516 – ERR_NOTIFICATION_WRONG_PARAMETER — неверный параметр, например, пустая строка;
- 4517 – ERR_NOTIFICATION_WRONG_SETTINGS — MetaQuotes ID неверно настроен или отсутствует;
- 4518 – ERR_NOTIFICATION_TOO_FREQUENT — слишком частые вызовы функции.

При наличии соединения с сервером отправка происходит моментально, и, если устройство пользователя находится в онлайн, сообщение должно достичь адресата, однако гарантировать доставку в общем случае нельзя. Обратного уведомления программы о доставке сообщения не предусмотрено. История push-сообщений на сервере для отложенной доставки не сохраняется.

Для функции установлены ограничения на частоту использования: не более 2-х вызовов в секунду и не более 10 в минуту.

В тестере стратегий функция *SendNotification* не выполняется.

К книге прилагается простой скрипт *NetNotification.mq5*, отправляющий тестовое уведомление при наличии правильных настроек.

```
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) или 0 (успех
}
```

7.5.2 Отправка уведомлений по электронной почте

Терминал позволяет посылать письма на адрес электронной почты, указанный в диалоге настроек на закладке Почта. В MQL5 для этого предусмотрена функция *SendMail*.

bool SendMail(const string subject, const string text)

В параметрах функции задается заголовок и текст (тело письма).

Функция возвращает *true*, если письмо поставлено в очередь на отправку на почтовом сервере, иначе — *false*. Ошибки возможны, если работа с почтой запрещена в настройках, или почтовые данные (SMTP-сервер, порт, логин, пароль) содержат ошибку или не указаны.

В тестере стратегий функция *SendMail* не выполняется.

Проверку входящей электронной почты и её чтение (т.е. протоколы POP, IMAP) MQL5 не поддерживает.

Вместе с книгой вы найдете пример скрипта *NetMail.mq5*, делающего попытку отправить тестовое сообщение.

```
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 (успех)
}
```

7.5.3 Отправка файлов на сервер FTP

MetaTrader 5 поддерживает отправку файлов на FTP-сервер. Для работы этой возможности следует ввести необходимые реквизиты FTP в диалоге настроек на закладке FTP: адрес FTP-сервера, логин, пароль и опционально путь размещения файлов на сервере. Если ваш компьютер находится в сети провайдера, который не выделил для вас публичный IP-адрес, то, вероятно, потребуется включить пассивный режим.

Непосредственно отправку файлов из MQL-программы обеспечивает функция *SendFTP*.

bool SendFTP(const string filename, const string path = NULL)

Функция посылает файл с указанным именем на FTP-сервер из настроек терминала. При необходимости можно указать другой путь, отличный от настроенного заранее. Если параметр *path* не указан, используется каталог, описанный в настройках.

Отсылаемый файл должен находиться в папке *MQL5/Files* или ее подпапках.

Функция возвращает признак успеха (*true*) или ошибки (*false*). Потенциальные ошибки в *_LastError*:

- 4514 — ERR_FTP_SEND_FAILED — не удалось отправить файл по FTP;
- 4519 — ERR_FTP_NOSERVER — не указан FTP-сервер;
- 4520 — ERR_FTP_NOLOGIN — не указан FTP-логин;
- 4521 — ERR_FTP_FILE_ERROR — не найден заданный файл в директории MQL5/Files;
- 4522 — ERR_FTP_CONNECT_FAILED — ошибка при подключении к FTP-серверу;
- 4523 — ERR_FTP_CHANGEDIR — на FTP-сервере не найдена директория для выгрузки файла;
- 4524 — ERR_FTP_CLOSED — подключение к FTP-серверу было закрыто.

Функция блокирует выполнение MQL-программы до тех пор, пока операция не завершится. В связи с этим функцию запрещено использовать в индикаторах.

Также функция *SendFTP* не выполняется в тестере стратегий.

Терминал поддерживает только отправку отдельно взятого файла на FTP-сервер — все другие FTP-команды недоступны из MQL5.

Пример скрипта *NetFtp.mq5* делает скриншот текущего графика и пытается отправить его по 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 (успех) or FTP_CONNECT_FAILED(4522), FTP_
}
```

7.5.4 Обмен данными с веб-сервером по протоколу HTTP/HTTPS

MQL5 позволяет интегрировать программы с веб-сервисами и запрашивать данные из Интернет. Отправка и получение данных по протоколам HTTP/HTTPS осуществляется функцией *WebRequest*, имеющей две версии: для упрощенного и для продвинутого взаимодействия с веб-серверами.

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)

Основное отличие между двумя функциями в том что, упрощенный вариант позволяет указать в запросе заголовки только двух типов: так называемые "куки" (*cookie*) и ссылку на адрес, откуда

производится переход (*referer*, здесь нет опечатки, так исторически сложилось, что английское слово "referrer" пишут в HTTP-заголовках через одно 'r'). Расширенный вариант принимает обобщенный параметр *headers* для передачи произвольного набора заголовков. Заголовки запроса имеют вид "имя: значение" и соединяются переносом строки "\r\n", если их больше одного.

В частности, если предположить, что строка *cookie* должна содержать "name1=value1; name2=value2", а ссылка *referer* равна "google.com", то для вызова второго варианта функции с тем же эффектом, что и первого, следует в параметр *headers* добавить: "Cookie: name1=value1; name2=value2\r\nReferer: google.com".

В параметре *method* указывается один из методов протокола, "HEAD", "GET" или "POST". Адрес запрашиваемого ресурса или сервиса передается в параметре *url*. По спецификации HTTP длина сетевого идентификатора ресурса ограничена 2048 байтами, однако на момент написания книги в MQL5 было свое ограничение - 1024 байта.

Максимальная длительность запроса определяется таймаутом (*timeout*) в миллисекундах.

Оба варианта функции позволяют передать на сервер данные из массива *data*. Первый вариант дополнительно требует указания размера этого массива в байтах (*size*).

Для отправки простых запросов со значениями нескольких переменных можно их соединить в строку вида "имя1=значение1&имя2=значение2&..." и добавить в адрес GET-запроса, после символа-разделителя '?' или поместить в массив *data* для POST-запроса с использованием заголовка "Content-Type: application/x-www-form-urlencoded". В более сложных случаях, когда требуется передать, например, файлы, используйте POST-запрос и "Content-Type: multipart/form-data".

Приемный массив *result* получит тело ответа сервера (если оно есть). Заголовки ответа сервера помещаются в строку *response*.

Функция возвращает HTTP-код ответа сервера или -1 в случае системной ошибки (например, проблемы со связью, ошибка в параметрах). Возможные коды в *_LastError*:

- 5200 — ERR_WEBREQUEST_INVALID_ADDRESS — некорректный URL;
- 5201 — ERR_WEBREQUEST_CONNECT_FAILED — не удалось подключиться к указанному URL;
- 5202 — ERR_WEBREQUEST_TIMEOUT — превышен таймаут получения ответа от сервера;
- 5203 — ERR_WEBREQUEST_REQUEST_FAILED — иная ошибка в результате выполнения запроса.

Напомним, что даже если запрос выполнен без ошибок на уровне MQL5, прикладная ошибка может содержаться в HTTP-коде ответа сервера (например, требуется авторизация, неверный формат данных, страница не найдена и т.д.). При этом результат окажется пустым, а инструкции по разрешению ситуации, как правило, выясняются путем анализа полученных заголовков *response*.

Для использования функции *WebRequest* следует добавить адреса серверов в список разрешенных URL во вкладке *Советники* в настройках терминала. Порт сервера выбирается автоматически на основе указанного протокола: 80 для адресов "http://" и 443 — для "https://".

> Функция *WebRequest* является синхронной, то есть она приостанавливает выполнение программы, пока ждет ответа от сервера. В связи с этим функция запрещена для вызова из индикаторов, поскольку они работают в общих потоках по каждому символу. Задержка

выполнения одного индикатора приведет к остановке обновления всех графиков по данному символу.

При работе в тестере стратегий функция *WebRequest* не выполняется.

Начнем практику с простого скрипта *WebRequestTest.mq5*, выполняющего единичный запрос. Во входных параметрах предоставим выбор для метода (по умолчанию "GET"), адреса тестовой веб-страницы, дополнительных заголовков (по желанию), а также таймаута.

```
input string Method = "GET"; // Method (GET,POST)
input string Address = "https://httpbin.org/headers";
input string Headers;
input int Timeout = 5000;
```

Адрес вводится как в строке браузера: все символы, которые запрещены спецификацией HTTP напрямую использовать в адресах (в том числе, символы локального алфавита) функция *WebRequest* автоматически "маскирует" перед отправкой по алгоритму *urlencode* (точно так же делает и браузер, но мы это не видим, так как это представление предназначено для передачи по сетевой инфраструктуре, а не для людей).

Также добавим опцию *DumpDataToFiles*: когда она равна *true*, скрипт сохранит ответ сервера в отдельный файл, поскольку он может быть довольно большим. Значение *false* предписывает выводить данные напрямую в журнал.

```
input bool DumpDataToFiles = true;
```

Сразу стоит сказать, что тестирование подобных скриптов требует наличия сервера. Желающие могут установить локальный веб-сервер, например, node.js, но это предполагает самостоятельную подготовку или установку серверных скриптов (в данном случае, с подключением модулей на JavaScript). Более простой способ заключается в использовании публичных тестовых веб-серверов, доступных в Интернет. Среди них, например, *httpbin.org*, *httpbingo.org*, *webhook.site*, *putsreq.com*, *www.mockable.io*, *reqbin.com*. Они предоставляют разный набор возможностей. Выберите или найдите для себя подходящий (удобный и понятный или максимально гибкий).

В параметре *Address* по умолчанию находится адрес "конечной точки" (*endpoint*) серверного API *httpbin.org* — эта динамическая "веб-страница" возвращает клиенту HTTP-заголовки его запроса (в формате JSON). Таким образом, мы сможем в своей программе увидеть, что именно пришло на веб-сервер из терминала.

Не забудьте добавить домен "httpbin.org" в список разрешенных в настройках терминала.

Кстати говоря, текстовый формат JSON является стандартом де-факто для веб-сервисов. Готовые реализации классов для разбора JSON можно найти на сайте *mql5.com*, но мы сейчас будем просто показывать JSON "как есть".

В обработчике *OnStart* вызовем *WebRequest* с заданными параметрами и обработаем результат, если код ошибки неотрицательный. Заголовки ответ сервера (*response*) всегда выводятся в журнал.

```
void OnStart()
{
   uchar data[], result[];
   string response;

   int code = PRTF(WebRequest(Method, Address, Headers, Timeout, data, result, respor
   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));
         }
      }
   }
}
```

Для формирования имени файла мы используем вспомогательный класс URL из заголовочного файла *URL.mqh* (здесь не будет описан полностью). Метод *URL::parse* разбирает переданную строку на составляющие URL согласно спецификации — общий вид URL всегда такой: "protocol://domain.com:port/path?query#hash", причем многие фрагменты опциональны. Результаты помещаются в приемный массив, индексы в котором соответствуют конкретным частям URL и описаны в перечислении URL_PARTS:

```
enum URL_PARTS
{
   URL_COMPLETE,   // полный адрес
   URL_SCHEME,     // протокол
   URL_USER,       // имя/пароль пользователя (устарело, не поддерживается)
   URL_HOST,       // сервер
   URL_PORT,       // номер порта
   URL_PATH,       // путь/каталоги
   URL_QUERY,      // строка запроса после '?'
   URL_FRAGMENT,   // фрагмент после '#' (не выделяется)
   URL_ENUM_LENGTH
};
```

Таким образом, когда получаемые данные должны записаться в файл, скрипт создает его в папке по имени сервера (*parts[URL_HOST]*) и далее с сохранением иерархии пути в URL

(*parts[URL_PATH]*): в простейшем случае это будет просто имя "конечной точки". Когда запрашивается главная страница сайта (путь содержит только наклонную черту '/'), файл получает название "_index_.htm".

Попробуем запустить скрипт с параметрами по умолчанию, не забыв предварительно разрешить данный сервер в настройках терминала. В журнале мы увидим следующие строки (HTTP-заголовки ответа сервера и сообщение об успешном сохранении файла):

```
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
```

Внутри файла *httpbin.org/headers* находятся заголовки нашего запроса, каким его увидел сервер (форматирование JSON сервер добавил сам при ответе нам).

```
{
  "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..." // <- это добавил реверс-прокси серв
  }
}
```

Таким образом, терминал сообщает о том, что готов принять данные любого типа, с поддержкой сжатия конкретными методами и списком предпочтительных языков. Кроме того, он представляется в поле User-Agent как MetaTrader 5. Последнее может быть нежелательным при работе с некоторыми сайтами, которые оптимизированы для работы исключительно с браузерами. Тогда мы можем во входном параметре *Headers* указать фиктивное название, например, "User-Agent: Mozilla/5.0 (Windows NT 10.0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/103.0.0.0 Safari/537.36".

> Некоторые из перечисленных выше тестовых сайтов позволяют организовать на сервере временное тестовое окружение со случайным именем под ваш личный эксперимент: для этого нужно зайти на сайт из браузера и получить уникальную ссылку, рабочую, как правило, в пределах 24 часов. Тогда вы сможете использовать эту ссылку в качестве адреса для запросов из MQL5 и отслеживать поведение запросов непосредственно из браузера. Там же вы сможете настраивать ответы сервера, в частности, пробовать отправку форм.

Слегка усложним пример. Сервер может потребовать от клиента дополнительных действий для выполнения запроса, в частности, авторизоваться, выполнить "редирект" (переход по другому адресу), снизить частоту запросов и т.д. Все подобные "сигналы" обозначаются особыми HTTP-

кодами, которые возвращает функция *WebRequest*. Например, коды 301 и 302 — это редирект по разным причинам, и функция *WebRequest* выполняет его внутри автоматически, перезапрашивая страницу по указанному сервером адресу (поэтому коды редиректа никогда не попадают в код MQL-программы). А код 401 требует от клиента предоставить имя пользователя и пароль, и здесь вся ответственность лежит на нас. Способов послать эти данные существует множество. В новом скрипте *WebRequestAuth.mq5* продемонстрирована обработка двух вариантов авторизации, которые сервер запрашивает с помощью ответных HTTP-заголовков: "WWW-Authenticate: Basic" или "WWW-Authenticate: Digest". В заголовках это может выглядеть так:

```
WWW-Authenticate:Basic realm="DemoBasicAuth"
```

Или так:

```
WWW-Authenticate:Digest realm="DemoDigestAuth",qop="auth", »
»  nonce="cuFAuHbb5UDvtFGkZEb2mNxjqEG/DjDr",opaque="fyNjGC4x8Zgt830PpzbXRvoqExsZeQSDZ
```

Первый из них — самый простой и небезопасный, а потому практически не используется: в книге он приведен из-за простоты изучения на первом этапе. Суть его работы в том, чтобы в ответ на требование сервера сформировать следующий HTTP-запрос, добавив специальный заголовок:

```
Authorization: Basic dXNlcjpwYXNzd29yZA==
```

Здесь после ключевого слова "Basic" идет Base64-кодировка строки "user:password" с актуальным именем и паролем, а символ ':' вставляется здесь и далее "как есть" в качестве связующего звена. Более наглядно процесс взаимодействия представлен на изображении.

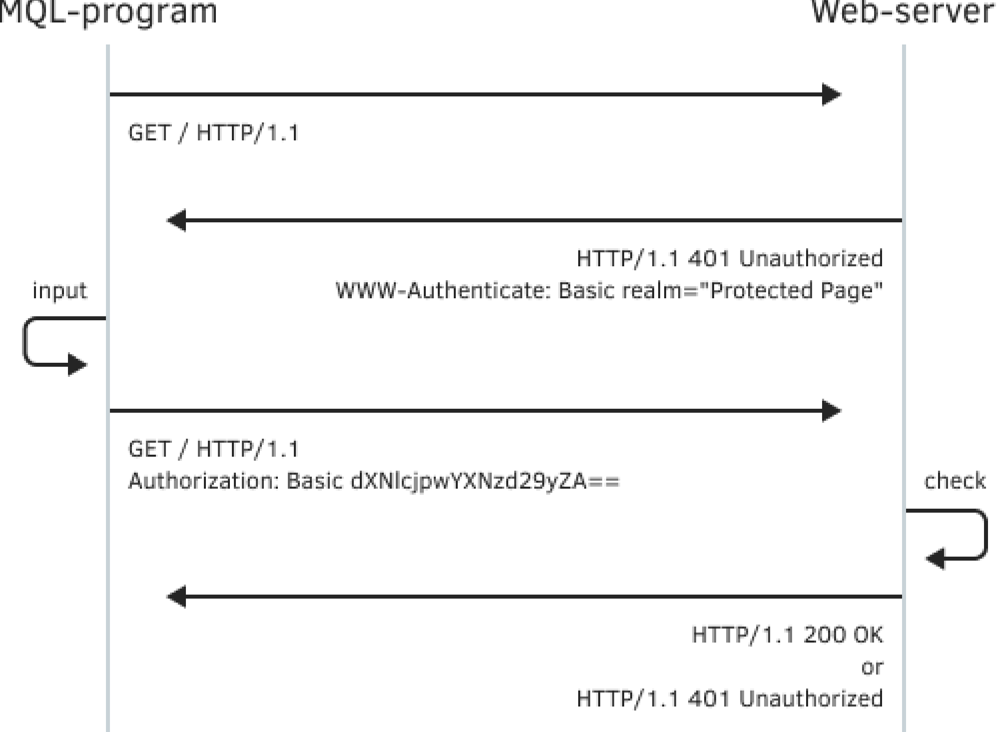


Схема простой авторизации на веб-сервере

Более продвинутой считается схема авторизации *Digest*. В этом случае сервер предоставляет некоторую дополнительную информацию в своем ответе:

- *realm* — название сайта (области сайта), куда производится вход;
- *qop* — разновидность метода Digest (мы затронем только "auth");
- *nonce* — случайная строка, которая будет использоваться для генерации данных авторизации;
- *opaque* — случайная строка, которую мы в своих заголовках передадим обратно "как есть";
- *algorithm* — опциональное название алгоритма хэширования, по умолчанию подразумевается MD5;

Для авторизации нужно выполнить следующие действия:

1. Сгенерировать собственную случайную строку *cnonce*;
2. Инициализировать или увеличить счетчик своих запросов *nc*;
3. Вычислить *hash1 = MD5(user:realm:password)*;
4. Вычислить *hash2 = MD5(method:uri)*, здесь *uri* — это путь и имя страницы;
5. Вычислить *response = MD5(hash1:nonce:nc:cnonce:qop:hash2)*.

После этого клиент может повторить запрос к серверу, добавив в свои заголовки строку вида:

```
Authorization: Digest username="user",realm="realm",nonce="...", »
» uri="/path/to/page",qop=auth,nc=00000001,cnonce="...",response="...",opaque="..."
```

Поскольку сервер имеет ту же информацию, что и клиент, он сможет повторить вычисления и проверить совпадение хэшей.

В параметры скрипта добавим переменные для ввода имени пользователя и пароля. В параметре *Address* по умолчанию прописан адрес "конечной точки" *digest-auth*, которая умеет запрашивать авторизацию с параметрами *qop* ("auth"), логином ("test") и паролем ("pass") — это все задается на выбор в самом пути "конечной точки" (вы можете тестировать другие методы и реквизиты пользователя, например, так: "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;
```

В параметре *Headers* мы просто так указали фиктивное название браузера для демонстрации возможности.

В функции *OnStart* добавим обработку HTTP-кода 401. Если имя пользователя и пароль не предоставлены, мы не сможем продолжить.

```
void OnStart()
{
   string parts[];
   URL::parse(Address, parts);
   uchar data[], result[];
   string response;
   int code = PRTF(WebRequest(Method, Address, Headers, Timeout, data, result, respor
   Print(response);
   if(code == 401)
   {
      if(StringLen(User) == 0 || StringLen(Password) == 0)
      {
         Print("Credentials required");
         return;
      }
      ...
```

Далее следует проанализировать заголовки, полученные с сервера. Для удобства был написан класс *HttpHeader* (*HttpHeader.mqh*). В его конструктор передается полный текст, а также разделитель элементов (в данном случае, символ перевода строки '\n') и символ, используемый между именем и значением внутри каждого элемента (в данном случае, двоеточие ':'). В процессе своего создания объект "парсит" текст, и затем элементы становятся доступны через перегруженный оператор [], причем тип его аргумента — строка. В результате, мы можем проверить наличие требования авторизации по имени "WWW-Authenticate". Если такой элемент есть в тексте и равен "Basic", формируем ответный заголовок "Authorization: Basic " с закодированными в Base64 логином и паролем.

```
      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, respc
         Print(response);
      }
      ...
```

Для Digest-авторизации все немного сложнее, в соответствии с изложенным выше алгоритмом.

```
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);
      }
   }
}
```

Статический метод *HttpHeader::hash* получает строку с шестнадцатеричным представлениям хэша (по умолчанию MD5) для всех требуемых составных строк. На основе этих данных формируется заголовок для очередного вызова *WebRequest*. Статический метод *HttpHeader::unquote* убирает обрамляющие кавычки.

Остальная часть скрипта осталась без изменений. Повторный HTTP-запрос может оказать успешным, и тогда мы получим содержимое защищенной страницы, либо в авторизации будет отказано, и сервер напишет что-то вроде "Access denied".

Поскольку параметры по умолчанию содержат правильные значения ("/digest-auth/auth/test/pass" соответствует пользователю "test" и паролю "pass"), мы должны получить следующий результат запуска скрипта (все основные этапы и данные выводятся в лог). Рассмотрим его по порядку.

```
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
...
```

Первый вызов *WebRequest* закончился с кодом 401, и среди ответных заголовков находится требование об авторизации ("WWW-Authenticate") с необходимыми параметрами. На их основе мы посчитали правильный ответ и подготовили заголовки для нового запроса.

```
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
...
```

Второй запрос уже возвращает код 200 и полезные данные, которые мы записываем в файл.

```
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
```

Внутри файла *MQL5/Files/httpbin.org/digest-auth/auth/test/pass* можно найти "веб-страницу", а точнее статус успешной авторизации в формате JSON.

```
{
  "authenticated": true,
  "user": "test"
}
```

Если при запуске скрипта указать неправильный пароль, получим пустой ответ от сервера, и файл не будет записан.

При использовании *WebRequest* мы автоматически оказываемся в области распределенных программных систем, в которых правильная работа зависит не только от нашего клиентского MQL-кода, но и сервера (не говоря уже о промежуточных звеньях, вроде прокси). Поэтому нужно быть готовым к возникновению чужих ошибок. В частности, на момент написания книги в реализации "конечной точки" *digest-auth* на *httpbin.org* имелась проблема: введенное в запрос имя пользователя не участвует в проверке авторизации, и потому любой логин приводит к успешной авторизации при указании правильного пароля. Чтобы все-таки проверить наш скрипт воспользуйтесь другими сервисами, например, аналогичным

httpbingo.org/digest-auth/auth/test/pass. Также вы можете настроить скрипт на адрес *jigsaw.w3.org/HTTP/Digest/* — он ожидает логин/пароль "guest"/"guest".

На практике большинство сайтов реализует авторизацию с помощью форм, встроенных непосредственно в веб-страницы: внутри HTML-кода они представляют собой тег-контейнер *form* с набором полей ввода (теги *input*, *select* и другие, см. далее), которые заполняются пользователем и отправляются на сервер методом POST. В связи с этим имеет смысл разобрать пример с отправкой формы. Однако, прежде чем заняться этим вплотную, желательно осветить еще один технический прием.

Дело в том, что взаимодействие клиента и сервера обычно сопровождается изменением состояния как клиента, так и сервера. На примере авторизации это можно понять наиболее наглядно, так как до авторизации пользователь был для системы неизвестным, а после — система уже знает его логин, и может применить предпочтительные настройки для сайта (например, язык, цвет, способ отображения форума), а также разрешить доступ к тем страницам, куда неавторизованные посетители попасть не могут (сервер такие попытки пресекает, возвращая HTTP-статус 403, Forbidden).

Поддержка и синхронизация согласованного состояния клиентской и серверной частей распределенного веб-приложения обеспечивается с помощью механизма "печеньиц" (cookies) — поименованных переменных и их значений в HTTP-заголовках. Термин восходит к "печеньям с предсказаниями", так как *cookie* тоже содержат маленькие послания, невидимые пользователю.

Любая из сторон — сервер и клиент — может добавлять *cookie* в HTTP-заголовок. Сервер это делает с помощью строки вида:

```
Set-Cookie: имя=значение; ⌈Domain=домен; Path=путь; Expires=дата; Max-Age=количество_
```

Только имя и значение являются обязательными, а остальные атрибуты опциональны: здесь приведены основные — *Domain*, *Path*, *Expires* и *Max-Age*, но по факту их больше.

Получив такой заголовок (или несколько), клиент должен запомнить у себя имя и значение переменной и отсылать их на сервер во всех запросах, которые адресуются к соответствующему домену (*Domain*), пути (*Path*) внутри этого домена и пока не истечет срок (*Expires* или *Max-Age*).

В исходящем от клиента HTTP-запросе *cookie* передаются строкой вида:

```
Cookie: имя⁽№⁾=значение⁽№⁾ [; имя⁽ⁱ⁾=значение⁽ⁱ⁾ ...]ᵒᵖᵗ
```

Здесь через точку с запятой с пробелом перечисляются все пары "имя=значение", установленные сервером и известные на данном клиенте, подходящие по домену и пути к текущему запросу, а также с неистекшим сроком жизни.

Сервер и клиент обмениваются всеми нужными куками при каждом HTTP-запросе — именно поэтому данный архитектурный стиль распределенных систем называется REST (*Representational State Transfer* или "передача самодостаточного состояния"). Например, после того как пользователь успешно авторизуется на сервере, последний установит (через заголовок "Set-Cookie:") специальную куку с идентификатором пользователя, после чего веб-браузер (или, в нашем случае, терминал с MQL-программой) будет отсылать её в следующих запросах (добавляя соответствующую строку в заголовок "Cookie:").

Функция *WebRequest* незаметно проделывает для нас всю эту работу: собирает "куки" из приходящих заголовков и добавляет подходящие куки в исходящие HTTP-запросы.

Куки хранятся терминалом и между сессиями, согласно их настройкам. Чтобы проверить это, достаточно дважды запросить веб-страницу с сайта, использующего куки.

> Внимание, куки хранятся в привязке к сайту и потому незаметно подставляются в исходящие заголовки всех MQL-программ, которые используют *WebRequest* для того же сайта.

Для упрощения последовательных запросов имеет смысл формализовать популярные действия в специальном классе *HTTPRequest* (*HTTPRequest.mqh*). В нем будем хранить общие HTTP-заголовки, которые, скорее всего, понадобятся для всех запросов (например, поддерживаемые языки, инструкции для прокси и т.д.). Кроме того, такая настройка как таймаут также является общей. Обе настройки передаются в конструктор объекта.

```
class HTTPRequest: public HttpCookie
{
protected:
   string common_headers;
   int timeout;

public:
   HTTPRequest(const string h, const int t = 5000):
      common_headers(h), timeout(t) { }
   ...
```

По умолчанию таймаут установлен в 5 секунд. Основной, в некотором смысле универсальный метод класса — *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;
   }
};
```

Опишем еще пару методов для запросов конкретных типов.

Запросы GET используют только заголовки, а тело документа (часто встречается термин *payload*, "нагрузка") у них пустое.

```
   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);
   }
```

В запросах POST "нагрузка", как правило, есть.

```
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);
}
```

Но отправка форм возможна в разных форматах. Наиболее простой из них "application/x-www-form-urlencoded". Он подразумевает, что полезная "нагрузка" будет представлять собой строку (может быть и очень длинную, так как спецификации не накладывают ограничений, и все зависит от настроек веб-серверов). Для таких форм предоставим более удобную перегрузку метода POST со строковым параметром "нагрузки".

```
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); // удаляем терминальный ноль
   return request("POST", address, custom_headers, bytes, result, response);
}
```

Для проверки нашего клиентского веб-движка напишем простой скрипт *WebRequestCookie.mq5*. Его задачей будет запросить одну и ту же веб-страницу дважды: первый раз сервер, скорее всего, предложит установить свои куки, и тогда они будут автоматически подставлены во второй запрос. Во входных параметрах укажем адрес страницы для теста — пусть это будет сайт *mql5.com*. Также сымитируем заголовки по умолчанию — пусть это будет откорректированная строка "User-Agent".

```
input string Address = "https://www.mql5.com";
input string Headers = "User-Agent: Mozilla/5.0 (Windows NT 10.0) Chrome/103.0.0.0";
```

В основной функции скрипта опишем объект *HTTPRequest* и выполним в цикле два запроса GET.

> Внимание! Данный тест работает в предположении, что MQL-программы еще не заходили на сайт www.mql5.com и не получали с него куки. После однократного запуска скрипта куки останутся в кэше терминала, и воспроизвести пример станет невозможно: на обеих итерациях цикла мы получим одинаковые записи в журнале.
>
> Не забудьте добавить домен "www.mql5.com" в список разрешенных в настройках терминала.

```
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) // покажем начало документа только первый раз
            {
               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));
            }
         }
      }
   }
}
```

Первая итерация цикла породит такие записи в журнале (с сокращениями):

```
>>> 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" />
```

Мы получили одну новую куку с именем *sid*. Чтобы убедиться в её эффективности, можно перейти к просмотру второй части журнала, для второй итерации цикла.

```
>>> 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
```

К сожалению, нам здесь не видны полные исходящие заголовки, формируемые внутри *WebRequest*, но то, что кука отправилась на сервер с помощью заголовка "Cookie:", доказывает тот факт, что сервер в своем втором ответе уже не просит её установить.

В принципе, данная кука просто идентифицирует посетителя (так делает большинство сайтов), но не означает его авторизации. Поэтому вернемся к упражнению по отправке формы в общем виде, подразумевая на перспективу частную задачу ввода логина и пароля.

Напомним, что для отправки формы мы можем воспользоваться методом POST со строковым параметром *payload*. Принцип подготовки данных по стандарту "x-www-form-urlencoded" заключается в том, что именованные переменные и их значения записываются в одну сплошную строку (в чем-то похожую на куки).

```
имя(№)=значение(№)[&имя(i)=значение(i)...]opt
```

Имя и значение связаны знаком равно '=', а пары состыковываются с помощью символа амперсанда '&'. Значение может отсутствовать. Например,

```
Name=John&Age=33&Education=&Address=
```

Важно отметить, что с технической точки зрения данная строка должна быть перед отправкой преобразована по алгоритму *urlencode* (именно отсюда и название формата), однако *WebRequest* сам выполняет это преобразование за нас.

Имена переменных обусловлены веб-формой (содержимым тега *form* в веб-странице) или логикой веб-приложения — в любом случае, имена и значения должен уметь трактовать веб-сервер. Поэтому для знакомства с технологией нам требуется тестовый сервер с формой.

Тестовая форма имеется по адресу *https://httpbin.org/forms/post*. Она представляет собой диалог для заказа пиццы.

Customer name:

Telephone:

E-mail address:

Pizza Size

Small

Medium

Large

Pizza Toppings

Bacon

Extra Cheese

Onion

Mushroom

Preferred delivery time: -- : -- --

Delivery instructions:

Submit order

Тестовая веб-форма

Её внутреннее устройство и поведение описывается следующим HTML-кодом. В нем нас в первую очередь интересуют теги *input*, которые и задают ожидаемые сервером переменные. Кроме того следует обратить внимание на атрибут *action* в самом теге *form*, так как он определяет адрес, на который должен отправляться POST-запрос, и в данном случае это "/post", что вместе с доменом дает строку "httpbin.org/post" — именно её будем использовать в MQL-программе.

```
<!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>
```

В скрипте *WebRequestForm.mq5* мы подготовили аналогичные входные переменные для указания пользователем перед отправкой на сервер.

```
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=";
```

Уже установленные строки приведены только для тестирования в одно нажатие: вы можете заменить их на собственные, но учтите, что внутри каждой строки следует редактировать только величину справа от '=', а имя слева от '=' нужно сохранить (неизвестные имена сервер проигнорирует).

В функции *OnStart* опишем HTTP-заголовок "Content-Type:" и подготовим объединенную строку со всеми переменными.

```
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 подразумевается для многих типов content-types,
         // но в некоторых может быть другая, анализируйте ответные заголовки
         Print(CharArrayToString(result, 0, WHOLE_ARRAY, CP_UTF8));
      }
   }
}
```

Затем выполним метод POST и выведем ответ сервера в журнал. Вот пример результата.

```
>>> 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"
}
```

Тестовый сервер подтверждает получение данных в виде их копии в формате JSON. На практике сервер, конечно, не станет возвращать сами данные, а просто сообщит статус успеха и, возможно, перенаправит на другую веб-страницу, на которую эти данные оказали эффект (например, покажет номер заказа).

С помощью таких POST-запросов, но поменьше, обычно выполняется и авторизация. Правда, большинство веб-сервисов специально усложняет данный процесс в целях безопасности, и предварительно требует рассчитать несколько хэш-сумм от реквизитов пользователя. Публичные, специально разработанные API обычно имеют в документации описания всех необходимых алгоритмов. Но так бывает не всегда. В частности, у нас не получится авторизоваться с

помощью *WebRequest* на *mql5.com*, потому что сайт не имеет открытого программного интерфейса.

При отправке запросов в веб-сервисы всегда придерживайтесь правила о непревышении частоты запросов: обычно каждый сервис указывает свои ограничения, и их нарушение приведет к последующей блокировке вашей клиентской программы, аккаунта или IP-адреса.

7.5.5 Установление и разрыв соединения сетевого сокета

В предыдущих разделах мы познакомились с высокоуровневыми сетевыми функциями MQL5: каждая из них предоставляет поддержку прикладному протоколу конкретной направленности. Например, SMTP — для отправки электронной почты (*SendMail*), FTP — для передачи файлов (*SendFTP*) или HTTP — для получения веб-документов (*WebRequest*). Все упомянутые стандарты опираются на более низкий, транспортный уровень TCP (Transmission Control Protocol). Он не последний в иерархии — есть и более низкие, но здесь мы не станем их касаться.

Стандартная реализация прикладных протоколов скрывает внутри много технических нюансов и освобождает программиста от лишних часов рутинного следования спецификациям. Однако, она не обладает гибкостью, не учитывает расширенные возможности, заложенные в стандарты. Поэтому иногда требуется запрограммировать сетевое взаимодействие на уровне TCP — на уровне сокетов.

Сокет можно рассматривать как аналог файла на диске: сокет тоже описывается целочисленным дескриптором, по которому можно читать или записывать данные, но происходит это в распределенной сетевой инфраструктуре. В отличие от файлов, количество сокетов на компьютере ограничено, и потому дескриптор сокета нужно заранее запросить у системы, прежде чем связать его с сетевым ресурсом (адресом, URL). Также заранее скажем, что доступ к информации через сокет — поточный, то есть нельзя "перемотать" некий "указатель" на начало, как в файле.

Потоки записи и чтения не пересекаются, но могут влиять на будущие читаемые или записываемые данные, поскольку передаваемая информация часто трактуется серверами и клиентскими программами как управляющие команды. Когда поток содержит команды, а когда данные — определяют стандарты протоколов.

Для создания "пустого" дескриптора сокета в MQL5 предназначена функция *SocketCreate*.

int SocketCreate(uint flags = 0)

Единственный параметр зарезервирован на будущее для указания битовой комбинации флагов, определяющих режим работы сокета, однако в данный момент поддерживается только один флаг-заглушка — SOCKET_DEFAULT — он соответствует текущему режиму, и его можно не указывать. На системном уровне это эквивалентно сокету в блокирующем режиме (это может быть интересно специалистам в области сетевого программирования).

В случае успешного выполнения функция возвращает дескриптор сокета, а иначе — INVALID_HANDLE.

Из одной MQL-программы можно создать максимум 128 сокетов. При превышении лимита в *_LastError* записывается ошибка 5271 (ERR_NETSOCKET_TOO_MANY_OPENED).

После того как мы открыли сокет, его следует связать с сетевым адресом.

bool SocketConnect(int socket, const string server, uint port, uint timeout)

Функция *SocketConnect* выполняет подключение сокета к серверу по указанному адресу и порту (например, веб-сервера обычно работают на портах 80 или 443 для протоколов HTTP и HTTPS, соответственно, а SMTP — на порту 25). Адресом может выступать как доменное имя, так и IP-адрес.

Параметр *timeout* позволяет задать таймаут в миллисекундах для ожидания ответа сервера.

Функция возвращает признак успешного подключения (*true*) или ошибки (*false*). Код ошибки записывается в *_LastError*, например, 5272 (ERR_NETSOCKET_CANNOT_CONNECT).

> Напоминаем: адрес для подключения должен быть добавлен в список разрешенных в настройках терминала (диалог *Сервис* -> *Настройки* -> *Советники*).

После завершения работы с сетью следует освободить сокет с помощью *SocketClose*.

bool SocketClose(const int socket)

Функция *SocketClose* закрывает сокет по его дескриптору, открытому ранее с помощью функции *SocketCreate*. Если для сокета ранее было создано соединение через *SocketConnect*, оно будет разорвано.

Функция также возвращает признак успеха (*true*) или ошибки (*false*). В частности, при передаче неверного дескриптора в *_LastError* записывается ошибка 5270 (ERR_NETSOCKET_INVALIDHANDLE).

Напоминаем, что все функции данного и последующих разделов запрещены в индикаторах: там попытка работать с сокетами приведет к ошибке 4014 (ERR_FUNCTION_NOT_ALLOWED, "Системная функция не разрешена для вызова").

Рассмотрим вводный пример — скрипт *SocketConnect.mq5*. Во входных параметрах можно задать адрес и порт сервера. Предполагается, что мы начнем тестирование с обычных веб-серверов, таких как mql5.com.

```
input string Server = "www.mql5.com";
input uint Port = 443;
```

В функции *OnStart* просто создадим сокет и свяжем его с сетевым ресурсом.

```
void OnStart()
{
   PRTF(Server);
   PRTF(Port);
   const int socket = PRTF(SocketCreate());
   if(PRTF(SocketConnect(socket, Server, Port, 5000)))
   {
      PRTF(SocketClose(socket));
   }
}
```

Если все настройки в терминале сделаны правильно, и он подключен к Интернет, получим следующий "отчет".

```
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 Проверка состояния сокета

В процессе работы с сокетом возникает необходимость проверки его статуса, потому что распределенные сети не столь надежны как файловая система. В частности, соединение может быть по той или иной причине потеряно. Выяснить это позволяет функция *SocketIsConnected*.

bool SocketIsConnected(const int socket)

Функция проверяет, подключен ли сокет с указанным дескриптором (полученным из *SocketCreate*) к своему сетевому ресурсу (указанному в *SocketConnect*), и возвращает *true* в случае успеха.

Другая функция *SocketIsReadable* позволяет узнать, не появились ли в системном буфере, связанном с сокетом, данные для чтения. Это означает, что компьютер, к которому мы подключились по сетевому адресу, прислал (и может быть продолжает присылать) нам данные.

uint SocketIsReadable(const int socket)

Функция возвращает количество байтов, которые можно прочитать из сокета. В случае ошибки возвращается 0.

> Программисты, знакомые с системными API сокетов в Windows/Linux, знают, что значение 0 также может быть нормальным состоянием, когда во внутреннем буфере сокета нет входящих данных. Однако данная функция в MQL5 ведет себя иначе. При пустом системном буфере сокета, она спекулятивно возвращает 1, откладывая реальную проверку на доступность данных на следующий вызов одной из функций чтения. В частности, данная ситуация с фиктивным результатом 1 байт возникает, как правило, при первом вызове функции на сокете, когда приемный внутренний буфер еще пуст.

При выполнении этой функции может произойти ошибка, означающая что соединение, установленное через *SocketConnect*, было разорвано (в *_LastError* попадет код 5273, ERR_NETSOCKET_IO_ERROR).

Функцию *SocketIsReadable* полезно использовать в программах, которые спроектированы под "неблокирующее" чтение данных с помощью *SocketRead*. Дело в том, что функция *SocketRead* при отсутствии данных в приемном буфере будет ожидать их поступления, приостановив исполнение программы (на заданную величину таймаута).

С другой стороны, блокирующее чтение более надежно в том плане, что ваша программа "проснется" сразу же по приходу новых данных, а вот проверки на их наличие с помощью *SocketIsReadable* нужно делать периодически, по неким другим событиям (как правило, по таймеру или в цикле).

Особую осторожность следует проявить при использование функции *SocketIsReadable* в защищенном режиме TLS. Функция возвращает количество "сырых" данных, которые в режиме TLS представляют собой зашифрованный блок. Если "сырые" данные еще не накоплены в размере блока дешифрования, то последующий вызов функции чтения *SocketTlsRead* заблокирует выполнение программы, ожидая недостающий фрагмент. Если "сырые" данные уже

содержат готовый для дешифрования блок, функция чтения вернет меньшее количество расшифрованных байтов, чем количество "сырых". В связи с этим, при включенном TLS, функцию *SocketIsReadable* рекомендуется всегда использовать в связке с функцией *SocketTlsReadAvailable*. В противном случае, поведение программы будет отличаться от ожидаемого. К сожалению, MQL5 не предоставляет функцию *SocketTlsIsReadable*, совместимую с режимом TLS и не накладывающую описанных условностей.

Похожая функция *SocketIsWritable* проверяет, возможна ли запись в данный сокет в текущий момент времени.

bool SocketIsWritable(const int socket)

Функция возвращает признак успеха (*true*) или ошибки (*false*). В последнем случае, соединение, установленное через *SocketConnect*, будет разорвано.

Приведем простой скрипт *SocketIsConnected.mq5* для проверки функций. Во входных параметрах, как и прежде, предусмотрим возможность ввести адрес и порт.

```
input string Server = "www.mql5.com";
input uint Port = 443;
```

В обработчике *OnStart* создадим сокет, подключимся к сайту и станем проверять в цикле статус сокета. После второй итерации сокет принудительно нами закрывается, и это должно привести к выходу из цикла.

```
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));
         }
      }
   }
}
```

В журнал выводятся такие записи.

```
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 Настройка таймаутов передачи и приема данных сокетами

Поскольку сетевые соединения ненадежны, все операции с *Socket*-функциями поддерживают централизованную настройку таймаутов: если чтение или отправка данных не завершится успешно за указанное время, функция прекратит попытки выполнить соответствующее действие.

Установить таймауты получения и отправки данных позволяет функция *SocketTimeouts*.

bool SocketTimeouts(int socket, uint timeout_send, uint timeout_receive)

Оба таймаута задаются в миллисекундах и влияют на все функции с указанным сокетом, на системном уровне.

Забегая вперед, скажем, что функция *SocketRead* имеет собственный параметр *timeout*, с помощью которого можно дополнительно управлять таймаутом во время конкретного вызова функции *SocketRead*.

SocketTimeouts возвращает *true* в случае успеха, а иначе — *false*.

По-умолчанию таймауты отсутствуют, что соответствует бесконечному ожиданию приема или отправки всех данных.

7.5.8 Чтение, запись данных по незащищенному сокет-соединению

Исторически так сложилось, что сокеты по-умолчанию обеспечивают передачу данных по простому соединению — в открытом виде, и это позволяет техническими средствами анализировать весь трафик. В последние годы к вопросам безопасности стали относиться более серьезно и потому практически повсеместно внедрена технология TLS (Transport Layer Security): она обеспечивает шифрование на лету всех данных между отправителем и получателем. В частности, для интернет-соединений разница заключается в протоколах HTTP (простое соединение) и HTTPS (защищенное).

В MQL5 существуют разные наборы *Socket*-функций для работы с простыми и защищенными соединениями. В этом разделе мы познакомимся с простым режимом, а позднее доберемся и до защищенного.

Для чтения данных из сокета используется функция *SocketRead*.

int SocketRead(int socket, uchar &buffer[], uint maxlen, uint timeout)

В функцию передается дескриптор сокета, полученный из *SocketCreate* и подключенный к сетевому ресурсу с помощью *SocketConnect*.

Параметр *buffer* представляет собой ссылку на массив, в который будут прочитаны данные. Если массив динамический, его размер увеличивается на количество прочитанных байт, но он не может превышать INT_MAX (2147483647). Ограничить количество читаемых байтов можно параметром *maxlen*. Данные, которые не поместятся, останутся во внутреннем буфере сокета: их можно будет получить следующим вызовом *SocketRead*. Величина *maxlen* должна быть в пределах от 1 до INT_MAX (2147483647).

Параметр *timeout* задает время (в миллисекундах) ожидания завершения чтения. Если в течение этого времени не удается получить данные, попытки завершаются, и функция завершает работу: результат при этом равен -1.

-1 также возвращается и при ошибке, а её код в *_LastError* — такой как 5273 (ERR_NETSOCKET_IO_ERROR), например, означает, что соединение, установленное через *SocketConnect*, разорвано.

В случае успеха функция возвращает количество прочитанных байтов.

При выставлении таймаута чтения в 0, используется значение по умолчанию 120000 (2 минуты).

Для записи данных в сокет используется функция *SocketSend*.

> К сожалению, названия функций *SocketRead* и *SocketSend* не являются "симметричными": обратной по смыслу операцией для "read" является "write", а для "send" — "receive". Это может оказаться непривычным для разработчиков со стажем, знакомым с сетевыми API на других платформах.

int SocketSend(int socket, const uchar &buffer[], uint maxlen)

В первом параметре передается дескриптор ранее созданного и открытого сокета. При передаче неверного хэндла в *_LastError* записывается ошибка 5270 (ERR_NETSOCKET_INVALIDHANDLE). В массиве *buffer* содержатся данные для отправки, их размер — в параметре *maxlen* (параметр введен для удобства отправки части данных из фиксированного массива).

В случае успеха функция возвращает количество байт, записанных в сокет, а в случае ошибки — -1.

Ошибки системного уровня (5273, ERR_NETSOCKET_IO_ERROR) сигнализируют о разрыве соединения.

В скрипте *SocketReadWriteHTTP.mq5* продемонстрируем, как с помощью сокетов можно реализовать работу по протоколу HTTP, то есть запросить информацию о странице с веб-сервера. Это малая часть того, что "за сценой" для нас делает функция *WebRequest*.

Оставим во входных параметрах адрес по умолчанию — сайт "www.mql5.com". Номер порта выбран равным 80, поскольку это значение по умолчанию для незащищенных HTTP-соединений (хотя некоторые сервера могут использовать и другой порт: 81, 8080 и т.д.). Порты, зарезервированные под защищенные соединения (в частности, самый популярный 443), данным примером пока не поддерживаются. Также в параметре *Server* важно вводить именно название домена, а не конкретной страницы, потому что скрипт умеет запрашивать только главную страницу, то есть корневой путь "/".

```
input string Server = "www.mql5.com";
input uint Port = 80;
```

В главной функции скрипта создадим сокет и откроем на нем соединение с указанными параметрами (таймаут равен 5 секундам).

```
void OnStart()
{
   PRTF(Server);
   PRTF(Port);
   const int socket = PRTF(SocketCreate());
   if(PRTF(SocketConnect(socket, Server, Port, 5000)))
   {
      ...
   }
}
```

Далее вспомним принцип работы протокола HTTP. Клиент отправляет запросы в виде специально оформленных заголовков (строк с предопределенными названиями и значениями), включающими, в частности, адрес веб-страницы, а сервер в ответ присылает всю веб-страницу или статус операции, используя для этого также специальные заголовки. Клиент может запросить веб-страницу с помощью GET-запроса, отправить некие данные с помощью POST-запроса или проверить статус веб-страницы с помощью экономного HEAD-запроса. В принципе, HTTP-методов гораздо больше — с ними можно познакомиться в спецификации протокола HTTP.

Таким образом, скрипт должен сформировать и отправить через сокет-соединение HTTP-заголовок. В простейшем виде получить мета-информацию о странице позволяет следующий HEAD-запрос (мы могли бы заменить HEAD на GET, чтобы запросить страницу целиком, но там есть некоторые сложности — об этом позднее).

```
HEAD / HTTP/1.1
Host: _server_
User-Agent: MetaTrader 5
                                        // <- два перевода строки подряд \r\n\r\n
```

Косая наклонная черта после "HEAD" (или другого метода) — это кратчайший возможный путь на любом сервере — корневой каталог, который обычно приводит к показу главной страницы. Если нам нужна конкретная веб-страница, мы могли бы написать что-то вроде "GET /en/forum/ HTTP/1.1" и получили бы оглавление англоязычных форумов на *mql5.com*. Вместо строки "_server_" должен быть подставлен реальный домен.

Наличие "User-Agent:" не обязательно, но позволяет программе "представиться" серверу, без чего некоторые сервера могут отклонить запрос.

Обратите внимание, на две пустые строки: они обозначают конец заголовка. В нашем скрипте заголовок удобно формировать таким выражением:

```
StringFormat("HEAD / HTTP/1.1\r\nHost: %s\r\n\r\n", Server)
```

Осталось отправить его на сервер. Для этой цели написана простая функция *HTTPSend*. Она принимает дескриптор сокета и строку заголовка.

```
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;
}
```

Внутри мы преобразуем строку в массив байтов и вызываем *SocketSend*.

Далее нам потребуется принять ответ сервера, для чего написана функция *HTTPRecv*. Она также ожидает дескриптор сокета и ссылку на строку, куда следует поместить данные, но устроена сложнее.

```
bool HTTPRecv(int socket, string &result, const uint timeout)
{
   char response[];
   int len;          // целое со знаком нужно для признака ошибки -1
   uint start = GetTickCount();
   result = "";

   do
   {
      ResetLastError();
      if(!(len = (int)SocketIsReadable(socket)))
      {
         Sleep(10); // ждем данных или таймаута
      }
      else          // читаем данные в имеющемся объеме
      if((len = SocketRead(socket, response, len, timeout)) > 0)
      {
         result += CharArrayToString(response, 0, len); // NB: без CP_UTF8 только 'HE
         const int p = StringFind(result, "\r\n\r\n");
         if(p > 0)
         {
            // HTTP-заголовок завершается двойным переводом строки, используем это
            // чтобы убедиться, что получен весь заголовок
            Print("HTTP-header found");
            StringSetLength(result, p); // отрезаем тело документа (на случай GET-зап
            return true;
         }
      }
   }
   while(GetTickCount() - start < timeout && !IsStopped() && !_LastError);

   if(_LastError) PRTF(_LastError);

   return StringLen(result) > 0;
}
```

Здесь мы в цикле проверяем появление данных в пределах заданного таймаута и читаем их в буфер *response*. Возникновение ошибки прерывает цикл.

Байты буфера сразу же конвертируются в строку и стыкуются в полный ответ в переменной *result*. Важно отметить, что функцию *CharArrayToString* с кодировкой по умолчанию мы можем использовать только для HTTP-заголовка, потому что в нем разрешены только латинские буквы и несколько специальных символов из ANSI.

Для приема полного веб-документа, который, как правило, имеет кодировку UTF-8 (но потенциально может иметь и другую — нелатинскую, которая указывается как раз в HTTP-заголовке) потребуется более хитрая обработка: сначала нужно собрать все присланные блоки в одном общем буфере, а потом всё целиком преобразовывать в строку с указанием CP_UTF8 (в противном случае, любой символ, закодированный двумя байтами, может быть "разрезан" при отправке и прибудет к нам в разных блоках — именно поэтому нельзя ожидать корректного потока байт UTF-8 в отдельно взятом фрагменте). Мы улучшим данный пример в следующих разделах.

Имея функции *HTTPSend* и *HTTPRecv*, завершим код *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);
         }
      }
      ...
}
```

В получаемом от сервера HTTP-заголовке интерес могут представлять следующие строки:

- 'Content-Length:' — общая длина документа в байтах;
- 'Content-Language:' — язык документа (например, "de-DE, ru");
- 'Content-Type:' — кодировка документа (например, "text/html; charset=UTF-8");
- 'Last-Modified:' — время последнего изменения документа, чтобы не скачивать то, что уже есть (в принципе, мы для этого можем в своем HTTP-запросе добавить заголовок 'If-Modified-Since:').

Про выяснение длины документа (размера данных) мы еще поговорим более подробно, потому что практически все заголовки являются опциональными, то есть сообщаются сервером по желанию, и при их отсутствии используются альтернативные механизмы. Размер важен, чтобы знать, когда закрывать соединение, то есть убедиться, что получены все данные.

Запуск скрипта с параметрами по умолчанию выдает такой результат.

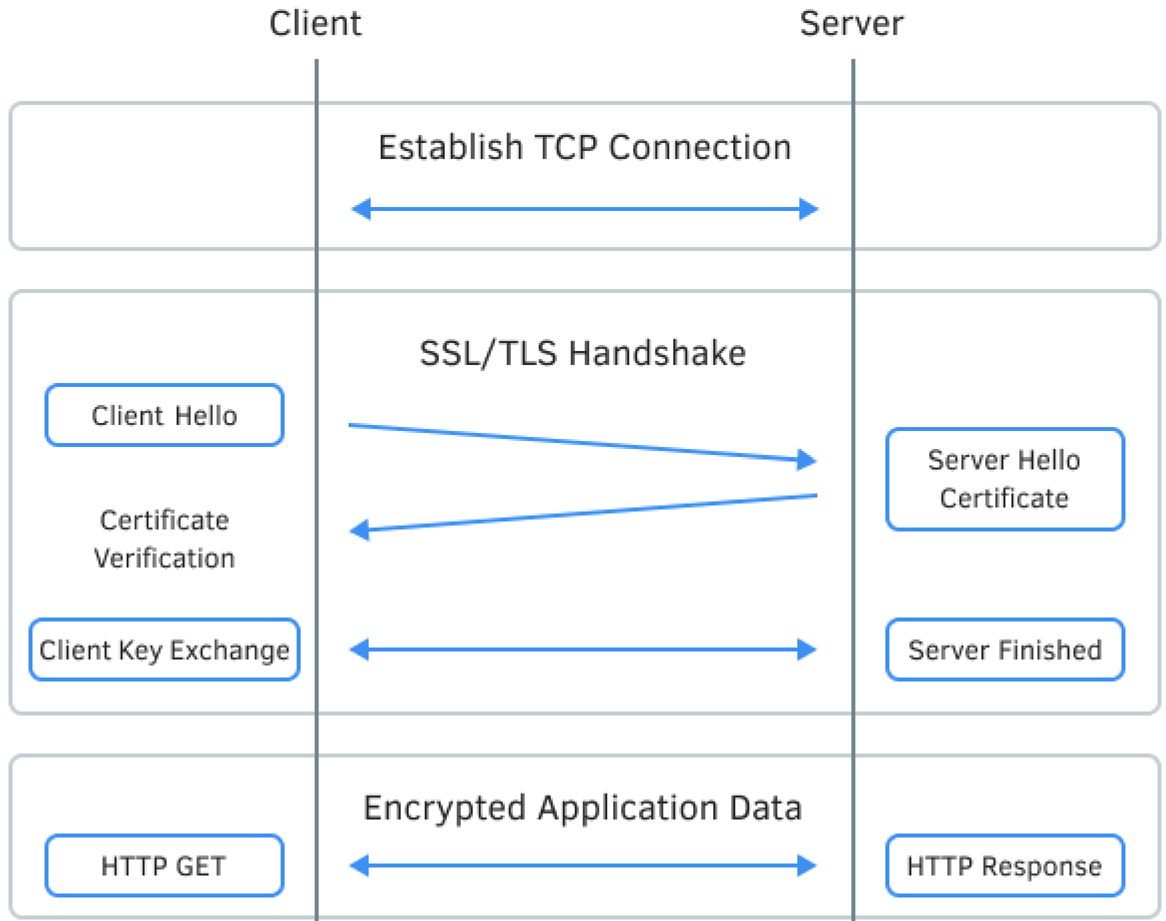
```
Server=www.mql5.com / ok
Port=80 / ok
SocketCreate()=1 / ok
SocketConnect(socket,Server,Port,5000)=true / ok
HTTPSend(socket,StringFormat(HEAD / HTTP/1.1
Host: %s
,Server))=true / ok
HTTP-header found
HTTPRecv(socket,response,5000)=true / ok
HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Sun, 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
```

Обратите внимание, что данный сайт, как и большинство сегодня, перенаправляет наш запрос на защищенное соединение: это достигается кодом статуса "301 Moved Permanently" и новым адресом "Location: https://www.mql5.com/" (здесь важен протокол "https"). Чтобы повторить запрос с поддержкой TLS, следует использовать несколько других функций, о которых и пойдет далее речь.

7.5.9 Подготовка защищенного сокет-соединения

Для перевода сокет-соединения в защищенное состояние и его проверки в MQL5 существует пара функций: *SocketTlsHandshake* и *SocketTlsCertificate*, соответственно. Как правило, нам не требуется "вручную" включать защиту вызовом *SocketTlsHandshake*, если соединение устанавливается на порту 443. Дело в том, что он является стандартным для HTTPS (TLS).

Принцип работы защиты основан на шифровании потока данных между клиентом и сервером, для чего изначально используется пара асимметричных ключей: открытого и закрытого. Мы уже касались этой темы в разделе Обзор доступных методов преобразования информации. Каждый уважающий себя сайт приобретает цифровой сертификат у одного из удостоверяющих центров, которым доверяет сетевое сообщество. Сертификат содержит публичный ключ сайта и подписан цифровой подписью центра. Браузеры и прочие клиентские приложения хранят (или могут импортировать) публичные ключи удостоверяющих центров и потому могут убедиться в качестве конкретного сертификата.



Установление защищенного TLS-соединения
(рисунок из инета)

Далее при подготовке защищенного соединения браузер или приложение генерирует некий "секрет", шифрует его открытым ключом сайта и отправляет ему, а сайт расшифровывает известным только ему закрытым ключом. Данный этап на практике выглядит более сложно, но в результате и клиент, и сервер обладают ключом шифрования для текущей сессии (соединения). Этот ключ используется обоими участниками коммуникации для шифрования последующих запросов и ответов на одном конце и расшифровки на другом.

Функция *SocketTlsHandshake* инициирует защищенное TLS-соединение с указанным хостом по протоколу TLS Handshake. При этом клиент и сервер согласовывают параметры соединения: версию используемого протокола и способ шифрования данных.

bool SocketTlsHandshake(int socket, const string host)

В параметрах функции передается дескриптор сокета и адрес сервера, с которым установлено соединение (по сути, это — то же имя, которое указывалось в *SocketConnect*).

До защищенного соединения программа должна предварительно установить обычное TCP-соединение с хостом при помощи *SocketConnect*.

Функция возвращает *true* в случае успеха, а иначе — *false*. При ошибке в *_LastError* записывается код 5274 (ERR_NETSOCKET_HANDSHAKE_FAILED).

Функция *SocketTlsCertificate* получает данные о сертификате, используемом для защиты сетевого соединения.

int SocketTlsCertificate(int socket, string &subject, string &issuer, string &serial, string &thumbprint, datetime &expiration)

Если для сокета установлено защищенное соединение (либо после явного и успешного вызова *SocketTlsHandshake*, либо после подключения по порту 443), то данная функция по дескриптору сокета заполняет все остальные ссылочные переменные соответствующей информацией: именем владельца сертификата (*subject*), именем издателя сертификата (*issuer*), серийным номером (*serial*), цифровым отпечатком (*thumbprint*) и сроком действия (*expiration*) сертификата.

Функция возвращает *true* в случае успешного получения сведений о сертификате или *false* в результате ошибки. Код ошибки при этом — 5275 (ERR_NETSOCKET_NO_CERTIFICATE). Это можно использовать для определения того факта, установилось ли соединение, открытое функцией *SocketConnect*, сразу в защищенном режиме. Мы воспользуемся этим в примере, в следующем разделе.

7.5.10 Чтение и запись данных по защищенному сокет-соединению

Для защищенного соединения имеется свой набор функций обмена данными между клиентом и сервером. Названия и принцип работы функций почти совпадают с уже рассмотренными функциями *SocketRead* и *SocketSend*.

int SocketTlsRead(int socket, uchar &buffer[], uint maxlen)

Функция *SocketTlsRead* читает данные из защищенного TLS-соединения, открытого на указанном сокете. Данные попадают в передаваемый по ссылке массив *buffer*. Если он динамический, его размер будет увеличен под объем данных, но не более чем до INT_MAX (2147483647) байтов.

Параметр *maxlen* задает количество расшифрованных байтов, которые требуется получить (их количество всегда меньше, чем объем "сырых" зашифрованных данных, поступающих во внутренний буфер сокета). Данные, которые не поместятся в массив, останутся в сокете, и их можно будет получить следующим вызовом *SocketTlsRead*.

Функция исполняется до тех пор, пока не получит указанное количество данных или не наступит таймаут, заданный с помощью *SocketTimeouts*.

В случае успеха функция возвращает количество прочитанных байтов, в случае ошибки — -1, при этом в *_LastError* записывается код 5273 (ERR_NETSOCKET_IO_ERROR). Наличие ошибки говорит о том, что соединение было разорвано.

int SocketTlsReadAvailable(int socket, uchar &buffer[], const uint maxlen)

Функция *SocketTlsReadAvailable* читает все доступные расшифрованные данные из защищенного TLS-соединения, но не более *maxlen* байтов. В отличие от *SocketTlsRead*, *SocketTlsReadAvailable* не ждет обязательного наличия заданного количества данных и сразу возвращает только то, что есть. Таким образом, если внутренний буфер сокета "пуст" (с сервера пока ничего не поступило, уже было прочитано или еще не сформировало готовый для дешифрации блок), функция вернет 0, и в приемный массив *buffer* ничего не будет записано. Это — штатная ситуация.

Величина *maxlen* должна быть в пределах от 1 до INT_MAX (2147483647).

int SocketTlsSend(int socket, const uchar &buffer[], uint bufferlen)

Функция *SocketTlsSend* отправляет данные из массива *buffer* через защищенное соединение, открытое на указанном сокете. Принцип действия тот же, что у описанной ранее функции *SocketSend* — разница только в типе соединения.

За основу нового примера скрипта *SocketReadWriteHTTPS.mq5* возьмем предыдущий *SocketReadWriteHTTP.mq5*, но добавим гибкости — выбор HTTP-метода (по умолчанию GET, а не HEAD), настройку таймаута и поддержку защищенных соединений. Стандартным таким портом является 443.

```
input string Method = "GET"; // Method (HEAD,GET)
input string Server = "www.google.com";
input uint Port = 443;
input uint Timeout = 5000;
```

Сервером по умолчанию указан "www.google.com" — не забудьте добавить его (и любой другой, который вы введете) в список разрешенных в настройках терминала.

Определять, является ли соединение защищенным или нет, будем с помощью функции *SocketTlsCertificate*: если она завершится успешно, значит сервер предоставил сертификат, и режим TLS активен. Если функция вернет *false* и взведет код ошибки NETSOCKET_NO_CERTIFICATE(5275) — значит, мы пользуемся обычным соединением, но ошибку можно проигнорировать и сбросить, поскольку нас устраивает и незащищенное соединение.

```
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;
      }
      ...
```

Остальная часть функции *OnStart* выполнена по прежнему плану: отправить запрос с помощью функции *HTTPSend* и принять ответ с помощью *HTTPRecv*. Но на этот раз мы дополнительно передаем в эти функции признак TLS, и они должны быть реализованы слегка иначе.

```
      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");
            // для больших документов предусмотрим сохранение в файл
            if(StringLen(response) > 1000)
            {
               int h = FileOpen(Server + ".htm", FILE_WRITE | FILE_TXT | FILE_ANSI, C
               FileWriteString(h, response);
               FileClose(h);
            }
            else
            {
               Print(response);
            }
         }
      }
```

На примере *HTTPSend* легко увидеть, что теперь в зависимости от флага TLS, мы используем либо *SocketTlsSend*, либо *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
}
```

С функцией *HTTPRecv* все несколько сложнее. Поскольку мы предоставляем возможность скачивать всю страницу (а не только заголовки), требуется неким образом узнать, получили ли мы все данные. Даже после передачи всего документа сокет, как правило, остается открытым для оптимизации последующих предполагаемых запросов. Но наша программа не будет знать, прекратилась ли передача штатно или, может быть, случился временный "затор" где-то в сетевой инфраструктуре (подобную неспешную, прерывистую загрузку страниц иногда можно наблюдать и в браузерах). Или наоборот, в случае обрыва соединения мы можем ошибочно посчитать, что приняли весь документ.

Дело в том, что сокеты сами по себе выступают лишь средством коммуникации программ и работают с абстрактными блоками данных: им неизвестно, какого типа данные, что они означают, и где у них логическое завершение. Всеми этими вопросами занимаются прикладные протоколы вроде HTTP. Поэтому нам потребуется углубиться в спецификацию и реализовать проверки самостоятельно.

```
bool HTTPRecv(int socket, string &result, const uint timeout, const bool TLS)
{
   uchar response[]; // накапливаем данные целиком (заголовки + тело веб-документа)
   uchar block[];    // отдельный блок чтения
   int len;          // размер текущего блока (целое со знаком для признака ошибки -1
   int lastLF = -1;  // позиция последнего найденного перевода строки LF(Line-Feed)
   int body = 0;     // смещение, где начинается тело документа
   int size = 0;     // размер документа согласно заголовку
   result = "";      // устанавливаем в начале пустой результат
   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;
   ...
```

Наиболее простой метод определения размера принимаемых данных основывается на анализе заголовка "Content-Length:". Для него нам потребовались переменные *lastLF*, *size*, *content_length*. Правда, заголовок этот присутствует не всегда, и в дело вступают "чанки" (chunks) — для их обнаружения введены переменные *chunk_size*, *chunk_start*, *crlf* и *crlf_length*.

Для демонстрации различных техник приема данных воспользуемся в этом примере "неблокирующей" функцией *SocketTlsReadAvailable*. Однако, аналогичная функция для незащищенного соединения отсутствует, и потому нам предстоит её написать самим (чуть позже). Общая схема алгоритма проста — это цикл с попытками получения новых блоков данных размером 1024 (или меньше) байтов. Если что-то удалось прочитать, мы это аккумулируем в массиве response. Если входной буфер сокета пуст, функции вернут 0, и мы делаем маленькую паузу. Наконец, если возникнет ошибка или таймаут, цикл прервется.

```
   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); // собираем все блоки вместе
         ...
         // основная работа здесь
      }
      else
      {
         if(len == 0) Sleep(10); // ждем немного прихода порции данных
      }
   }
   while(GetTickCount() - start < timeout && !IsStopped() && !_LastError);
   ...
```

Прежде всего, необходимо дождаться во входном потоке данных завершения HTTP-заголовока. Как мы уже видели из предыдущего примера, заголовки отделены от документа двойным переводом строки, то есть последовательностью символов "\r\n\r\n". Её легко обнаружить по двум символам '\n' (LF), расположенным через один.

Результатом поиска станет смещение в байтах от начала данных, где кончается заголовок и начинается документ. Мы сохраним его в переменную *body*.

```
         if(body == 0) // ищем завершение заголовков, пока не найдем
         {
            for(int i = n; i < ArraySize(response); ++i)
            {
               if(response[i] == '\n') // LF
               {
                  if(lastLF == i - crlf_length) // найдена последовательность "\r\n\r
                  {
                     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; // найдена граница заголовков/тела
                  }
                  lastLF = i;
               }
            }
         }
         
         if(size == ArraySize(response) - body) // документ целиком
         {
            Print("* Complete document");
            break;
         }
         ...
```

При этом сразу выполняется поиск заголовка "Content-Length:" и извлечение из него размера. Заполненная переменная *size* дает возможность написать дополнительный условный оператор для выхода из цикла приема данных, когда получен весь документ.

Некоторые сервера "отдают" содержимое по частям — именно они называются "chunks". В таких случаях в HTTP-заголовке присутствует строка "Transfer-Encoding: chunked", а строка "Content-Length:" отсутствует. Каждый "чанк" начинается с шестнадцатеричного числа, указывающего размер "чанка", после чего следует перевод строки и указанное количество байтов с данными. Завершается "чанк" еще одним переводом строки. Последний "чанк", обозначающий конец документа, имеет нулевой размер.

Обратите внимание, что деление на подобные сегменты выполняется сервером, исходя из собственных, текущих "предпочтений" по оптимизации отправки, и никак не связано с блоками (пакетами) данных, на которые информация делится на уровне сокетов для передачи по сети.

Иными словами, "чанки" сами, как правило, произвольным образом фрагментированы, и граница между сетевыми пакетами может случиться даже между цифр в размере "чанка".

Схематично это можно изобразить так (слева "чанки" документа, справа блоки данных из сокет-буфера).

Фрагментация веб-документа при передаче на уровнях HTTP и TCP

В нашем алгоритме пакеты — это то, что попадает на каждой итерации в массив *block*, но анализировать их по одиночке не имеет смысла и вся основная работа идет с общим массивом response.

Итак, если HTTP-заголовок полностью получен, но в нем не обнаружена строка "Content-Length:", мы переходим в ветвь алгоритма с режимом "Transfer-Encoding: chunked". По текущей позиции *body* в массиве *response* (сразу после завершения HTTP-заголовков) выделяется фрагмент строки и преобразуется в число из предположения шестнадцатеричного формата: это делает вспомогательная функция *HexStringToInteger* (см. прилагаемый исходный код). Если там действительно обнаруживается число, мы записываем его в *chunk_size*, помечаем позицию, как начало "чанка" в *chunk_start* и удаляем из *response* байты с самим числом и обрамляющими переводами строк.

```
...
if(lastLF == i - crlf_length) // найдена последовательность "\r\n\r
{
   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; // сервер не предоставил длину документа
      // пытаемся найти чанки и размер первого из них
      if(StringFind(headers, "Transfer-Encoding: chunked") > 0)
      {
         // синтаксис чанка:
         // <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; // найдена граница заголовков/тела
}
lastLF = i;
...
```

Теперь для проверки завершенности документа нужно анализировать не только переменную *size* (которая, как мы видели, может быть фактически выключена из работы присвоением -1 в отсутствии "Content-Length:"), но и новые переменные для "чанков": *chunk_start* и *chunk_size*.

Принцип действий тот же самый, что и после HTTP-заголовков: по смещению в массиве *response*, где завершился предыдущий "чанк", вычленяем размер следующего "чанка". Продолжаем процесс, пока не найдем "чанк" нулевого размера.

```
...
if(size == ArraySize(response) - body) // документ целиком
{
   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              // есть '\r\n...\r\n' ?
      && StringFind(preview, crlf, crlf_length) > crlf_length)
   {
      chunk_size = HexStringToInteger(preview, crlf_length);
      if(chunk_size > 0)
      {                               // дважды '\r\n': до и после размера ча
         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;
      }
   } // иначе ждем еще данных
}
```

Таким образом, мы обеспечили выход из цикла по результатам анализа входящего потока двумя разными способами (помимо выхода по таймауту и по ошибке). При штатном завершении цикла мы преобразуем в строку ту часть массива *response*, которая начинается с позиции *body* и содержит целый документ, а иначе вернем просто все, что успели получить, вместе с заголовками — для "разбора полетов".

```
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)
      {
         ... // основная работа здесь - рассмотрено выше
      }
      else
      {
         if(len == 0) Sleep(10); // ждем немного прихода порции данных
      }
   }
   while(GetTickCount() - start < timeout && !IsStopped() && !_LastError);
   
   if(_LastError) PRTF(_LastError);
   
   if(ArraySize(response) > 0)
   {
      if(body != 0)
      {
         // TODO: желательно проверить 'Content-Type:' на 'charset=UTF-8'
         result = CharArrayToString(response, body, WHOLE_ARRAY, CP_UTF8);
      }
      else
      {
         // для анализа нештатных ситуаций вернем неполные заголовки как есть
         result = CharArrayToString(response);
      }
   }
   
   return StringLen(result) > 0;
}
```

Осталось показать функцию *SocketReadAvailable* — аналог *SocketTlsReadAvailable* для незащищенных соединений.

```
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;
}
```

Скрипт готов к работе.

От нас потребовалось довольно много усилий, чтобы реализовать простой запрос веб-страницы с помощью сокетов. Это служит демонстрацией того, какая большая рутинная работа обычно

скрывается в поддержке сетевых протоколов на низком уровне. Конечно, в случае HTTP нам проще и правильнее использовать встроенную реализацию WebRequest, но она не включает всех возможностей HTTP (причем мы вскользь затронули HTTP 1.1, а есть еще HTTP/2), да и количество других прикладных протоколов огромно, поэтому для их интеграции в MetaTrader 5 не обойтись без *Socket*-функций.

Запустим *SocketReadWriteHTTPS.mq5* с параметрами по умолчанию.

```
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
```

Как мы видим, документ передается "чанками" и был сохранен во временный файл (вы можете найти его в *MQL5/Files/www.mql5.com.htm*).

Запустим теперь скрипт для сайта "www.mql5.com" и порта 80. Из предыдущего раздела мы знаем, что сайт в этом случае выдает перенаправление на свою защищенную версию, но этот "редирект" не пустой: у него есть документ-заглушка, и теперь мы можем получить его полностью. Для нас здесь важно, что в данном случае корректно используется заголовок "Content-Length:".

```
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>
```

Еще один, большой пример использования сокетов на практике мы рассмотрим в главе Проекты.

7.6 База данных SQLite

Относительно недавно, по меркам долгой истории развития платформы, в MetaTrader 5 была встроена "родная" поддержка базы данных SQLite. Это — легкая, но полнофункциональная система управления базами данных (СУБД). Традиционно, подобные системы ориентированы на обработку таблиц данных, где хранятся однотипные записи с общим набором атрибутов, причем между записями разных типов (т.е. таблиц) могут быть установлены различные соответствия (связи или "реляции"), в связи с чем такие базы еще называют реляционными. Мы уже рассматривали примеры таких связей между структурами экономического календаря, но база календаря хранится внутри терминала, а функции данного раздела позволят создавать произвольные базы из MQL-программ.

Специализация СУБД на указанных структурах данных позволяет оптимизировать — ускорить и упростить — многие востребованные операции: сортировку, поиск, фильтрацию, суммирование или вычисление других агрегатных функций для больших объемов данных.

Однако у этой медали есть и другая сторона: для программирования СУБД требуется свой собственный язык SQL (Structured Query Language), и знания чистого MQL5 будет недостаточно. В отличие от MQL5, который относится к *императивным* языкам (использующим операторы-указания, что, как, в какой последовательности делать), SQL является *декларативным* — он

описывает исходные данные и желаемый результат, без указания, как и в какой последовательности производить вычисления. Суть алгоритма на SQL описывается в виде SQL-запросов. Запрос — это аналог отдельного оператора MQL5, формируемый в виде строки по особому синтаксису.

Вместо программирования сложных циклов и сравнений достаточно вызвать функции SQLite (например, *DatabaseExecute* или *DatabasePrepare*), передавая им SQL-запросы. Для получения результатов запроса в готовую структуру MQL5 можно использовать функцию *DatabaseReadBind* — это позволит прочитать сразу все поля записи (структуры) за один вызов.

С помощью функций базы данных легко создавать таблицы, добавлять в них записи, производить модификации и делать выборки по сложным условиям, например, для таких задач как:

- получение торговой истории и котировок,
- сохранение результатов оптимизации и тестирования,
- подготовка и обмен данными с другими пакетами анализа,
- анализ данных экономического календаря,
- хранение настроек и состояния MQL5-программ.

Кроме того, в SQL-запросах можно использовать широкий набор общеупотребительных, статистических и математических функций. Причем выражения с их участием можно вычислять даже без создания таблицы.

Особенностью SQLite является то, что эта СУБД не требует отдельного приложения, настройки и администрирования, нетребовательна к ресурсам, поддерживает большинство команд популярного стандарта SQL92. Дополнительное удобство, что вся база данных находится в единственном файле на жестком диске на компьютере пользователя, и её можно легко переносить или создавать резервные копии. Однако для ускорения операций чтения, записи и модификации базу данных можно открывать/создавать и в оперативной памяти с флагом DATABASE_OPEN_MEMORY, правда при этом такая база будет доступна только данной конкретной программе и не может использоваться для совместной работы нескольких программ.

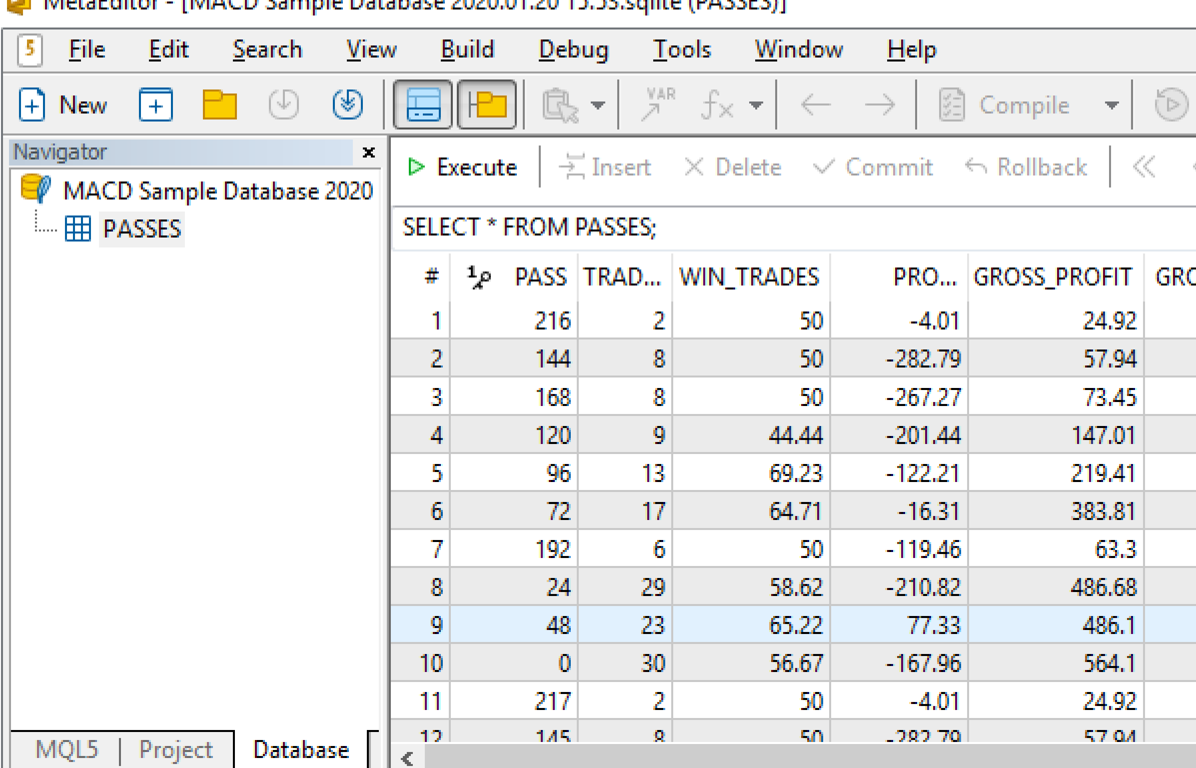
> Важно отметить, что относительная простота SQLite по сравнению с полнофункциональными СУБД, обуславливает некоторые ограничения. В частности, SQLite не имеет выделенного процесса (системного сервиса или приложения), который бы предоставлял централизованный доступ к API управления базами и таблицами, из-за чего параллельный, разделяемый доступ к одной и той же базе (файлу) из разных процессов не гарантируется самой СУБД. Так, если потребуется одновременное чтение и запись в базу из агентов оптимизации, выполняющих экземпляры одного и того же эксперта, потребуется самостоятельно написать в нем код для синхронизации доступа (иначе, записываемые и считываемые данные окажутся в противоречивом состоянии: ведь порядок записи, модификации, удаления, чтения из параллельных несинхронизированных процессов случаен). Более того, попытки одновременного изменения базы могут приводить к получению MQL-программой ошибок "база данных занята" (и запрошенная операция при этом не выполняется). Единственный сценарий, не требующий синхронизации параллельных операций с SQLite, — это, когда задействованы только операции чтения.

Мы представим лишь основы SQL в объеме необходимом для начала прикладного применения. Полное описание синтаксиса и принципов работы SQL выходит за рамки этой книги. Изучите документацию на сайте SQLite. Вместе с тем, учтите, что MQL5 и MetaEditor поддерживают ограниченное подмножество команд и синтаксических конструкций SQL.

В редактор MetaEditor, в состав *Мастера MQL*, встроена опция создания базы данных, которая сразу предлагает сформировать и первую таблицу, определив список ее полей. Также для работы с базами данных в *Навигаторе* предусмотрена отдельная вкладка.

Воспользовавшись *Мастером* или контекстным меню *Навигатора*, вы можете создать пустую базу данных (файл на диске, по умолчанию, в каталоге *MQL5/Files*) одного из нескольких поддерживаемых форматов (*.db, *.sql, *.sqlite и других). Кроме того, в контекстном меню можно импортировать базу целиком из sql-файла или отдельные таблицы из csv-файлов.

Существующую или созданную базу легко открыть через то же меню. После этого ее таблицы появятся в *Навигаторе*, а в правой, основной области окна выводится панель с инструментами для отладки SQL-запросов и таблица с результатами. Например, двойной щелчок мышью на названии таблицы осуществляет быстрый запрос всех полей записей, что соответствует оператору "SELECT * FROM 'table'", который отображается в поле ввода вверху.



Просмотр базы данных SQLite в MetaEditor

Вы можете отредактировать запрос и нажать кнопку *Выполнить* для его активации. Потенциальные ошибки в синтаксисе SQL выводятся в журнал.

Подробнее о *Мастере*, импорте/экспорте баз данных и интерактивной работе с ними см. документацию MetaEditor.

7.6.0 Знакомство с принципами работы с базой данных в MQL5

Базы данных хранят информацию в виде таблиц. Получение, модификация и добавление новых данных в них делается с помощью запросов на языке SQL. Про его специфику мы расскажем в следующих разделах. А пока продемонстрируем на примере скрипта *DatabaseRead.mq5*, никак не связанного с трейдингом, как создать простую базу данных и получить из неё информацию. Все

упомянутые здесь функции будут подробно описаны позднее. Сейчас важно представить себе общие принципы.

Создание и закрытие базы с помощью встроенных функций *DatabaseOpen/DatabaseClose* аналогичны работе с файлами — так же создаем дескриптор для базы данных, проверяем его и закрываем в конце.

```
void OnStart()
{
   string filename = "company.sqlite";
   // создадим или откроем базу данных
   int db = DatabaseOpen(filename, DATABASE_OPEN_READWRITE | DATABASE_OPEN_CREATE);
   if(db == INVALID_HANDLE)
   {
      Print("DB: ", filename, " open failed with code ", _LastError);
      return;
   }
   ... // дальнейшая работа с базой данных
   // закрываем базу данных
   DatabaseClose(db);
}
```

После открытия базы убедимся в отсутствии в ней таблицы под нужным нам именем — если таблица уже существует, то при попытке вставить в неё такие же данные, как в нашем примере, произойдет ошибка, поэтому используем функцию *DatabaseTableExists*.

Удаление и создание таблицы совершается с помощью запросов, которые отправляются в базу двумя вызовами функции *DatabaseExecute* и сопровождаются контролем ошибок.

```
...
// если таблица COMPANY существует, то удалим её
if(DatabaseTableExists(db, "COMPANY"))
{
   if(!DatabaseExecute(db, "DROP TABLE COMPANY"))
   {
      Print("Failed to drop table COMPANY with code ", _LastError);
      DatabaseClose(db);
      return;
   }
}
// создаем таблицу 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;
}
...
```

Поясним суть SQL-запросов. В таблице COMPANY у нас всего 5 полей: ID записи, имя, возраст, адрес и зарплата. Причем поле ID является ключом, то есть уникальным индексом. Индексы позволяют однозначно определять каждую запись и могут использоваться в разных таблицах для того чтобы связывать их между собой. Это аналогично тому, как идентификатор позиции связывает между собой все сделки и ордера, которые относятся к конкретной позиции.

Теперь необходимо заполнить таблицу данными, делается это с помощью запроса "INSERT":

```
// вставляем данные в таблицу
if(!DatabaseExecute(db,
   "INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) VALUES (1,'Paul',32,'Califorr
   "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-Mor
{
   Print("DB: ", filename, " insert failed with code ", _LastError);
   DatabaseClose(db);
   return;
}
...
```

Здесь в таблицу COMPANY добавляются 4 записи, для каждой записи указывается список полей и значений, которые будут записаны в эти поля. Записи вставляются отдельными запросами "INSERT...", которые объединены в одну строку, через специальный символ-разделитель ';', но мы могли бы каждую запись вставлять в таблицу отдельным вызовом *DatabaseExecute*.

Так как по окончании работы скрипта база будет сохранена в файл "company.sqlite", то при последующем его запуске мы попытались бы записать те же самые данные в таблицу COMPANY с

таким же ID. Это привело бы к ошибке — именно поэтому мы ранее удалили таблицу, чтобы при каждом запуске скрипта начинать работу с нуля.

Теперь получим все записи из таблицы COMPANY, где поле *SALARY > 15000*. Делается это с помощью функции *DatabasePrepare*, которая "компилирует" текст запроса и возвращает его дескриптор для последующего использования в функциях *DatabaseRead* или *DatabaseReadBind*.

```
// подготовим запрос с дескриптором
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;
}
...
```

После того как запрос успешно создан, необходимо получить результаты его выполнения. Это можно сделать с помощью функции *DatabaseRead*, которая при первом вызове выполнит запрос и перейдет на первую запись в результатах. При каждом последующем вызове она будет считывать следующую запись, пока не дойдет до конца. В этом случае она вернет *false*, что означает "записей больше нет".

```
// распечатаем все записи с зарплатой больше 15000
int id, age;
string name, address;
double salary;
Print("Persons with salary > 15000:");
for(int i = 0; DatabaseRead(request); i++)
{
   // прочитаем значения каждого поля из полученной записи по его номеру
   if(DatabaseColumnInteger(request, 0, id) && DatabaseColumnText(request, 1, name
      DatabaseColumnInteger(request, 2, age) && DatabaseColumnText(request, 3, adc
      DatabaseColumnDouble(request, 4, salary))
      Print(i, ":  ", id, " ", name, " ", age, " ", address, " ", salary);
   else
   {
      Print(i, ": DatabaseRead() failed with code ", _LastError);
      DatabaseFinalize(request);
      DatabaseClose(db);
      return;
   }
}
// удалим дескриптор после использования
DatabaseFinalize(request);
```

Результатом выполнения будет:

```
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
```

Таким образом, функция *DatabaseRead* позволяет пройти по всем записям из результата запроса и далее получить полную информацию о каждом столбце в полученной таблице через *DatabaseColumn*-функции. Эти функции предназначены для универсальной работы с результатами любого запроса, но платой за это становится избыточный код.

Если структура результатов запроса заранее известна, то лучше воспользоваться функцией *DatabaseReadBind*, которая позволяет считать сразу всю запись целиком в структуру. Мы можем переделать предыдущий пример таким образом и представить под новым названием *DatabaseReadBind.mq5*. Сначала объявим структуру *Person*:

```
struct Person
{
   int    id;
   string name;
   int    age;
   string address;
   double salary;
};
```

Затем сделаем вычитку каждой записи из результатов запроса с помощью *DatabaseReadBind(request, person)* в цикле, пока функция возвращает значение *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);
```

Таким образом, мы сразу получаем из текущей записи значения всех полей и нам не требуется вычитывать их по отдельности.

Этот вводный пример был взят из статьи SQLite: нативная работа с базами данных на SQL в MQL5, где кроме него рассмотрено несколько вариантов прикладного применения базы данных для трейдеров. В частности, там можно найти восстановление истории позиций из сделок, анализ торгового отчета в разрезе стратегий, рабочих символов или наиболее предпочтительных торговых часов, а также приемы работы с результатами оптимизации.

Для освоения этого материала может потребоваться некоторое минимальное знание языка SQL, поэтому рассмотрим его вкратце в следующих разделах.

7.6.1 Основы SQL

Все задачи, выполняемые в SQLite, предполагают наличие рабочей базы данных (одной или нескольких), поэтому создание и открытие базы данных (по аналогии с файлом) являются обязательными рамочными операциями, устанавливающими необходимую программную среду. Средств для программного удаления базы в SQLite не предусмотрено — считается, что вы можете просто удалить файл базы с диска.

В контексте открытой базы нам становятся доступны действия, которые можно условно разделить на следующие основные группы:

- создание и удаление таблиц, а также модификация их схем, то есть описаний столбцов, с указанием типов, имен и ограничений;
- создание (добавление), чтение, редактирование и удаление записей в таблицах — данные операции часто обозначают общей аббревиатурой CRUD (Create, Read, Update, Delete);
- построение запросов для выборки записей из одной или комбинации нескольких таблиц по сложным условиям;
- оптимизация алгоритмов за счет построения индексов по избранным столбцам, использования представлений (view), заключения пакетных действий в транзакции, объявления триггеров обработки событий и других инструментов продвинутого уровня.

В базах данных SQL все эти действия выполняются с помощью зарезервированных SQL-команд (или иначе операторов). В силу специфики интеграции с MQL5, часть действий выполняется встроенными функциями MQL5. Например, открытие, применение или отмена транзакции выполняется тройкой *DatabaseTransaction*-функций, хотя в стандарте SQL (и в публичной реализации SQLite) имеются соответствующие SQL-команды (BEGIN TRANSACTION, COMMIT, ROLLBACK).

Большинство SQL-команд доступно и в MQL-программах: они передаются в исполняющий "движок" SQLite как строковые параметры функций *DatabaseExecute* или *DatabasePrepare*. Разница между этими двумя вариантами заключается в нескольких нюансах.

DatabasePrepare позволяет подготовить запрос для его последующего массового циклического исполнения, причем на каждой итерации — с различными значениями параметров (сами параметры, то есть их имена в запросе, — одни и те же). Кроме того, подобные подготовленные запросы предоставляют механизм чтения результатов с помощью *DatabaseRead* и *DatabaseReadBind*. То есть с их помощью можно "пролистать" набор отобранных записей.

В отличие от этого, функция *DatabaseExecute* выполняет переданный одиночный запрос в одностороннем порядке: команда уходит внутрь "движка" SQLite, производит некие действия над данными, но ничего не возвращает. Это обычно используется для создания таблиц или пакетной модификации данных.

В последующем нам часто придется оперировать несколькими базовыми понятиями.

Таблица — структурированная совокупность данных, состоящая из строк и столбцов. Каждая строка — это отдельная запись данных с полями (свойствами), описанными с помощью имени и типа соответствующих столбцов. Все таблицы базы физически хранятся в файле базы, и доступны на чтение и запись (если при открытии базы не были ограничены права).

Представление — своего рода виртуальная таблица, рассчитываемая движком SQLite на основе заданного SQL-запроса, других таблиц или представлений. Представления доступны только на чтение. В отличие от любых таблиц (включая временные, которые SQL позволяет создавать в памяти на период сеанса работы программы), представления динамически пересчитываются каждый раз, когда к ним идет обращение.

Индекс — служебная структура данных (так называемое сбалансированное дерево, B-tree) для быстрого поиска записей по значениям предопределенных полей (свойств) или их комбинаций.

Триггер — подпрограмма из одной или нескольких SQL-инструкций, назначенная для автоматического запуска в ответ на события (до или после) добавления, изменения или удаления записи в конкретной таблице.

Приведем краткий список наиболее востребованных SQL-операторов и выполняемые ими действия:

- CREATE — создаёт объект базы данных (таблицу, представление, индекс, триггер);
- ALTER — изменяет объект (таблицу);
- DROP — удаляет объект (таблицу, представление, индекс, триггер);
- SELECT — выбирает записи или вычисляет значения, удовлетворяющие заданным условиям;
- INSERT — добавляет новые данные (одну или набор записей);
- UPDATE — изменяет существующие записи;
- DELETE — удаляет записи из таблицы;

Здесь указаны лишь ключевые слова, начинающие соответствующую языковую конструкцию SQL, а более развернутый синтаксис — чуть ниже. Как это выглядит на практике — станет ясно из последующих примеров.

Каждый оператор может располагаться на нескольких строках (переводы строк, так же как и лишние пробелы игнорируются). При необходимости можно отправить в SQLite сразу несколько команд — тогда после каждой команды следует использовать символ завершения команды ';' (точка с запятой).

Текст в командах разбирается системой независимо от регистра, однако в SQL обычно принято писать ключевые слова прописными буквами.

При создании таблицы мы должны указать её имя, а также список столбцов в круглых скобках, через запятую. Для каждого столбца указывается имя и тип, и опционально ограничения. Наиболее простая форма:

```
CREATE TABLE имя_таблицы
    ( имя_столбца тип [ ограничения ...] [, имя_столбца тип [ ограничения ...] ...] );
```

Что такое ограничения, и какие типы бывают в SQL, мы разберем в следующем разделе. А пока наглядный пример (с разными типами и опциями):

```
CREATE TABLE IF NOT EXISTS example_table
    (id INTEGER PRIMARY KEY,
     name TEXT,
     timestamp INTEGER DEFAULT CURRENT_STAMP,
     income REAL,
     data BLOB);
```

Синтаксис создания индекса таков:

```
CREATE [ UNIQUE ] INDEX имя_индекса
    ON имя_таблицы ( имя_столбца [, имя_столбца ...] );
```

Имеющиеся индексы автоматически используются при запросах с условиями отбора по соответствующим столбцам. Без индексов процесс происходит медленнее.

Удаление таблицы (вместе с данными, если в неё что-то было записано) довольно просто:

```
DROP TABLE имя_таблицы;
```

Вставить данные в таблицу можно следующим образом:

```
INSERT INTO имя_таблицы [ ( имя_столбца [, имя_столбца...] ) ]
    VALUES ( значение [, значение ...] );
```

Первый список в круглых скобках — с именами столбцов, и он необязательный (см. пояснение ниже) — должен соответствовать второму списку со значениями для них. Например,

```
INSERT INTO example_table (name, income) VALUES ('Morning Flat Breakout', 1000);
```

Обратите внимание, строковые литералы помещаются в SQL в одинарные кавычки.

Если имена столбцов в INSERT-инструкции опущены, предполагается, что после ключевого слова VALUES указаны значения для всех столбцов таблицы, причем именно в том порядке, в котором они описаны в таблице.

Существуют и более сложные формы оператора, позволяющие, в частности, вставлять записи из других таблиц или результатов запросов.

Выборка записей по условию, с опциональным ограничением перечня возвращаемых полей (столбцов), выполняется командой SELECT.

```
SELECT имя_столбца [, имя_столбца ...] FROM имя_таблицы [ WHERE условие ];
```

Если нужно вернуть каждую подходящую запись целиком (все столбцы), используем нотацию со звездочкой:

```
SELECT * FROM имя_таблицы [ WHERE условие ];
```

Когда условия нет, система возвращает все записи таблицы.

В качестве условия можно подставить логическое выражение, включающее имена столбцов и различные операторы сравнения, а также встроенные функции SQL и результаты вложенного запроса SELECT (подобные запросы записывается в круглых скобках). Операторы сравнения включают:

- AND — логическое И;
- OR — логическое ИЛИ;
- IN — значение из списка;
- NOT IN — значение вне списка;
- BETWEEN — значение в диапазоне;
- LIKE — похожее по написанию на шаблон со специальными символами подстановки ('%', '_');
- EXISTS — проверка на непустоту результатов вложенного запроса.

Например, выборка имен записей с доходом не меньше 1000 и не старше одного года (с предварительным округлением до месяца):

```
SELECT name FROM example_table
   WHERE income >= 1000 AND timestamp > datetime('now', 'start of month', '-1 year');
```

Дополнительно выборку можно отсортировать по возрастанию или убыванию (ORDER BY), сгруппировать по признакам (GROUP BY) и отфильтровать группы (HAVING), а также ограничить в

ней количество записей (LIMIT, OFFSET). Для каждой группы можно вернуть значение какой-либо агрегатной функции, в частности, COUNT, SUM, MIN, MAX, AVG, посчитанной на всех записях группы.

```
SELECT [ DISTINCT ] имя_столбца [, имя_столбца ...] FROM имя_таблицы
   [ WHERE условие ]
   [ ORDER BY имя_столбца [ ASC | DESC ]
      [ LIMIT количество OFFSET смещение_от_начала ] ]
   [ GROUP BY имя_столбца [ HAVING условие ] ];
```

Необязательное ключевое слово DISTINCT позволяет убрать дубликаты (если они обнаружатся в результатах по текущим условиям отбора). Оно имеет смысл только в отсутствии группировки.

LIMIT даст воспроизводимые результаты только при наличии сортировки.

При необходимости выборку SELECT можно делать не из одной таблицы, а из нескольких, комбинируя их по требуемому сочетанию полей. Для этого используется ключевое словое JOIN.

```
SELECT [...] FROM имя_таблицы_1
   [ INNER | OUTER | CROSS ] JOIN имя_таблицы_2
   ON логическое_условие
```

или

```
SELECT [...] FROM имя_таблицы_1
   [ INNER | OUTER | CROSS ]opt JOIN имя_таблицы_2
   USING ( имя_общего_столбца [, имя_общего_столбца ...] )
```

В принципе, SQLite поддерживает три разновидности JOIN — INNER JOIN, OUTER JOIN и CROSS JOIN, — но подробности мы оставим для самостоятельного изучения: в книге общее представление о них можно будет составить из примеров.

С помощью JOIN можно, в частности, построить все сочетания записей одной таблицы с записями из другой или сопоставить сделки из таблицы сделок (допустим, под названием deals) со сделками этой же таблицы по принципу совпадения идентификаторов позиции, но чтобы направление сделок (вход в рынок/выход из рынка) было противоположным, что в результате даст виртуальную таблицу трейдов.

```
SELECT         // перечисляем столбцы таблицы результатов с алиасами (после 'as')
   d1.time as time_in, d1.position_id as position, d1.type as type, // таблица d1
   d1.volume as volume, d1.symbol as symbol, d1.price as price_in,
   d2.time as time_out, d2.price as price_out,                      // таблица d2
   d2.swap as swap, d2.profit as profit,
   d1.commission + d2.commission as commission                      // комбинация
   FROM deals d1 INNER JOIN deals d2      // d1 и d2 - алиасы одной таблицы deals
   ON d1.position_id = d2.position_id     // условие слияния по позиции
   WHERE d1.entry = 0 AND d2.entry = 1    // условие отбора "вход/выход"
```

Это SQL-запрос из справки по MQL5, где примеры JOIN имеются в описаниях функций *DatabaseExecute*, *DatabasePrepare*.

Основополагающее свойство SELECT в том, что он всегда возвращает результаты в вызывающую программу, в отличие от других запросов типа CREATE, INSERT и т.д. Правда, с версии SQLite 3.35 для операторов INSERT, UPDATE и DELETE также появилась возможность, при

необходимости, вернуть значения с помощью дополнительного ключевого слова RETURNING. Например,

```
INSERT INTO example_table (name, income) VALUES ('Morning Flat Breakout', 1000)
   RETURNING id;
```

В любом случае, в MQL5 результаты запроса доступы через группу *DatabaseColumn*-функций, *DatabaseRead*, *DatabaseReadBind*.

Кроме того, SELECT позволяет рассчитывать результаты выражений и возвращать их сами по себе или комбинировать с результатами из таблиц. Выражения могут включать большинство операторов, знакомых нам по выражениям MQL5, а также встроенные функции SQL. С их полным перечнем следует ознакомиться в документации SQLite. Вот, например, как можно узнать текущую версию сборки SQLite в вашем экземпляре терминала и редактора, что может быть важно для выяснения того, какие опции доступны.

```
SELECT sqlite_version();
```

Здесь выражение целиком состоит из одного вызова функции *sqlite_version*. По аналогии с выбором нескольких столбцов из таблицы, вы можете вычислить несколько выражений, указанных через запятую.

Среди функций имеется несколько востребованных статистических и математических.

Редактировать записи следует оператором UPDATE.

```
UPDATE имя_таблицы SET имя_столбца = значение [, имя_столбца = значение ...]
  WHERE условие;
```

Синтаксис команды удаления записей таков:

```
DELETE FROM имя_таблицы WHERE условие;
```

7.6.2 Структура (схема) таблиц: типы данных и ограничения

При описании полей таблицы требуется указать для них типы данных, однако понятие типа данных в SQLite сильно отличается от MQL5.

MQL5 является строго типизированным языком: каждая переменная или поле структуры всегда сохраняет тип данных согласно декларации. SQL же является слабо типизированным языком: те типы, которые мы укажем в описании таблицы — не более, чем рекомендация. При этом программе не запрещено записать в любую "ячейку" (поле в записи) значение произвольного типа, и "ячейка" изменит свой тип, что, в частности, можно будет обнаружить и встроенной MQL-функцией *DatabaseColumnType*.

Разумеется, на практике все, как правило, придерживаются правила "уважать" типы столбцов.

Второе существенное отличие в механизме типов SQL — наличие большого количества ключевых слов, описывающих типы, однако все эти слова сводятся, в конечном счете, к пяти классам хранения. Будучи упрощенной версией SQL, SQLite в большинстве случаев не делает различий между ключевыми словами одной группы (например, в описании строки с лимитом длины VARCHAR(80) этот лимит не контролируется, и описание эквивалентно классу хранения TEXT), поэтому более логично описывать тип именем группы. Конкретные типы оставлены только для совместимости с другими СУБД (но для нас это неважно).

В следующей таблице приведены типы MQL5 и соответствующие им "аффинности" (affinity, обобщающие признаки типов SQL).

Типы MQL5	Обобщенные типы SQL
NULL (не является типом в MQL5)	NULL (значение отсутствует)
bool, char, short, int, long, uchar, ushort, uint, ulong, datetime, color, enum	INTEGER
float, double	REAL
(вещественное число фиксированной точности, нет аналога в MQL5)	NUMERIC
string	TEXT
(произвольные "сырые" данные, аналог массива uchar[] или других)	BLOB (binary large object), NONE

При записи значения в базу SQL определяет его тип по нескольким правилам:

- отсутствие кавычек, десятичной точки или показателя степени дают INTEGER;
- наличие точки и экспоненты означают REAL;
- обрамление из одиночных или двойных кавычек сигнализирует о типе TEXT;
- значение NULL без кавычек соответствует классу NULL;
- литералы (константы) с двоичными данными записываются как шестнадцатеричная строка с префиксом 'x'.

Специальная функция SQL *typeof* позволяет проверить тип значения. Например, следующий запрос можно выполнить в редакторе MetaEditor.

```
SELECT typeof(100), typeof(10.0), typeof('100'), typeof(x'1000'), typeof(NULL);
```

Он выведет в таблицу результатов:

```
integer    |    real    |    text    |    blob    |    null
```

Проверять значения на NULL сравнением '=' нельзя (потому что результат тоже даст NULL), следует использовать специальный оператор NOT NULL.

SQLite налагает некоторые лимиты на хранимые данные: часть из них трудно достижима (и потому мы их здесь опустим), но другая может учитываться при проектировании программы. Так, максимальное количество столбцов в таблице равно 2000, а размер одной строки, BLOB-а и в целом одной записи не может превышать миллион байтов. Эта же величина выбрана пределом длины SQL-запроса.

Что касается даты и времени, SQL в принципе может хранить их в трех форматах, но лишь первый из них соответствует *datetime* в MQL5:

- INTEGER — количество секунд с начала 1970.01.01 (оно еще называется "эпохой Unix");
- REAL — количество дней (с долями) от 24 ноября 4714 года до нашей эры;

- TEXT — дата и время с точностью до миллисекунды в формате "YYYY-MM-DD HH:mm:SS.sss", опционально с часовым поясом, для чего добавляют суффикс "[±]HH:mm" со смещением от UTC.

Вещественный тип хранения даты (называемый также "днем по юлианскому календарю", для чего есть встроенная SQL-функция *julianday*) интересен тем, что позволяет хранить время с точностью до миллисекунд. В принципе, это можно делать и в виде строки формата 'YYYY-MM-DDTHH:mm:SS.sssZ', но такое хранение очень неэкономно. Пересчет "дня" в количество секунд с дробной частью, начиная с привычной нам даты 1970.01.01 00:00:00, производится по формуле: *julianday('now') - 2440587.5) * 86400.0*. Здесь 'now' обозначает текущее время UTC, но может быть заменено на другие значения, описанные в документации SQLite. Константа 2440587.5 как раз равна количеству дней "календаря" на указанную "нулевую" дату — точку отсчета "эпохи Unix".

Помимо типа каждое поле может иметь особую характеристику — одно или несколько ограничений (constraints) — они записываются специальными ключевыми словами после типа. Ограничение описывает, какие значения может принимать поле, и даже позволяет автоматизировать заполнение согласно предопределенному назначению поля.

Рассмотрим основные ограничения по порядку.

```
... DEFAULT выражение
```

При добавлении новой записи, если значение поля не указано, система автоматически проставит указанное здесь значение (константу) или вычислит выражение (функцию).

```
... CHECK ( логическое_выражение )
```

При добавлении новой записи система проверит, чтобы выражение, которое может содержать названия полей как переменные, было истинным. Если выражение ложно, запись не будет вставлена, а система вернет ошибку.

```
... UNIQUE
```

Система проверяет, чтобы во всех записях таблицы отличались значения данного поля. Попытка добавить запись со значением, которое уже есть, вызовет ошибку, и добавления не произойдет.

Для отслеживания уникальности система неявным образом создает индекс по указанному полю.

```
... PRIMARY KEY
```

Поле, помеченное данным признаком, используется системой для идентификации записей в таблице и ссылок на них из других таблиц (так образуются реляционные связи, дающие название рассматриваемым реляционным базам данных вроде SQLite). Очевидно, что данный признак также включает уникальный индекс.

Если в таблице нет поля типа INTEGER с признаком PRIMARY KEY, система автоматически неявным образом создает такой столбец под именем *rowid*. Если в вашей таблице есть целочисленное поле, объявленное первичным ключом, то оно также доступно и под алиасом *rowid*.

Если в таблицу добавляется запись, в которой *rowid* опущен или равен NULL, SQLite автоматически присвоит ей следующее целое число (64-битное, соответствующее *long* в MQL5), на единицу большее максимального *rowid* в таблице. Начальное значение — 1.

Обычно счетчик просто увеличивается каждый раз на 1, но если количество когда-либо вставленных (и, возможно, затем удаленных) в одну таблицу записей превысит *long*, счетчик перескочит на начало, и система будет пытаться найти свободные числа. Но такое маловероятно. Например, если писать в таблицу тики со средней скоростью 1 тик в миллисекунду, то переполнение случится через 292 миллиона лет.

Первичный ключ может быть только один, но он может состоять из нескольких столбцов — это делается с помощью иного синтаксиса, нежели ограничения — непосредственно в описании таблицы.

```
CREATE TABLE имя_таблицы (
   имя_столбца тип [ ограничения ]
   [, имя_столбца тип [ ограничения ] ...]
   , PRIMARY KEY ( имя_столбца [, имя_столбца ...] ) );
```

Но вернемся к ограничениям.

```
... AUTOINCREMENT
```

Данное ограничение может указываться только как дополнение PRIMARY KEY, гарантируя постоянное увеличение идентификаторов. Это означает, что любые прежние идентификаторы — даже те, что использовались у удаленных записей — не будут выбраны повторно. Однако данный механизм реализован в SQLite менее эффективно, чем простой PRIMARY KEY, в плане потребления вычислительных ресурсов, и поэтому не рекомендуется к применению.

```
... NOT NULL
```

Это ограничение запрещает добавлять в таблицу запись, в которой данное поле не заполнено. По умолчанию, когда ограничения нет, любое неуникальное поле можно опустить в добавляемой записи и ему будет присвоено значение NULL.

```
... CURRENT_TIME
... CURRENT_DATE
... CURRENT_TIMESTAMP
```

Данные инструкции позволяют автоматически заполнять поле временем (без даты), датой (без времени) или полным временем UTC на момент вставки записи (при условии, что SQL-оператор INSERT ничего не записывает в это поле явным образом, даже NULL). SQLite не умеет аналогичным автоматическим образом засекать время изменения записи — для этой цели придётся написать триггер (что выходит за рамки книги).

К сожалению, ограничения группы CURRENT_TIMESTAMP реализованы в SQLite с упущением: временная метка не проставляется, если поле имеет значение NULL. Это отличает SQLite от других движков SQL и от того, как сам SQLite обрабатывает NULL в полях первичных ключей. Получается, что для автоматического проставления метки нельзя записывать в базу весь объект целиком, а нужно явно указать все поля за исключением поля с датой и временем. Для решения проблемы нам потребуется альтернативный вариант с подстановкой SQL-функции STRFTIME('%s') в компилируемый запрос для в соответствующих столбцов.

7.6.3 Интеграция ООП (MQL5) и SQL: концепция ORM

Использование базы данных в MQL-программе подразумевает, что алгоритм поделен на 2 части: управляющая пишется на MQL5, а исполнительная — на SQL. В результате исходный код может начать представлять собой лоскутное одеяло и требовать внимания на поддержание

согласованности. Чтобы избежать этого, в объектно-ориентированных языках разработана концепция Object-Relational Mapping (ORM), то есть отображения объектов в записи реляционных таблиц и обратно.

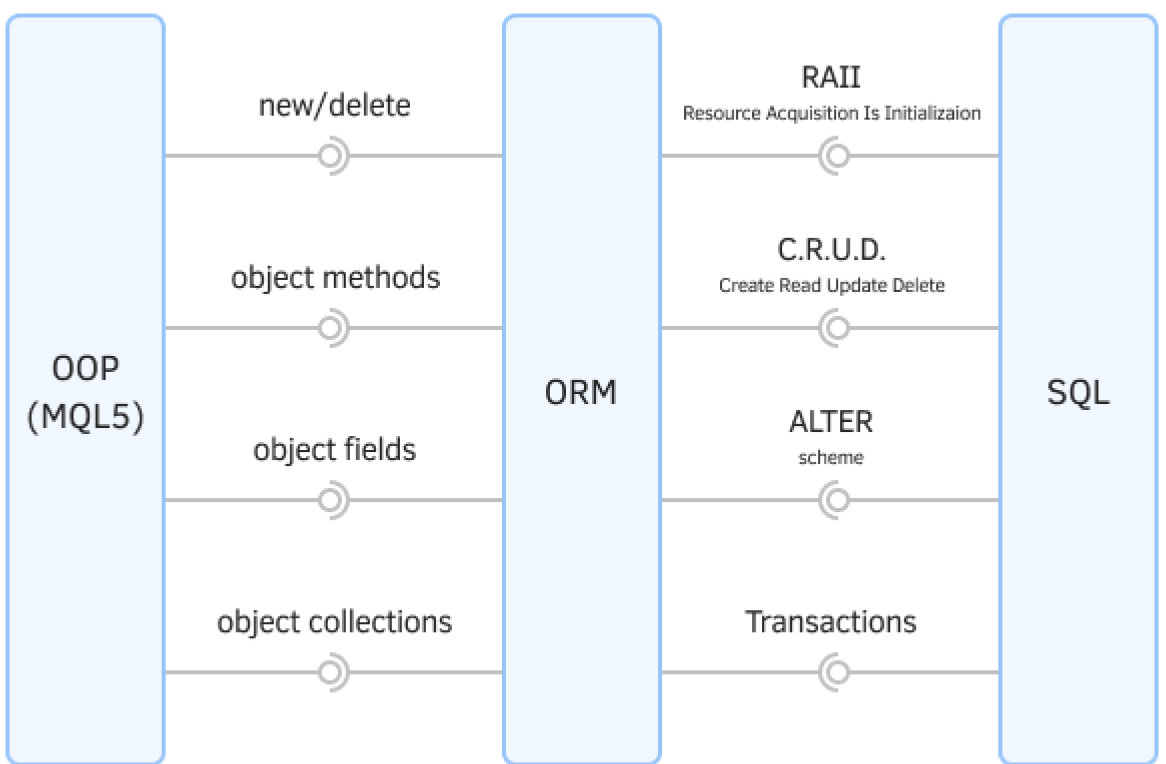
Суть подхода в том, чтобы инкапсулировать все действия на языке SQL в классах/структурах специальной прослойки. В результате прикладную часть программы можно писать на чистом ООП-языке (например, MQL5), не отвлекаясь более на нюансы SQL.

При наличии полноценной реализации ORM (в виде "черного ящика" с набором всех команд) у прикладного разработчика вообще появляется возможность не изучать SQL.

Кроме того ORM позволяет "незаметно" поменять "движок" СУБД при необходимости. Это не особо актуально для MQL5, потому что в него встроена только база SQLite, но некоторые разработчики предпочитают применять полновесные СУБД и подключают их к MetaTrader 5 с помощью импорта DLL-библиотек.

Наконец, использование объектов с конструкторами и деструкторами очень удобно для автоматического захвата и освобождения ресурсов. Мы рассматривали эту концепцию (RAII, Resource Acquisition Is Initialization) в разделе Управление дескрипторами файлов, однако, как мы увидим далее, работа с базой данных также строится на выделении и освобождении дескрипторов различного вида.

На следующей картинке схематично изображено взаимодействие разных слоев ПО при интеграции ООП и SQL в виде ORM.



Объектно-реляционное отображение ORM

В качестве бонуса объектная "обертка" (не только ORM, специфическая для БД) позволит автоматизировать подготовку и преобразование данных, а также проверить их на корректность в целях предотвращения некоторых ошибок.

В следующих разделах, в процессе знакомства со встроенными функциями для работы с базой, мы будем реализовывать примеры, постепенно формируя собственную простую прослойку ORM. Из-за некоторой специфики MQL5 наши классы не смогут обеспечить универсализм, покрывающий 100% задач, но окажутся полезными для многих проектов.

7.6.4 Создание, открытие и закрытие базы данных

Для создания и открытия баз данных предназначена пара функций: *DatabaseOpen* и *DatabaseClose*.

int DatabaseOpen(const string filename, uint flags)

Функция открывает или создаёт базу данных в файле с именем *filename*. Параметр может содержать не только имя, но и путь с вложенными папками относительно *MQL5/Files* (конкретного экземпляра терминала или в общей папке, см. флаги ниже). Расширение можно не указывать: при этом к имени по умолчанию добавляется ".sqlite".

Если в параметре *filename* указано значение NULL или пустая строка "", то база создается во временном файле, который будет автоматически удален после закрытия базы данных.

Если в параметре *filename* указана строка ":memory:", то база данных будет создана в памяти. Такая временная база будет автоматически удалена после закрытия.

Параметр *flags* содержит комбинацию флагов, описывающих дополнительные условия для создания или открытия базы, из перечисления ENUM_DATABASE_OPEN_FLAGS.

Идентификатор	Описание
DATABASE_OPEN_READONLY	Открыть только на чтение
DATABASE_OPEN_READWRITE	Открыть на чтение и запись
DATABASE_OPEN_CREATE	Создать файл на диске, если он не существует
DATABASE_OPEN_MEMORY	Создать базу данных в оперативной памяти
DATABASE_OPEN_COMMON	Файл находится в общей папке всех терминалов

Если в параметре *flags* не указан ни один из флагов DATABASE_OPEN_READONLY или DATABASE_OPEN_READWRITE, то будет использован флаг DATABASE_OPEN_READWRITE.

При успешном выполнении функция возвращает дескриптор базы данных, который затем используется в параметрах других функций для доступа к ней. В противном случае возвращается значение INVALID_HANDLE, а код ошибки можно узнать в *_LastError*.

void DatabaseClose(int database)

Функция *DatabaseClose* закрывает базу данных по её дескриптору, который был ранее получен из функции *DatabaseOpen*.

После вызова *DatabaseClose* все дескрипторы запросов, которые мы научимся создавать для открытой базы в следующих разделах, автоматически удаляются и становятся недействительными.

Функция ничего не возвращает, однако в случае передачи ей некорректного дескриптора установит _LastError в ERR_DATABASE_INVALID_HANDLE.

Начнем разрабатывать объектно-ориентированную обертку для баз данных в файле *DBSQLite.mqh*.

Класс *DBSQLite* обеспечит создание, открытие и закрытие баз данных. Позднее мы дополним его.

```
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;
   }
};
```

Обратите внимание, что база данных автоматически создается или открывается при создании объекта, и закрывается при его уничтожении.

С помощью данного класса напишем простой скрипт *DBinit.mq5*, который будет создавать или открывать указанную базу.

```
input string Database = "MQL5Book/DB/Example1";

void OnStart()
{
   DBSQLite db(Database);                       // создаем или открываем базу в конструкт
   PRTF(db.getHandle());                        // 65537 / ok
   PRTF(FileIsExist(Database + ".sqlite")); // true / ok
}                                               // база закрывается в деструкторе
```

После первого запуска, при настройках по умолчанию, мы должны получить новый файл *MQL5/Files/MQL5Book/DB/Example1.sqlite* — это подтверждается в коде проверкой на существование файла. При последующих запусках с тем же именем скрипт просто открывает базу и выводит в журнал действующий дескриптор (некое целое число).

7.6.5 Выполнение запросов без привязки к данным MQL5

Некоторые SQL-запросы представляют собой команды, которые достаточно отправить в "движок" как есть, и не требуют ни переменных входных данных, ни получения результатов. Например, если наша MQL-программа должна создать в базе таблицу, индекс или представление с определенной структурой и именем, мы можем прописать её константной строкой с оператором "CREATE ...". Кроме того, такие запросы удобно использовать для пакетной обработки записей или их комбинирования (слияния, вычисления агрегированных показателей, однотипной модификации). То есть одним запросом можно преобразовать данные таблиц целиком или заполнить на их основе другие таблицы. В последующих запросах можно будет анализировать эти результаты.

Во всех этих случаях важно лишь получить подтверждение успешности действия. Запросы такого типа выполняются с помощью функции *DatabaseExecute*.

bool DatabaseExecute(int database, const string sql)

Функция исполняет запрос в базе данных, указанной дескриптором *database*. Сам запрос передается в виде готовой строки *sql*.

Функция возвращает признак успеха (*true*) или ошибки (*false*).

Например, мы можем дополнить свой класс *DBSQLite* таким методом (дескриптор уже имеется внутри объекта).

```
class DBSQLite
{
   ...
   bool execute(const string sql)
   {
      return DatabaseExecute(handle, sql);
   }
};
```

Тогда скрипт, создающий новую таблицу (а при необходимости, предварительно, и саму базу данных), может выглядеть так (*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
   }
}
```

После выполнения скрипта попробуйте открыть указанную базу в MetaEditor и убедиться, что в ней имеется пустая таблица с единственным текстовым полем "msg". Но это можно сделать и программным способом (см. следующий раздел).

Интересно, что если мы запустим скрипт второй раз с теми же параметрами, то получим ошибку (хотя и некритическую, без принудительного закрытия программы).

```
database error, table table1 already exists
db.execute(StringFormat(CREATE TABLE %s (msg text),Table))=false / DATABASE_ERROR(560
```

Дело в том, что нельзя повторно создать уже существующую таблицу. Но SQL позволяет подавить эту ошибку и создать таблицу, только если её до сих пор не было, а в противном случае практически ничего не делать и вернуть признак успеха. Для этого достаточно добавить в запрос "IF NOT EXISTS" перед именем.

```
   db.execute(StringFormat("CREATE TABLE IF NOT EXISTS %s (msg text)", Table));
```

На практике таблицы требуются для хранения информации об объектах прикладной области: котировках, сделках, торговых сигналах. Поэтому желательно автоматизировать создание таблиц на основе описания объектов в MQL5. Как мы увидим далее, функции SQLite предоставляют возможность привязать результаты запросов к структурам MQL5 (но не классам). В связи с этим, в рамках ORM-обертки, разработаем механизм по генерации SQL-запроса "CREATE TABLE" по описанию *struct* конкретного вида в MQL5.

Для этого требуется зарегистрировать имена и типы полей структуры каким-либо образом в общем списке в момент компиляции, а затем, уже на стадии исполнения программы, по этому списку можно формировать SQL-запросы.

На стадии компиляции происходит разбор нескольких категорий сущностей MQL5, которыми можно воспользоваться для выявления типов и имен:

- макросы
- наследование
- шаблоны

Прежде всего, следует напомнить, что собираемые описания полей относятся к контексту конкретной структуры и их нельзя перемешивать, потому что в программе может оказаться много разных структур с потенциально совпадающими названиями и типами. Иными словами, информацию желательно аккумулировать в раздельных списках по каждому типу структуры. Для этого идеально подойдет шаблонный тип, параметром шаблона которого (S) будет выступать прикладная структура. Назовем шаблон *DBEntity*.

```
template<typename S>
struct DBEntity
{
   static string prototype[][3]; // 0 - тип, 1 - имя, 2 - ограничения
   ...
};

template<typename T>
static string DBEntity::prototype[][3];
```

Внутри шаблона — многомерный массив *prototype*, в который и будем записывать описание полей. Чтобы перехватить тип и имя прикладного поля потребуется объявить внутри *DBEntity* еще одну шаблонную структуру *DBField*: на этот раз её параметр T является типом самого поля. В конструкторе мы имеем информацию об этом типе (*typename(T)*), а также получаем название поля (и опционально, ограничение — о нем чуть ниже) в виде параметров.

```
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;
      }
   };
```

Поле *f* не используется, но оно нужно, потому что структуры не могут быть пустыми.

Допустим, что у нас есть прикладная структура *Data* (*DBmetaProgramming.mq5*).

```
struct Data
{
   long id;
   string name;
   datetime timestamp;
   double income;
};
```

Мы можем сделать её аналог, унаследованный от *DBEntity<DataDB>*, но с подмененными полями на основе *DBField*, идентичными исходному набору.

```
struct DataDB: public DBEntity<DataDB>
{
   DB_FIELD(long, id);
   DB_FIELD(string, name);
   DB_FIELD(datetime, timestamp);
   DB_FIELD(double, income);
} proto;
```

За счет подстановки названия структуры в параметр родительского шаблона, структура предоставляет программе информацию о собственных свойствах.

Обратите внимание на одномоментное определение переменной *proto* вместе с декларацией структуры. Это нужно, потому что в шаблонах каждый конкретный параметризованный тип компилируется только в случае, если в исходном коде создается хотя бы один объект такого типа. И для нас важно, чтобы создание этого прото-объекта происходило в самом начале запуска программы, в момент инициализации глобальных переменных.

Под идентификатором DB_FIELD скрывается макрос:

```
#define DB_FIELD(T,N) struct T##_##N: DBField<T> { T##_##N() : DBField<T>(#N) { } } \
   _##T##_##N;
```

Вот как он раскрывается для отдельного поля:

```
   struct Type_Name: DBField<Type>
   {
      Type_Name() : DBField<Type>(Name) { }
   } _Type_Name;
```

Здесь структура также не только определяется, но и сразу же создается: фактически она подменяет собой оригинальное поле.

Поскольку структура *DBField* содержит единственную переменную *f* нужного типа, размеры и внутреннее двоичное представление *Data* и *DataDB* идентично. В этом легко убедиться, запустив скрипт *DBmetaProgramming.mq5*.

```
void OnStart()
{
   PRTF(sizeof(Data));
   PRTF(sizeof(DataDB));
   ArrayPrint(DataDB::prototype);
}
```

Он выводит в журнал:

```
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"    ""
```

Правда, для доступа к полям в *DataDB* нужно писать нечто неудобное: *data._long_id.f*, *data._string_name.f*, *data._datetime_timestamp.f*, *data._double_income.f*.

Мы не будем так делать не только и не столько из-за неудобства, а потому что данный способ конструирования мета-структур не совместим с принципами привязки данных к запросам SQL. В следующих разделах мы приступим к изучению *Database*-функций, позволяющих получать записи таблиц и результатов SQL-запросов в структуры MQL5, однако там разрешено использовать только простые структуры без наследования и статических членов объектных типов. Поэтому требуется слегка изменить принцип выявления мета-информации.

Нам придется оставлять исходные типы структур неизменными, а описание для базы данных фактически повторять, следя за тем, чтобы не было разночтений (опечаток). Это не очень удобно, но иного способа в данный момент нет.

Мы перенесем декларацию экземпляров *DBEntity* и *DBField* за пределы прикладных структур. При этом макрос DB_FIELD получит дополнительный параметр (S), в котором нужно будет передать тип прикладной структуры (ранее он неявно брался за счет объявления внутри самой структуры).

```
#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;
```

Поскольку у столбцов таблицы могут быть ограничения, их также потребуется при необходимости передавать в конструктор *DBField*. Для этой цели добавим пару макросов с соответствующими параметрами (в принципе, ограничений у одного столбца может быть несколько, но обычно, не более двух).

```
#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;
```

Все три макроса, как и дальнейшие наработки, попадают в заголовочный файл *DBSQLite.mqh*.

Важно отметить, что данная "самодельная" привязка объектов к таблице востребована только для ввода данных в базу, потому что чтение данных из таблицы в объект реализовано в MQL5 с помощью функции *DatabaseReadBind*.

Реализацию *DBField* также усовершенствуем. Напомним, что типы MQL5 не соответствуют один в один классам хранения SQL, в связи с чем нужно выполнить преобразование при заполнении элемента *prototype[n][0]*. Этим занимается статический метод *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;
   }
};
```

Использованные здесь текстовые константы обобщенных типов SQL вынесены в отдельное пространство имен: потребность в них может возникнуть в разных местах MQL-программ, и следует гарантировать отсутствие конфликтов имен.

```
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";
}
```

Заготовки возможных ограничений также описаны в своей группе для удобства (как подсказки).

```
namespace DB_CONSTRAINT
{
   const string PRIMARY_KEY = "PRIMARY KEY";
   const string UNIQUE = "UNIQUE";
   const string NOT_NULL = "NOT NULL";
   const string CHECK = "CHECK (%s)"; // требует выражения
   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)"; // требует выражения (константы, функции)
}
```

Поскольку среди ограничений есть такие, которые требуют параметров (места под них помечены привычным форматным модификатором '%s'), добавим проверку их наличия — вот окончательный вид конструктора *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       // обходим ошибку STRING_SMALL_LEN(5035)
         && StringFind(constraints, "%") >= 0)
      {
         Print("Constraint requires an expression (skipped): ", constraints);
      }
      else
      {
         prototype[n][2] = constraints;
      }
   }
```

Благодаря тому, что комбинация макросов и вспомогательных объектов *DBEntity<S>* и *DBField<T>* заполняет массив прототипов, в классе *DBSQLite* появляется возможность реализовать автоматическую генерацию SQL-запроса на создание таблицы структур.

Метод *createTable* шаблонизирован типом прикладной структуры и содержит заготовку запроса ("CREATE TABLE %s %s (%s);"). Первым аргументом для неё является опциональная инструкция "IF NOT EXISTS", вторым — имя таблицы, которое по умолчанию берется как тип параметра шаблона *typename(S)*, но при необходимости его можно заменить чем-то еще с помощью входного параметра name (если он не равен NULL). Наконец третий аргумент в скобках — это список столбцов таблицы: он формируется вспомогательным методом *columns* на основе массива *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;
      }
      // попытка создать уже существующую таблицу даст ошибку,
      // если не использовать 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;
   }
};
```

Для каждого столбца описание составляется из имени, типа и необязательного ограничения. Дополнительно существует возможность передать общее ограничение на таблицу (*table_constraints*).

Перед тем как отправить сформированный SQL-запрос в функцию *DatabaseExecute*, метод *createTable* производит отладочный вывод текста запроса в журнал (весь такой вывод в классах ORM можно централизованно отключить подменой макроса PRTF).

Теперь все готово для написания тестового скрипта *DBcreateTableFromStruct.mq5*, который по декларации структуры создал бы соответствующую таблицу в SQLite. Во входном параметре зададим только имя базы, а имя таблицы программа выберет сама по типу структуры.

```
#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);
```

В главной функции *OnStart* создаем таблицу вызовом *createTable* с параметрами по умолчанию. Если не хотим получить признак ошибки при повторных попытках создания, нужно передать *true* первым параметром (*db.createTable<Struct>(true)*).

```
void OnStart()
{
   DBSQLite db(Database);
   if(db.isOpen())
   {
      PRTF(db.createTable<Struct>());
      PRTF(db.hasTable(typename(Struct)));
   }
}
```

Метод *hasTable* проверяет наличие таблицы в базе по её (таблицы) имени. Реализацию этого метода мы покажем в следующем разделе, а пока запустим скрипт. После первого запуска создание таблицы пройдет успешно, и в журнале можно увидеть SQL-запрос (он отображается с переводами строк, как мы его формировали в коде).

```
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
```

Второй запуск вернет ошибку из вызова *DatabaseExecute*, потому что данная таблица уже существует, о чем дополнительно говорит и результат *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 Проверка существования таблицы в базе данных

Встроенная функция *DatabaseTableExists* позволяет проверить наличие таблицы по её имени.

bool DatabaseTableExists(int database, const string table)

Дескриптор базы и имя таблицы задаются в параметрах.

Результат вызова функции равен *true*, если таблица существует.

Дополним класс *DBSQLite* соответствующим методом *hasTable*.

```
class DBSQLite
{
   ...
   bool hasTable(const string table) const
   {
      return DatabaseTableExists(handle, table);
   }
```

В скрипте *DBcreateTable.mq5* убедимся в появлении таблицы.

```
void OnStart()
{
   DBSQLite db(Database);
   if(db.isOpen())
   {
      PRTF(db.execute(StringFormat("CREATE TABLE %s (msg text)", Table)));
      PRTF(db.hasTable(Table));
   }
}
```

Опять же, не смущаемся потенциальной возможности получить ошибку при попытке повторного создания. Это никак не влияет на наличие таблицы.

```
database error, table table1 already exists
db.execute(StringFormat(CREATE TABLE %s (msg text),Table))=false / DATABASE_ERROR(560
db.hasTable(Table)=true / ok
```

Поскольку мы пишем универсальный вспомогательный класс *DBSQLite*, предусмотрим в нем механизм удаления таблиц. Напомним, что в SQL для этой цели есть команда DROP.

```
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;
   }
```

Перед выполнением запроса мы проверяем наличие таблицы и сразу выходим, если её нет.

После выполнения запроса мы дополнительно проверяем, удалилась ли таблица, повторным вызовом *DatabaseTableExists*. Поскольку отсутствие таблицы будет помечено кодом ошибки DATABASE_NO_MORE_DATA, а для данного метода это ожидаемый результат, мы очищаем код ошибки с помощью *ResetLastErrorOnCondition*.

В принципе, более эффективно использовать возможности самого SQL для исключения попытки удаления несуществующей таблицы: достаточно добавить в запрос фразу "IF EXISTS". Поэтому окончательный вариант метода *deleteTable* упрощается:

```
   bool deleteTable(const string name) const
   {
      const static string query = "DROP TABLE IF EXISTS '%s';";
      return DatabaseExecute(handle, StringFormat(query, name));
   }
```

Вы можете попробовать написать проверочный скрипт для удаления таблицы самостоятельно, но будьте осторожны, чтобы в зависимости от входных переменных по ошибке не удалить какую-нибудь рабочую таблицу. Таблицы удаляются сразу со всеми данными, без запросов подтверждения и без возможности восстановления. Для важных проектов сохраняйте бэкапы баз.

7.6.7 Подготовка запросов с привязкой: DatabasePrepare

Во многих случаях в SQL-запросы необходимо встроить параметры. В принципе, поскольку SQL-запрос представляет собой "изначально" строку, отвечающую специальному синтаксису, её можно сформировать обычным вызовом *StringFormat* или конкатенацией, дополнив в нужных местах значениями параметров. Данный прием уже использовался нами в запросах на создание таблицы ("CREATE TABLE %s '%s' (%s);"), но здесь только часть параметров содержала данные (список значений подставлялся вместо %s внутри круглых скобок), а остальные представляли

собой опцию и имя таблицы. В данном разделе речь пойдет исключительно о подстановке данных в запрос. Делать это "родным" для SQL способом важно по нескольким причинам.

Прежде всего, SQL-запрос лишь передается "движку" SQLite в виде строки, а там разбирается на компоненты, проверяется на корректность и неким образом "компилируется" (конечно, это не компилятор MQL5). Затем откомпилированный запрос выполняется базой. Именно поэтому мы взяли слово "изначально" в кавычки.

Когда один и тот же запрос нужно выполнить с различными параметрами (например, вставить в таблицу много записей — а мы к этой задаче медленно приближаемся), отдельная компиляция и проверка запроса для каждой записи — довольно неэффективна. Правильнее откомпилировать запрос один раз, а затем выполнять массовым образом, просто подставляя разные значения.

Данная операция компиляции называется подготовкой запроса и выполняется функцией *DatabasePrepare*.

Подготовленные запросы имеют и еще одно предназначение — с их помощью "движок" SQLite возвращает результаты выполнения запросов в код MQL5 (подробнее об этом рассказано в разделах Выполнение подготовленных запросов и Раздельное чтение полей записи результатов запроса).

Последний, не менее важный нюанс, связанный с параметризованными запросами, заключается в том, что они защищают вашу программу от потенциальных хакерских атак под названием "внедрение SQL" (SQL injection). Это, в первую очередь, критично для баз данных публичных сайтов, где вводимая пользователями информация записывается в базу путем встраивания в SQL-запросы: если в этом случае применить простую форматную подстановку '%s', пользователь сможет ввести вместо предполагаемых данных некую длинную строку с дополнительными командами SQL, и она станет частью исходного SQL-запроса, исказив его смысл. Если же SQL-запрос скомпилирован, его не удастся изменить входными данными: они всегда обрабатываются как данные.

Хотя MQL-программа не является серверной, все же и она может сохранять в базе информацию, получаемую от пользователя.

int DatabasePrepare(int database, const string sql, ...)

Функция *DatabasePrepare* создает в указанной базе данных дескриптор для запроса в строке *sql*. База данных *database* должна быть заранее открыта функцией *DatabaseOpen*.

Места расположения параметров запроса указываются в строке *sql* с помощью фрагментов '?1', '?2', '?3', и так далее. Нумерация означает индекс параметра, используемый в будущем, при назначении ему входной величины, в *DatabaseBind*-функциях. Номера в строке *sql* не обязаны идти по порядку и могут повторяться, если один и тот же параметр нужно вставить в разные места запроса.

> Внимание! Индексация в подстановочных фрагментах '?n' начинается с 1, в то время как в *DatabaseBind*-функциях — с 0. Например, параметр '?1' в тексте запроса получит значение при вызове *DatabaseBind* с индексом 0, параметр '?2' — по индексу 1, и так далее. Такое постоянное смещение на 1 сохраняется даже в том случае, если в нумерации параметров '?n' есть пропуски (случайные или намеренные).

Если все параметры планируется привязывать строго по порядку, можно применить сокращенную запись: на месте каждого параметра просто указать символ '?' без номера: в этом случае параметры автоматически нумеруются. Любой параметр '?' без номера получает номер на 1

больше максимального из прочитанных левее параметров (с явно указанными номерами или рассчитанные по такому же принципу, а самый первый получит номер 1, то есть '?1').

Таким образом, запрос:

```
SELECT * FROM table WHERE risk > ?1 AND signal = ?2
```

эквивалентен:

```
SELECT * FROM table WHERE risk > ? AND signal = ?
```

Если часть параметров постоянна или запрос подготавливается для однократного исполнения с целью получить результат, значения параметров можно передать непосредственно в функцию *DatabasePrepare*, списком через запятую, вместо многоточия (также как в *Print* или *Comment*).

> Параметры запроса разрешено использовать только для задания значений в столбцах таблицы (при записи, изменении или в условиях отбора). Названия таблиц, столбцов, опции, ключевые слова SQL — нельзя передавать через параметры '?'/'?n'.

Сама функция *DatabasePrepare* не выполняет запрос. Возвращаемый из неё дескриптор затем должен передаваться в вызовы функций *DatabaseRead* или *DatabaseReadBind* — именно они выполняют запрос и делают доступным для чтения результат (это может быть одна запись или много). Разумеется, если в запросе есть заместители параметров ('?' или '?n'), и значения для них не были указаны в *DatabasePrepare*, перед выполнением запроса требуется осуществить привязку параметров и данных с помощью соответствующих *DatabaseBind*-функций.

Если какому-либо параметру так и не было назначено значение, во время выполнения запроса вместо него подставляется NULL.

В случае ошибки функция *DatabasePrepare* вернет INVALID_HANDLE.

Пример использования *DatabasePrepare* мы представим в следующих разделах, после изучения других функций, связанных с подготовленными запросами.

7.6.8 Удаление и сброс подготовленных запросов

Поскольку подготовленные запросы могут выполняться многократно, в цикле для различных значений параметров, на каждой итерации требуется сбрасывать запрос в начальное состояние. Для этого предназначена функция *DatabaseReset*. Но её не имеет смысла вызывать, если подготовленный запрос выполняется однократно.

bool DatabaseReset(int request)

Функция приводит внутренние откомпилированные структуры запроса в начальное состояние, как после вызова *DatabasePrepare*. Однако *DatabaseReset* не компилирует запрос заново и потому выполняется очень быстро.

Также важно, что функция не сбрасывает уже установленные привязки данных в запросе, если они были сделаны. Таким образом, при необходимости можно менять значение только одного или малого числа параметров — достаточно после вызова *DatabaseReset* вызывать *DatabaseBind*-функции только для изменившихся параметров.

> На момент написания книги MQL5 API не предоставляло функцию для сброса привязки данных, аналог функции *sqlite_clear_bindings* в стандартном дистрибутиве SQLite.

В параметре *request* следует указать действующий дескриптор запроса, полученный ранее из *DatabasePrepare*. Если же передать дескриптор запроса, который был перед этим удален с помощью *DatabaseFinalize* (см. ниже), это приведёт к ошибке.

Функция возвращает признак успеха (*true*) или ошибки (*false*).

Общий принцип работы с повторяющимися запросами показан в следующем псевдо-коде. Часть из примененных функций — *DatabaseBind*, *DatabaseRead* — описана в следующих разделах и будет "упакована" в классы ORM.

```
struct Data                                  // пример структуры
{
   long count;
   double value;
   string comment;
};
Data data[];
...                                          // получаем массив данных
int r =
     DatabasePrepare(db, "INSERT... (?, ?, ?)")); // компилируем запрос с параметрами
for(int i = 0; i < ArraySize(data); ++i)     // цикл по данным
{
   DatabaseBind(r, 0, data[i].count);        // делаем привязку данных к парамет
   DatabaseBind(r, 1, data[i].value);
   DatabaseBind(r, 2, data[i].comment);
   DatabaseRead(r);                          // выполняем запрос
   ...                                       // анализ или сохранение результатс
   DatabaseReset(r);                         // на каждой итерации начальное сос
}
DatabaseFinalize(r);
```

После того как необходимость в подготовленном запросе отпала, следует освободить занимаемые им ресурсы компьютера с помощью *DatabaseFinalize*.

void DatabaseFinalize(int request)

Функция удаляет запрос с указанным дескриптором, созданный в *DatabasePrepare*.

В случае передачи некорректного дескриптора функция выставит в *_LastError* ошибку ERR_DATABASE_INVALID_HANDLE.

При закрытии базы с помощью *DatabaseClose* все дескрипторы запросов, созданные для неё, автоматически удаляются и становятся недействительными.

Дополним наш ORM-слой (*DBSQLite.mqh*) новым классом *DBQuery* для работы с подготовленными запросами. Пока в нем будет лишь функционал инициализации и деинициализации, присущий концепции RAII, но скоро мы его расширим.

```
class DBQuery
{
protected:
   const string sql;  // запрос
   const int db;      // дескриптор базы данных (аргумент конструктора)
   const int handle;  // дескриптор подготовленного запроса

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);
   }
   ...
};
```

В классе *DBSQLite* инициируем подготовку запроса в методе *prepare* за счет создания экземпляра *DBQuery*. Все объекты запросов будем сохранять во внутреннем массиве *queries* в виде автоуказателей, что позволяет вызывающему коду не следить за их явным удалением.

```
class DBSQLite
{
   ...
protected:
   AutoPtr<DBQuery> queries[];
public:
   DBQuery *prepare(const string sql)
   {
      return PUSH(queries, new DBQuery(handle, sql));
   }
   ...
};
```

7.6.9 Привязка данных к параметрам запроса:DatabaseBind/Array

После того как SQL-запрос был откомпилирован функцией *DatabasePrepare*, можно использовать полученный дескриптор запроса для привязки данных к параметрам запроса, для чего

предназначены функции *DatabaseBind* и *DatabaseBindArray*. Обе функции можно вызывать не только сразу после создания запроса в *DatabasePrepare*, но и после сброса запроса в начальное состояние с помощью *DatabaseReset* (если запрос выполняется много раз в цикле).

Этап привязки данных требуется не всегда, поскольку подготовленные запросы могут и не иметь параметров. Как правило, такая ситуация возникает, когда запрос возвращает данные из SQL в MQL5, в связи с чем требуется дескриптор запроса: о том, как читать результаты запросов по их дескриптору, рассказано в разделах о функциях *DatabaseRead/DatabaseReadBind* и *DatabaseColumn*-функциях.

bool DatabaseBind(int request, int index, T value)

Функция *DatabaseBind* устанавливает для запроса с дескриптором *request* значение параметра с индексом *index*. По умолчанию нумерация начинается с 0, если параметры в запросе помечены подстановочными символами '?' (без номера). Однако, параметры могут обозначаться в строке запроса и с номером (?1, '?5', ?21): в таком случае фактические индексы, которые следует передавать в функцию, должны быть на 1 меньше, чем соответствующий номер в строке. Дело в том, что в строке запроса нумерация ведется с 1.

Например, для следующего запроса требуется задать один параметр (индекс 0):

```
int r = DatabasePrepare(db, "SELECT * FROM table WHERE id=?");
DatabaseBind(r, 0, 1234);
```

Если бы в строке запроса использовалась подстановка "... id=?10", потребовалось бы вызвать *DatabaseBind* с индексом 9.

Значение *value* в прототипе *DatabaseBind* может быть любого простого типа или строкой. Если параметру требуется сопоставить данные составного типа (структуры) или произвольные двоичные данные, которые можно представить в виде массива байтов, используйте функцию *DatabaseBindArray*.

Функция возвращает *true* в случае успеха, а иначе — *false*.

bool DatabaseBindArray(int request, int index, T &array[])

Функция *DatabaseBindArray* устанавливает для запроса с дескриптором *request* значение параметра с индексом *index* как массив простого типа или простых структур (включая строки). Эта функция позволяет записывать в базу данных BLOB и NULL (отсутствие значения, которое считается в SQL самостоятельным типом и не равно 0).

Вернемся к классу *DBQuery* в файле *DBSQLite.mqh* и поддержим в нем привязку данных.

```
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 подойдет для переноса в базу любого файла в неизменном виде, например, если предварительно прочитать его в байтовый массив с помощью функции *FileLoad*.

Необходимость явной привязки значения NULL не так очевидна. Дело в том, что при вставке новых записей в базу вызывающая программа обычно передает только известные ей поля, а все недостающие (если они не помечены ограничением NOT NULL или не имеют иного значения DEFAULT в описании таблицы), "движок" автоматически оставит равными NULL. Однако при использовании подхода ORM бывает удобно записать в базу объект целиком, включая и поле с уникальным первичным ключом (PRIMARY KEY). В новом объекте этого идентификатора еще нет, так как его проставляет сама база при первой записи объекта, поэтому важно привязать это поле в новом объекте к значению NULL.

7.6.10 Выполнение подготовленных запросов: DatabaseRead/Bind

Выполнение подготовленных запросов производится с помощью функций *DatabaseRead* и *DatabaseReadBind*. Первая из них извлекает результаты из базы таким образом, что впоследствии можно считывать отдельные поля из каждой записи, поочередно получаемой в ответ, а вторая извлекает каждую подходящую запись целиком, в виде структуры.

bool DatabaseRead(int request)

При первом вызове после *DatabasePrepare* или *DatabaseReset* функция *DatabaseRead* выполняет запрос и устанавливает внутренний указатель результатов запроса на первую полученную запись (если запрос предполагает возврат записей). Прочитать значения полей записи, то есть указанных в запросе столбцов, позволяют *DatabaseColumn*-функции.

При последующих вызовах функция *DatabaseRead* осуществляет переход к следующей записи в результатах запроса, пока не будет достигнут их конец.

Функция возвращает *true* при успешном выполнении. Значение *false* используется как индикатор ошибки (например, база может быть заблокирована или занята), а также при штатном достижении конца результатов, поэтому следует анализировать код в *_LastError*. В частности, значение ERR_DATABASE_NO_MORE_DATA (5126) указывает на то, что результаты закончились.

> Внимание! В тех случаях, когда *DatabaseRead* используется для выполнения запросов, которые не возвращают данных, такие как INSERT, UPDATE и т.д., функция сразу возвращает *false* и устанавливает код ошибки ERR_DATABASE_NO_MORE_DATA, если запрос выполнился успешно.

Обычная схема использования функции иллюстрируется следующим псевдо-кодом (группа *DatabaseColumn*-функций для различных типов представлена в следующем разделе).

```
int r = DatabasePrepare(db, "SELECT... WHERE...?",
   param));                          // компилируем запрос (опционально с параметра
while(DatabaseRead(r))               // выполнение запроса (на первой итерации)
{                                    //    и цикл по записям результата
   int count;
   DatabaseColumnInteger(r, 0, count); // чтение одного поля из текущей записи
   double number;
   DatabaseColumnDouble(r, 1, number); // чтение другого поля из текущей записи
   ...                               // типы и кол-во колонок в записи определяет п
                                     // обрабатываем полученные значения count, num
}                                    // цикл прерывается по достижению конца резуль
DatabaseFinalize(r);
```

Обратите внимание: поскольку запрос (чтение данных по условиям) фактически выполняется только однажды (на самой первой итерации), не нужно вызывать *DatabaseReset*, как мы это делали при записи меняющихся данных. Однако, если мы захотим еще раз выполнить запрос и "пройтись" по новым результатам, вызов *DatabaseReset* был бы необходим.

bool DatabaseReadBind(int request, void &object)

Функция *DatabaseReadBind* работает по аналогичному принципу с *DatabaseRead*: первый вызов исполняет SQL-запрос и, в случае успеха (наличия подходящих данных в результате), заполняет переданную по ссылке структуру *object* полями первой записи; последующие вызовы продолжают перемещение внутреннего указателя по записям в результатах запроса, заполняя структуру данными очередной записи.

Структура должна иметь в качестве членов только числовые типы и/или строки (массивы не разрешены), не может быть наследником или содержать статические члены объектных типов.

Количество полей в структуре *object* не должно превышать количество столбцов в результатах запроса: в противном случае получим ошибку. Количество столбцов можно узнать динамически с помощью функции *DatabaseColumnsCount*, однако вызывающая программа, как правило, должна заранее "знать" ожидаемую конфигурацию данных согласно исходному запросу.

Если количество полей в структуре меньше количества полей в записи, будет произведено частичное чтение. Оставшиеся данные можно получить с помощью соответствующих *DatabaseColumn*-функций.

Предполагается, что типы полей структуры совпадают с типами данных в столбцах результата. Иначе будет произведена автоматическая неявная конвертация, которая может привести к неожиданным последствиям (например, строка, прочитанная в числовое поле, даст 0).

В простейшем случае, когда мы рассчитываем некую общую величину по записям базы, например, вызвав агрегатную функцию вроде *SUM(столбец)*, *COUNT(столбец)* или *AVERAGE(столбец)*, результатом запроса будет единственная запись с единственным полем.

```
SELECT SUM(swap) FROM trades;
```

Поскольку чтение результатов связано *DatabaseColumn*-функциями, мы отложим разработку примера до следующего раздела, где они представлены.

7.6.11 Раздельное чтение полей: DatabaseColumn-функции

В результате выполнения запроса функциями *DatabaseRead* или *DatabaseReadBind* программа получает возможность пролистывать записи, отобранные по заданным условиям. На каждой итерации во внутренних структурах "движка" SQLite выделена одна конкретная запись, поля (столбцы) которой доступны через группу *DatabaseColumn*-функций.

int DatabaseColumnsCount(int request)

По дескриптору запроса функция возвращает количество полей (столбцов) в результатах запроса.

В случае ошибки получим -1.

Узнать количество полей запроса, созданного в *DatabasePrepare*, можно еще до вызова функции *DatabaseRead*. Для остальных *DatabaseColumn*-функций предварительно требуется вызвать *DatabaseRead* (хотя бы один раз).

Для каждого поля в результатах запроса программа может по его порядковому номеру узнать название (*DatabaseColumnName*), тип (*DatabaseColumnType*), размер (*DatabaseColumnSize*), а также значение соответствующего типа (под каждый тип выделена своя функция).

bool DatabaseColumnName(int request, int column, string &name)

Функция заполняет переданный по ссылке строковый параметр (*name*) названием указанного по номеру столбца (*column*) в результатах запроса (*request*).

Нумерация полей начинается с 0 и не может превышать значение *DatabaseColumnsCount() - 1*. Это касается не только данной, но и всех остальных функций раздела.

Функция возвращает *true* в случае успеха или *false* в случае ошибки.

ENUM_DATABASE_FIELD_TYPE DatabaseColumnType(int request, int column)

Функция *DatabaseColumnType* возвращает тип значения в указанном столбце в текущей записи результатов запроса. Возможные типы собраны в перечислении ENUM_DATABASE_FIELD_TYPE.

Идентификатор	Описание
DATABASE_FIELD_TYPE_INVALID	Ошибка получения типа, код ошибки в _LastError
DATABASE_FIELD_TYPE_INTEGER	Целое число
DATABASE_FIELD_TYPE_FLOAT	Вещественное число
DATABASE_FIELD_TYPE_TEXT	Строка
DATABASE_FIELD_TYPE_BLOB	Двоичные данные
DATABASE_FIELD_TYPE_NULL	Пустота (специальный тип NULL)

Более подробно про типы SQL и их соответствие типам MQL5 было рассказано в разделе Структура (схема) таблицы: типы данных и ограничения.

int DatabaseColumnSize(int request, int column)

Функция возвращает размер значения в байтах для поля под индексом *column* в текущей записи результатов запроса *request*. Например, целочисленные значения могут быть представлены различным количеством байтов (мы это знаем и по типам MQL5, в частности, *short/int/long*).

Следующая группа функций позволяет получить само значение конкретного типа из соответствующего поля записи. Для чтения значений из следующей записи нужно снова вызвать *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[])

Все функции при успешном выполнении возвращают *true* и помещают значение поля в приемную переменную *value*. Особый случай представляет только функция *DatabaseColumnBlob*, которой в качестве выходной переменной передается массив произвольного простого типа или простых структур. Указав массив *uchar[]*, как наиболее универсальный вариант, вы можете прочитать байтовое представление любого значения (включая двоичные файлы, помечаемые типом DATABASE_FIELD_TYPE_BLOB).

> Движок SQLite не проверяет, чтобы для столбца вызывалась функция, соответствующая его типу. Если типы по недосмотру или намеренно отличаются, система автоматически неявным образом сконвертирует значение поля под тип приемной переменной.

Теперь, после знакомства с большинством *Database*-функций, мы можем завершить разработку набора SQL-классов в файле *DBSQLite.mqh* и обратиться к практическим примерам.

7.6.12 Примеры CRUD-операций в SQLite через объекты ORM

Мы изучили все функции, необходимые для реализации полного жизненного цикла информации в базе данных, то есть CRUD (Create, Read, Update, Delete). Но прежде чем приступать к практике, нужно завершить прослойку ORM.

Из нескольких предыдущих разделов нам уже ясно, что единицей работы с базой данных является запись: это может быть запись в таблице базы или элемент в результатах выполнения запроса. Для чтения одной записи на уровне ORM введем класс DBRow. Каждая запись порождается SQL-запросом, поэтому в конструктор передается его дескриптор.

Как мы знаем, запись может состоять из нескольких столбцов, количество и типы которых позволяют узнать *DatabaseColumn-функции*. Для экспозиции этой информации в MQL-программу, использующую *DBRow*, мы зарезервировали соответствующие переменные: *columns* и массив структур *DBRowColumn* (в последней — три поля для хранения имени, типа и размера столбца).

Кроме того, объекты *DBRow* могут при необходимости кэшировать в себе значения, полученные из базы. Для этой цели применен массив *data* типа *MqlParam*. Поскольку мы заранее не знаем, значения какого типа окажутся в конкретной колонке, используем *MqlParam* как разновидность универсального типа *Variant*, доступного в других средах программирования.

```
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;
   }
   ...
};
```

В переменной *cursor* отслеживается номер текущей записи из результатов запроса. Пока запрос не выполнен, *cursor* равен -1.

За выполнение запроса отвечает виртуальный метод *DBread*, вызывающий *DatabaseRead*.

```
protected:
   virtual bool DBread()
   {
      return PRTF(DatabaseRead(query));
   }
```

Зачем нам потребовался виртуальный метод, мы раскроем чуть позже. Публичный метод *next*, в котором используется *DBread*, обеспечивает "пролистывание" записей результата и выглядит следующим образом.

```
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]; // перегрузка operator[](int)
            }
         }
         ++cursor;
      }
      return success;
   }
```

Если обращение к запросу выполняется первый раз, мы выделяем память и считываем информацию о столбцах. Если было запрошено кэширование, дополнительно заполняем массив *data*. Для этого для каждого столбца вызывается перегруженный оператор '[]'. В нем, в зависимости от типа значения, мы вызываем соответствующую *DatabaseColumn*-функцию и помещаем полученное значение в то или иное поле структуры *MqlParam*.

```
virtual MqlParam operator[](const int i = 0) const
{
   MqlParam param = {};
   if(i < 0 || i >= columns) return param;
   if(ArraySize(data) > 0 && cursor != -1) // если есть кэш, возвращаем из него
   {
      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: // возвращаем base64 только для информации, т.к.
      {                           // нет способа вернуть двоичные данные в MqlPara
         uchar blob[];            // точное представление двоичных полей выдает ge
         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;
}
```

Для полноценного считывания двоичных данных из BLOB-полей предусмотрен метод *getBlob* (используйте тип *uchar* в качестве S, чтобы получить массив байтов, если нет более конкретной информации о формате содержимого).

```
template<typename S>
int getBlob(const int i, S &object[])
{
   ...
   return DatabaseColumnBlob(query, i, object);
}
```

Для описанных методов процесс выполнения запроса и чтения его результатов можно представить таким псевдо-кодом (в нем оставлены за кадром уже существующие классы *DBSQLite* и *DBQuery* — скоро мы сведем их воедино):

```
int query = ...
DBRow *row = new DBRow(query);
while(row.next())
{
   for(int i = 0; i < row.length(); ++i)
   {
      StructPrint(row[i]); // печать i-го столбца в виде структуры MqlParam
   }
}
```

Прописывать каждый раз цикл по столбцам в явном виде не очень красиво, поэтому в классе предусмотрен метод для получения значений всех полей записи.

```
void readAll(MqlParam &params[]) const
{
   ArrayResize(params, columns);
   for(int i = 0; i < columns; ++i)
   {
      params[i] = this[i];
   }
}
```

Также для удобства в класс добавлены перегрузки оператора '[]' и метода *getBlob* для чтения полей по их именам вместо индексов. Например,

```
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]; // перегрузка operator()[int]
      static MqlParam param = {};
      return param;
   }
   ...
};
```

Таким образом, вы можете обращаться к избранным столбцам.

```
int query = ...
DBRow *row = new DBRow(query);
for(int i = 1; row.next(); )
{
   Print(i++, " ", row["trades"], " ", row["profit"], " ", row["drawdown"]);
}
```

Но все же получение элементов записи по отдельности, в виде массива *MqlParam* нельзя назвать по-настоящему объектным подходом. Было бы желательно считывать запись таблицы базы целиком в объект — прикладную структуру. Напомним, что MQL5 API предоставляет подходящую функцию: *DatabaseReadBind*. Именно здесь нам приходит на помощь возможность описать класс-наследник *DBRow* и переопределить в нем виртуальный метод *DBRead*.

Этот класс *DBRowStruct* является шаблоном и ожидает в качестве параметра S одну из простых структур, разрешенных для привязки в *DatabaseReadBind*.

```
template<typename S>
class DBRowStruct: public DBRow
{
protected:
   S object;

   virtual bool DBread() override
   {
      // NB: унаследованные структуры и с вложенными структурами не разрешены;
      // количество полей структуры не должно превышать количество столбцов в таблице
      return PRTF(DatabaseReadBind(query, object));
   }

public:
   DBRowStruct(const int q, const bool c = false): DBRow(q, c)
   {
   }

   S get() const
   {
      return object;
   }
};
```

С производным классом мы можем практически бесшовно получать объекты из базы.

```
int query = ...
DBRowStruct<MyStruct> *row = new DBRowStruct<MyStruct>(query);
MyStruct structs[];
while(row.next())
{
   PUSH(structs, row.get());
}
```

Теперь настало время для превращения псевдо-кода в рабочий код за счет увязки *DBRow/DBRowStruct* с *DBQuery*. Добавим в *DBQuery* автоуказатель на объект *DBRow*, который будет содержать данные о текущей записи из результатов запроса (если он был выполнен). Использование автоуказателя освобождает вызывающий код от забот об освобождении объектов *DBRow*: они удаляются вместе с *DBQuery* или при повторном создании из-за перезапуска запроса (если такое потребуется). Инициализацию объекта *DBRow* или *DBRowStruct* производит шаблонный метод *start*.

```
class DBQuery
{
protected:
   ...
   AutoPtr<DBRow> row;    // текущая запись
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[];
   }
```

Тип *DBValue* представляет собой структуру-пустышку, которая нужна только для того, чтобы инструктировать программу о создании базового объекта *DBRow*, не нарушая при этом компилируемость строки с вызовом *DatabaseReadBind*.

С методом *start* все вышеприведенные фрагменты псевдокода становятся рабочими за счет такой подготовки запроса:

```
DBSQLite db("MQL5Book/DB/Example1");                            // открываем базу
DBQuery *query = db.prepare("PRAGMA table_xinfo('Struct')");    // подготавливаем зап
DBRowStruct<DBTableColumn> *row = query.start<DBTableColumn>(); // получаем объектный
DBTableColumn columns[];                                        // приемный массив об
while(row.next())             // цикл пока есть записи в результате запроса
{
   PUSH(columns, row.get());  // получаем объект из текущей записи
}
ArrayPrint(columns);
```

Данный пример считывает из базы данных мета-информацию о конфигурации конкретной таблицы (мы её создали в примере *DBcreateTableFromStruct.mq5* в разделе Выполнение запросов без привязки к данным MQL5): каждый столбец описывается отдельной записью с несколькими полями (стандарт SQLite), что формализовано в структуре *DBTableColumn*.

```
struct DBTableColumn
{
   int cid;              // идентификатор (порядковый номер)
   string name;          // название
   string type;          // тип
   bool not_null;        // признак NOT NULL (да/нет)
   string default_value; // значение по умолчанию
   bool primary_key;     // признак PRIMARY KEY (да/нет)
};
```

Чтобы избавить пользователя от необходимости каждый раз писать цикл с переводом записей результатов в объекты-структуры, в классе *DBQuery* есть шаблонный метод *readAll*, который заполняет передаваемый по ссылке массив структур информацией из результатов запроса. Аналогичный метод *readAll* позволяет заполнить массив указателей на объекты *DBRow* (это больше подойдет для приема результатов синтетических запросов с колонками из разных таблиц).

В квартете операций CRUD метод *DBRowStruct::get* отвечает за букву R (Read). Чтобы сделать чтение объекта более функционально полным поддержим точечное восстановление объекта из базы по его идентификатору.

Подавляющее большинство таблиц в базах SQLite имеет первичный ключ *rowid* (если только разработчик по тем или иным причинам не использовал опцию "WITHOUT ROWID" в описании), поэтому новый метод *read* будет принимать в качестве параметра значение ключа. По умолчанию, название таблицы подразумевается равным типу принимающей структуры, но можем быть изменено на альтернативное через параметр *table*. Учитывая, что подобный запрос носит разовый характер и должен вернуть одну запись, имеет смысл поместить метод *read* непосредственно в класс *DBSQLite* и управлять короткоживущими объектами *DBQuery* и *DBRowStruct<S>* внутри.

```
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;
   }
};
```

Основную работу выполняет SQL-запрос "SELECT * FROM '%s' WHERE %s=%ld;", возвращающий запись со всеми полями из указанной таблицы по совпадению ключа *rowid*.

Теперь создать конкретный объект из базы можно так (подразумевается, что интересующий нас идентификатор должен быть где-то сохранен).

```
   DBSQLite db("MQL5Book/DB/Example1");
   long rowid = ... // заполняем идентификатор
   Struct s;
   if(db.read(rowid, s))
      StructPrint(s);
```

Наконец, в некоторых сложных случаях, когда требуется максимальная гибкость в составлении запроса (например, комбинация нескольких таблиц — как правило, SELECT с JOIN, или вложенные запросы), нам все же придется разрешить задавать SQL-команду в явном виде для получения выборки, хотя это и нарушает принцип ORM. Эту возможность открывает метод *DBSQLite::prepare*, который мы уже представляли раньше в контексте управления подготовленными запросами.

На этом все основные способы чтения можно считать рассмотренными.

Однако нам пока нечего читать из базы, потому что мы перескочили через этап добавления записей.

Попробуем реализовать создание объекта (C). Напомним, что в нашей объектной концепции типы структур полуавтоматически определяют под себя таблицы базы данных (с помощью макросов DB_FIELD). Например, структура *Struct* позволила создать в базе таблицу "Struct" с набором столбцов, соответствующих полям структуры. Мы обеспечили это шаблонным методом

createTable в классе *DBSQLite*. Теперь по аналогии необходимо написать шаблонный метод *insert*, который добавлял бы запись в эту таблицу.

В метод передается объект структуры, для типа которой должен существовать заполненный массив *DBEntity<S>::prototype* (он заполняется макросами). Благодаря этому массиву мы можем сформировать список параметров (точнее, их заместителей '?n'): это поручено статическому методу *qlist*. Однако подготовка запроса еще полдела — чуть ниже нужно будет выполнить привязку входных данных на основе свойств объекта.

В команду "INSERT" добавлена инструкция "RETURNING rowid", поэтому при успешном выполнении запроса мы ожидаем единственную строку результата с одним значением: новым *rowid*.

```
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()) // результат должен быть одной записью с одним значением нового
         {
            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;
   }
};
```

В исходном коде метода *insert* есть один нюанс, на который следует обратить особое внимание. Для привязки значений к параметрам запроса вызывается метод *object.bindAll(q)*. Это означает,

что в прикладной структуре, которую требуется интегрировать с базой, нужно реализовать такой метод, предоставляющий для "движка" все переменные-члены.

Кроме того, для идентификации объектов предполагается наличие поля с первичным ключом, и лишь объект "знает", что это за поле, поэтому в структуре имеется метод *rowid*, позволяющий, с одной стороны, передать в объект идентификатор, присвоенный его записи в базе, а с другой стороны — узнать у объекта этот идентификатор, если тот уже был присвоен ранее.

Метод для изменения записи *DBSQLite::update* (U) во многом схож с *insert*, и потому с ним предлагается ознакомиться самостоятельно. Его основой является SQL-запрос "UPDATE '%s' SET (%s)=(%s) WHERE rowid=%ld;", в который предполагается передача всех полей структуры (*bindAll()* объекта) и ключ (*rowid()* объекта).

Наконец, упомянем, что точечное удаление (D) записи по объекту реализовано в методе *DBSQLite::remove* (слово *delete* является оператором MQL5).

Покажем все методы в примере скрипта *DBfillTableFromStructArray.mq5*, где определена новая структура *Struct*.

Полями структуры сделаем несколько значений общеупотребительных типов.

```
struct Struct
{
   long id;
   string name;
   double number;
   datetime timestamp;
   string image;
   ...
};
```

В строковом поле *image* вызывающий код будет указывать название графического ресурса или имя файла, а в момент привязки к базе соответствующие двоичные данные будут копироваться как BLOB. Впоследствии, когда мы будем считывать данные из базы в объекты *Struct*, двоичные данные будут попадать в строку *image*, но, разумеется, с искажениями (потому что строка прервется на первом нулевом байте). Для точного извлечения BLOB-ов из базы нужно будет вызвать метод *DBRow::getBlob* (на основе *DatabaseColumnBlob*).

Создание мета-информации о полях структуры *Struct* обеспечивают следующие макросы. На их основе MQL-программа может автоматически создать таблицу в базе под объекты *Struct*, а также инициировать привязку передаваемых в запросы данных на основе свойств объектов (не следует путать эту привязку с обратной привязкой для получения результатов запроса, т.е. *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);
```

Для наполнения небольшого тестового массива структур в скрипте имеются входные переменные: в них указывается тройка валют, котировки которых попадут в поле *number*. Также мы встроили в скрипт два стандартных изображения с целью проверки работы с BLOB-ами: они "отправятся" в поле *image*. Поле *timestamp* будет автоматически заполняться нашими ORM-

классами текущей временной меткой вставки или модификации записи. Первичный ключ в поле *id* должна будет заполнять сама 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";
```

Поскольку значения для входных переменных запросов (те самые '?n') привязываются, в конечном счете, с помощью функций *DatabaseBind* или *DatabaseBindArray* под номерами, наша структура должна в методе *bindAll* установить соответствие между номерами и своими полями: предполагается простая нумерация по порядку декларирования.

```
struct Struct
{
   ...
   bool bindAll(DBQuery &q) const
   {
      uint pixels[] = {};
      uint w, h;
      if(StringLen(image))                // загрузка двоичных данных
      {
         if(StringFind(image, "::") == 0) // это ресурс
         {
            ResourceReadImage(image, pixels, w, h);
            // отладка/проверка примера (не BMP, без заголовка)
            FileSave(StringSubstr(image, 2) + ".raw", pixels);
         }
         else                             // это файл
         {
            const string res = "::" + image;
            ResourceCreate(res, image);
            ResourceReadImage(res, pixels, w, h);
            ResourceFree(res);
         }
      }
      // когда id = NULL, база присвоит новый rowid
      return (id == 0 ? q.bindNull(0) : q.bind(0, id))
         && q.bind(1, name)
         && q.bind(2, number)
         // && q.bind(3, timestamp) // это поле автозаполнится CURRENT_TIMESTAMP
         && q.bindBlob(4, pixels);
   }
   ...
};
```

Метод *rowid* очень прост.

```
struct Struct
{
   ...
   long rowid(const long setter = 0)
   {
      if(setter) id = setter;
      return id;
   }
};
```

Определив структуру, опишем тестовый массив из 4 элементов. Только 2 из них имеют привязанные изображения. Идентификаторы у всех объектов нулевые, т.к. они еще не в базе.

```
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},
};
```

В главной функции *OnStart* создадим или откроем базу (по умолчанию *MQL5Book/DB/Example2.sqlite*), на всякий случай попытаемся удалить таблицу "Struct", чтобы при повторных запусках скрипта обеспечить воспроизводимость результатов и отладки, затем создадим таблицу под структуру *Struct*.

```
void OnStart()
{
   DBSQLite db(Database);
   if(!PRTF(db.isOpen())) return;
   PRTF(db.deleteTable(typename(Struct)));
   if(!PRTF(db.createTable<Struct>(true))) return;
   ...
```

Вместо того, чтобы добавлять объекты по одному, в цикле вроде такого:

```
   // -> этот вариант (отложен в сторону)
   for(int i = 0; i < ArraySize(demo); ++i)
   {
      PRTF(db.insert(demo[i])); // получаем новый rowid при каждом вызове
   }
```

воспользуемся альтернативной реализацией метода *insert*, который принимает на вход сразу массив объектов и обрабатывает их в едином запросе — так эффективнее (но общая канава метода — такая же, как в показанном выше методе *insert* для одного объекта).

```
   db.insert(demo);  // в объектах проставляются новые rowid
   ArrayPrint(demo);
   ...
```

Теперь попробуем отобрать из базы записи по некоторому условию, например, те, у которых не назначено изображение. Для этого подготовим SQL-запрос, обернутый в объект *DBQuery*, и далее получим его результаты двумя способами: через привязку к структурам *Struct* или через экземпляры универсального класса *DBRow*.

```
DBQuery *query = db.prepare(StringFormat("SELECT * FROM %s WHERE image IS NULL",
   typename(Struct)));

// подход 1: прикладной тип структуры Struct
Struct result[];
PRTF(query.readAll(result));
ArrayPrint(result);

query.reset(); // сбрасываем запрос, чтобы повторить

// подход 2: универсальный контейнер записи DBRow со значениями MqlParam
DBRow *rows[];
query.readAll(rows); // получаем объекты DBRow с кэшированными значениями
for(int i = 0; i < ArraySize(rows); ++i)
{
   Print(i);
   MqlParam fields[];
   rows[i].readAll(fields);
   ArrayPrint(fields);
}
...
```

Оба варианта должны дать один и тот же результат, хотя и представленный по-разному (см. лог ниже).

Далее наш скрипт делает паузу на 1 секунду, чтобы можно было заметить изменения во временных метках следующих записей, которые мы изменим.

```
Print("Pause...");
Sleep(1000);
...
```

Назначим объектам в массиве *result[]* изображение "yuan.bmp", расположенное в папке рядом со скриптом. И тут же обновим объекты в базе.

```
for(int i = 0; i < ArraySize(result); ++i)
{
   result[i].image = "yuan.bmp";
   db.update(result[i]);
}
...
```

После запуска скрипта вы можете убедиться в наличии BLOB-ов у всех четырех записей в навигаторе БД, встроенном в MetaEditor, а также в различии временных меток у первых двух и последних двух записей.

Продемонстрируем извлечение двоичных данных. Сначала покажем, как BLOB отображается в строковое поле *image* (двоичные данные — не для журнала, мы делаем так только для демонстрации).

```
const long id1 = 1;
Struct s;
if(db.read(id1, s))
{
   Print("Length of string with Blob: ", StringLen(s.image));
   Print(s.image);
}
...
```

Затем прочитаем данные целиком с помощью *getBlob* (полная длина больше, чем у строки выше).

```
DBRow *r;
if(db.read(id1, r, "Struct"))
{
   uchar bytes[];
   Print("Actual size of Blob: ", r.getBlob("image", bytes));
   FileSave("temp.bmp.raw", bytes); // не BMP, без заголовка
}
```

Мы должны получить файл *temp.bmp.raw*, идентичный *MQL5/Files/Images/dollar.bmp.raw*, который в целях отладки создается в методе *Struct::bindAll*. Таким образом, легко убедиться в точном соответствии записанных и прочитанных двоичных данных.

> Обратите внимание, что поскольку мы сохраняем в базу двоичное содержимое ресурса, оно не является исходным файлом формата BMP: ресурсы производят нормализацию цвета и хранят массив пикселей без заголовка с мета-информацией об изображении.

В процессе работы скрипт формирует подробный лог. В частности, создание базы и таблицы отмечается следующими строками.

```
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
```

SQL-запрос вставки массива объектов подготавливается однократно, а затем многократно исполняется с предварительной привязкой разных данных (здесь показана только одна итерация). Количество вызовов *DatabaseBind*-функций соответствует переменным '?n' в запросе ('?4' автоматически заменена нашими классами на вызов функции SQL STRFTIME('%s') для получения метки текущего времени UTC).

```
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
...
```

Далее в журнал выводится массив структур с уже назначенными первичными ключами *rowid* в первой колонке.

```
    [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
```

Выбор записей без изображений выдает следующий результат (мы выполняем этот запрос дважды разными методами: первый раз заполняем массив структур *Struct*, а второй — массив *DBRow*, из которого для каждого поля получаем "значение" в виде *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
...
```

Вторая часть скрипта обновляет пару найденных записей без изображений и добавляет в них BLOB-ы.

```
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)
...
```

Наконец, получение двоичных данных двумя способами — несовместимым, через строковое поле *image* в результате чтения объекта целиком *DatabaseReadBind* (это сделано только для визуализации последовательности байтов в журнале) и совместимым, через *DatabaseRead* и *DatabaseColumnBlob* — дает отличные результаты: разумеется, второй способ правильный: длина и содержимое BLOB-а в 4096 байтах восстановлены.

```
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�ƚ7�ö6�ũ6�Ĩ5���5�¦5�Ĩ5�ʅ6�ö6�Ō7�ƚ7�ƚ7�7�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
```

Подводя промежуточный итог разработки собственной оболочки ORM, приведем обобщенную схему её классов.

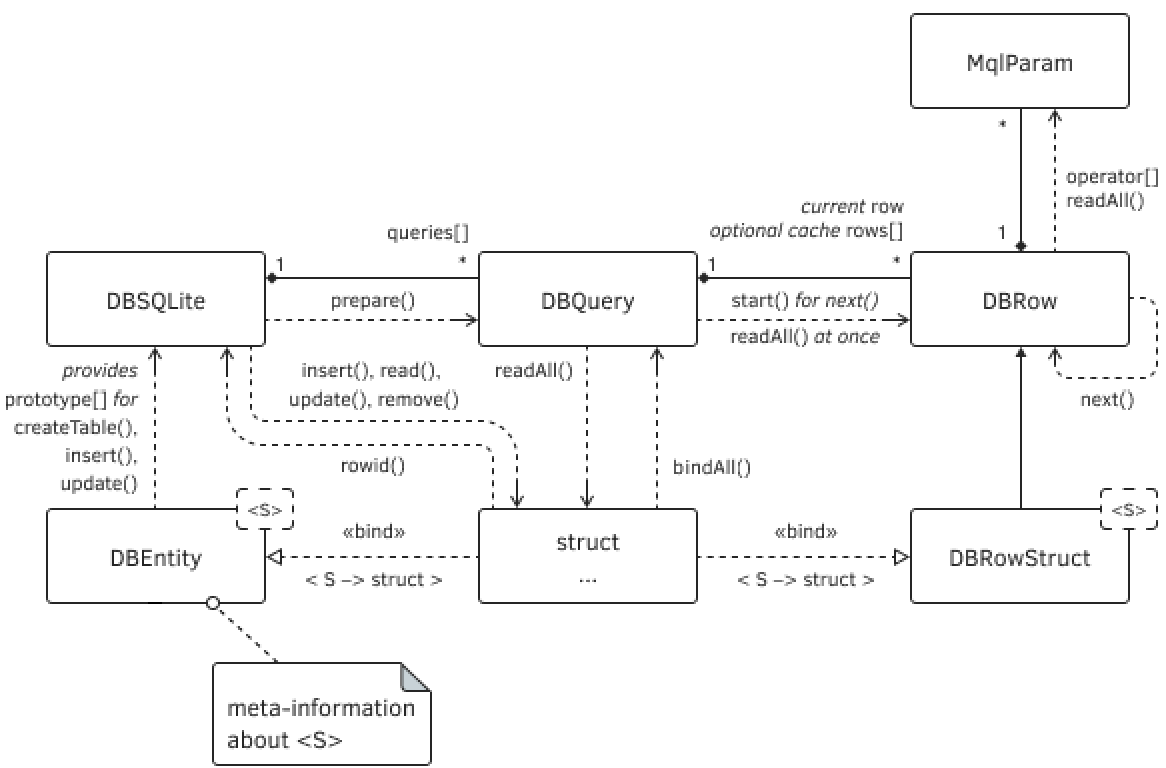


Диаграмма классов ORM (MQL5<->SQL)

7.6.13 Транзакции

SQLite поддерживает механизм *транзакций* — логически связанных наборов действий, которые могут быть выполнены либо целиком, либо не выполнены вовсе, что обеспечивает непротиворечивость данных в базе.

> Понятие *транзакция* имеет в контексте баз данных новый смысл, отличающийся от того, что мы использовали при описании торговых транзакций: там под транзакцией подразумевалась отдельная операция над сущностями торгового счета — ордерами, сделками, позициями.

Транзакции обеспечивают 4 основных характеристики изменений базы:

- Атомарность (неделимость) — при успешном завершении транзакции в базу попадут все входящие в неё изменения, а в случае ошибки — не попадет ничего;
- Согласованность — текущее правильное состояние базы может измениться только на другое правильное (промежуточные, по прикладной логике, состояния исключены);
- Изолированность — изменения в транзакции текущего подключения не видны до конца этой транзакции в других подключениях к той же базе и наоборот — изменения из других подключений не видны в текущем подключении, пока здесь есть незавершенная транзакция;
- Надежность — изменения из успешной транзакции гарантированно сохраняются в базе.

Англоязычные термины этих характеристик — Atomic, Consistent, Isolated, Durable — формируют известный в теории баз данных акроним ACID.

Даже если нормальный ход программы будет прерван из-за системного сбоя, база данных сохранит свое рабочее состояние.

Наиболее часто использование транзакций иллюстрируют примером банковской системы, в базе которой выполняется перевод средств со счета одного клиента на счет другого. Он должен затронуть две записи с балансами клиентов: в одной остаток уменьшается на размер перевода, а в другой — увеличивается. Ситуация, когда применилось бы только одно из этих изменений, нарушило бы баланс банковских счетов: в зависимости от того, какая операция отказала, перечисленная сумма могла бы пропасть или, наоборот, взяться ниоткуда.

Можно привести более приближенный к трейдерской практике пример, но по принципу "от противного". Дело в том, что система учета ордеров, сделок и позиций в MetaTrader 5 не является транзакционной.

В частности, как мы знаем из главы про Создание экспертов, сработавший ордер (рыночный или отложенный), пропав из списка активных, может не сразу отобразиться в списке позиций. Поэтому для анализа фактического результата приходится в MQL-программе реализовывать ожидание обновления (актуализации) торгового окружения. Если бы система учета строилась на транзакциях, то исполнение ордера, регистрация сделки в истории и появление позиции были бы заключены в транзакцию и согласованы друг с другом. Разработчики терминала выбрали другой подход: максимально быстро и асинхронно возвращать любые модификации торгового окружения, а за их целостностью должна следить MQL-программа.

Любая SQL-команда, которая изменяет базу (то есть фактически все, кроме SELECT), автоматически будет обернута в транзакцию, если это не было сделано предварительно явным образом.

MQL5 API предоставляет 3 функции для управления транзакциями: *DatabaseTransactionBegin*, *DatabaseTransactionCommit*, *DatabaseTransactionRollback*. Все функции возвращают *true* в случае успеха или *false* в случае ошибки.

bool DatabaseTransactionBegin(int database)

Функция *DatabaseTransactionBegin* начинает выполнение транзакции в базе данных с указанным дескриптором, полученным из *DatabaseOpen*.

Все последующие изменения, производимые в базе, накапливаются во внутреннем кэше транзакции и не попадают в базу, пока не будет вызвана функция *DatabaseTransactionCommit*.

Транзакции в MQL5 не могут быть вложенными: если транзакция уже начата, то повторный вызов *DatabaseTransactionBegin* вернет признак ошибки и выдаст сообщение в журнал.

```
database error, cannot start a transaction within a transaction
DatabaseTransactionBegin(db)=false / DATABASE_ERROR(5601)
```

Соответственно, нельзя пытаться и завершить транзакцию многократно.

bool DatabaseTransactionCommit(int database)

Функция *DatabaseTransactionCommit* завершает транзакцию, предварительно начатую в базе с указанным дескриптором, и применяет все накопленные изменения (сохраняет их). Если MQL-программа начнет транзакцию, но не применит её до закрытия базы, все изменения будут потеряны.

При необходимости программа может отменить транзакцию и, тем самым, все изменения с начала транзакции.

bool DatabaseTransactionRollback(int database)

Функция *DatabaseTransactionRollback* выполняет "откат" всех действий, попавших в начатую ранее транзакцию для базы с дескриптором *database*.

Дополним класс *DBSQLite* методами для работы с транзакциями, с учетом ограничения на их вложенность, которую будем подсчитывать в переменной *transaction*. Если она равна 0, метод *begin* начинает транзакцию вызовом *DatabaseTransactionBegin*. Все последующие попытки начать транзакцию просто увеличивают счетчик. В методе *commit* уменьшаем счетчик, и по достижении 0 вызываем *DatabaseTransactionCommit*.

```
class DBSQLite
{
protected:
   int transaction;
   ...
public:
   bool begin()
   {
      if(transaction > 0)   // уже в транзакции
      {
         transaction++;     // отслеживаем уровень вложенности
         return true;
      }
      return (bool)(transaction = PRTF(DatabaseTransactionBegin(handle)));
   }

   bool commit()
   {
      if(transaction > 0)
      {
         if(--transaction == 0) // самая внешняя транзакция
            return PRTF(DatabaseTransactionCommit(handle));
      }
      return false;
   }
   bool rollback()
   {
      if(transaction > 0)
      {
         if(--transaction == 0)
            return PRTF(DatabaseTransactionRollback(handle));
      }
      return false;
   }
};
```

Кроме того, создадим класс *DBTransaction*, который позволит описывать внутри блоков (например, функций) объекты, обеспечивающие автоматическое начало транзакции с её последующим применением (или отменой) при выходе программы из блока.

```
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;
   }
};
```

Политика использования таких объектов избавляет от необходимости обрабатывать различные варианты выхода из блока (функции).

```
void DataFunction(DBSQLite &db)
{
   DBTransaction tr(db);
   DBQuery *query = db.prepare("UPDATE..."); // пакетные изменения
   ... // модификация базы
   if(... /* ошибка1 */) return;             // автоматический rollback
   ... // модификация базы
   if(... /* ошибка2 */) return;             // автоматический rollback
   tr.commit();
}
```

Чтобы объект автоматически применял изменения на любой стадии, следует передать *true* во втором параметре его конструктора.

```
void DataFunction(DBSQLite &db)
{
   DBTransaction tr(db, true);
   DBQuery *query = db.prepare("UPDATE..."); // пакетные изменения
   ... // модификация базы
   if(... /* условие1 */) return;            // автоматический commit
   ... // модификация базы
   if(... /* условие2 */) return;            // автоматический commit
   ...
}                                            // автоматический commit
```

Вы можете описать объект *DBTransaction* внутри цикла, и тогда на каждой итерации будет начинаться и закрываться отдельная транзакция.

Демонстрация транзакций будет приведена в разделе Пример поиска торговой стратегии средствами SQLite.

7.6.14 Импорт и экспорт таблицы базы данных

MQL5 позволяет экспортировать и импортировать отдельные таблицы базы данных в/из файлов формата CSV. Экспорт/импорт базы целиком, в виде файла с SQL-командами, к сожалению, не предусмотрен.

long DatabaseImport(int database, const string table, const string filename, uint flags,
  const string separator, ulong skip_rows, const string comment_chars)

Функция *DatabaseImport* импортирует в таблицу данные из указанного файла. Дескриптор открытой базы и имя таблицы задаются двумя первыми параметрами.

Если таблицы с именем *table* не существует, она будет создана автоматически. Имена и типы полей в таблице будут распознаны автоматически на основе данных, содержащихся в файле.

Импортируемый файл может представлять собой не только готовый CSV-файл, но и ZIP-архив с CSV-файлом. Имя файла может содержать путь. Файл ищется относительно каталога *MQL5/Files*.

Допустимые флаги, из которых можно составить побитовую комбинацию, описаны в перечислении ENUM_DATABASE_IMPORT_FLAGS:

- DATABASE_IMPORT_HEADER — первая строка содержит имена полей таблицы;
- DATABASE_IMPORT_CRLF — для переносов строк используется последовательность символов CRLF;
- DATABASE_IMPORT_APPEND — добавлять данные к существующей таблице;
- DATABASE_IMPORT_QUOTED_STRINGS — строковые значения в двойных кавычках;
- DATABASE_IMPORT_COMMON_FOLDER — общая папка терминалов.

Параметр *separator* задает символ-разделитель в CSV-файле.

Параметр *skip_rows* позволяет пропустить указанное количество начальных строк в файле.

Параметр *comment_chars* содержит символы, используемые в файле в качестве признака комментария. Строки, начинающиеся с любого из этих символов, будут считаться комментариями и не будут импортироваться.

Функция возвращает количество импортированных строк или -1 в случае ошибки.

long DatabaseExport(int database, const string table_or_sql, const string filename, uint flags, const string separator)

Функция *DatabaseExport* экспортирует таблицу или результат выполнения SQL-запроса в CSV-файл. Дескриптор базы, а также имя таблицы или текст запроса задаются в первых двух параметрах.

Если экспортируются результаты запроса, то SQL-запрос должен начинаться с "SELECT" или "select". Другими словами, SQL-запрос не может изменять состояние базы данных, в противном случае *DatabaseExport* завершится ошибкой.

Имя файла *filename* может содержать путь внутри каталога *MQL5/Files* текущего экземпляра терминала или общей папки терминалов, в зависимости от флагов.

Параметр *flags* позволяет задать комбинацию флагов, управляющую форматом и размещением файла.

- DATABASE_EXPORT_HEADER — выводить строку с именами полей;
- DATABASE_EXPORT_INDEX — выводить номера строк;
- DATABASE_EXPORT_NO_BOM — не вставлять метку BOM в начале файла (по умолчанию BOM вставляется);
- DATABASE_EXPORT_CRLF — для переноса строки использовать CRLF (по умолчанию LF);
- DATABASE_EXPORT_APPEND — дописывать данные в конец существующего файла (по умолчанию файл перезаписывается), если файл не существует — он будет создан;
- DATABASE_EXPORT_QUOTED_STRINGS — выводить строковые значения в двойных кавычках;
- DATABASE_EXPORT_COMMON_FOLDER — CSV-файл будет создан в общей папке всех терминалов *MetaQuotes/Terminal/Common/File*.

Параметр *separator* задает символ-разделитель столбцов. Если он равен NULL, то в качестве разделителя будет использоваться символ табуляции '\t'. Пустая строка "" считается допустимым разделителем, но полученный CSV-файл не сможет быть прочитан как таблица — это будет набор строк.

Текстовые поля в базе данных могут содержать переводы строк ('\r' или '\r\n'), а также символ-разделитель, указанный в параметре separator. В этом случае нужно обязательно использовать флаг DATABASE_EXPORT_QUOTED_STRINGS в параметре *flags*. При наличии этого флага все выводимые строки будут заключены в двойные кавычки, а если в строке содержится двойная кавычка, то она будет заменена на две двойные кавычки.

Функция возвращает количество экспортированных записей или отрицательное значение в случае ошибки.

7.6.15 Печать таблиц и SQL-запросов в журнал

При необходимости MQL-программа может вывести содержимое таблицы или результаты выполнения SQL-запроса в журнал с помощью функции *DatabasePrint*.

long DatabasePrint(int database, const string table_or_sql, uint flags)

Дескриптор базы передается в первом параметре, далее идет имя таблицы или текст запроса (*table_or_sql*). SQL-запрос должен начинаться с "SELECT" или "select", то есть он не должен изменять состояние базы данных — или функция *DatabasePrint* завершится ошибкой.

В параметре *flags* указывается комбинация флагов, определяющая форматирование вывода.

- DATABASE_PRINT_NO_HEADER – не выводить названия столбцов таблицы (имена полей);
- DATABASE_PRINT_NO_INDEX – не выводить номера строк;
- DATABASE_PRINT_NO_FRAME – не выводить фрейм, разделяющий заголовок и данные;
- DATABASE_PRINT_STRINGS_RIGHT – выравнивать строки вправо.

Если *flags = 0*, то выводятся столбцы и строки, заголовок и данные разделяются фреймом, строки выравниваются влево.

Функция возвращает количество выведенных записей или -1 в случае ошибки.

Воспользуемся функцией в следующем разделе.

К сожалению, функция не позволяет выводить подготовленные запросы с параметрами. При наличии параметров их потребуется внедрить в текст запроса на уровне MQL5.

7.6.16 Пример поиска торговой стратегии средствами SQLite

Попробуем применить SQLite для решения практических задач. Импортируем в базу структуры *MqlRates* с историей котировок и проведем их анализ с целью выявления закономерностей и поиска потенциальных торговых стратегий. Разумеется, любая выбранная логика может быть реализована и на MQL5, однако SQL позволяет делать это иным способом, во многих случаях более эффективно и с привлечением множества интересных встроенных функций SQL. Тематика книги, нацеленной на изучение MQL5, не позволяет углубляться в эту технологию, но мы упоминаем её, как заслуживающую внимания алготрейдера.

Скрипт для перевода истории котировок в формат базы данных называется *DBquotesImport.mq5*. Во входных параметрах можно задать префикс названия базы данных и размер транзакции (количество записей в одной транзакции).

```
input string Database = "MQL5Book/DB/Quotes";
input int TransactionSize = 1000;
```

Для добавления структур *MqlRates* в базу с использованием нашей ORM-прослойки в скрипте определена вспомогательная структура *MqlRatesDB*, в которую "зашиты" правила привязки полей структуры в столбцы базы. Поскольку наш скрипт только записывает данные в базу и не читает их оттуда, ему не нужна привязка с помощью функции *DatabaseReadBind*, что накладывало бы ограничение на "простоту" структуры. Отсутствие ограничения позволяет сделать структуру *MqlRatesDB* производной от *MqlRates* (и не повторять описание полей).

```
struct MqlRatesDB: public MqlRates
{
   /* для справки:

      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 устанавливается нами по времени бара
      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);
```

Название базы формируется из префикса *Database*, имени и таймфрейма текущего графика, на котором запущен скрипт. В базе создается единственная таблица "MqlRatesDB" с конфигурацией полей, заданной макросами DB_FIELD. Обратите внимание, что первичный ключ не будет генерироваться базой, а берется непосредственно из баров, из поля *time* (время открытия бара).

```
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;
   ...
```

Далее пакетами по *TransactionSize* баров запрашиваем бары из истории и добавляем в таблицу. Этим занимается вспомогательная функция *ReadChunk*, вызываемая в цикле до тех пор, пока есть данные (функция при этом возвращает *true*) или пользователь не остановит скрипт вручную. Код функции показан чуть ниже.

```
   int offset = 0;
   while(ReadChunk(db, offset, TransactionSize) && !IsStopped())
   {
      offset += TransactionSize;
   }
```

По завершению процесса запрашиваем у базы количество сформированных записей в таблице и выводим его в журнал.

```
   DBRow *rows[];
   if(db.prepare(StringFormat("SELECT COUNT(*) FROM %s",
      typename(MqlRatesDB))).readAll(rows))
   {
      Print("Records added: ", rows[0][0].integer_value);
   }
}
```

Обещанная функция *ReadChunk* выглядит следующим образом.

```
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;
}
```

В ней вызывается встроенная функция *CopyRates* и, тем самым, заполняется массив баров *rates*. Затем бары переносятся в массив *ratesDB*, чтобы одним оператором *db.insert(ratesDB)* записать информацию в базу (именно в *MqlRatesDB* нами формализовано, как это правильно делать).

Наличие объекта *DBTransaction* (с включенной опцией автоматического "коммита") внутри блока означает, что все операции с массивом "обложены" транзакцией. Для индикации прогресса, во время обработки каждого блока баров в журнал выводится метка первого бара.

Пока функция *CopyRates* возвращает данные и их вставка в базу происходит успешно, цикл в *OnStart* продолжается со сдвигом номеров копируемых баров вглубь истории. Когда достигается конец доступной истории или лимит баров, заданный в настройках терминала, *CopyRates* вернет ошибку 4401 (HISTORY_NOT_FOUND), и работа скрипта завершится.

Запустим скрипт на графике EURUSD,H1. В журнале должны появиться примерно такие сообщения.

```
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
```

Теперь у нас имеется база *QuotesEURUSDH1.sqlite*, над которой можно ставить эксперименты для проверки различных торговых гипотез. Вы можете открыть её в редакторе MetaEditor, чтобы убедиться в правильности переноса данных.

Проверим одну из простейших стратегий, основанных на регулярностях в истории. Найдем статистику двух последовательных баров в одном направлении в разбивке по внутридневному времени и дню недели. Если для какого-то сочетания времени и дня недели найдется ощутимое преимущество, его можно рассматривать в будущем как сигнал для входа в рынок по направлению первого бара.

Для начала спроектируем SQL-запрос, запрашивающий котировки за некоторый период и вычисляющий движение цены на каждом баре, то есть разницу между соседними ценами открытия.

Поскольку время баров хранится как количество секунд (по стандартам *datetime* MQL5 и, по совместительству, "эпоха Unix" SQL), их отображение для удобного чтения желательно преобразовать в строку, поэтому начнем запрос SELECT с поля *datetime* на основе функции DATETIME:

```
SELECT
   DATETIME(time, 'unixepoch') as datetime, open, ...
```

В анализе это поле участвовать не будет и добавлено только для пользователя. После для справки выведена цена, чтобы мы могли по отладочной печати проверить расчет приращений цен.

Поскольку мы сбираемся, при необходимости, выбирать некий период из всего файла, в условии потребуется поле *time* в "чистом виде", и его тоже следует добавить в запрос. Кроме того, согласно планируемому анализу котировок нам потребуется вычленить из метки бара его внутрисуточное время, а также день недели (их нумерация соответствует принятой в MQL5, 0 — воскресенье). Два последних столбца запроса назовем *intraday* и *day*, соответственно, а для их получения задействованы функции TIME и STRFTIME.

```
SELECT
   DATETIME(time, 'unixepoch') as datetime, open,
   time,
   TIME(time, 'unixepoch') AS intraday,
   STRFTIME('%w', time, 'unixepoch') AS day, ...
```

Для вычисления приращения цены в SQL можно применить функцию LAG — она возвращает значение указанной колонки со смещением на заданное количество строк. Например, *LAG(X, 1)* означает получение величины *X* в предыдущей записи, причем второй параметр 1, означающий смещение, по умолчанию равен 1 и потому его можно опустить, получив эквивалентную запись *LAG(X)*. Для получения величины следующей записи следует вызвать *LAG(X,-1)*. В любом случае, при использовании LAG требуется дополнительная синтаксическая конструкция, задающая порядок сортировки записей, в простейшем случае, вида *OVER (ORDER BY столбец)*.

Таким образом, для получения приращения цены между ценами открытия двух соседних баров напишем:

```
   ...
   (LAG(open,-1) OVER (ORDER BY time) - open) AS delta, ...
```

Данная колонка является предсказательной, потому что заглядывает в будущее.

Выявить факт, что два бара сформировались в одном направлении можно по произведению приращений на них — положительные значения укажут согласованный рост или падение:

```
...
(LAG(open,-1) OVER (ORDER BY time) - open) * (open - LAG(open) OVER (ORDER BY time
   AS product, ...
```

Данный показатель выбран как наиболее простой в расчете: для реальных торговых систем можно выбрать более сложный критерий.

Чтобы оценить прибыль, генерируемую системой на бэктесте, нужно направление предыдущего бара (выступающего индикатором будущего движения) умножить на приращение цены на следующем баре. Направление рассчитывается в колонке *direction* (с помощью функции SIGN), только для справки. Оценка прибыли — в колонке *estimate* — это произведение прежнего направления движения (*direction*) на приращение следующего бара (*delta*): если направление сохранится, получим положительный результат (в пунктах).

```
...
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
   AS estimate ...
```

В выражениях в SQL-команде нельзя использовать AS-алиасы, определенные в той же команде. Именно поэтому мы не можем определить *estimate* как *delta * direction*, и приходится повторять вычисление произведения в явном виде. Однако напомним, что колонки *delta* и *direction* для программного анализа не нужны и добавлены здесь только для визуализации таблицы перед пользователем.

В завершение SQL-команды укажем таблицу, из которой делается выборка, и условия фильтрации по диапазону дат бэктеста: два параметра "от" и "до".

```
...
FROM MqlRatesDB
WHERE (time >= ?1 AND time < ?2)
```

Опционально мы можем добавить ограничение *LIMIT ?3* (и вводить какое-нибудь малое значение, например 10), чтобы визуальная проверка результатов запроса на первых порах не вынуждала просматривать десятки тысяч записей.

Проверить работу SQL-команды можно с помощью функции *DatabasePrint*, однако функция, к сожалению, не позволяет работать с подготовленными запросами с параметрами. Поэтому нам придется заменить подготовку параметров SQL '?n' на форматирование строки запроса с помощью *StringFormat* и подставлять значения параметров там. Альтернативно можно было бы полностью отказаться от *DatabasePrint* и выводить результаты в журнал самостоятельно, построчно (через массив *DBRow*).

Таким образом, завершающий фрагмент запроса превратится в:

```
...
WHERE (time >= %ld AND time < %ld)
ORDER BY time LIMIT %d;
```

Следует отметить, что в данном запросе значения *datetime* будут поступать из MQL5 в "машинном" формате количества секунд с начала 1970 года. Если же мы захотим отлаживать этот же SQL-запрос в редакторе MetaEditor, то условие на диапазон дат удобнее записывать с применением литералов (строк) дат, следующим образом:

```
   WHERE (time >= STRFTIME('%s', '2015-01-01') AND time < STRFTIME('%s', '2021-01-01'
```

Здесь опять возникает необходимость использования функции STRFTIME (модификатор '%s' в SQL задает перевод указанной строки с датой в метку "эпохи Unix"; тот факт, что '%s' напоминает форматную строку MQL5 — просто совпадение).

Спроектированный SQL-запрос сохраним в отдельном текстовом файле *DBQuotesIntradayLag.sql* и подключим его как ресурс в одноименный тестовый скрипт *DBQuotesIntradayLag.mq5*.

```
#resource "DBQuotesIntradayLag.sql" as string sql1
```

Первый параметр скрипта позволяет задать префикс в имени базы данных, которая уже должна существовать после запуска *DBquotesImport.mq5* на графике с тем же символом и таймфреймом. Последующие входные параметры предназначены для диапазона дат и ограничения длины отладочной распечатки в журнал.

```
input string Database = "MQL5Book/DB/Quotes";
input datetime SubsetStart = D'2022.01.01';
input datetime SubsetStop = D'2023.01.01';
input int Limit = 10;
```

Таблица с котировками известна заранее, из предыдущего скрипта.

```
const string Table = "MqlRatesDB";
```

В функции *OnStart* откроем базу и убедимся в наличии таблицы котировок.

```
void OnStart()
{
   Print("");
   DBSQLite db(Database + _Symbol + PeriodToString());
   if(!PRTF(db.isOpen())) return;
   if(!PRTF(db.hasTable(Table))) return;
   ...
```

Далее подставляем параметры в строку SQL-запроса. Уделяем внимание не только подмене SQL-параметров '?n' на форматные последовательности, но и удваиваем сперва символы процента '%', потому что иначе функция *StringFormat* воспримет их, как свои команды, и не пропустит в 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);
```

Все эти манипуляции потребовались только для выполнения запроса в контексте функции *DatabasePrint*. В рабочей версии аналитического скрипта мы бы считывали результаты запроса и анализировали их программным способом, минуя форматирование и вызов *DatabasePrint*.

Наконец выполним SQL-запрос и выведем таблицу с результатами в журнал.

```
    DatabasePrint(db.getHandle(), sqlfmt, 0);
  }
```

Вот что мы увидим для 10 баров EURUSD,H1 в начале 2022 года.

```
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 B
            AS product,
         (LAG(open,-1) OVER (ORDER BY time) - open) * SIGN(open - LAG(open) OVER (ORD
            AS estimate
      FROM MqlRatesDB
      WHERE (time >= 1640995200 AND time < 1672531200)
      ORDER BY time LIMIT 10;
 #| datetime               open       time intraday day       delta dir       product
--+------------------------------------------------------------------------------------
 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
 3| 2022-01-03 02:00:00 1.13728 1641175200 02:00:00 1  -0.001060006  1 -3.1799748e-08
 4| 2022-01-03 03:00:00 1.13622 1641178800 03:00:00 1 -0.0003400007 -1  3.6040028e-07
 5| 2022-01-03 04:00:00 1.13588 1641182400 04:00:00 1  -0.001579991 -1  5.3719982e-07
 6| 2022-01-03 05:00:00  1.1343 1641186000 05:00:00 1  0.0005299919 -1 -8.3739827e-07
 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
 9| 2022-01-03 08:00:00  1.1338 1641196800 08:00:00 1   0.000510001 -1 -1.3260079e-07
10| 2022-01-03 09:00:00 1.13431 1641200400 09:00:00 1  0.0004800036  1  2.4480023e-07
...
```

Легко убедиться, что внутридневное время бара выделено правильно, как и день недели — 1, что соответствует понедельнику. Также можно проверить и приращение delta. Значения *product* и *estimate* пусты в первой строке, потому что для их расчета требуется отсутствующая предыдущая строка.

Усложним наш SQL-запрос, сгруппировав записи с одинаковыми сочетаниями времени суток (*intraday*) и дня недели (*day*), и вычислив для каждого из этих сочетаний некий целевой показатель, характеризующий успешность торговли. Возьмем в качестве такого показателя средний размер ячейки *product*, деленный на стандартное отклонение этих же произведений. Чем больше среднее произведение приращений цен соседних баров, тем больше ожидаемая прибыль, а чем меньше разброс этих произведений, тем стабильнее прогноз. Название показателя в SQL-запросе — *objective*.

Помимо целевого показателя будем также рассчитывать оценку прибыли (*backtest_profit*) и профит-фактор (*backtest_PF*). Оценку прибыли получим как сумму приращений цен (*estimate*) по всем барам в разрезе внутридневного времени и дня недели (размер открывающегося бара, как приращение цены — аналог будущей прибыли в пунктах за один бар). Профит-фактор — это, традиционно, частное от положительных и отрицательных приращений.

```
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 B
         AS product,
      (LAG(open,-1) OVER (ORDER BY time) - open) * SIGN(open - LAG(open) OVER (ORD
         AS estimate
   FROM MqlRatesDB
   WHERE (time >= STRFTIME('%s', '2015-01-01') AND time < STRFTIME('%s', '2021-01-
)
GROUP BY intraday, day
ORDER BY objective DESC
```

Первый SQL-запрос стал вложенным, из которого мы теперь аккумулируем данные внешним SQL-запросом. Группировку по всем сочетаниям времени и дня недели обеспечивает "довесок" *GROUP BY intraday, day*. Кроме того, мы добавили сортировку по целевому показателю (*ORDER BY objective DESC*), чтобы лучшие варианты оказались наверху таблицы.

Во вложенном запросе мы убрали параметр LIMIT, потому что количество групп стало приемлемым, значительно меньше количества анализируемых баров. Так, для H1 получим 120 вариантов (24*5).

Расширенный запрос помещен в текстовый файл *DBQuotesIntradayLagGroup.sql*, который в свою очередь подключен в виде ресурса в одноименный тестовый скрипт *DBQuotesIntradayLagGroup.mq5*. Его исходный код мало отличается от предыдущего, поэтому сразу покажем результат его запуска для диапазона дат по умолчанию: с начала 2015 год по начало 2021 (исключая 2021-й и 2022-й).

```
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
```

Таким образом, анализ подсказывает нам, что 16-часовой бар H1 в пятницу является лучшим кандидатом для торговли в продолжение тренда на основании предыдущего бара. Следующий по предпочтительности — 20-часовой бар в среду. И так далее.

Однако желательно проверить найденные настройки на форвард-периоде.

Для этого мы можем выполнить текущий SQL-запрос не только на "прошлом" диапазоне дат (у нас в тесте до 2021-го года), но и еще раз — в "будущем" (с начала 2021-го). Результаты обоих запросов следует объединить (JOIN) по нашим группам (*intraday*, *day*). Тогда при сохранении сортировки по целевому показателю мы увидим в соседних колонках прибыли и профит-фактор для тех же сочетаний времени и дня недели, и насколько они просели.

Приведем здесь окончательный SQL-запрос (в сокращенном варианте):

```
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-
   )
   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
```

Полный текст с запросом находится в файле *DBQuotesIntradayBackAndForward.sql*. Он подключен как ресурс в скрипте *DBQuotesIntradayBackAndForward.mq5*.

Запустив скрипт с настройками по умолчанию, получим такие показатели (с сокращениями):

```
 #|          objective    backtest_profit    backtest_PF intraday day forward_profit
--+------------------------------------------------------------------------------------
 1|   0.16713214428916     0.073200000001  1.46040631486 16:00:00 5   0.004920000048
 2|  0.118128291843983    0.0433099999995  1.33678071539 20:00:00 3   0.007880000055
 3|  0.103701251751617   0.00929999999853  1.14148790506 05:00:00 2   0.002210000082
 4|  0.102930330078208    0.0164399999973   1.1932071923 08:00:00 4   0.001409999969
 5|  0.089531492651001    0.0064300000006  1.10167615433 07:00:00 2  -0.009119999869
 6| 0.0827628326995007 -8.99999999970e-05 0.999601152226 17:00:00 4   0.009070000091
 7| 0.0823433025146974    0.0159700000012  1.21665988332 21:00:00 1    0.00250999999
 8| 0.0767938336191962   0.00522999999874  1.04226945769 13:00:00 1  -0.008490000055
 9| 0.0657741522256548    0.0162299999986  1.09699976093 15:00:00 2    0.01423999997
10| 0.0635243373432768   0.00932000000044  1.08294766820 22:00:00 3   -0.00456999993
...
```

Итак, торговая система с найденными лучшими торговыми расписаниями продолжает показывать прибыль в "будущем" периоде, хотя и не такую большую как на бэктесте.

Разумеется, рассмотренный пример лишь частный случай торговой системы. Мы могли бы, например, найти сочетания времени и дня недели, когда на соседних барах работает разворотная стратегия, или основанная вообще на других принципах (анализ тиков, календаря, портфеля торговых сигналов и т.д.).

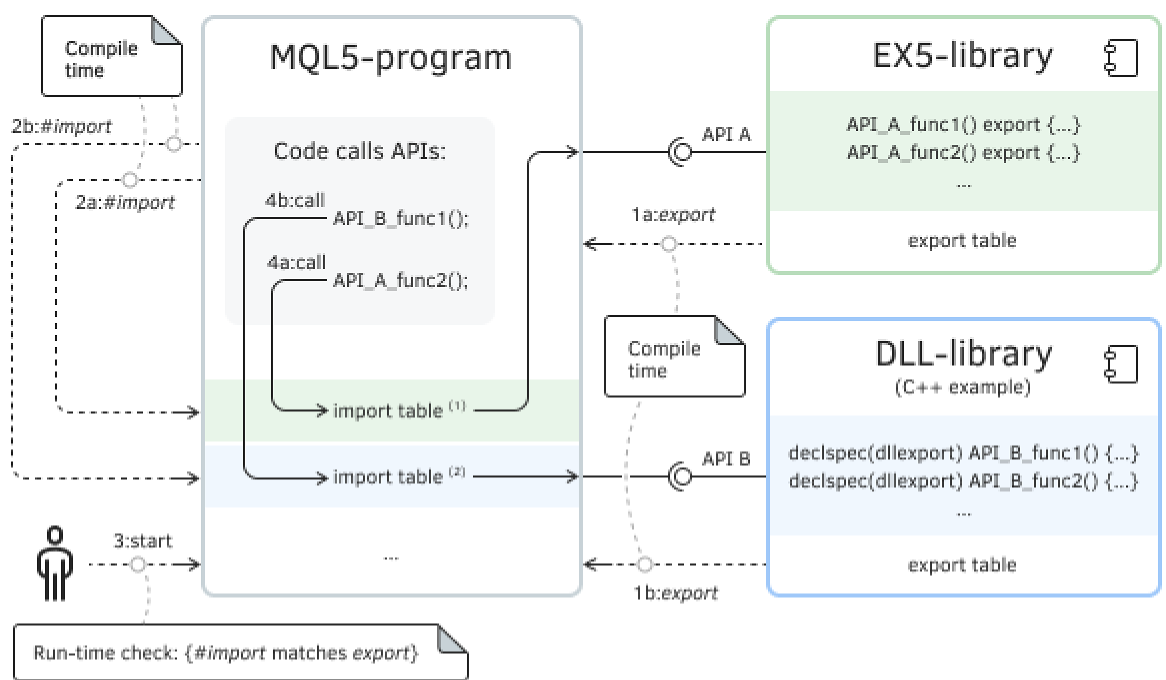
Суть заключается в том, что движок SQLite предоставляет многие удобные инструменты, которые потребовалось бы реализовывать на MQL5 самостоятельно. Правда, и изучение SQL требует времени. Платформа позволяет выбрать оптимальное сочетание двух технологий для эффективного программирования.

7.7 Разработка и подключение библиотек двоичных форматов

Помимо специализированных типов MQL-программ — *экспертов*, *индикаторов*, скриптов и сервисов — платформа MetaTrader 5 позволяет создавать и подключать независимые двоичные модули с произвольной функциональностью, откомпилированные в виде ex5-файлов или общеупотребительных dll-библиотек (Dynamic Link Library), стандартных для Windows. Это могут быть аналитические алгоритмы, графическая визуализация, сетевое взаимодействие с веб-сервисами, управление внешними программами или самой операционной системой. В любом случае подобные библиотеки работают в терминале не как самостоятельные MQL-программы, а в связке с программой какого-либо из вышеупомянутых 4-х типов.

Суть интеграции библиотеки и основной (родительской) программы заключается в том, что библиотека особым образом экспортирует, то есть объявляет доступными для использования извне, некие функции, а программа импортирует их прототипы. Именно описание прототипов — совокупность названий, списков параметров и возвращаемых значений — позволяет вызывать в коде эти функции, не имея их реализации.

Далее во время запуска MQL-программы выполняется раннее динамическое связывание — загрузка библиотеки вслед за основной программой и установление соответствий между импортированными прототипами и имеющимися в библиотеке экспортированными функциями. Установление однозначных соответствий по именам, спискам параметров и возвращаемым типам — обязательное условие успешной загрузки. Если для описания импорта хотя бы одной функции не удастся найти соответствующую экспортируемую реализацию, выполнение MQL-программы будет отменено (она завершится ошибкой на стадии запуска).



Коммуникационно-компонентная диаграмма MQL-программы с библиотеками

Выбирать подключаемую библиотеку при старте MQL-программы нельзя — эту связь задает разработчик при компиляции главной программы вместе с "импортами" библиотек. Однако пользователь может вручную подменить один библиотечный ex5/dll-файл на другой между стартами программы (при условии, что прототипы реализованных экспортируемых функций совпадают в библиотеках). Это можно использовать, например, для переключения языка пользовательского интерфейса, если в библиотеках содержатся строковые ресурсы с надписями. Однако чаще всего библиотеки применяются как коммерческий продукт с неким ноу-хау, которое автор не готов распространять в виде открытых заголовочных файлов.

Для программистов, пришедших в MQL5 из других сред и уже знакомых с технологией DLL, сделаем замечание относительно позднего динамического связывания, которое является одним из преимуществ DLL. Полноценное динамические подключение одной MQL-программы (или модуля DLL) к другой MQL-программе в процессе выполнения невозможно. Единственное похожее действие, которое MQL5 позволяет делать "на ходу" — связка эксперта и индикатора через *iCustom* или *IndicatorCreate*, где индикатор выступает в роли динамически подключаемой библиотеки (однако программное взаимодействие с ним придется осуществлять через API индикаторов, что означает повышенные накладные расходы на *CopyBuffer*, по сравнению с прямым вызовом функций через *export/#import*).

Отметим, что в обычных случаях, когда MQL-программа компилируется из исходных текстов без импорта внешних функций, используется статическое связывание, то есть генерируемый двоичный код напрямую обращается к вызываемым функциям, поскольку они известны в момент компиляции.

Строго говоря, библиотека также может полагаться на другие библиотеки, то есть может импортировать часть функций. В принципе, цепочка таких зависимостей может быть еще длиннее: например, MQL-программа подключает библиотеку A, библиотека A использует библиотеку B, а та, в свою очередь, библиотеку C, но такая практика не приветствуется, поскольку усложняет распространение и установку продукта, а также выявление причин

потенциальных проблем с запуском. Поэтому обычно библиотеки подключаются непосредственно к родительской MQL-программе.

В данной главе мы опишем процесс создания библиотек на MQL5, экспорт и импорт функций (включая ограничения на используемые в них типы данных), а также подключение внешних (готовых) DLL-библиотек. Разработка DLL-библиотек выходит за рамки данной книги.

7.7.1 Создание ex5-библиотек и *export* функций

Для описания библиотеки следует добавить в исходный код главного (компилируемого) модуля директиву #property library (обычно, в начало файла).

```
#property library
```

Указание этой директивы в любых других файлах, подключаемых в процесс компиляции через *#include*, эффекта не имеет.

Свойство *library* указывает компилятору, что данный ex5-файл является библиотекой: пометка об этом хранится в заголовке ex5-файла.

Для библиотек в MetaTrader 5 зарезервирована отдельная папка *MQL5/Libraries*. В ней можно организовать иерархию вложенных папок, как и для других типов программ в MQL5.

Библиотеки не участвуют напрямую в обработке событий, и потому компилятор не требует в коде наличия каких-либо стандартных обработчиков. Однако вы сможете вызывать экспортированные функции библиотеки из обработчиков событий той MQL-программы, к которой библиотека подключена.

Для экспорта функции из библиотеки достаточно пометить её специальным ключевым словом *export*. Данный модификатор должен располагаться в самом конце заголовка функции.

```
тип_результата идентификатор_функции ( [ тип_параметра идентификатор_параметра
                                        [ = значение_по_умолчанию] ...] ) export
{
   ...
}
```

Параметры должны быть простых типов или строками, структурами с полями таких типов или их массивами. Для объектных типов MQL5 допустимы указатели и ссылки (про ограничения при импорте DLL-библиотек см. соответствующий раздел).

Приведем несколько примеров. Параметр — простое число:

```
double Algebraic2(const double x) export
{
   return x / sqrt(1 + x * x);
}
```

Параметры — указатель на объект и ссылка на указатель (позволяет присвоить указатель внутри функции).

```
class X
{
public:
   X() { Print(__FUNCSIG__); }
};
void setObject(const X *obj) export { ... }
void getObject(X *&obj) export { obj = new X(); }
```

Параметр — структура:

```
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 { ... }
```

Экспортировать можно только функции, но не классы или структуры целиком. Частично данные ограничения можно обойти с помощью указателей и ссылок, что мы рассмотрим подробнее в дальнейшем.

Шаблоны функций нельзя объявлять с ключевым словом *export* и в директиве *#import*.

Модификатор *export* указывает компилятору внести функцию в таблицу экспортируемых функций внутри данного исполняемого ex5-файла. За счет этого такие функции становятся доступными ("видимыми") из других MQL-программ, где их можно использовать после импорта с помощью специальной директивы *#import*.

Все функции, которые планируется экспортировать, должны быть помечены модификатором export, хотя главная программа не обязана затем импортировать их все, а может — лишь необходимые.

Если забыть проэкспортировать какую-либо функцию, но включить её в директиву импорта в главной MQL-программе, то при запуске последней возникнет ошибка вида:

```
cannot find 'функция' in 'библиотека.ex5'
unresolved import function call
```

Аналогичная проблема возникнет, если в описании экспортированной функции и её импортируемом прототипе есть разночтения. Это может произойти, например, если забыть перекомпилировать библиотеку или главную программу после внесения изменений в программный интерфейс, который обычно описан в отдельном заголовочном файле.

> Отладка библиотек невозможна, поэтому при необходимости следует иметь вспомогательный скрипт или другую MQL-программу, которая собирается из исходных кодов библиотеки в режиме отладчика и может исполняться с точками остановки или пошагово. Разумеется, при этом потребуется эмулировать вызовы экспортируемых функций с использованием неких реальных или искусственных данных.
>
> Для dll-библиотек описание экспортируемых функций делается по-разному, в зависимости от языка программирования, на котором они создаются. Ищите подробности в документации избранных вами сред разработки.

Рассмотрим пример простой библиотеки *MQL5/Libraries/MQL5Book/LibRand.mq5*, из которой экспортируется несколько функций с различными типами параметров и результатов. Библиотека предназначена для генерации случайных данных:

- числовых данных с псевдо-нормальным распределением;
- строк со случайными символами из заданных наборов (может пригодиться для паролей).

В частности, получить одно случайное число можно с помощью функции *PseudoNormalValue*, в которой параметрами задаются матожидание и дисперсия.

```
double PseudoNormalValue(const double mean = 0.0, const double sigma = 1.0,
   const bool rooted = false) export
{
   // используем готовый sqrt при массовой генерации в цикле в PseudoNormalArray
   const double s = !rooted ? sqrt(sigma) : sigma;
   const double r = (rand() - 16383.5) / 16384.0; // [-1,+1] исключая границы
   const double x = -(log(1 / ((r + 1) / 2) - 1) * s) / M_PI * M_E + mean;
   return x;
}
```

Для заполнения массива случайными значениями в заданном количестве (*n*) и с нужным распределением реализована функция *PseudoNormalArray*.

```
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)); // передаем готовый sqrt при вызове PseudoNorm
   ArrayResize(array, n);
   for(int i = 0; i < n; ++i)
   {
      array[i] = PseudoNormalValue(mean, s, true);
      success = success && MathIsValidNumber(array[i]);
   }
   return success;
}
```

Для генерации одной случайной строки написана функция *RandomString*, которая "выбирает" из предоставленного набора символов (*pattern*) заданное количество (*length*) произвольных символов. Когда параметр *pattern* оставлен пустым (по умолчанию), подразумевается полный набор букв и цифр. Для его получения используются вспомогательные функции *StringPatternAlpha* и *StringPatternDigit*, которые также являются экспортируемыми (в книге не приводятся, см. исходный код).

```
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;
}
```

В целом, для работы с библиотекой необходимо опубликовать заголовочный файл с описанием всего, что в ней должно быть доступно извне (а подробности внутренней реализации можно и нужно скрывать). В нашем случае такой файл называется *MQL5Book/LibRand.mqh*. В частности, в нем описываются пользовательские типы (в нашем случае, перечисление STRING_PATTERN) и прототипы функций.

Хотя точный синтаксис блока *#import* нам еще не известен, это не должно сказываться на понятности деклараций внутри него: здесь повторяются заголовки экспортируемых функций, но уже без ключевого слова *export*.

```
enum STRING_PATTERN
{
   STRING_PATTERN_LOWERCASE = 1, // только строчные буквы
   STRING_PATTERN_UPPERCASE = 2, // только заглавные буквы
   STRING_PATTERN_MIXEDCASE = 3  // оба регистра
};

#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
```

Тестовый скрипт, использующий данную библиотеку, напишем в следующем разделе, после изучения директивы *#import*.

7.7.2 Подключение библиотек и *#import* функций

Импорт функций осуществляется из откомпилированных модулей MQL5 (файлы *.ex5) и из модулей динамических библиотек Windows (файлы *.dll). Имя модуля указывается в директиве *#import*, после чего следуют описания прототипов импортируемых функций. Такой блок должен заканчиваться еще одной директивой *#import*, причем она может быть без имени и просто закрывать собой блок, либо в директиве может быть указано имя другой библиотеки, и тем самым одновременно начинается следующей блок импорта. В конце серии блоков импорта всегда должна идти директива без имени библиотеки.

В простейшем случае директива выглядит следующим образом:

```
#import "[путь] имя_модуля [.расширение]"
   тип_функции имя_функции([список_параметров]);
   [тип_функции имя_функции([список_параметров]);]
   ...
#import
```

Имя файла библиотеки можно указывать без расширения: тогда по умолчанию предполагается dll-библиотека. Расширение *ex5* указывать обязательно.

Перед именем может идти путь размещения библиотеки. По умолчанию, если пути нет, библиотеки ищутся в папке *MQL5/Libraries* или в папке рядом с MQL-программой, куда подключена библиотека. В противном случае для поиска библиотек применяются разные правила в зависимости от типа — DLL или EX5. Эти правила освещены в отдельном разделе.

Вот пример последовательных блоков импорта из двух библиотек:

```
#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
```

При наличии таких директив импортируемые функции можно вызывать из исходного кода точно так же, как и функции, определенные непосредственно в самой MQL-программе. Все технические "сложности" с загрузкой библиотек и переадресации вызовов в сторонние модули берет на себя среда исполнения MQL-программ.

Для того чтобы компилятор мог правильно оформить вызов импортируемой функции и организовать передачу параметров, необходимо полное описание — с типом результата, со всеми параметрами, модификаторами и значениями по умолчанию, если они присутствуют в источнике.

Так как импортируемые функции находятся вне компилируемого модуля, компилятор не может проверить правильность передаваемых параметров и возвращаемых значений. Любое несовпадение формата ожидаемых и получаемых данных приведет к ошибке во время выполнения программы, причем это может проявляться как критическая остановка программы, так и непредвиденное поведение.

В случае если библиотека не смогла загрузиться или вызываемая импортируемая функция не была найдена, MQL-программа останавливает свою работу с соответствующим сообщением в журнале. Программа не сможет запускаться, пока проблема не будет решена, например, путем

модификации и перекомпиляции, размещения искомой библиотеки в одном из мест по пути поиска или разрешением на использование DLL (только для DLL-библиотек).

При совместном использовании нескольких библиотек (не важно — DLL или EX5) следует помнить, что они должны иметь разные имена вне зависимости от каталогов их размещения. Все импортируемые функции получают область видимости, совпадающую с именем файла библиотеки, то есть это своего рода пространство имен, неявно выделяемое под каждую подключенную библиотеку.

Импортируемые функции могут иметь любые имена, в том числе, совпадающие с именами встроенных функций (хотя это не рекомендуется). Более того можно одновременно импортировать из разных модулей функции с одинаковыми именами. В подобных случаях следует применять операцию разрешения контекста для определения того, какая из функций должна вызываться.

Например:

```
#import "kernel32.dll"
   int GetLastError();
#import "lib.ex5"
   int GetLastError();
#import

class Foo
{
public:
   int GetLastError() { return(12345); }
   void func()
   {
      Print(GetLastError());           // вызов метода класса
      Print(::GetLastError());         // вызов встроенной (глобальной) функции MQL5
      Print(kernel32::GetLastError()); // вызов функции из kernel32.dll
      Print(lib::GetLastError());      // вызов функции из lib.ex5
   }
};

void OnStart()
{
   Foo foo;
   foo.func();
}
```

Покажем простой пример скрипта *LibRandTest.mq5*, использующего функции из EX5-библиотеки, созданной в предыдущем разделе.

```
#include <MQL5Book/LibRand.mqh>
```

Во входных параметрах можно выбрать количество элементов в массиве чисел, параметры распределения, а также шаг гистограммы, которую мы посчитаем, чтобы убедиться в приблизительном соответствии распределения нормальному закону.

```
input int N = 10000;
input double Mean = 0.0;
input double Sigma = 1.0;
input double HistogramStep = 0.5;
input int RandomSeed = 0;
```

Инициализация встроенного в MQL5 генератора случайных чисел (равномерного распределения) производится значением *RandomSeed* или, если здесь оставлен 0, берется *GetTickCount* (новое при каждом запуске).

Для построения гистограммы используем *MapArray* и *QuickSortStructT* (мы с ними уже работали в разделах о мультивалютных индикаторах и о сортировке массивов, соответственно). В карте будут накапливаться счетчики попадания случайных чисел в ячейки гистограммы с шагом *HistogramStep*.

```
#include <MQL5Book/MapArray.mqh>
#include <MQL5Book/QuickSortStructT.mqh>
```

Для отображения гистограммы на основе карты нужно уметь сортировать карту в порядке значений-ключей. Для этого пришлось определить производный класс.

```
#define COMMA ,

template<typename K,typename V>
class MyMapArray: public MapArray<K,V>
{
public:
   void sort()
   {
      SORT_STRUCT(Pair<K COMMA V>, array, key);
   }
};
```

Обратите внимание, что макрос COMMA становится альтернативным представлением символа запятой ',' и используется при вызове другого макроса SORT_STRUCT. Если бы не эта подстановка, запятая внутри пары Pair<K,V> трактовалась бы препроцессором, как обычный разделитель параметров макроса, в результате чего на входе SORT_STRUCT получилось бы 4 параметра вместо ожидаемых 3-х — это вызвало бы ошибку компиляции. Препроцессор ничего не знает о синтаксисе MQL5.

В начале *OnStart*, после инициализации генератора, проверим получение одиночной случайной строки и массива строк разной длины.

```
void OnStart()
{
   const uint seed = RandomSeed ? RandomSeed : GetTickCount();
   Print("Random seed: ", seed);
   MathSrand(seed);
   
   // вызываем 2 библиотечных функции: StringPatternDigit и RandomString
   Print("Random HEX-string: ", RandomString(30, StringPatternDigit() + "ABCDEF"));
   Print("Random strings:");
   string text[];
   RandomStrings(text, 5, 10, 20);         // 5 строк длиной от 10 до 20 символов
   ArrayPrint(text);
   ...
```

Далее тестируем нормально-распределенные случайные числа.

```
   // вызываем еще одну библиотечную функцию: PseudoNormalArray
   double x[];
   PseudoNormalArray(x, N, Mean, Sigma);   // заполнили массив x
   
   Print("Random pseudo-gaussian histogram: ");
   
   // берем 'long' как тип ключей, т.к. 'int' уже использован для доступа по индексу
   MyMapArray<long,int> map;
   
   for(int i = 0; i < N; ++i)
   {
      // величина x[i] определяет ячейку гистограммы, где увеличиваем статистику
      map.inc((long)MathRound(x[i] / HistogramStep));
   }
   map.sort();                             // сортируем по ключу (т.е. по значению)
   
   int max = 0;                            // ищем максимум для нормировки
   for(int i = 0; i < map.getSize(); ++i)
   {
      max = fmax(max, map.getValue(i));
   }
   
   const double scale = fmax(max / 80, 1); // в гистограмме максимум 80 символов
   
   for(int i = 0; i < map.getSize(); ++i)  // печатаем гистограмму
   {
      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);
   }
```

Вот какой результат получился при запуске с настройками по умолчанию (рандомизация таймером — каждый запуск будет выбирать новый *seed*).

```
Random seed: 8859858
Random HEX-string: E58B125BCCDA67ABAB2F1C6D6EC677
Random strings:
"K4ZOpdIy5yxq4ble2" "NxTrVRl6q5j3Hr2FY" "6qxRdDzjp3WNA8xV"  "UlOPYinnGd36"      "6OCm
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)
```

В данной библиотеке мы экспортировали и импортировали только функции со встроенными типами. Однако гораздо интереснее и более востребованы с практической точки зрения объектные интерфейсы со структурами, классами и шаблонами. Про нюансы их применения в библиотеках мы поговорим в отдельном разделе.

При тестировании экспертов и индикаторов в тестере следует иметь в виду важный момент, связанный с библиотеками. Библиотеки, необходимые для основной тестируемой MQL-программы определяются автоматически из директив #import. Однако если из основной программы вызывается пользовательский индикатор, к которому подключена какая-либо библиотека, то необходимо в явном виде указать в свойствах программы, что она опосредованно зависит от конкретной библиотеки. Это делается с помощью директивы:

```
#property tester_library "путь_имя_библиотеки.расширение"
```

7.7.3 Порядок поиска файлов библиотек

Если имя библиотеки указано без пути или с относительным путем, поиск производится по различным правилам в зависимости от типа библиотеки.

Системные библиотеки (DLL) загружаются по правилам операционной системы. Если библиотека уже загружена (например, другим экспертом и даже из другого клиентского терминала, запущенного параллельно), то обращение идет к уже загруженной библиотеке. В противном случае поиск идет в следующей последовательности:

1. Папка, из которой была запущена откомпилированная EX5-программа, импортирующая DLL;
2. Папка *MQL5/Libraries*;
3. Папка, где находится запущенный терминал MetaTrader 5;
4. Системная папка (обычно внутри Windows);
5. Каталог Windows;
6. Текущая рабочая папка процесса терминала (может отличаться от папки размещения терминала);
7. Папки, перечисленные в системной переменной PATH.

В директивах *#import* не рекомендуется использовать полностью квалифицированное имя загружаемого модуля вида *Drive:/Directory/FileName.dll*.

Если библиотека DLL использует в своей работе другую DLL, то в случае отсутствия второй DLL первая не сможет загрузиться.

Поиск импортируемой библиотеки EX5 производится в следующей последовательности:

1. Папка запуска импортирующей EX5-программы;
2. Папка *MQL5/Libraries* конкретного экземпляра терминала;
3. Папка *MQL5/Libraries* в общей папке всех терминалов MetaTrader 5 (*Common/MQL5/Libraries*).

Перед загрузкой MQL-программы формируется общий список всех библиотечных модулей EX5, которые предполагается использовать как из самой программы, так и из библиотек из этого списка. Он называется списком зависимостей и может стать очень разветвленным "деревом".

Для EX5-библиотек терминалом также обеспечивается однократная загрузка многократно используемых модулей.

Вне зависимости от типа библиотеки, в каждом её экземпляре идет работа с собственными данными, относящимися к контексту вызвавшего эксперта, скрипта, сервиса или индикатора. Библиотеки не являются инструментом для разделяемого доступа к переменным или массивам MQL5.

Библиотеки EX5 и DLL выполняются в потоке вызывающего модуля.

Не предусмотрено штатных средств, чтобы выяснить в коде библиотеки, откуда она загружена.

7.7.4 Особенности подключения DLL-библиотек

В импортируемые из DLL функции нельзя передавать в качестве параметров:

- Классы (объекты и указатели на них);

- Структуры, содержащие динамические массивы, строки, классы, другие сложные структуры;
- Массивы строк или вышеперечисленных сложных объектов.

Все параметры простых типов передаются по значению, если явно не указано, что они передаются по ссылке. При передаче строки передается адрес буфера скопированной строки; если строка передается по ссылке, то в функцию, импортируемую из DLL, передается адрес буфера именно этой строки без копирования.

При передаче в DLL массива всегда (независимо от флага AS_SERIES) передается адрес начала буфера данных. Функция внутри DLL ничего не знает о флаге AS_SERIES, переданный массив является массивом неизвестной длины, для указания его размера нужен дополнительный параметр.

При описании прототипа импортируемой функции можно использовать параметры со значениями по умолчанию.

При импорте DLL-библиотек требуется дать разрешение на их использование в свойствах конкретной MQL-программы или в общих настройках терминала. В связи с этим в разделе Разрешения мы представляли скрипт *EnvPermissions.mq5*, в котором, в частности, имеется функция чтения содержимого системного буфера обмена Windows, использующая системные DLL-библиотеки. Там эта функция была приведена в факультативном порядке: её вызов был закомментирован, потому что мы еще не знали, как работать с библиотеками, но сейчас перенесем её в отдельный скрипт *LibClipboard.mq5*.

Запуск скрипта может инициировать запрос подтверждения у пользователя (так как по умолчанию DLL-библиотеки запрещены из соображений безопасности). При необходимости включите опцию в диалоге, на закладке с зависимостями.

Вместе с терминалом поставляются заголовочные файлы в каталоге *MQL5/Include/WinApi*, где уже прописаны директивы *#import* для востребованных системных функций, таких как работа с буфером обмена (*OpenClipboard*, *GetClipboardData*, *CloseClipboard*), управление памятью (*GlobalLock*, *GlobalUnlock*), окнами Windows и многие другие. Мы подключим лишь два файла — *winuser.mqh* и *winbase.mqh*. В них есть требуемые директивы импорта и, опосредовано — через подключение *windef.mqh*, макросы терминов Windows (HANDLE и 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
```

Кроме того, мы самостоятельно импортируем функцию *lstrcatW* из библиотеки *kernel32.dll*, потому что нас не устраивает её описание в *winbase.mqh*, предоставленное по умолчанию: тем самым у функции появляется второй прототип, подходящий для передачи в первом параметре значения PVOID.

```
#include <WinApi/winuser.mqh>
#include <WinApi/winbase.mqh>

#define CF_UNICODETEXT 13 // один из стандартных форматов обмена - текст Unicode
#import "kernel32.dll"
string lstrcatW(PVOID string1, const string string2);
#import
```

Суть работы с буфером обмена заключается в "захвате" доступа к нему с помощью *OpenClipboard*, после чего следует получить дескриптор данных (*GetClipboardData*), преобразовать его в адрес в памяти (*GlobalLock*) и, наконец, скопировать данные из системной памяти в свою переменную (*lstrcatW*). Далее выполняется освобождение занятых ресурсов в обратном порядке (*GlobalUnlock*, *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();
   }
}
```

Попробуйте скопировать в буфер обмена текст и затем запустить скрипт: в журнал должно вывестись содержимое буфера. Если в буфере находится изображение или другие данные, не имеющие текстового представления, результат окажется пустым.

Функции, импортируемые из DLL, подчиняются соглашению о связывании (linking) двоичных исполняемых файлов, принятому для функций Windows API. Для обеспечения такого соглашения в исходном тексте программ используются специфические для конкретного компилятора ключевые слова, такие как, например, *__stdcall* в C или C++. Данные правила связывания подразумевают следующее:

- Вызывающая функция (в нашем случае, MQL-программа) должна "видеть" прототип вызываемой (импортируемой из DLL) функции, для того чтобы правильно сложить параметры на стек;
- Вызывающая функция (в нашем случае, MQL-программа) складывает параметры на стек в обратном порядке, справа налево — именно в таком порядке импортируемая функция считывает переданные ей параметры;
- Параметры передаются по значению, за исключением тех, которые явно передаются по ссылке (в нашем случае, строк);
- Импортируемая функция, считывая переданные ей параметры, сама очищает стек.

Приведем еще один пример скрипта, использующего DLL, — *LibWindowTree.mq5*. Его задача — проход по дереву всех окон терминала и получение их имен классов (согласно регистрации в системе средствами WinApi) и заголовков. Под окнами здесь имеются в виду стандартные элементы интерфейса Windows, включающие также и элементы управления. Данная процедура может пригодиться для автоматизации работы с терминалом: эмуляции нажатия кнопок в окнах, переключения режимов, которые недоступны через MQL5 и так далее.

Для импорта требуемых системных функций подключим заголовочный файл *WinUser.mqh*, использующий *user32.dll*.

```
#include <WinAPI/WinUser.mqh>
```

Получить название класса окна и его заголовок можно с помощью функций *GetClassNameW* и *GetWindowTextW*: они вызываются в функции *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);
   }
}
```

Суффикс 'W' в названиях функций означает, что они предназначены для строк формата Unicode (2 байта на символ) — наиболее распространенного сегодня (суффикс 'A' для ANSI-строк имеет смысл использовать только для обратной совместимости со старыми библиотеками).

При наличии некоего начального дескриптора окна Windows проход вверх по иерархии его родительских окон обеспечивает функция *TraverseUp*: её работа основывается на системной функции *GetParent*. Для каждого найденного окна *TraverseUp* вызывает *GetWindowData* и выводит полученные имя класса и заголовок в журнал.

```
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;
}
```

Обход вглубь иерархии производится функцией *TraverseDown*: для перечисления дочерних окон применяется системная функция *FindWindowExW*.

```
HANDLE TraverseDown(const HANDLE w, const int level = 0)
{
   // запрашиваем первое дочернее окно (если есть)
   HANDLE child = FindWindowExW(w, NULL, NULL, NULL);
   while(child)          // цикл, пока есть дочерние окна
   {
      string clazz, title;
      GetWindowData(child, clazz, title);
      Print(StringFormat("%*s", level * 2, ""), "'", clazz, "' '", title, "'");
      TraverseDown(child, level + 1);
      // запрашиваем следующее дочернее окно
      child = FindWindowExW(w, child, NULL, NULL);
   }
   return child;
}
```

В функции *OnStart* найдем главное окно терминала за счет обхода окон вверх от дескриптора текущего графика, на котором запущен скрипт. А затем построим все дерево окон терминала.

```
void OnStart()
{
   HANDLE h = TraverseUp(ChartGetInteger(0, CHART_WINDOW_HANDLE));
   Print("Main window handle: ", h);
   TraverseDown(h, 1);
}
```

По идее мы можем искать требуемые окна по имени класса и/или заголовку, а потому главное окно можно было бы сразу получить, вызвав *FindWindowW*, поскольку его атрибуты известны.

```
   h = FindWindowW("MetaQuotes::MetaTrader::5.00", NULL);
```

Вот пример журнала (фрагмент):

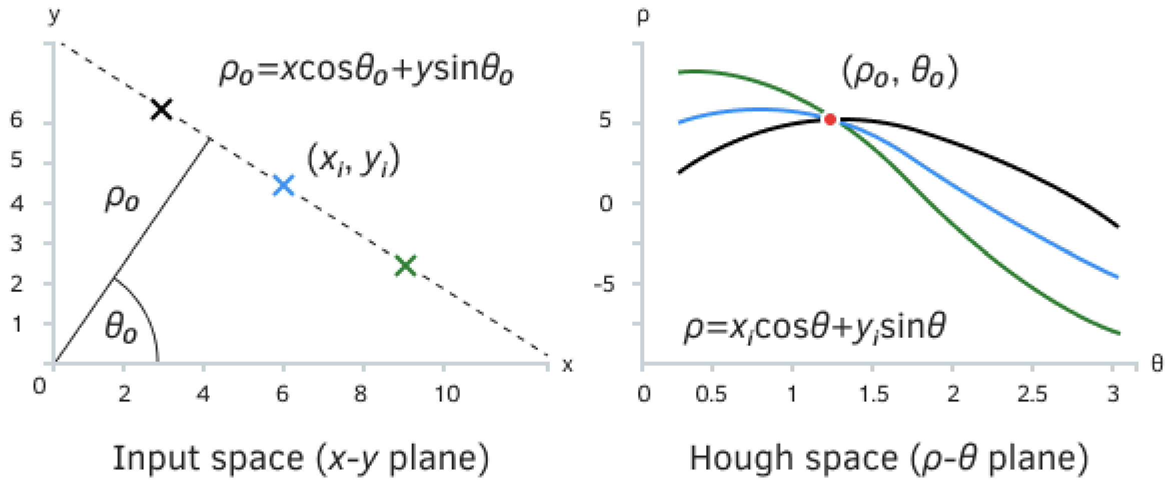
```
  'AfxFrameOrView140su' ''
  'Afx:000000013F110000:b:0000000000010003:0000000000000006:00000000000306BA' 'EURUSD,
  'MDIClient' ''
  'MetaQuotes::MetaTrader::5.00' '12345678 - MetaQuotes-Demo: Demo Account - Hedge - .
Main window handle: 263576
   'msctls_statusbar32' 'For Help, press F1'
   'AfxControlBar140su' 'Standard'
     'ToolbarWindow32' 'Timeframes'
     'ToolbarWindow32' 'Line Studies'
     'ToolbarWindow32' 'Standard'
   'AfxControlBar140su' 'Toolbox'
     'Afx:000000013F110000:b:0000000000010003:0000000000000000:0000000000000000' 'Tool
       'AfxWnd140su' ''
         'ToolbarWindow32' ''
...
   'MDIClient' ''
     'Afx:000000013F110000:b:0000000000010003:0000000000000006:00000000000306BA' 'EURU
       'AfxFrameOrView140su' ''
         'Edit' '0.00'
     'Afx:000000013F110000:b:0000000000010003:0000000000000006:00000000000306BA' 'XAUU
       'AfxFrameOrView140su' ''
         'Edit' '0.00'
     'Afx:000000013F110000:b:0000000000010003:0000000000000006:00000000000306BA' 'EURU
       'AfxFrameOrView140su' ''
         'Edit' '0.00'
```

7.7.5 Классы и шаблоны в библиотеках MQL5

Хотя экспорт и импорт классов и шаблонов в целом запрещен, разработчик может обойти данные ограничения за счет выноса описания абстрактных базовых интерфейсов в заголовочный файл библиотеки и передачи указателей. Проиллюстрируем данную концепцию примером библиотеки, которая выполняет преобразование Хафа для изображения.

Преобразованием Хафа называется алгоритм выделения особенностей изображения за счет сопоставления с некоторой формальной моделью (формулой), описываемой набором параметров.

Наиболее простым преобразованием Хафа является выделение на изображении прямых линий путем пересчета в полярные координаты. При такой обработке последовательности "закрашенных" пикселей, выстроенные более или менее в ряд, формируют в пространстве полярных координат пики на пересечении конкретного угла ("тета") наклона прямой и её сдвига ("ро") относительно центра координат.



Преобразование Хафа для прямых линий

Каждая из трех цветных точек на левом (исходном) изображении оставляет след в пространстве полярных координат (справа), потому что через точку можно провести бесконечное количество прямых линий под разными углами и перпендикулярами до центра. Каждый фрагмент следа "отмечается" только один раз, за исключением красной метки: в этом месте все три следа пересекаются и дают максимальный отклик (3). И действительно, как мы видим на исходном изображении, есть прямая линия, которая проходит через все три точки. Таким образом, два параметра прямой и выявляются максимумом в полярных координатах.

Мы можем использовать такое преобразование Хафа на графиках котировок для выделения альтернативных линий поддержки и сопротивления. Если обычно такие линии проводятся по отдельным экстремумам и, фактически, производят анализ выбросов, то линии преобразования Хафа могут учитывать все цены *High* или все цены *Low*, или даже распределение тиковых объемов внутри баров: все это позволяет получить более обоснованную оценку уровней.

Проработку начнем с заголовочного файла *LibHoughTransform.mqh*. Поскольку исходные данные для анализа поставляет некое абстрактное изображение, определим шаблон интерфейса *HoughImage*.

```
template<typename T>
interface HoughImage
{
   virtual int getWidth() const;
   virtual int getHeight() const;
   virtual T get(int x, int y) const;
};
```

Все, что нужно знать об изображении при его обработке, это его размеры и содержимое каждого пикселя, которое из соображений общности представлено параметрическим типом T. Понятно, что в простейшем случае это может быть *int* или *double*.

С вызовом аналитической обработки изображения все немного сложнее: для неё требуется описать в библиотеке класс, объекты которого мы станем возвращать из специальной фабричной функции (в виде указателей), и именно эта функция должна экспортироваться из библиотеки. Предположим, что так:

```
template<typename T>
class HoughTransformDraft
{
public:
   virtual int transform(const HoughImage<T> &image, double &result[],
      const int elements = 8) = 0;
};

HoughTransformDraft<?> *createHoughTransform() export { ... } // Проблема - шаблон!
```

Однако шаблонные типы и шаблонные функции нельзя экспортировать. Поэтому мы сделаем промежуточный нешаблонный класс *HoughTransform*, а в нем — шаблонный метод под параметр изображения. К сожалению, шаблонные методы не могут быть виртуальными, и потому выполним внутри метода диспетчеризацию вызовов вручную (с помощью *dynamic_cast*), переадресовывая обработку классу-наследнику с виртуальным методом.

```
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;
};
```

Внутреннюю реализацию класса *HoughTransformConcrete* напишем в файле библиотеки *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) { }
   ...
```

Поскольку мы собираемся пересчитывать точки изображения в пространство в новых — полярных — координатах, следует выделить под задачу некоторый размер. Уточним, что речь идет о дискретном преобразовании Хафа, поскольку мы и исходное изображение рассматриваем как дискретный набор точек (пикселей), и значения углов с перпендикулярами станем аккумулировать ячейками (квантами). Для простоты остановимся на варианте с квадратным пространством, где количество отсчетов и по углу, и по расстоянию до центра, равно. Этот параметр передается в конструктор класса.

```
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();
   }
   ...
```

Для подсчета статистики "следов", оставляемых "закрашенными" пикселями в преобразованном пространстве размера *size* на *size*, описан массив *data*. Для него использован вспомогательный

класс-шаблон *Plain2DArray* (с параметром-типом T), позволяющий эмулировать двумерный массив произвольных размеров. Тот же класс, но с параметром-типом *double*, применен для таблицы предварительно рассчитанных значений синусов и косинусов углов *trigonometric*: она потребуется для быстрого отображения пикселей в новое пространство.

Метод детектирования параметров наиболее заметных прямых линий называется *extract*. Он принимает на вход изображение и должен заполнить выходной массив *result* найденными парами параметров прямых линий. В уравнении:

```
y = a * x + b
```

параметр *a* (наклон, "тета") будет записываться по четным номерам массива *result*, а параметр **b** (отступ, "ро") — по нечетным. Например, первая, наиболее заметная, прямая после завершения работы метода описывается выражением:

```
y = result[0] * x + result[1];
```

Для второй прямой индексы увеличатся до 2 и 3, соответственно, и так далее, вплоть до максимально запрошенного количества линий (*lines*). Размер массива *result* равен удвоенному количеству прямых.

```
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 градусов, например
      const double rstep = MathSqrt(w * w + h * h) / size;
      ...
```

В блоке поиска прямых линий организованы вложенные циклы по пикселям изображения. Для каждой "закрашенной" (ненулевой) точки производится цикл по углам наклона, и соответствующие пары полярных координат помечаются в преобразованном пространстве. В данном случае мы просто вызываем метод увеличения содержимого ячейки на то значение, что вернул пиксель — *data.inc((int)r, i, v)*, но в зависимости от прикладной задачи и типа T может потребовать и более сложная обработка.

```
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); // диапазон [-size, +size]
         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);
      }
   }
}
...
```

Во второй части метода производится поиск максимумов в новом пространстве и заполнение выходного массива *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;
   }
```

Использованный здесь вспомогательный метод *findMax* (см. исходный код) записывает в переменные *x* и *y* координаты максимального значения в новом пространстве, дополнительно затирая окрестность этого места, чтобы не находить его снова и снова.

Имея готовый класс *LinearHoughTransform*, мы можем написать экспортируемую фабричную функцию для порождения объектов.

```
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;
}
```

Поскольку шаблоны запрещены при экспорте, мы используем перечисление ENUM_DATATYPE во втором параметре, чтобы варьировать тип данных в ходе преобразования и в представлении исходного изображения.

Чтобы проверить экспорт/импорт структур мы также описали структуру с мета-информацией о преобразовании в данной версии библиотеки и экспортировали функцию, возвращающую такую структуру.

```
struct HoughInfo
{
   const int dimension; // количество параметров в формуле модели
   const string about;  // словесное описание
   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];");
}
```

Дело в том, что различные модификации преобразований Хафа могут выявлять не только прямые линий, но и другие построения, отвечающие заданной аналитической формуле (например, окружности). Такие модификации будут выявлять другое количество параметров и нести другой смысл. Наличие самодокументирующей функции способно упростить интеграцию библиотек (особенно, когда их становится много; заметьте, что наш заголовочный файл содержит только информацию общего плана, относящуюся к любой библиотеке, реализующей данный интерфейс преобразования Хафа, а не только для прямых линий).

Конечно, данный пример экспорта класса с единственным публичным методом является несколько условным, потому что можно было бы экспортировать непосредственно функцию трансформации. Однако на практике классы, как правило, содержат больше функционала. В частности, и в наш класс легко добавить настройку чувствительности алгоритма, хранение образцовых паттернов из линий для детектирования сигналов, проверенных на истории, и так далее.

Задействуем библиотеку в индикаторе, рассчитывающем линии поддержки и сопротивления по ценам *High* и *Low* на заданном количестве баров. В принципе, благодаря преобразованию Хафа и программному интерфейсу, библиотека позволяет вывести по несколько наиболее важных таких линий.

Исходный код индикатора находится в файле *MQL5/Indicators/MQL5Book/p7/LibHoughChannel.mq5*. Он также подключает заголовочный файл *LibHoughTransform.mqh*, куда мы добавили директиву импорта.

```
#import "MQL5Book/LibHoughTransform.ex5"
HoughTransform *createHoughTransform(const int quants,
   const ENUM_DATATYPE type = TYPE_INT);
HoughInfo getHoughInfo();
#import
```

В анализируемом изображении обозначим пикселями положение специфических типов цен (OHLC) в котировках. Для реализации изображения потребуется описать класс *HoughQuotes*, производный от *HoughImage<int>*.

Предусмотрим "закрашивание" пикселей несколькими способами: внутри тела свечей, внутри полного размаха свечей, а также непосредственно в максимумах и минимумах. Всё это

формализовано в перечислении PRICE_LINE. Пока в индикаторе будут использованы только *HighHigh* и *LowLow*, но это можно вынести в настройки.

```
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
   };
   ...
```

В параметрах конструктора и внутренних переменных укажем диапазон баров для анализа. Количество баров *size* определяет размер изображение по горизонтали. Для простоты возьмем такое же количество отсчетов и по вертикали. Поэтому шаг дискретизации цен (*step*) равен фактическому размаху цен (*pp*) за эти *size* баров, деленному на *size*. Для переменной *base* вычислим нижнюю границу цен, которые подпадают под рассмотрение в указанных барах. Эта переменная потребуется для привязки построения линий на основе найденных параметров преобразования Хафа.

```
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);
   }
   ...
```

Напомним, что интерфейс *HoughImage* обязывает нас реализовать 3 метода: *getWidth*, *getHeight* и *get*. С первыми двумя все просто.

```
    virtual int getWidth() const override
    {
       return size;
    }

    virtual int getHeight() const override
    {
       return size;
    }
```

Метод *get* для получения "пикселей" на основе котировок возвращает 1, если указанная точка попадает внутрь диапазона бара или ячейки, согласно выбранному методу расчета из PRICE_LINE. В противном случае возвращается 0. Данный метод можно существенно усовершенствовать, если оценивать фракталы, последовательно увеличивающиеся экстремумы или "круглые" цены с более высоким весом (жирностью пикселей).

```
   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;
   }
```

Вспомогательный метод *convert* обеспечивает пересчет из пиксельных *y*-координат в ценовые значения.

```
   double convert(const double y) const
   {
      return base + y * step * _Point;
   }
};
```

Теперь все готово для написания технической части индикатора. Прежде всего объявим три входных переменных для выбора анализируемого фрагмента и количества линий. Все линии будут идентифицироваться по общему префиксу.

```
input int BarOffset = 0;
input int BarCount = 21;
input int MaxLines = 3;

const string Prefix = "HoughChannel-";
```

Объект, предоставляющий сервис по преобразованию, опишем как глобальный: именно здесь вызывается фабричная функция *createHoughTransform* из библиотеки.

```
HoughTransform *ht = createHoughTransform(BarCount);
```

В функции *OnInit* просто выведем в журнал описание библиотеки, воспользовавшись второй импортируемой функцией *getHoughInfo*.

```
int OnInit()
{
   HoughInfo info = getHoughInfo();
   Print(info.dimension, " per ", info.about);
   return INIT_SUCCEEDED;
}
```

Расчет в *OnCalculate* будем выполнять однократно на открытии бара.

```
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))
   {
      ... // см. следующий блок
      now = iTime(NULL, 0, 0);
   }
   return rates_total;
}
```

Сам расчет преобразования запускается дважды на паре изображений (*highs* и *lows*), сформированных по разным типам цен. При этом работу последовательно выполняет один и тот же объект *ht*. Если выявление прямых линий завершилось успешно, отображаем их на графике с помощью функции *DrawLine*. Поскольку линии перечислены в массиве результатов в порядке убывания важности, линиям назначается убывающая жирность.

```
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);
   }
}
```

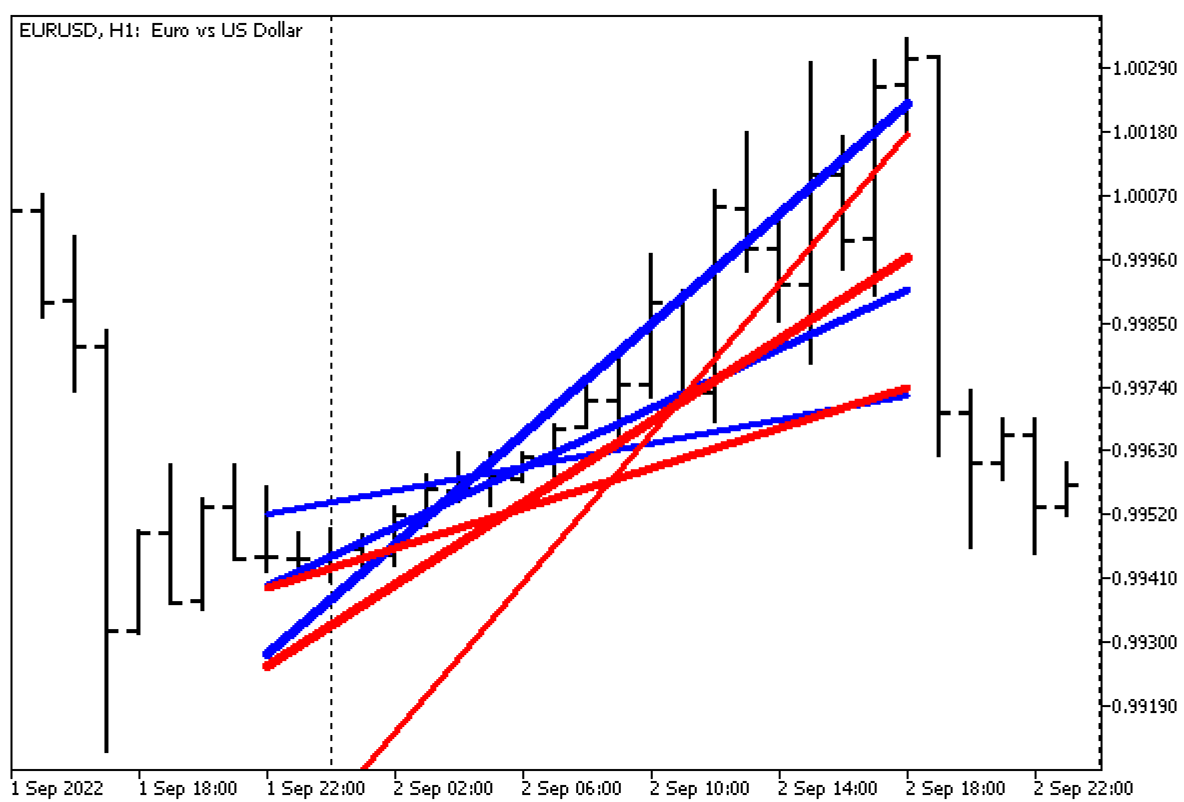
Функция *DrawLine* основана на трендовых графических объектах (OBJ_TREND, см. исходный код).

При деинициализации индикатора удаляем линии и аналитический объект.

```
void OnDeinit(const int)
{
   AutoPtr<HoughTransform> destructor(ht);
   ObjectsDeleteAll(0, Prefix);
}
```

Прежде чем тестировать новую разработку, не забываем откомпилировать и библиотеку, и индикатор.

Запуск индикатора с настройками по умолчанию дает примерно такую картину.



Индикатор с основными линиями по ценам High/Low на базе библиотеки с преобразованием Хафа

В нашем случае тест прошел успешно. Но что делать, если требуется отладка библиотеки? Встроенных средств для этого не предусмотрено, поэтому обычно применяется следующий прием. Исходный тест библиотеки включается в режиме условной компиляции в некую отладочную версию продукта, и продукт тестируется со встроенной библиотекой. Покажем это на примере нашего индикатора.

Предусмотрим макрос LIB_HOUGH_IMPL_DEBUG для включения внедрения исходника библиотеки непосредственно в индикатор. Макрос следует размещать перед включение заголовочного файла.

```
#define LIB_HOUGH_IMPL_DEBUG
#include <MQL5Book/LibHoughTransform.mqh>
```

В самом заголовочном файле обложим блок импорта из двоичной обособленной копии библиотеки инструкциями условной компиляции препроцессора. Когда макрос включен, будет работать другая ветка, с инструкцией *#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
```

В исходном файле библиотеки *LibHoughTransform.mq5*, внутри функции *getHoughInfo* добавим вывод в журнал информации о способе компиляции, в зависимости от включенного или отключенного макроса.

```
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];");
}
```

Если в коде индикатора, в файле *LibHoughChannel.mq5* раскомментировать инструкцию *#define LIB_HOUGH_IMPL_DEBUG*, вы можете попробовать пошагово выполнить анализ изображений.

7.7.6 Импорт функций из .NET библиотек

MQL5 предоставляет особый сервис для работы с функциями .NET библиотек — достаточно импортировать саму DLL без указания конкретных функций. MetaEditor автоматически импортирует все функции, с которыми возможна работа:

- простые структуры (POD, plain old data) – структуры, которые содержат только простые типы данных;
- публичные статические функции, в параметрах которых используются только простые типы и структуры POD или их массивы;

К сожалению, в данный момент нет средств, чтобы увидеть прототипы функций в том виде, как их распознал MetaEditor.

Например, пусть C# код функции *Inc* класса *TestClass* в библиотеке *TestLib.dll* выглядит следующим образом:

```
public class TestClass
{
   public static void Inc(ref int x)
   {
      x++;
   }
}
```

Тогда для её импорта и вызова достаточно написать:

```
#import "TestLib.dll"

void OnStart()
{
   int x = 1;
   TestClass::Inc(x);
   Print(x);
}
```

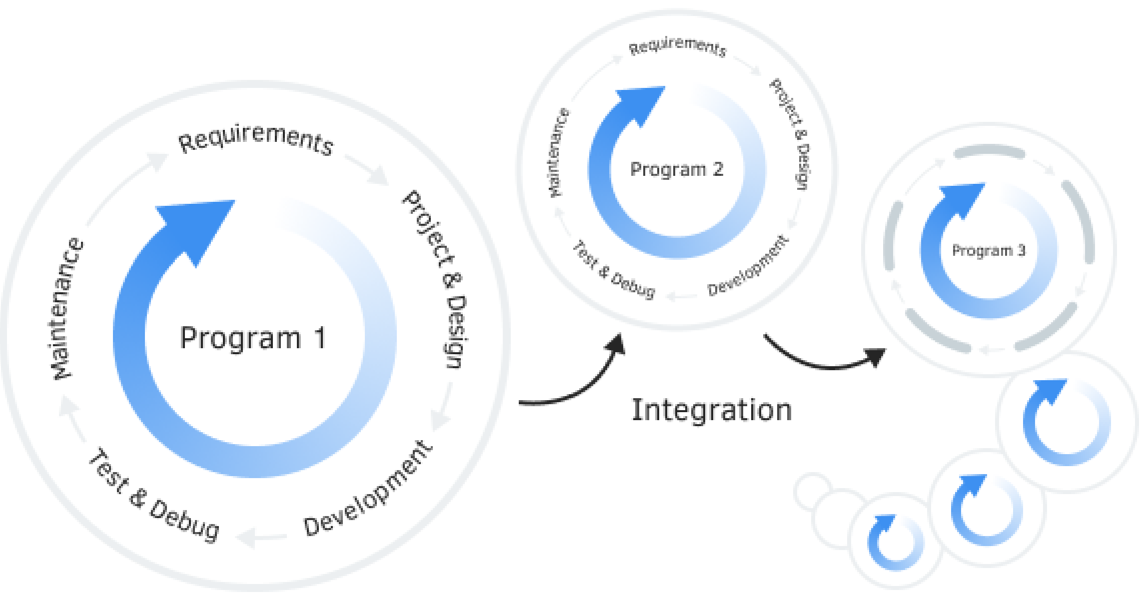
В результате выполнения скрипт вернет значение 2.

7.8 Проекты

Программные продукты, как правило, развиваются в рамках стандартного жизненного цикла:

- Сбор и дополнение требований
- Проектирование
- Разработка
- Тестирование
- Эксплуатация

В результате постоянного совершенствования и расширения функционала обычно возникает необходимость систематизации исходных файлов, ресурсов и используемых сторонних библиотек (здесь имеются в виду не только библиотеки двоичного формата, но и, в более общем смысле, любые наборы файлов, например, заголовочных). Более того, отдельные программы интегрируются в некий общий продукт, воплощающий прикладную идею.



Структура и жизненный цикл проекта

Например, при разработке торгового робота зачастую требуется подключение готовых или собственных индикаторов, использование внешних алгоритмов машинного обучения

подразумевает написание скрипта экспорта котировочных данных и скрипта для обратного импорта обученных моделей, а программы, связанные с обменом данными через Интернет (например, торговыми сигналами) могут требовать веб-сервера и его настройки на других языках программирования, хотя бы для отладки и тестирования, если не для разворачивания публичного сервиса.

Весь комплекс нескольких взаимосвязанных продуктов, вместе с их "зависимостями" (то есть использованными ресурсами и библиотеками, написанными самостоятельно или взятыми из сторонних источников), образуют программный проект.

При превышении программой некоего размера её удобное и эффективное развитие затруднено без специальных средств управления проектами. Это в полной мере относится и к программам на MQL5, поскольку многие трейдеры эксплуатируют комплексные торговые системы.

Поэтому с недавнего времени концепция проектов стала развиваться и в MetaEditor. Пока данный функционал находится в начале своего развития, и к моменту выхода книги, вероятно, изменится.

При работе с проектами на MQL5 следует учитывать, что на платформе термин "проект" применяется для двух разных сущностей:

- локального проекта в виде mqproj-файла;
- папки в облачном хранилище MQL5.

Локальный проект позволяет систематизировать, собрать воедино всю информацию об исходных кодах, ресурсах и настройках, необходимых для сборки конкретной MQL-программы. Такой проект находится только на вашем компьютере и может ссылаться на файлы из разных папок.

Файл с расширением *mqproj* имеет широко распространенный, универсальный, текстовый формат JSON (JavaScript Object Notation). Он удобен, прост и хорошо подходит для описания данных любой предметной области: вся информации группируется в объекты или массивы с именованными свойствами, с поддержкой значений разных типов. Все это делает JSON особенно близким по духу к ООП языкам, да и сам он родом из объектно-ориентированного JavaScript, как легко догадаться по названию.

Облачное хранилище функционирует на основе системы контроля версий и коллективной работы над ПО под названием SVN (Subversion). Здесь под проектом понимается папка верхнего уровня, внутри локальной папки *MQL5/Shared Projects*, причем для этой папки назначается одноименная папка, расположенная на сервере MQL5 Хранилища. Внутри папки проекта можно организовать иерархию вложенных папок. Как видно из названия, сетевые проекты можно "разделять" с другими разработчиками и делать вообще публичными (содержимое смогут скачивать любые желающие, зарегистрированные на mql5.com).

Система обеспечивает синхронизацию по запросу (с помощью специальных команд пользователя) между образом папки в облаке и на локальном диске, и наоборот — то есть можно как "подтягивать" чужие изменения проекта к себе на компьютер, так и "отправлять" в облако свои правки. Синхронизироваться может как полный образ папки, так и выборочные файлы, включая, разумеется, mq5-файлы, заголовочные mqh-файлы, мультимедиа, настройки (set-файлы), а также и mqproj-файлы. Более подробно об облачном хранилище читайте документацию MetaEditor и системы SVN.

> Важно отметить, что существование mqproj-файла не означает создание на его основе какого-либо облачного проекта, точно так же как и создание разделяемой папки не обязывает вас использовать mqproj-проект.

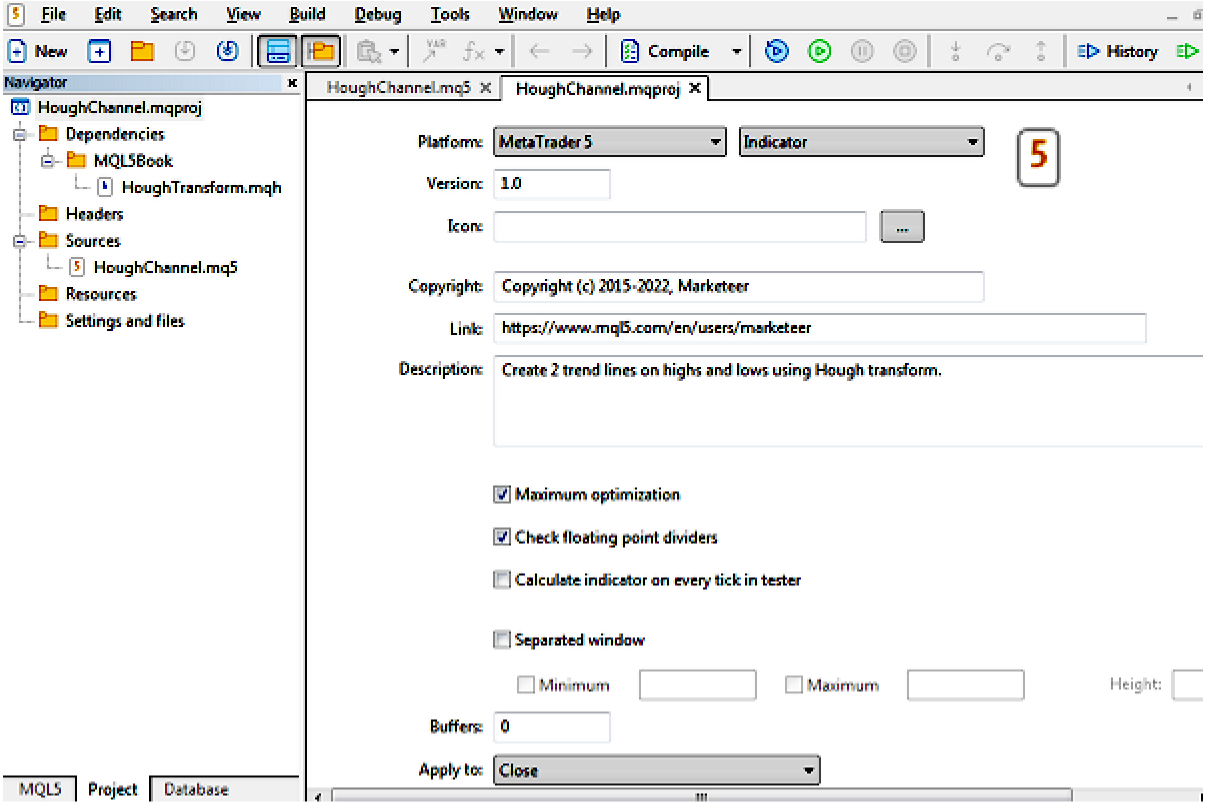
На момент написания книги mqproj-файл может описывать структуру только одной программы, но не нескольких, однако поскольку такое требование является частым при разработке сложных проектов, в будущем данный функционал, вероятно, будет добавлен в MetaEditor.

В данной главе мы опишем основные функции по созданию и организации mqproj-проектов и приведем серию примеров.

7.8.1 Общие принципы работы с локальными проектами

Локальный проект (mqproj-файл) можно создать из главного меню MetaEditor или из контекстного меню *Навигатора* командами *Новый проект* или *Новый проект из исходного файла*. В последнем случае файл следует предварительно выделить в *Навигаторе* или выбрать в открывшемся диалоге *Открыть* — в результате в проект сразу будет включен указанный mq5-файл. Первая из упомянутых команд запускает Мастер MQL, в котором следует выбрать тип программы или вариант пустого проекта (исходные файлы в него можно добавить позже). При выборе типа MQL-программы под проект, далее следуют привычные шаги Мастера по настройке свойств.

Проект содержит в себе несколько логических разделов — своего рода дерево (иерархию) со всеми компонентами — они отображаются в левой панели *Навигатора*, в отдельной закладке *Проект*.



Навигатор и свойства проекта индикатора

Сразу после создания проекта или затем по двойному щелчку мыши на корне дерева — в правой части окна открывается панель настройки свойств MQL-программы. Набор свойств меняется в зависимости от типа программы.

Большинство свойств соответствует каким-либо директивам *#property* в исходном коде, но имеет приоритет: если вы укажете свойства и в проекте, и в исходном коде, будут использоваться значения из проекта.

Некоторым разработчиком может понравиться устанавливать свойства интерактивно в диалоге, а не прописывать в исходном коде. Кроме того, вы можете использовать один и тот же mq5-файл в разных проектах и собирать версии MQL-программы с разными настройками (не меняя исходный код).

Некоторые свойства доступны только в проекте. К ним относятся, например, включение/отключение оптимизации компиляции и встроенные проверки на деление на ноль.

В процессе компиляции проекта автоматически анализируются зависимости, то есть включенные заголовочные файлы, ресурсы и так далее. Зависимости отображаются в различных ветвях иерархии проекта. В частности, заголовочные файлы из стандартных папок MQL5/Include, включенные в директивах *#include* с помощью угловых скобок (*<filename>*), попадают в *Dependecies*, а пользовательские заголовочные файлы, включенные с помощью двойных кавычек (*#include "filename"*) - в раздел *Headers*.

Дополнительно пользователь может добавить в проект файлы, которые относятся к законченному программному продукту и, возможно, требуются для его нормальной работы или демонстрации (например, файлы с обученными моделями нейронных сетей), но напрямую не встроены в исходный код. Для этих целей подойдет ветвь *Settings and files*. В её контекстном меню есть команды для добавления в проект одного файла или целиком какого-либо каталога.

В частности, мы далее рассмотрим примеры проектов, которые будут включать не только клиентские MQL-программы, но и серверную часть.

Команды *Новый файл* и *Новая папка* добавляют новый элемент в папку с файлом проекта: такие элементы всегда ищутся относительно самого проекта (в mqproj-файле они помечаются свойством *relative_to_project*, равным *true*, см. далее).

Команды *Добавить существующий файл* и *Добавить существующую папку* позволяют выбрать один или несколько элементов из существующей структуры каталогов внутри папки MQL5, причем на эти элементы внутри mqproj-файла делается ссылка относительно корневой MQL5 (свойство *relative_to_project* равно *false*).

Свойство *relative_to_project* — лишь одно из немногих, определенных разработчиками MetaTrader 5 для представления проекта в формате JSON. Напомним, что в результате редактирования проекта (иерархии и свойств) формируется mqproj-файл формата JSON.

Вот как выглядит этот файл для проекта на вышеприведенном изображении.

```
{
  "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
    }
  ]
}
```

О технических особенностях формата JSON мы поговорим более подробно в последующих разделах, потому что применим его и в своих демонстрационных проектах.

> Важно отметить, что все файлы, на которые ссылается проект, не хранятся внутри mqproj-файла, и потому копирование в новое место или перемещение на другой компьютер только файла проекта не обеспечит его восстановление. Чтобы иметь возможность переноса проекта, организуйте для него разделяемый проект и загрузите в облако все содержимое проекта. Однако это может потребовать реорганизации структуры локальной файловой системы, т.к. все компоненты должны находиться внутри папки разделяемого проекта, в то время как mqproj-формат этого не требует.

7.8.2 План проекта веб-сервиса копирования сделок и сигналов

В качестве сквозного демонстрационного проекта, который мы будем развивать на протяжении данной главы, возьмем простой, но вместе с тем довольно технологичный продукт: клиент-серверную систему копирования торговли. Клиентская часть, разумеется, будет представлять собой MQL-программы, общающиеся с центральной частью с помощью технологии сокетов. Учитывая, что MQL5 позволяет работать только с клиентскими сокетами, для сокет-сервера

потребуется выбрать альтернативную платформу (об этом чуть ниже). Таким образом, проект потребует симбиоза нескольких разных технологий и использования многих, уже изученных нами, разделов MQL5 API, включая и прикладные коды, разработанные на их основе.

Благодаря клиент-серверной архитектуре на основе сокетов, систему можно будет использовать по разным сценариям:

- для простого копирования сделок между терминалами на одном компьютере;
- для установления частного (личного) канала связи между терминалами на разных компьютерах, в том числе, не только в локальной сети, но и через интернет;
- для организации публично открытого или требующего регистрации закрытого сервиса сигналов;
- для мониторинга торговли;
- для удаленного управления своим собственным счетом.

Во всех случаях клиентские программы будут выступать в 2-х ролях: публикатора (издателя, отправителя) и подписчика (получателя) данных.

Мы не станем изобретать собственный сетевой протокол, а возьмем уже существующий и популярный стандарт WebSocket-ов. Их клиентская реализация встроена во все браузеры, и нам потребуется повторить её (с большей или меньшей степенью полноты) на MQL5. Разумеется, поддержка WebSocket-ов имеется и для большинства популярных веб-серверов. Поэтому наши наработки в любом случае можно будет не только адаптировать под другие сервера (если кому-то подойдет другой), но и интегрировать с известными площадками, предоставляющими аналогичные веб-сервисы. Здесь весь вопрос заключается в строгом следовании спецификации их API, надстроенного над WebSockets.

При разработке программных комплексов, более сложных, чем одна обособляемая программа, важно составить план действий и, возможно, даже технический проект — структуру модулей, их взаимодействие, последовательность кодирования.

Итак, наш план включает:

1. Теоретический разбор протокола WebSocket-ов;
2. Выбор и установку веб-сервера с реализацией сервера WebSocket-ов;
3. Создание простого эхо-сервера (отправляющего копию приходящих сообщений обратно клиенту) для знакомства с технологией;
4. Создание простой клиентской веб-страницы для проверки работоспособности эхо-сервера из браузера;
5. Создание простого чат-сервера, отправляющего сообщения всем подключившимся клиентам, и проверочной веб-страницы для него;
6. Создание сервера обмена сообщениями между идентифицируемыми поставщиками и подписчиками, и проверочного веб-клиента для него;
7. Проектирование и реализация WebSockets на MQL5;
8. Создание простого скрипта в качестве клиента для эхо-сервера;
9. Создание простого эксперта в качестве клиента чат-сервера;
10. Наконец, создание копировщика сделок на MQL5: он будет выступать и поставщиком информации (монитором изменений и состояния счета), и потребителем информации (воспроизводить трейды), в зависимости от настроек.

Но прежде чем приступать к реализации плана, необходимо установить веб-сервер.

7.8.3 Веб-сервер на основе nodejs

Для организации серверной части наших проектов нужен веб-сервер. Выберем в качестве него наиболее легкий и при том технологичный nodejs. Серверные скрипты для него можно писать на JavaScript, то есть на том же языке, что используется в браузерах для интерактивных веб-страниц. Это удобно с точки зрения унифицированного написания клиентской и серверной частей системы, а клиентская часть у любого веб-сервиса, как правило, рано или поздно требуется, например, для администрирования, регистрации и показа красивой статистики использования сервиса.

Тот, кто знает MQL5, почти знает и JavaScript. Основные отличия рассмотрены во врезке.

> **MQL5 vs JavaScript**
>
> JavaScript является интерпретируемым языком, в отличие от компилируемого MQL5. Для нас, как разработчиков это облегчает жизнь, потому что не нужна отдельная фаза компиляции, чтобы получить работающую программу. По поводу эффективности JavaScript не стоит беспокоиться: все среды исполнения JavaScript применяют прием компиляции JavaScript по требованию JIT(just-in-time) — при первом обращении к модулю. Этот процесс происходит автоматически, неявным образом, однократно за сеанс работы, после чего скрипт выполняется в откомпилированном виде.
>
> MQL5 относится к языкам со статической типизацией, то есть при описании переменных мы должны явно задать их тип, и компилятор следит за совместимостью типов. В отличие от этого, JavaScript — язык с динамической типизацией: тип переменной определяется тем, какое значение мы в неё положили, и может изменяться в процессе жизни переменной. С одной стороны это дает гибкость, но и требует осторожности, чтобы избежать непредвиденных ошибок.
>
> JavaScript является, в некотором смысле, более объектным языком, чем MQL5, потому что в нем объектами являются почти все сущности. Например, функция — тоже объект, и класс, как описатель свойств объектов, сам тоже — объект (прототипа).
>
> JavaScript самостоятельно "занимается" "уборкой мусора", то есть освобождает память, выделенную прикладной программой под объекты. В MQL5 мы должны следить за своевременным вызовом *delete* для динамических объектов.
>
> В синтаксис JavaScript заложено много удобных "сокращений" для записи конструкций, которые в MQL5 приходится реализовывать более длинным способом. Например, чтобы в MQL5 передать в некую функцию параметр, указывающий на другую функцию, нам потребуется описать тип такого указателя с помощью typedef, отдельно определить функцию, подходящую под этот прототип, и только затем передать её идентификатор в качестве параметра. В JavaScript можно определить указываемую функцию (целиком!) непосредственно в списке аргументов вместо параметра-указателя.

Если вы веб-разработчик или уже знакомы с nodejs, можете пропустить этапы установки и настройки.

Скачать nodejs следует с официального сайта nodejs.org. Установка доступна в разных вариантах, например, с помощью инсталлятора или распаковкой архива. В результате установки вы

получите в указанном каталоге исполняемый файл *node.exe* и несколько вспомогательных файлов и папок.

Если nodejs не был прописан в системном пути инсталлятором, это можно сделать для текущего пользователя Windows, выполнив следующую команду в той папке, куда установлен nodejs (где находится файл *node.exe*):

```
setx PATH "%CD%"
```

Альтернативно вы можете редактировать переменные среды Windows из диалога свойств системы (*Компьютер* -> *Свойства* -> *Дополнительные параметры* -> *Переменные среды* — конкретный вид диалогов зависит от версии операционной системы). В любом случае, мы таким образом обеспечим возможность запуска nodejs из любой папки на компьютере, что нам пригодится в дальнейшем.

Проверить работоспособность nodejs можно, выполнив команды (в командной строке Windows):

```
node -v
npm version
```

Первая выводит версию nodejs, а вторая — версию важного встроенного сервиса nodejs — менеджера пакетов npm.

Пакет — это готовый к использованию модуль, дополняющий nodejs конкретным функционалом. Сам по себе nodejs — очень небольшой, и без пакетов требовал бы много рутинного кодирования.

Наиболее востребованные пакеты хранятся в централизованном репозитарии в Интернете и могут быть скачаны и установлены в конкретную копию nodejs или глобально (для всех копий nodejs, если их несколько на машине). Установка пакета в конкретную копию выполняется такой командой:

```
npm install <package name>
```

Запускайте её в папке, куда производилась установка nodejs — эта команда разместит пакет локально и не затронет неожиданными правками другие копии nodejs, которые уже есть или, может быть, появятся на компьютере впоследствии.

Нам, в частности, потребуется пакет ws, реализующий протокол WebSocket-ов. То есть необходимо выполнить команду:

```
npm install ws
```

и дождаться завершения процесса. В результате мы должны обнаружить в папке *<путь_установки_nodejs>/node_modules/* новую вложенную папку *ws* с необходимым содержимым (вы можете заглянуть в файл README.md с описанием пакета, чтобы убедиться, что это библиотека WebSocket-протокола).

В принципе, пакет содержит реализации как сервера, так и клиента, но вместо последнего мы напишем свой на MQL5.

Весь функционал сервера nodejs сконцентрирован именно в папке */node_modules*. Её можно сравнить по назначению со стандартной папкой *MQL5/Include* в MetaTrader 5. При написании прикладных программ на JavaScript мы будем особым образом подключать, "импортировать" необходимые модули, по аналогии с включением заголовочных mqh-файлов с помощью директивы *#include* в MQL5.

7.8.4 Теоретические основы протокола WebSockets

Протокол WebSocket-ов строится поверх сетевых соединений TCP/IP, которые характеризуются IP-адресом (или заменяющим его доменным именем), а также номером порта. По такому же принципу работает и протокол HTTP/HTTPS, с которым мы уже практиковались в главе про сетевые функции. Там стандартными номерами портов были 80 (для незащищенных соединений) и 443 (для защищенных). Для WebSocket не существует выделенного номера порта, поэтому поставщики веб-сервиса могут выбрать любой свободный номер. Во всех наших примерах будет использоваться порт 9000.

При задании URL в качестве префиксов протокола WebSocket-ов применяются *ws* (для незащищенных соединений) и *wss* (для защищенных).

Формат WebSocket-ов более эффективен в плане передачи данных, чем HTTP, так как использует гораздо меньший объем управляющих данных.

Начальное установление соединение для WebSocket-сервиса полностью повторяет запрос веб-страницы HTTP/HTTPS: нужно отправить GET-запрос со специально подготовленными заголовками. Особенностью этих заголовков является наличие строк:

```
Connection: Upgrade
Upgrade: websocket
```

а также некоторых дополнительных, сообщающих версию протокола WebSocket-ов и специальные случайно сгенерированные строки — ключи, участвующие в процедуре "рукопожатия" (handshaking) клиента и сервера.

```
Sec-WebSocket-Key: ...
Sec-WebSocket-Version: 13
```

На практике "рукопожатие" заключается в проверке сервером доступности тех опций, которые запросил клиент, и в ответе со стандартными HTTP-заголовками, подтверждающими переключение в режим WebSocket-ов или отклоняющими его. Самой простой причиной отклонения может быть, если вы пытаетесь подключиться по протоколу WebSocket-ов к простому веб-серверу, где WebSocket-сервер не предусмотрен или не поддерживается нужная версия.

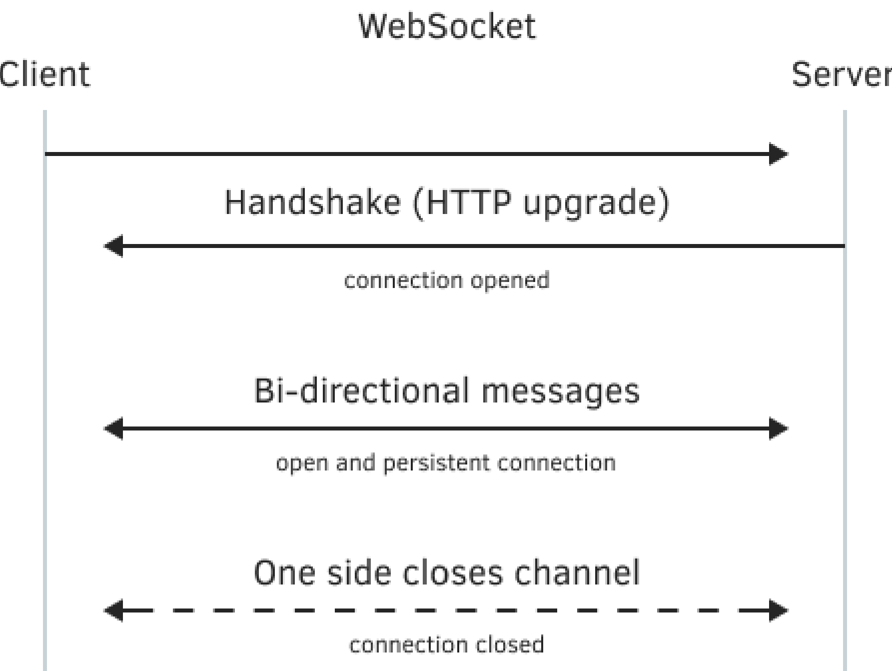
Актуальная версия протокола WebSockets известна под символическим именем Hybi и номером 13. Существующая более ранняя и простая версия Hixie может быть полезна для обеспечения обратной совместимости. Далее мы будем использовать только Hybi, хотя реализация Hixie также прилагается.

Успешное подключение обозначается такими HTTP-заголовками в ответе сервера:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: ...
```

Поле Sec-WebSocket-Accept здесь рассчитывается и заполняется сервером на основе ключа Sec-WebSocket-Key, чтобы подтвердить соответствие протоколу. Все это регламентировано спецификацией RFC6455 и будет поддержано также и в наших MQL-программах.

Для наглядности процедура показана на следующем изображении:



Взаимодействие клиента и сервера по протоколу WebSocket-ов

После установления WebSocket-соединения клиент и сервер могут обмениваться информацией, упакованной в специальные блоки: фреймы и сообщения. Сообщение может состоять из одного или нескольких фреймов. В принципе, размер фрейма, согласно спецификации, ограничен астрономическим числом 2^{63} байтов (9223372036854775807 ~ 9.22 эксабайт!), но конкретные реализации, разумеется, могут иметь более приземленные ограничения, поскольку данный теоретический лимит не кажется практичным для пересылки одним пакетом.

В любой момент клиент или сервер могут разорвать соединение, предварительно "вежливо попрощавшись" (см. ниже) или просто закрыв сетевой сокет.

Фреймы бывают разных типов, что задается в их заголовке (длиной от 4 до 16 байтов), который идет в начале каждого фрейма. Перечислим для справки операционные коды (они присутствуют в первом байте заголовка) и назначение фреймов разных типов.

- 0 — фрейм продолжения (наследует свойства предыдущего фрейма);
- 1 — фрейм с текстовой информацией;
- 2 — фрейм с двоичной информацией;
- 8 — фрейм запроса закрытия и подтверждения закрытия соединения (посылаются для "вежливого прощания");
- 9 — пинг-фрейм, может периодически посылаться любой из сторон, чтобы убедиться в физическом сохранении соединения;
- 10 — понг-фрейм, отправляется в ответ на пинг-фрейм.

Последний фрейм в сообщении помечается особым битом в заголовке. Разумеется, когда сообщение состоит из одного фрейма, он же является и последним. Также в заголовке передается длина полезных данных.

7.8.5 Серверная часть веб-сервисов на базе WebSocket-протокола

Для организации общей серверной части всех проектов заведем отдельную папку *Web* внутри *MQL5/Experts/MQL5Book/p7/*. В идеале было бы удобно расположить *Web* как подпапку в *Shared Projects*. Дело в том, что *MQL5/Shared Projects* имеется в стандартной поставке MetaTrader 5 и зарезервирована для проектов облачного хранилища. Поэтому впоследствии, за счет использования функционала разделяемых проектов, можно было бы загрузить все файлы наших проектов на сервер (не только веб-файлы, но и MQL-программы).

Позднее, когда мы создадим mqproj-файл с клиентскими программами на MQL5, мы добавим все файлы в данной папке в раздел проекта *Settings and files*, так как все эти файлы составляют неотъемлемую часть проекта — серверную часть.

Поскольку под сервер проекта выделен отдельный каталог, нужно обеспечить в нем доступность "импорта" модулей из nodejs. По умолчанию, nodejs "ищет" модули в подпапке */node_modules* текущего каталога, а запускать сервер мы будем из проекта. Поэтому, находясь в папке, где мы разместим веб-файлы проекта, выполните команду:

```
mklink /j node_modules {диск:/путь/к/папке/nodejs}/node_modules
```

В результате, внутри нашего проекта появится "символическая" ссылка-каталог под названием *node_modules*, указывающая на исходную одноименную папку в установленном nodejs.

Самым простым способом проверить работоспособность WebSocket-ов считается эхо-сервис. Принцип его действия: вернуть любое полученное сообщение обратно отправителю. Рассмотрим, как можно было бы организовать такой сервис в минимальной конфигурации. Пример прилагается в файле *wsintro.js*.

Первым делом подключаем пакет (модуль) *ws*, который предоставляет функционал WebSocket-ов для nodejs, и который мы установили вместе с веб-сервером.

```
// JavaScript
const WebSocket = require('ws');
```

Функция *require* работает аналогично директиве *#include* MQL5, но дополнительно возвращает объект модуля с программным интерфейсом всех файлов пакета *ws*. Благодаря этому мы можем вызывать методы и свойства объекта *WebSocket*. В данном случае нам требуется создать WebSocket-сервер на порту 9000.

```
// JavaScript
const port = 9000;
const wss = new WebSocket.Server({ port: port });
```

Здесь мы видим привычный по MQL5 вызов конструктора оператором *new*, но в качестве параметра передается безымянный объект (структура), в котором как в карте может храниться набор поименованных свойств и их значений. В данном случае используется только одно свойство *port*, и его значение устанавливается равным переменной (точнее, константе) *port*, описанной выше. В принципе, мы можем передавать номер порта (и другие настройки) в командной строке при запуске скрипта.

Объект сервера попадает в переменную *wss*. При успешном выполнении мы сигнализируем в окно командной строки о том, что сервер работает (ожидает подключений).

```
// JavaScript
console.log('listening on port: ' + port);
```

Вызов *console.log*, как легко понять, аналогичен привычным *Print*-ам в MQL5. По ходу отметим, что строки в JavaScript можно заключать не только в двойные, но и в одинарные кавычки, и даже в "косые" кавычки `` `this is a ${template} text` ``, которые добавляют кое-какие полезные возможности.

Далее назначим для объекта *wss* обработчик события "connection", то есть о подключении нового клиента. Очевидно, что перечень поддерживаемых событий объекта определен разработчиками пакета, в данном случае, используемого нами пакета *ws*. Все это отражено в документации.

Привязка обработчика производится методом *on*, в котором указывается название события и сам обработчик.

```
// JavaScript
wss.on('connection', function(channel)
{
   ...
});
```

Обработчик представляет собой безымянную (анонимную) функцию, определенную непосредственно в том месте, где ожидается параметр-ссылка для обратного вызова кода, который следует выполнить при новом соединении. Функция сделана анонимной, потому что используется только здесь, а JavaScript позволяет делать такие упрощения в синтаксисе. У функции — единственный параметр — объект нового соединения. Имя для параметра мы вольны выбирать сами, и в данном случае это — *channel*.

Внутри обработчика следует установить другой обработчик — на этот раз для события "message" о приходе нового сообщения в конкретном канале.

```
// JavaScript
   channel.on('message', function(message)
   {
      console.log('message: ' + message);
      channel.send('echo: ' + message);
   });
   ...
```

Здесь также используется анонимная функция с единственным параметром — объектом полученного сообщения. Мы также выводим его в журнал консоли для отладки. Но самое главное происходит во второй строке: вызовом *channel.send* мы отправляем ответное сообщение клиенту.

Для полноты картины добавим в обработчик "connection" отправку своего приветственного сообщения. Целиком это выглядит так:

```
// 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!');
});
```

Важно понимать, что, хотя привязка обработчика "message" сделана выше по коду, чем отправка "приветствия", обработчик сообщений будет вызван позже, и только при условии, что клиент пришлет сообщение.

Мы рассмотрели набросок скрипта для организации эхо-сервиса. Однако нам желательно его протестировать. Наиболее просто и быстро это можно сделать с помощью обычного браузера, но для этого потребуется слегка усложнить скрипт: превратить его в минимально-возможный веб-сервер, отдающий веб-страницу с минимально-возможным WebSocket-клиентом.

Эхо-сервис и тестовая веб-страница

Скрипт эхо-сервера, который мы сейчас рассмотрим, находится в файле *wsecho.js*. Одним из основных моментов является то, что на сервере желательно поддержать не только открытые протоколы *http/ws*, но и защищенные *https/wss*. Такая возможность будет обеспечена во всех наших примерах (включая и клиенты на MQL5), но на сервере для этого необходимо выполнить кое-какие действия.

Начать следует с пары файлов, содержащих ключи шифрования и сертификаты. Файлы обычно получают от авторизованных источников, удостоверяющих центров, но для ознакомительных целей можно сгенерировать файлы самостоятельно. Их, разумеется, нельзя использовать на публичный серверах, и в любом браузере страницы с подобными сертификатами будут вызывать предупреждения (значок страницы слева от адресной строки подсвечивается красным).

Описание устройства сертификатов и процесса их генерации собственными силами выходит за рамки книги, но с книгой поставляются два готовых файла *MQL5Book.crt* и *MQL5Book.key* (бывают и другие расширения) с ограниченным сроком действия. Эти файлы нужно передать в конструктор объекта веб-сервера, чтобы сервер заработал по протоколу HTTPS.

Имя файлов сертификата мы будем передавать в командной строке запуска скрипта. Например, так:

```
node wsecho.js MQL5Book
```

Если запустить скрипт без дополнительного параметра, сервер будет работать по протоколу HTTP.

```
node wsecho.js
```

Внутри скрипта аргументы командной строки доступны через встроенный объект *process.argv*, причем первые два аргумента всегда содержат, соответственно, имя самого сервера *node.exe* и имя запускаемого скрипта (в данном случае, *wsecho.js*), поэтому мы их отбрасываем методом *splice*.

```
// JavaScript
const args = process.argv.slice(2);
const secure = args.length > 0 ? 'https' : 'http';
```

В зависимости от наличия имени сертификата переменная *secure* получает название пакета, который следует далее загрузить для создания сервера: *https* или *http*. Всего у нас в коде 3 зависимости:

```
// JavaScript
const fs = require('fs');
const http1 = require(secure);
const WebSocket = require('ws');
```

Про пакет *ws* мы уже все знаем, пакеты *https* или *http* предоставят реализацию веб-сервера, а встроенный пакет *fs* обеспечивает работу с файловой системой.

Настройки веб-сервера оформлены в виде объекта *options*. Здесь мы видим, как в строках с косыми кавычками с помощью выражения *${args[0]}* подставляется имя сертификата из командной строки. Затем соответствующая пара файлов читается методом *fs.readFileSync*.

```
// JavaScript
const options = args.length > 0 ?
{
   key : fs.readFileSync(`${args[0]}.key`),
   cert : fs.readFileSync(`${args[0]}.crt`)
} : null;
```

Веб-сервер создается вызовом метода *createServer*, в который передается объект опций, а также анонимная функция — обработчик HTTP-запросов. У обработчика имеется два параметра: объект *req* с HTTP-запросом, и объект *res*, с помощью которого мы должны послать ответ (HTTP-заголовки и веб-страницу).

```
// 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);
```

Главная индексная страница (и единственная) называется у нас, как принято, *index.htm* (её сейчас предстоит написать). Кроме неё обработчик "умеет" "отдавать" js-файлы и css-файлы, что пригодится нам в дальнейшем. В зависимости от включения защищенного режима, сервер запускается вызовом метода *listen* на стандартных портах 443 или 80 (измените на другие, если эти уже заняты на вашем компьютере).

Чтобы принимать соединения на порту 9000 для веб-сокетов нам требуется "поднять" еще один экземпляр веб-сервера с такими же опциям. Но в данном случае сервер предназначен для единственной цели — обработать HTTP-запрос об "апгрейде" соединения до протокола веб-сокетов.

```
// JavaScript
const server = new http1.createServer(options).listen(9000);
server.on('upgrade', function(req, socket, head)
{
   console.log(req.headers); // TODO: можем добавить авторизацию!
});
```

Здесь в обработчике события "upgrade" мы принимаем любые соединения, которые уже прошли "рукопожатие", и выводим заголовки в лог, но потенциально мы могли бы запросить авторизацию пользователя, если бы делали закрытый (платный) сервис.

Наконец, мы создаем объект WebSocket-сервера, как в предыдущем ознакомительном примере, с единственным отличием, что в конструктор передается уже готовый веб-сервер. Все подключающиеся клиенты подсчитываются и приветствуются по порядковому номеру.

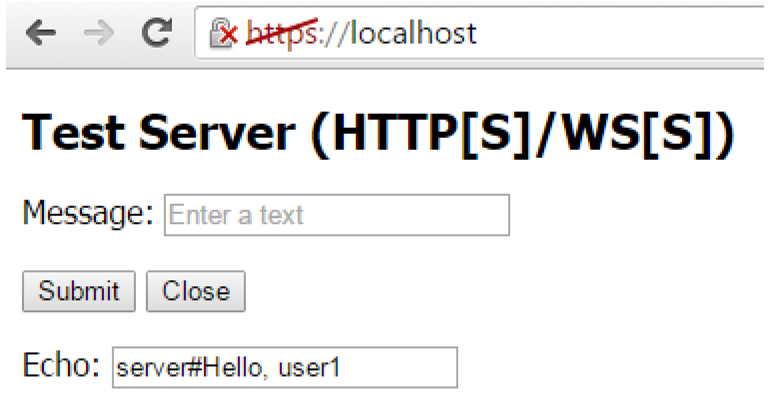
```
// JavaScript
var count = 0;

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);
   });
});
```

Для всех событий — подключение, отключение и сообщение — в консоль выводится отладочная информация.

Что ж, веб-сервер с поддержкой веб-сокет-сервера готов. Осталось создать для него клиентскую веб-страницу *index.htm*.

```html
<!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>
```

Как нетрудно понять, страница представляет собой форму с единственным полем ввода и кнопкой для отправки сообщения.



Веб-страница эхо-сервиса на WebSocket

Страница использует скрипт *wsecho_client.js*, который обеспечивает клиентскую реакцию веб-сокетов. В браузерах веб-сокеты встроены как "родные" объекты JavaScript, поэтому ничего внешнего подключать не надо: достаточно вызвать конструктор *WebSocket* с нужным протоколом и номером порта.

```javascript
// JavaScript
const proto = window.location.protocol.startsWith('http') ?
              window.location.protocol.replace('http', 'ws') : 'ws:';
const ws = new WebSocket(proto + '//' + window.location.hostname + ':9000');
```

URL формируется из адреса текущей веб-страницы (*window.location.hostname*), поэтому веб-сокет-соединение устанавливается с тем же сервером.

Далее объект *ws* позволяет реагировать на события и отправлять сообщения. В браузере событие открытия соединения называется "open" и подключается через свойство *onopen*. Этот же, слегка

отличный от серверной реализации, синтаксис применяется и для события прихода нового сообщения — обработчик для него присваивается свойству *onmessage*.

```
// JavaScript
ws.onopen = function()
{
   console.log('Connected');
};

ws.onmessage = function(message)
{
   console.log('Message: %s', message.data);
   document.getElementById('echo').value = message.data;
};
```

Текст входящего сообщения отображается в элементе формы с идентификатором "echo". Обратите внимание, что объект события сообщения (параметр обработчика) не есть само сообщение, которое доступно в свойстве *data*. Это особенность реализации в JavaScript.

Реакция на кнопки формы назначается с помощью метода *addEventListener* для каждого из двух объектов-тегов *button*. Здесь мы видим еще один способ описания анонимной функции в JavaScript: круглые скобки со списком аргументов, который может быть пустым, и тело функции после стрелки — *(arguments) => { ... }*.

```
// JavaScript
const button = document.querySelectorAll('button'); // запрашиваем все кнопки
// кнопка "отправить"
button[0].addEventListener('click', (event) =>
{
   const x = document.getElementById('message').value;
   if(x) ws.send(x);
});
// кнопка "закрыть"
button[1].addEventListener('click', (event) =>
{
   ws.close();
   document.getElementById('echo').value = 'disconnected';
   Array.from(document.getElementsByTagName('button')).forEach((e) =>
   {
      e.disabled = true;
   });
});
```

Для отправки сообщений вызываем метод *ws.send*, и для закрытия соединения — метод *ws.close*.

На этом разработка первого примера клиент-серверных скриптов для демонстрации эхо-сервиса закончена. Вы можете запустить *wsecho.js* одной из показанных ранее команд и затем открыть в своем браузере страницу по адресу *http://localhost* или *https://localhost* (в зависимости от настроек сервера). После появления формы на экране попробуйте переписываться с сервером и убедитесь, что сервис работает.

Постепенно усложняя данный пример, мы подготовим почву и для веб-сервиса копирования торговых сигналов. Но следующим шагом будет чат-сервис, принцип работы которого

напоминает сервис торговых сигналов: сообщения одного пользователя передаются другим пользователям.

Чат-сервис и тестовая веб-страница

Новый серверный скрипт называется *wschat.js*, и он во многом повторяет *wsecho.js*. Перечислим основные отличия. В обработчике HTTP-запросов веб-сервера поменяем начальную страницу с *index.htm* на *wschat.htm*.

```
// JavaScript
http1.createServer(options, function (req, res)
{
   if(req.url == '/') req.url = "wschat.htm";
   ...
});
```

Для хранения информации о подключившихся к чату пользователях опишем массив-карту *clients*. *Map* — это стандартный ассоциативный контейнер JavaScript, в который по ключам произвольного типа можно записывать произвольные значения, в том числе и объекты.

```
// JavaScript
const clients = new Map();                    // добавили эту строку
var count = 0;
```

В обработчике события подключения нового пользователя будем добавлять объект *client*, получаемый как параметр функции, в карту под текущим порядковым номером клиента.

```
// 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);                // добавили эту строку
   ...
```

Напомним, что внутри функции *onConnect* мы устанавливаем обработчик для события о приходе нового сообщения для конкретного клиента, и именно внутри вложенного обработчика производим рассылку сообщений. Только на этот раз пробегаемся в цикле по всем элементам карты (то есть по всем клиентам) и отправляем текст каждому из них. Цикл организован с помощью вызова метода *forEach* для массива из карты, причем в метод передается по месту очередная анонимная функция, которая будет выполняться для каждого элемента (*elem*). На примере данного цикла вновь наглядно демонстрируется преобладающая в JavaScript *функционально-декларативная* парадигма программирования (отличная от *императивного* подхода MQL5).

```
// JavaScript
   client.on('message', function(message)
   {
      console.log('%d : %s', client.id, message);
      Array.from(clients.values()).forEach(function(elem) // добавили цикл
      {
         elem.send('user' + client.id + '#' + message);
      });
```

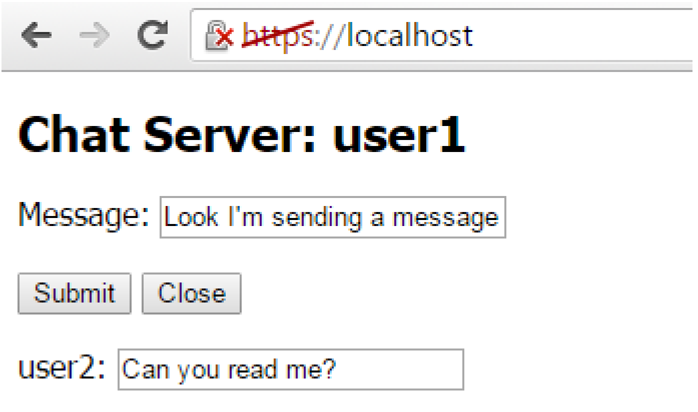
```
   });
```

Важно отметить, что мы отправляем копию сообщения всем клиентам, включая и автора, изначально написавшего его. Его можно было бы отфильтровать, но для целей отладки лучше иметь подтверждение отправки сообщения.

Последним отличием от предыдущего эхо-сервиса является то, что при отключении клиента требуется удалить запись о нем из карты.

```
// JavaScript
   client.on('close', function()
   {
      console.log('User disconnected:', client.id);
      clients.delete(client.id);                    // добавили эту строку
   });
```

Что касается замены страницы *index.htm* на *wschat.htm*, то в ней мы добавили "поле" для отображения автора сообщения (*origin*) и подключили новый скрипт для браузера *wschat_client.js*. В нем производится разбор сообщений (мы применяем символ '#', чтобы отделить автора от текста) и заполнение полей формы полученной информацией. Поскольку с точки зрения WebSocket-протокола ничего не поменялось, мы не станем приводить исходный код.



Веб-страница чат-сервиса на WebSocket

Вы можете запустить nodejs с чат-сервером *wschat.js*, а затем подключиться к нему из нескольких закладок браузера. Каждое соединение получает уникальный номер, выводимый в заголовке. Текст из поля *Message* отправляется всем клиентам по нажатию *Submit*, и у них в форме выводится как автор сообщения (метка слева внизу), так и сам текст (поле в центре внизу).

Итак, мы убедились в готовности веб-сервера с поддержкой веб-сокетов. Обратимся к написанию клиентской части протокола на MQL5.

7.8.6 Протокол WebSocket-ов на MQL5

Мы уже ранее рассмотрели Теоретические основы протокола WebSockets. Полная спецификация довольно обширна, и подробное описание её реализации потребовало бы много места и времени. Поэтому мы приведем общую структуру уже готовых классов и их программных интерфейсов. Все файлы расположены в каталоге *MQL5/Include/MQL5Book/ws/*.

- wsinterfaces.mqh — общее абстрактное описание всех интерфейсов (см. далее), констант и типов;

- wstransport.mqh — класс MqlWebSocketTransport, реализующий низкоуровневый сетевой интерфейс передачи данных IWebSocketTransport на базе Socket-функций MQL5;
- wsframe.mqh — классы WebSocketFrame и WebSocketFrameHixie, реализующие интерфейс IWebSocketFrame, за которым скрыты алгоритмы формирования (кодирования и декодирования) фреймов для протоколов Hybi и Hixie, соответственно;
- wsmessage.mqh — классы WebSocketMessage и WebSocketMessageHixie, реализующие интерфейс IWebSocketMessage, формализующий формирование сообщений из фреймов для протоколов Hybi и Hixie, соответственно;
- wsprotocol.mqh — классы WebSocketConnection, WebSocketConnectionHybi, WebSocketConnectionHixie, унаследованные от IWebSocketConnection; именно здесь происходит координированное управление формированием фреймов, сообщений, приветствия и разрыва соединения согласно спецификации, для чего используются вышеперечисленные интерфейсы;
- wsclient.mqh — готовая реализация Web-Socket-клиента — шаблонный класс WebSocketClient, поддерживающий интерфейс IWebSocketObserver (для обработки событий), и ожидающий WebSocketConnectionHybi или WebSocketConnectionHixie в качестве параметризованного типа;
- wstools.mqh — полезные утилиты в пространстве имен WsTools;

В наши будущие mqporj-проекты данные заголовочные файлы попадут автоматически как зависимости из директив *#include*.

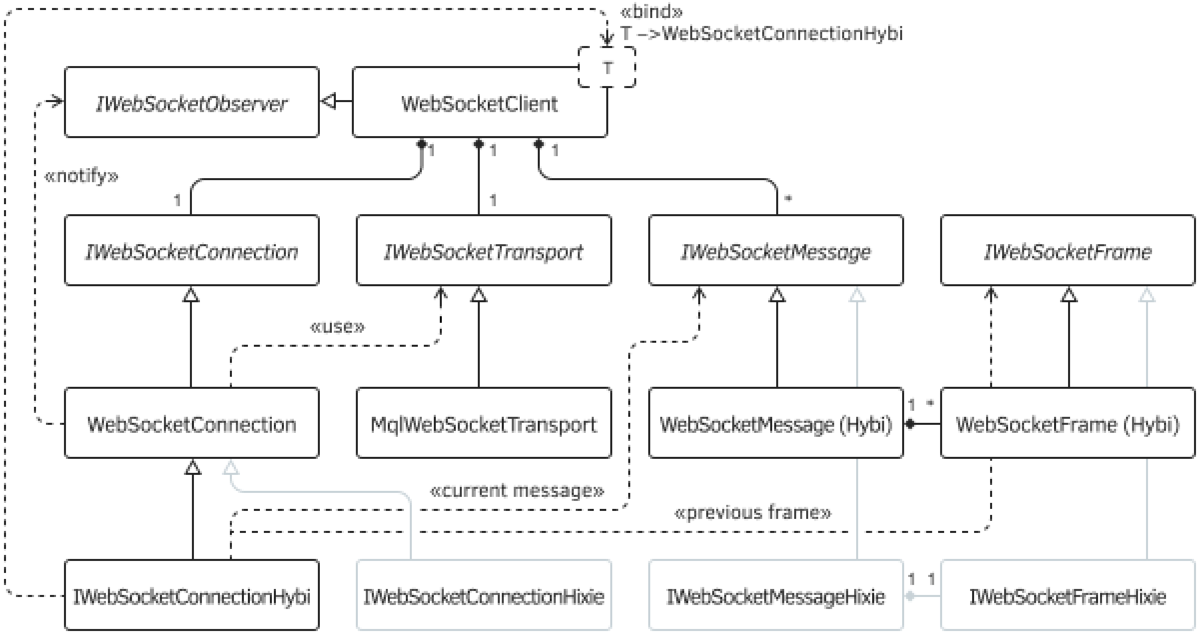


Диаграмма классов WebSocket в MQL5

Низкоуровневый сетевой интерфейс *IWebSocketTransport* имеет следующие методы.

```
interface IWebSocketTransport
{
   int write(const uchar &data[]); // запись массива байтов в сеть
   int read(uchar &buffer[]);      // чтение данных из сети в массив байтов
   bool isConnected(void) const;   // проверка на наличие связи
   bool isReadable(void) const;    // проверка на возможность чтения из сети
   bool isWritable(void) const;    // проверка на возможность записи в сеть
   int getHandle(void) const;      // системный дескриптор сокета
   void close(void);               // закрытие связи
};
```

Из названий методов нетрудно догадаться, с помощью каких Socket-функций MQL5 API они будут строиться. Но при необходимости, желающие могут воплотить данный интерфейс собственными средствами, например, через DLL.

Класс *MqlWebSocketTransport*, реализующий данный интерфейс, при создании экземпляра требует указания протокола, имени хоста и номера порта, куда производится сетевое подключение. Дополнительно можно указать величину таймаута.

Типы фреймов собраны в перечислении 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
};
```

Интерфейс для работы с фреймами содержит как статические, так и обычные методы, относящиеся к экземплярам фреймов. Статические методы выступают фабриками для создания фреймов необходимого типа передающей стороной (*create*) и приходящих фреймов (*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;
   };
   ...
```

Наличие методов-фабрик в классах-наследниках делается обязательным за счет наличия шаблона *Creator* и возвращающего его экземпляр метода *getCreator* (предполагается возврат "синглтона").

```
protected:
   template<typename P>
   class Creator: public StaticCreator
   {
   public:
      // декодируем полученные двоичные данные в IWebSocketFrame
      // (в случае продолжения, предыдущий фрейм в 'head')
      virtual IWebSocketFrame *decode(uchar &data[],
         IWebSocketFrame *head = NULL) override
      {
         return P::decode(data, head);
      }
      // создаем фрейм нужного типа (текст/закрытие/другой) с опциональным текстом
      virtual IWebSocketFrame *create(WS_FRAME_OPCODE type, const string data = NULL,
         const bool deflate = false) override
      {
         return P::create(type, data, deflate);
      };
      // создаем фрейм нужного типа (двоичный/текст/закрытия/другой) с данными
      virtual IWebSocketFrame *create(WS_FRAME_OPCODE type, const uchar &data[],
         const bool deflate = false) override
      {
         return P::create(type, data, deflate);
      };
   };
public:
   // требуем наличия экземпляра Creator
   virtual IWebSocketFrame::StaticCreator *getCreator() = 0;
   ...
```

Остальные методы интерфейса обеспечивают все необходимые манипуляции с данными во фреймах (кодирование/декодирование, получение данных и различных флагов).

```
    // закодировать "чистое" содержимое фрейма в данные для передачи по сети
    virtual int encode(uchar &encoded[]) = 0;

    // получить данные как текст
    virtual string getData() = 0;

    // получить данные как байты, вернуть размер
    virtual int getData(uchar &buf[]) = 0;

    // вернуть тип фрйема (opcode)
    virtual WS_FRAME_OPCODE getType() = 0;

    // проверка, является ли фрейм управляющим или с данными:
    // управляющие фреймы обрабатываются внутри классов
    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; }
 };
```

Интерфейс *IWebSocketMessage* содержит методы для выполнения похожих действий, но уже на уровне сообщений.

```
class IWebSocketMessage
{
public:
   // получить массив фреймов, составляющих данное сообщение
   virtual void getFrames(IWebSocketFrame *&frames[]) = 0;

   // задать текст как содержимое сообщения
   virtual bool setString(const string &data) = 0;

   // вернуть содержимое сообщения как текст
   virtual string getString() = 0;

   // задать двоичные данные как содержимое сообщения
   virtual bool setData(const uchar &data[]) = 0;

   // вернуть содержимое сообщения в "сыром" двоичном виде
   virtual bool getData(uchar &data[]) = 0;

   // признак полноты сообщения (получены все фреймы)
   virtual bool isFinalised() = 0;

   // добавить фрейм в сообщение
   virtual bool takeFrame(IWebSocketFrame *frame) = 0;
};
```

С учетом интерфейсов фреймов и сообщений определен общий интерфейс WebSocket-соединений *IWebSocketConnection*.

```
interface IWebSocketConnection
{
   // открыть соединение с указанным URL и его частями,
   // и опциональными кастом-заголовками
   bool handshake(const string url, const string host, const string origin,
      const string custom = NULL);

   // низко-уровневое чтение фреймов с сервера
   int readFrame(IWebSocketFrame *&frames[]);

   // низко-уровневая отправка фрейма (например, закрытие или ping)
   bool sendFrame(IWebSocketFrame *frame);

   // низко-уровневая отправка сообщения
   bool sendMessage(IWebSocketMessage *msg);

   // пользовательская проверка новых сообщений (генерация событий)
   int checkMessages();

   // пользовательская отправка текста
   bool sendString(const string msg);

   // пользовательская отправка двоичных данных
   bool sendData(const uchar &data[]);

   // закрытие соединения
   bool disconnect(void);
};
```

Уведомления о разрыве связи и получении новых сообщений поступают через методы интерфейса *IWebSocketObserver*.

```
interface IWebSocketObserver
{
  void onConnected();
  void onDisconnect();
  void onMessage(IWebSocketMessage *msg);
};
```

В частности, класс *WebSocketClient* сделан наследником этого интерфейса и по умолчанию просто выводит информацию в журнал. Конструктор класса ожидает адрес для соединения с протоколом *ws* или *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: сообщение может быть двоичным, печатаем его просто для уведомления
      Print(" > Message ", url, " " , msg.getString());
      WsTools::push(messages, msg);
```

```
      }
      ...
   };
```

Класс *WebSocketClient* собирает все объекты сообщений в массив и заботится об их удалении, если это не сделает MQL-программа.

Установление соединения осуществляется в методе *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);
      }
      ...
```

Наиболее удобные способы отправки данных предоставляют перегруженные методы *send* для текста и двоичных данных.

```
      bool send(const string str)
      {
         return connection ? connection.sendString(str) : false;
      }

      bool send(const uchar &data[])
      {
         return connection ? connection.sendData(data) : false;
      }
```

Для проверки наличия новых поступивших сообщений можно вызывать метод *checkMessages*. В зависимости от его параметра *blocking*, метод будет в цикле ожидать сообщения вплоть до

таймаута или сразу вернет управление, если сообщений нет. Сообщения поступят в обработчик *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())
   {
      // все фреймы собираются в соответствующие сообщения, и те становятся
      // доступными через уведомления о событии IWebSocketObserver::onMessage,
      // однако управляющие фреймы к этому моменту уже обработаны внутри и удалень
      if(!connection.checkMessages()) // пока нет сообщений, сделаем микро-паузу
      {
         Sleep(100);
      }
   }
}
```

Альтернативный способ получения сообщений реализован в методе *readMessage*: он возвращает указатель на сообщение в вызывающий код (иначе говоря, прикладной обработчик *onMessage* не требуется). После этого уже MQL-программа ответственна за освобождение объекта.

```
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;
}
```

Также класс позволяет изменить таймаут, проверить соединение и закрыть его.

```
   void setTimeOut(const int ms)
   {
      timeOut = fabs(ms);
   }

   bool isConnected() const
   {
      return socket && socket.isConnected();
   }

   void close()
   {
      if(isConnected())
      {
         if(connection)
         {
            connection.disconnect(); // это закроет socket после подтверждения сервер
            delete connection;
            connection = NULL;
         }
         if(socket)
         {
            delete socket;
            socket = NULL;
         }
      }
   }
};
```

Библиотека рассмотренных классов позволяет создать клиентские приложения для эхо- и чат-сервисов.

7.8.7 Клиентские программы эхо и чат-сервисов на MQL5

Для подключения к эхо-сервису напишем простой скрипт *MQL5/Experts/MQL5Book/p7/wsEcho/wsecho.mq5* (обратите внимание, что это именно скрипт, но мы расположили его внутри папки *MQL5/Experts/MQL5Book/p7/*, сделав её единым контейнером для MQL-программ, связанных с веб, поскольку все последующие примеры будут экспертами). Поскольку в данной главе мы рассматриваем создание комплексов программ в рамках проектов, оформим скрипт как часть mqproj-проекта, в который также включим и серверную составляющую.

Входные параметры скрипта позволяют указать адрес сервиса и текст сообщения. По умолчанию используется незащищенное соединение. Если сервер *wsecho.js* будет запущен с поддержкой TLS, нужно поменять протокол на защищенный *wss*. Имейте в виду, что установление защищенного соединения требует больше времени (например, пару секунд), чем обычного.

```
input string Server = "ws://localhost:9000/";
input string Message = "My outbound message";

#include <MQL5Book/AutoPtr.mqh>
#include <MQL5Book/ws/wsclient.mqh>
```

В функции *OnStart* создадим экземпляр WebSocket-клиента (*wss*) для заданного адреса и вызовем метод *open*. В случае успешного подключения мы ждем приветственного сообщения от сервиса с помощью вызова *wss.readMessage* в блокирующем режиме (ожидание до 5 секунд, по умолчанию). Мы использует автоуказатель для получаемого объекта, чтобы не вызывать в конце *delete* вручную.

```
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());
      ...
```

Напомним, что класс *WebSocketClient* содержит заглушки обработчиков событий, в том числе и простой метод *onMessage*, который распечатает приветствие в журнал.

Затем мы отправляем свое сообщение и снова ждем ответа от сервера. Эхо-сообщение также будет выведено в журнал.

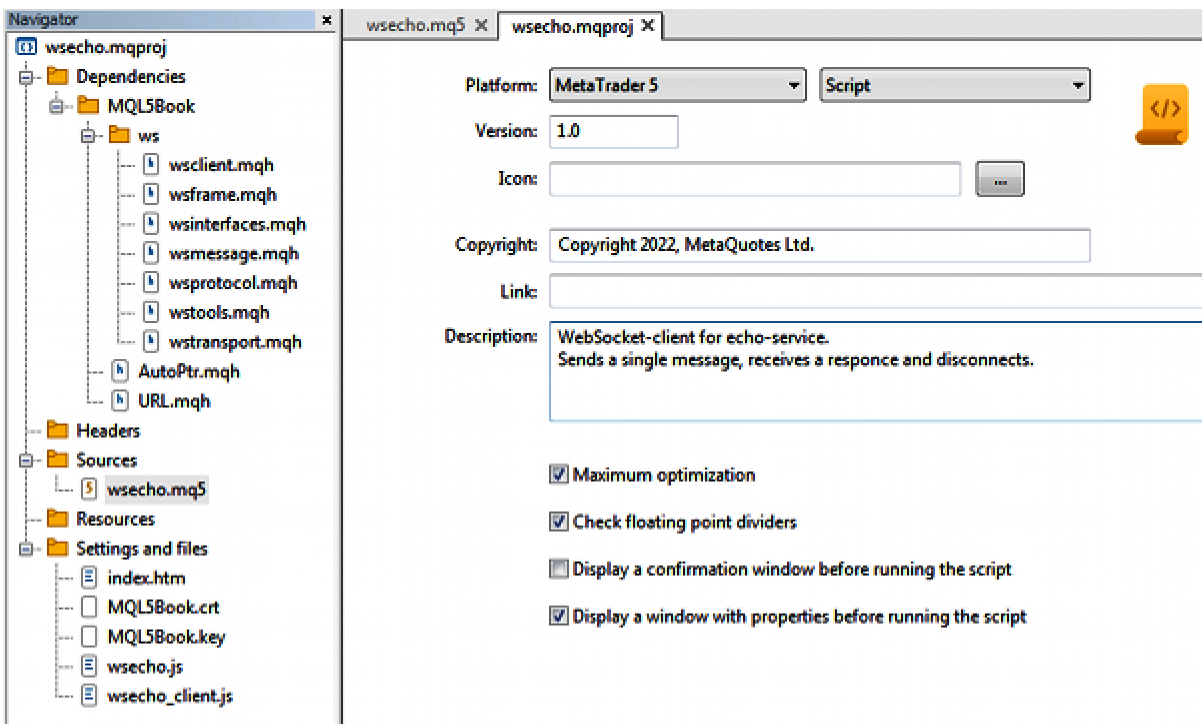
```
      Print("Sending message...");
      wss.send(Message);
      Print("Receiving echo...");
      AutoPtr<IWebSocketMessage> echo(wss.readMessage());
   }
   ...
```

В завершение мы закрываем соединение.

```
   if(wss.isConnected())
   {
      Print("Closing...");
      wss.close();
   }
}
```

На основе файла скрипта создадим файл проекта (*wsecho.mqproj*). Заполним в свойствах проекта номер версии (1.0), копирайт, и описание. Добавим в ветвь *Settings and files* серверные файлы эхо-сервиса (это, как минимум, станет напоминанием разработчику о наличии тестового сервера). После компиляции в иерархии появятся зависимости (заголовочные файлы).

Все должно выглядеть примерно как на скриншоте.



Проект эхо-сервиса, клиентский скрипт и сервер

Если бы скрипт располагался внутри папки *Shared Projects*, например, в *MQL5/Shared Projects/MQL5Book/wsEcho/*, то после успешной компиляции его ex5-файл был бы автоматически перемещен в папку *MQL5/Scripts/Shared Projects/MQL5Book/wsEcho/*, о чем была бы выведена соответствующая запись в журнал компиляции. Это стандартное поведение компиляции любых MQL-программ в разделяемых проектах.

Во всех примерах данной главы не забываем стартовать сервер, прежде чем тестировать MQL-скрипт. В данном случае выполняем команду: *node.exe wsecho.js*, находясь в папке *Web*.

Далее запустим скрипт *wsecho.ex5*. В журнале появятся записи о происходящих действиях, а также уведомления о сообщениях.

```
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
```

Приведенные выше HTTP-заголовки — это ответ сервера в процессе "рукопожатия". Если заглянуть в окно консоли, где запущен сервер, обнаружим HTTP-заголовки, полученные сервером от нашего клиента.

```
C:\Program Files\MetaTrader 5\MQL5\Projects\MQL5Book\Web>node wsecho.js
{
  connection: 'Upgrade',
  host: 'localhost',
  'sec-websocket-key': 'i1MkqeHVVi1J/u1qyhA5QQ==',
  origin: 'http://localhost',
  'sec-websocket-version': '13',
  upgrade: 'websocket'
}
New user: 1
1 : My outbound message
User disconnected: 1
```

Серверный журнал эхо-сервиса

Также здесь отмечено подключение пользователя, его сообщение и отключение.

Проделаем аналогичную работу для чат-сервиса: создадим WebSocket-клиент на MQL5, проект под него, и протестируем. На этот раз тип клиентской программы будет эксперт, потому что для чата необходима поддержка интерактивных событий от клавиатуры на графике. Эксперт прилагается к книге в папке *MQL5/MQL5Book/p7/wsChat/wschat.mq5*.

Для демонстрации технологии получения событий в методах-обработчиках определим собственный класс *MyWebSocket*, производный от *WebSocketClient*.

```
class MyWebSocket: public WebSocketClient<Hybi>
{
public:
   MyWebSocket(const string address, const bool compress = false):
      WebSocketClient(address, compress) { }

   /* void onConnected() override { } */

   void onDisconnect() override
   {
      // можем сделать что-то еще и вызвать (или не вызывать) унаследованный код
      WebSocketClient<Hybi>::onDisconnect();
   }

   void onMessage(IWebSocketMessage *msg) override
   {
      // TODO: мы могли бы отсекать копии собственных сообщений,
      // но они оставлены для отладки
      Alert(msg.getString());
      delete msg;
   }
};
```

При получении сообщения мы будем выводить его не в журнал, а алертом, после чего объект следует удалить.

В глобальном контексте опишем объект нашего класса *wss* и строку *message*, где будет аккумулироваться ввод пользователя с клавиатуры.

```
MyWebSocket wss(Server);
string message = "";
```

Функция *OnInit* содержит необходимую подготовку, в частности запускает таймер и открывает соединение.

```
int OnInit()
{
  ChartSetInteger(0, CHART_QUICK_NAVIGATION, false);
  EventSetTimer(1);
  wss.setTimeOut(1000);
  Print("Opening...");
  return wss.open() ? INIT_SUCCEEDED : INIT_FAILED;
}
```

Таймер нужен, чтобы проверять новые сообщений от других пользователей.

```
void OnTimer()
{
   wss.checkMessages(false); // в таймере используем неблокирующую проверку
}
```

В обработчике *OnChartEvent* реагируем на нажатия клавиш: все буквенно-цифровые клавиши транслируются в символы и присоединяются к строке *message*. При необходимости можно нажать *Backspace*, чтобы удалить последний знак. Весь набранный текст обновляется в

комментарии графика. Когда сообщение набрано полностью, нажатие *Enter* отправляет его на сервер.

```
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);
   }
}
```

Если ввести текст "long", программа отправит специально подготовленный довольно длинный текст. Если текст сообщения "bye", программа закрывает соединение. Также соединение закроется при выходе из программы.

```
void OnDeinit(const int)
{
   if(wss.isConnected())
   {
      Print("Closing...");
      wss.close();
   }
}
```

Создадим под эксперт проект (файл *wschat.mqproj*), заполним его свойства и добавим серверную часть в ветвь *Settings and files*. На этот раз покажем, как файл проекта выглядит изнутри. В mqproj-файле ветвь *Dependencies* хранится в свойстве "files", а ветвь *Settings and files* — в свойстве "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 f
  "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
      }
    ]
  }
```

Если бы эксперт находился внутри папки *Shared Projects*, например, в *MQL5/Shared Projects/MQL5Book/wsChat/*, после успешной компиляции его ex5-файл был бы автоматически перемещен в папку *MQL5/Experts/Shared Projects/MQL5Book/wsChat/*.

Стартуем сервер *node.exe wschat.js*. Теперь можно запустить пару копий эксперта на разных графиках. В принципе, сервис предполагает "общение" между разными терминалами и даже разными компьютерами, но никто не запрещает тестировать его из одного терминала.

Вот пример общения между чартами EURUSD и 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!
```

Поскольку у нас сообщения рассылаются всем, включая отправителя, они задвоены в журнале, но на разных чартах.

Общение видно и на стороне сервера.

```
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!
```

Серверный журнал чат-сервиса

Теперь у нас готова вся техническая составляющая для организации сервиса торговых сигналов.

7.8.8 Сервис торговых сигналов и тестовая веб-страница

Сервис торговых сигналов технически идентичен чат-сервису, однако его пользователи (а точнее клиентские соединения) должны выполнять одну из двух ролей:

- поставщик сообщений;
- потребитель сообщений;

Кроме того, информация должна быть доступна не всем, а по некоторой схеме подписки.

Чтобы обеспечить это, при подключении к сервису пользователи должны будут указать определенную идентификационную информацию, различающуюся в зависимости от роли.

Поставщик должен указать публичный идентификатор сигнала (PUB_ID), уникальный среди всех сигналов. В принципе, одно и то же лицо может потенциально генерировать более одного сигнала и, соответственно, должно иметь возможность получить несколько идентификаторов. В этом смысле мы не станем усложнять сервис, вводя раздельные идентификаторы поставщика (как конкретного лица) и идентификаторы его сигналов. Вместо этого будут поддерживаться только идентификаторы сигналов. Для реального сервиса сигналов этот вопрос нужно проработать, вместе с авторизацией, которую мы оставили за кадром.

Идентификатор потребуется, чтобы рекламировать его или просто передавать лицам, заинтересованным в подписке на данный сигнал. Но "каждый встречный" не должен получить доступ к сигналу, зная только публичный идентификатор. В простейшем случае, для открытого мониторинга счета это было бы приемлемо, но мы продемонстрируем вариант ограничения доступа именно в разрезе сигналов.

Для этой цели поставщик должен предоставить серверу секретный ключ (PUB_KEY), известный только ему, но не общественности. Этот ключ потребуется для генерации ключа доступа конкретного подписчика.

Потребитель (подписчик) также должен иметь уникальный идентификатор (SUB_ID, и здесь также обойдемся без авторизации). Чтобы подписаться на интересующий его сигнал, пользователь

должен сообщить поставщику сигнала свой идентификатор (на практике подразумевается, что на этом же этапе нужно подтвердить оплату, и обычно это все автоматизируется сервером). Поставщик формирует тем или иным способом слепок, состоящий из своего идентификатора, идентификатора подписчика и своего секретного ключа. В нашем сервисе это будет делаться вычислением хэша SHA256 от строки PUB_ID:PUB_KEY:SUB_ID, после чего полученные байты переводятся в строку шестнадцатеричного формата. Это и будет ключ доступа (SUB_KEY или ACCESS_KEY) к сигналу конкретного поставщика для конкретного подписчика. Поставщик (а в реальных системах — сам сервер автоматически) пересылает этот ключ подписчику.

Таким образом, подписчик при подключении к сервису должен будет указать свой идентификатор (SUB_ID), идентификатор желаемого сигнала (PUB_ID) и ключ доступа (SUB_KEY). Поскольку сервер знает секретный ключ поставщика, он может повторно рассчитать ключ доступа для данного сочетания PUB_ID и SUB_ID, и сравнить с предоставленным SUB_KEY. Совпадение означает продолжение нормального процесса с обменом сообщениями. Различие приведет к сообщению об ошибке и отключению псевдо-подписчика от сервиса.

Важно отметить, что в нашем демо, в угоду простоте, отсутствует нормальная регистрация пользователей и сигналов, и потому выбор идентификаторов — произвольный. Для нас лишь важно отслеживать уникальность идентификаторов, чтобы знать, кому и от кого посылать информацию онлайн. Так что, наш сервис не гарантирует, что идентификатор, например, "Super Trend" принадлежит вчера, сегодня и завтра одному и тому же пользователю. Резервирование имен производится по принципу: кто первый встал, того и тапки. Пока некий поставщик беспрерывно подключен под данным идентификатором, сигнал исходит от него. Если он отсоединится, то идентификатор станет доступен для выбора в любом следующем подключении.

Единственный идентификатор, который будет всегда "занят" — это "Server": его сервер использует для рассылки своих сообщений о статусах подключений.

Для генерации ключей доступа в папке сервера имеется простой JavaScript *access.js*. При его запуске в командной строке нужно единственным параметром передать строку указанного выше вида PUB_ID:PUB_KEY:SUB_ID (идентификаторы и секретный ключ между ними, соединенные символом ':')

Если параметр не указать, скрипт генерирует ключ доступа для неких демонстрационных идентификаторов (PUB_ID_001, SUB_ID_100) и секрета (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'));
```

Запустив скрипт командой:

```
node access.js PUB_ID_001:PUB_KEY_FFF:SUB_ID_100
```

мы получим такой результат:

```
fd3f7a105eae8c2d9afce0a7a4e11bf267a40f04b7c216dd01cf78c7165a2a5a
```

Между прочим, вы можете проверить и повторить данный алгоритм на чистом MQL5 с помощью функции *CryptEncode*.

Разобрав концептуальную часть, приступим к практической реализации.

Серверный скрипт сигнального сервиса разместим в файле *MQL5/Experts/MQL5Book/p7/Web/wspubsub.js*. Настройка серверов в нем совпадает с тем, что мы уже делали ранее. Но дополнительно потребуется подключить тот же модуль "crypto", который был использован в *access.js*. Начальная страница будет называться *wspubsub.htm*.

```
// JavaScript
const crypto = require('crypto');
...
http1.createServer(options, function (req, res)
{
   ...
   if(req.url == '/')
   {
      req.url = "wspubsub.htm";
   }
   ...
});
```

Вместо одной карты подключившихся клиентов определим две карты — раздельно под поставщиков и потребителей сигналов.

```
// JavaScript
const publishers = new Map();
const subscribers = new Map();
```

В обеих картах ключом выступает идентификатор поставщика, однако в первой хранятся объекты самих поставщиков, а во второй — объекты подписанных на каждого поставщика подписчиков (массивы объектов).

Для передачи идентификаторов и ключей во время "рукопожатия" будем использовать специальный заголовок, разрешенный спецификацией WebSocket-ов, а именно Sec-Websocket-Protocol. Договоримся, что идентификаторы и ключи будут склеены символом '-': в случае поставщика ожидается строка вида X-MQL5-publisher-PUB_ID-PUB_KEY, а в случае подписчика — X-MQL5-subscriber-SUB_ID-PUB_ID-SUB_KEY.

Любые попытки подсоединиться к нашему сервису без заголовка Sec-Websocket-Protocol: X-MQL5-... будут пресекаться немедленным закрытием.

В объекте нового клиента (в параметре обработчика события "connection" — *onConnect(client)*) данный заголовок легко извлечь из свойства *client.protocol*.

Покажем процедуру регистрации и рассылки сообщений поставщика сигнала в упрощенном виде, без обработки ошибок (полный код прилагается). Важно отметить, что текст сообщений формируется в формате JSON (о котором мы более подробно поговорим в следующем разделе). В частности, отправитель сообщения передается в свойстве "origin" (причем, когда сообщение посылает сам сервис, в этом поле — строка "Server"), а прикладные данные от поставщика помещаются в свойство "msg", и это может быть не просто текст, но и вложенная структура любого содержания.

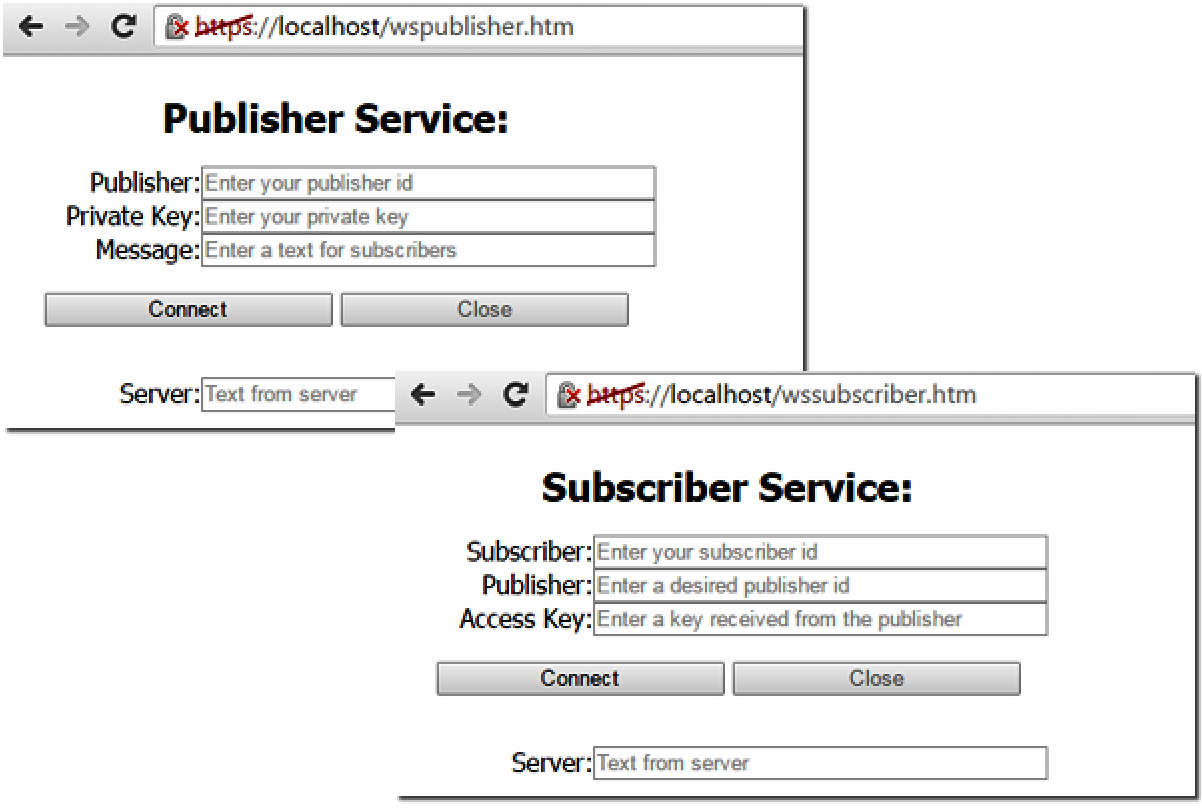
```
// JavaScript
const wsServer = new WebSocket.Server({ server });
wsServer.on('connection', function onConnect(client)
{
   console.log('New user:', ++count, client.protocol);
   if(client.protocol.startsWith('X-MQL5-publisher'))
   {
      const parts = client.protocol.split('-');
      client.id = 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);
      });
   }
   ...
```

Половина алгоритма для подписчиков похожа, но здесь добавилось вычисление ключа доступа и его сличение с тем, что передал подключающийся клиент.

```
// 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);
            }
         }
      });
```

```
}
```

Пользовательский интерфейс на клиентской странице *wspubsub.htm* просто предлагает перейти по ссылке на одну из двух страниц с формами для поставщиков (*wspublisher.htm* + *wspublisher_client.js*) или подписчиков (*wssubscriber.htm* + *wssubscriber_client.js*).



Веб-страницы тестовых клиентов сигнального сервиса

Их реализация наследует черты предыдущих рассмотренных JavaScript-клиентов, но с учетом кастомизации заголовка Sec-Websocket-Protocol: X-MQL5- и еще одного нюанса.

До сих пор мы обменивались простыми текстовыми сообщениями. Но для сигнального сервиса потребуется передавать много структурированной информации, а для этого лучше подойдет JSON. Поэтому клиенты умеют парсить JSON, хотя и не используют его по прямому назначению, потому что даже если в JSON-е обнаружится команда покупать или продавать конкретный тикер заданным объемом, браузер не умеет этого делать.

Поддержку JSON нам потребуется добавить и в свой клиент сигнального сервиса на MQL5. Но пока можно запустить на сервере *wspubsub.js* и протестировать избирательное подключение поставщиков и потребителей сигналов в соответствии с указанными ими реквизитами. Проделайте это самостоятельно.

7.8.9 Клиентская программа сигнального сервиса на MQL5

Итак, мы решили, что текст в сообщениях сервиса будет в формате JSON.

В наиболее распространенном варианте, JSON представляет собой текстовое описание объекта, похожее на то, как это делается для структур в MQL5. Объект заключен в фигурные скобки, внутри которых через запятую пишутся его свойства: каждое свойство имеет идентификатор в кавычках, после чего идет двоеточие и значение свойства. Поддерживаются свойства нескольких

примитивных типов: строки, целые и вещественные числа, логические *true/false* и пустое значение *null*. Кроме того, значением свойства может быть, в свою очередь, объект или массив. Массивы описываются с помощью квадратных скобок, внутри которых элементы перечисляются через запятую. Например,

```
{
   "string": "this is a text",
   "number": 0.1,
   "integer": 789735095,
   "enabled": true,
   "subobject" :
   {
      "option": null
   },
   "array":
   [
      1, 2, 3, 5, 8
   ]
}
```

В принципе, массив на верхнем уровне также является валидным JSON. Например,

```
[
   {
      "command": "buy",
      "volume": 0.1,
      "symbol": "EURUSD",
      "price": 1.0
   },
   {
      "command": "sell",
      "volume": 0.01,
      "symbol": "GBPUSD",
      "price": 1.5
   }
]
```

Для сокращения трафика в прикладных протоколах, использующих JSON, принято сокращать названия полей до нескольких букв (часто — до одной).

Названия свойств и строковые значения заключаются в двойные кавычки. Если требуется указать кавычку внутри строки, её следует экранировать обратной косой чертой.

Применение JSON делает протокол универсальным и расширяемым. Например, для проектируемого сервиса (торговых сигналов и, в более общем случае, копирования состояния счета) можно чисто теоретически предположить следующую структуру сообщения:

```
{
   "origin": "publisher_id",    // отправитель сообщения ("Server" в техническом сооб
   "msg" :                      // сообщение (текст или JSON), как поступило от отпра
   {
      "trade" :                 // текущие торговые команды (если есть сигнал)
      {
         "operation": ...,      // покупка/продажа/закрытие
         "symbol": "ticker",
         "volume": 0.1,
         ... // другие параметры сигнала
      },
      "account":                // статус счета
      {
         "positions":           // позиции
         {
            "n": 10,            // количество открытых позиций
            [ { ... },{ ... } ] // массив свойств открытых позиций
         },
         "pending_orders":      // отложенные ордера
         {
            "n": ...
            [ { ... } ]
         }
         "drawdown": 2.56,
         "margin_level": 12345,
         ... // другие параметры статуса
      },
      "hardware":               // удаленный контроль "здоровья" ПК
      {
         "memory": ...,
         "ping_to_broker": ...
      }
   }
}
```

Какие-то из этих возможностей могут поддерживать, а могут не поддерживать конкретные реализации клиентских программ (всё, что им не "понятно", они просто проигнорируют). Кроме того, при соблюдении условия отсутствия конфликтов в названиях свойств на одном уровне, каждый поставщик информации может добавлять в JSON свои специфические данные. Сервис обмена сообщениями просто будет пересылать эту информацию. Разумеется, программа на приемной стороне должна уметь интерпретировать эти специфические данные.

К книге прилагается парсер JSON под названием *ToyJson* ("игрушечный" JSON, файл *toyjson.mqh*) — маленький, неэффективный, и без поддержки полных возможностей спецификации формата (например, в части обработки *escape*-последовательностей). Он был написан специально для данного демо-сервиса, с поправкой на предполагаемую, не особо сложную, структуру информации о торговых сигналах. Мы не станем его подробно здесь описывать, а принципы его использования станут ясны из исходного кода MQL-клиента сигнального сервиса.

Для своих проектов и в случае развития данного проекта вы можете выбрать другие парсеры JSON, доступные в кодовой базе на сайте *mql5.com*.

Один элемент (контейнер или свойство) в *ToyJson* описывается объектом класса *JsValue*. В нем определено несколько перегрузок метода *put(key, value)* для добавления именованных внутренних свойств как в JSON-объект или *put(value)* для добавления значения как в JSON-массив. Также данный объект может представлять и отдельное значение примитивного типа. Для чтения свойств JSON-объекта можно применить к *JsValue* нотацию оператора [] с указанием имени требуемого свойства в скобках. Очевидно, что целочисленные индексы поддерживаются для доступа внутрь JSON-массива.

Сформировав требуемую конфигурацию связанных объектов *JsValue*, можно её сериализовать в текст JSON-формата с помощью метода *stringify(string &buffer)*.

Второй класс в *toyjson.mqh* — *JsParser* — позволяет выполнять обратную операцию: превращать текст с описанием JSON в иерархическую структуру *JsValue*-объектов.

С учетом классов для работы с JSON приступим к написанию эксперта *MQL5/Experts/MQL5Book/p7/wsTradeCopier/wstradecopier.mq5*, который сможет выполнять обе роли в сервисе копирования сделок: поставщика информации о трейдах, совершаемых на счету, или получателя этой информации от сервиса для воспроизведения этих трейдов.

Объем и содержание пересылаемой информации остается, с политической точки зрения, на усмотрении поставщика и может существенно отличаться в зависимости от сценария (назначения) использования сервиса. В частности, можно копировать только совершаемые сделки или полностью состояние счета вместе с отложенными ордерами и защитными уровнями. В нашем примере мы лишь обозначим техническую реализацию передачи информации, а конкретный набор объектов и свойств можно затем подобрать по своему усмотрению.

Опишем в коде 3 структуры, унаследованные от встроенных структур и обеспечивающие "упаковку" информации в JSON:

- MqlTradeRequestWeb — MqlTradeRequest;
- MqlTradeResultWeb — MqlTradeResult;
- DealMonitorWeb — DealMonitor*.

Последняя в списке структура, строго говоря, не является встроенной, а определена нами в файле *DealMonitor.mqh*, но она заполняется на стандартном наборе свойств сделок.

Конструктор каждой из производных структур заполняет поля на основе переданного первоисточника (торгового запроса, его результата или сделки). В каждой структуре реализован метод *asJsValue*, возвращающий указатель на объект *JsValue*, в котором отражены все свойства структуры: они добавляются в JSON-объект с помощью метода *JsValue::put*. Вот, например, как это сделано в случае *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();
      // главный блок: 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);

      // объемы
      if(volume != 0) req.put("v", TU::StringOf(volume));
      req.put("f", VerboseJson ? EnumToString(type_filling) : (string)type_filling);

      // блок с ценами
      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));

      // блок отложенных ордеров
      if(TU::IsPendingType(type))
      {
         req.put("t", VerboseJson ? EnumToString(type_time) : (string)type_time);
         if(expiration != 0) req.put("d", TimeToString(expiration));
      }

      // блок модификации
      if(order != 0) req.put("o", order);
      if(position != 0) req.put("q", position);
```

```
      if(position_by != 0) req.put("b", position_by);

      // вспомогательный блок
      if(magic != 0) req.put("m", magic);
      if(StringLen(comment)) req.put("c", comment);

      return req;
   }
};
```

Мы переносим в JSON все свойства (это подойдет для сервиса мониторинга счета), но вы можете оставить только ограниченный набор.

Для свойств, которые являются перечислениями, мы предусмотрели 2 способа представления в JSON: как целое число и как строковое имя элемента перечисления. Выбор способа производится с помощью входного параметра *VerboseJson* (в идеале, он должен прописываться в коде структур не напрямую, а через параметр конструктора).

```
input bool VerboseJson = false;
```

Передача только чисел упростила бы кодирование, потому что на приемной стороне достаточно привести их к нужному типу перечисления, чтобы выполнить "зеркальные" действия. Однако числа затрудняют восприятие информации человеком, а ему может потребоваться проанализировать ситуацию (сообщение). Поэтому имеет смысл поддержать опцию для строкового представления, как более "дружественного", хотя оно и требует дополнительных операций в приемном алгоритме.

Во входных параметрах также указывается адрес сервера, роль программы и реквизиты подключения — раздельно для поставщика и подписчика.

```
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;
```

Параметры *SymbolFilter* и *MagicFilter* в группе поставщика позволяют ограничить отслеживаемую торговую активность заданным символом и магическим номером. Пустое значение в *SymbolFilter* означает контроль только текущего символа графика, для перехвата любых сделок введите

символ '*'. В поставщике сигнала для этой цели будет применяться функция *FilterMatched*, принимающая символ и "магик" сделки.

```
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;
}
```

Параметр *SymbolSubstitute* во входной группе подписчика позволяет подменить приходящий в сообщениях символ на другой, по которому и будет вестись копирующая торговля. Эта возможность пригодится, если названия тикеров одного и того же финансового инструмента отличаются у брокеров. Но данный параметр выполняет также и функцию разрешающего фильтра для повторения сигналов: только указанные здесь символы будут торговаться. Например, чтобы разрешить торговлю сигналом по символу EURUSD (даже без подмены тикеров), нужно задать в параметре строку "EURUSD=EURUSD". Слева от знака '=' указывается символ из сообщений сигнала, справа — символ для торговли.

Список подстановки символов обрабатывается функцией *FillSubstitutes* во время инициализации и затем используется для подстановки и разрешения торговли функцией *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;
}
```

Для общения с сервисом определен производный класс от *WebSocketClient*. Он нужен, в первую очередь, для запуска торговли по сигналу — по приходу сообщения в обработчик *onMessage*. Мы вернемся к этому вопросу чуть позже, после того как рассмотрим формирование и отправку сигналов на стороне поставщика.

```
class MyWebSocket: public WebSocketClient<Hybi>
{
public:
   MyWebSocket(const string address): WebSocketClient(address) { }

   void onMessage(IWebSocketMessage *msg) override
   {
      ...
   }
};

MyWebSocket wss(Server);
```

Инициализация в *OnInit* включает запуск таймера (для периодического вызова *wss.checkMessages(false)*) и подготовку кастом-заголовков с реквизитами пользователя, в

зависимости от выбранной роли. Затем открываем соединение с помощью вызова *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;
}
```

Механизм копирования, то есть перехвата сделок и отправки информации о них на веб-сервис, запускается в обработчике *OnTradeTransaction*. Как мы знаем, это не единственный способ — можно было бы анализировать "слепок" состояния счета в *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           // успешное действие
         || result.retcode == TRADE_RETCODE_DONE
         || result.retcode == TRADE_RETCODE_DONE_PARTIAL)
      {
         if(FilterMatched(request.symbol, request.magic))
         {
            ... // см. следующий блок кода
         }
      }
   }
}
```

Мы отслеживаем события об успешно выполненных торговых запросах, которые удовлетворяют условиям заданных фильтров. Далее структуры запроса, результата запроса и сделки превращаются в объекты JSON. Все они помещаются в один общий контейнер *msg* под названиями "req", "res" и "deal", соответственно. Напомним, что сам контейнер попадёт в сообщение веб-сервиса как свойство "msg".

```
    // контейнер для вложения в сообщение сервиса будет виден как свойство "m
      // {"origin" : "this_publisher_id", "msg" : { наши данные здесь }}
    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);
```

После заполнения контейнер выводится в виде строки в *buffer*, печатается в журнал и отправляется на сервер.

По идее, мы можем добавлять в этот контейнер другую информацию: статус счета (просадка, загрузка), количество и свойства отложенных ордеров и так далее. Так, просто для демонстрации возможностей по расширению содержимого сообщений, мы выше добавили количество открытых позиций. Для отбора позиций согласно фильтрам мы использовали объект класса *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);
   ...
}
```

В принципе, для повышения надежности "копировщикам" имеет смысл анализировать состояние позиций, а не просто перехватывать сделки.

На этом рассмотрение той части эксперта, которая задействована в роли поставщика сигналов, завершено.

В роли подписчика, как мы уже анонсировали, эксперт получает сообщения в методе *MyWebSocket::onMessage*. Здесь входящее сообщение разбирается с помощью *JsParser::jsonify*, и тот контейнер, который был сформирован передающей стороной, извлекается из свойства *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"])) { /* обработка ошибок */ }
         delete obj;
      }
      delete msg;
   }
};
```

Непосредственно анализом сигнала и торговыми операциями занимается функция *RemoteTrade*. Здесь она приводится с сокращениями, без обработки потенциальных ошибок. В функции обеспечена поддержка обоих способов представления перечислений: как целочисленных значений или как строковых названий элементов. Входящий JSON-объект "исследуется" на наличие необходимых свойств (команд и атрибутов сигнала) путем применения оператора [], в том числе последовательно по несколько раз (для доступа во вложенные JSON-объекты).

```
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; // не найден или запрещен
      }
      
      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")
         {
            // действие закрытия предполагает наличие подходящей позиции, ищем её
            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)
            {
               // нужна позиция, противоположная по направлению сделке
               if(!TU::IsSameType((ENUM_ORDER_TYPE)values[i]._2, type))
               {
                  // нужен достаточный объем (здесь точно равный!)
                  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;
}
```

В данной реализации не анализируется цена сделки, возможные ограничения на лот, стоп-уровни и прочие нюансы. Мы просто повторяем торговлю по текущей локальной цене. Также при закрытии позиции делается проверка на точное равенство объема, что подходит для счетов с хеджинговым учетом, но не для неттинга, где возможно частичное закрытие, если объем сделки меньше позиции (а может быть и больше, в случае переворота, но вариант DEAL_ENTRY_INOUT здесь не поддержан). Все эти моменты следует доработать для реального применения.

Давайте запустим сервер *node.exe wspubsub.js* и две копии эксперта *wstradecopier.mq5* на разных графиках, в одном и том же терминале. Обычный сценарий предполагает, что эксперт нужно запустить на разных счетах, но для проверки работоспособности подойдет и "парадоксальный" вариант: будем копировать сигналы от одного символа на другой.

В одной копии эксперта оставим настройки по умолчанию, с ролью публикатора. Его следует разместить на графике EURUSD. Во второй копии, на графике GBPUSD, сменим роль на подписчика. Строка "EURUSD=GBPUSD" во входном параметре *SymbolSubstitute* как раз разрешает торговлю GBPUSD по сигналам о EURUSD.

В журнал будут выведены строки о подключении, с HTTP-заголовками и приветствиями, которые мы уже видели, и потому опустим их.

Совершим покупку EURUSD и убедимся, что она "продублировалась" в том же объеме по GBPUSD.

Далее приведены фрагменты журнала (имейте в виду, что из-за того, что оба эксперта работают в одной копии терминала, сообщения о транзакциях будут поступать в оба чарта, в связи с чем для облегчения анализа журнала можно попеременно устанавливать фильтры "EURUSD" и "GBPUSD"):

```
(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" : 10
»   "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" : "
»   "entry" : "DEAL_ENTRY_IN", "pid" : 1461313378, "r" : "DEAL_REASON_CLIENT", "v" :
»   "s" : "EURUSD"}, "pos" : {"n" : 1}}
```

Здесь показано содержание выполненного запроса и его результат, а также буфер с JSON-строкой, отправленной на сервер.

Почти моментально на приемной стороне, на графике GBPUSD, выводится алерт с сообщением от сервера: в "сыром" виде и отформатированный после успешного парсинга в *JsParser*. В "сыром" виде сохранено свойство "origin", в котором сервер дает нам знать, кто является источником сигнала.

```
(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, "
»   "p" : 0.99886, "b" : 0.99886, "a" : 0.99886}, "deal" : {"d" : 1439023682, "o" : 1
»   "t" : "2022.09.19 16:45:50", "tmsc" : 1663605950086, "type" : "DEAL_TYPE_BUY",
»   "entry" : "DEAL_ENTRY_IN", "pid" : 1461313378, "r" : "DEAL_REASON_CLIENT", "v" :
»   "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 -
```

Последняя из приведенных записей сигнализирует об успешно выполненной сделке по GBPUSD. На торговой закладке счета должны отображаться 2 позиции.

Спустя некоторое время закроем позицию EURUSD — позиция GBPUSD должна закрыться автоматически.

```
(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}, "r
»   "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" :
»   "v" : 0.01, "p" : 0.99881, "m" : -0.05, "s" : "EURUSD"}, "pos" : {"n" : 0}}
```

Если в первый раз сделка имела тип DEAL_ENTRY_IN, то теперь — DEAL_ENTRY_OUT. Алерт подтверждает прием сообщения и успешное закрытие дублирующей позиции.

```
(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", "
»   "o" : 1461315206, "q" : 1461313378}, "res" : {"code" : 10009, "d" : 1439025490, "
»   "v" : 0.01, "p" : 0.99881, "b" : 0.99881, "a" : 0.99881}, "deal" : {"d" : 1439025
»   "o" : 1461315206, "t" : "2022.09.19 16:46:52", "tmsc" : 1663606012990, "type" : "
»   "entry" : "DEAL_ENTRY_OUT", "pid" : 1461313378, "r" : "DEAL_REASON_CLIENT", "v" :
»   "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 -
```

Напоследок создадим рядом с экспертом *wstradecopier.mq5* файл проекта *wstradecopier.mqproj*, чтобы добавить в него описание и необходимые серверные файлы (в прежнем каталоге *MQL5/Experts/p7/MQL5Book/Web/*).

Подведем итог: мы организовали технически расширяемую, многопользовательскую систему для обмена торговой информацией через сокет-сервер. В силу технических особенностей веб-сокетов (постоянное открытое соединение), данная реализация сервиса сигналов больше подходит для краткосрочной и высокочастотной торговли, а также для контроля арбитражных ситуаций в котировках.

Решение задачи потребовало объединения нескольких программ на разных платформах, подключение большого количества зависимостей, чем обычно и характеризуется переход на уровень проекта. Среда разработки при этом также расширяется, выходя за рамки компилятора и редактора исходных кодов. В частности, наличие в проекте клиентской или серверной частей обычно предполагает вовлечение разных программистов, отвечающих за них. В этом случае разделяемые проекты в облаке и с контролем версий становятся незаменимы.

Правда, в случае MetaEditor, при разработке проекта в папке *MQL5/Shared Projects* следует учитывать, что заголовочные файлы из стандартного каталога *MQL5/Include* не попадают в общее хранилище. С другой стороны, создание выделенной папки *Include* внутри вашего проекта и перенос в неё необходимых стандартных mqh-файлов приведет к дублированию информации и потенциальным расхождениям в версиях заголовочных файлов. Это поведение, вероятно, будет совершенствоваться в MetaEditor.

Еще одним моментом для проектов, выходящих на уровень публичного, сетевого сервиса, является необходимость администрирования пользователей и их авторизация. В нашем последнем примере этот вопрос был только обозначен, но не прорабатывался. Однако сайт mql5.com предоставляет готовое решение на базе широко известного протокола OAuth. Ознакомиться с принципом работы OAuth и настроить его для своего веб-сервиса может каждый, у кого есть учетная запись mql5.com: достаточно найти раздел *Приложения* (ссылка вида *https://www.mql5.com/en/users/<login>/apps*) в своем профиле. За счет регистрации веб-сервиса

в приложениях mql5.com, вы получите возможность авторизовывать пользователей через сайт mql5.com.

7.9 Встроенная поддержка Python

Потенциальный успех автоматизированной торговли во многом определяется широтой технологий, которые доступны при реализации идеи. Как мы уже убедились по предыдущим разделам, MQL5 позволяет не ограничиваться строго прикладными трейдерскими задачами и дает возможности для интеграции с внешними сервисами (например, на основе сетевых функций и пользовательских символов), обработки и хранения данных средствами реляционной базы, а также подключения произвольных библиотек.

Последний пункт позволяет обеспечить взаимодействие с любым программным обеспечением, API которого есть в формате DLL. Некоторые разработчики используют этот способ для подключения к промышленным распределенным СУБД (вместо встроенной SQLite), математическим пакетам вроде R или MATLAB и даже другим языкам программирования.

Одним из наиболее популярных языков программирования последнего времени стал Python. Его особенностью является компактное ядро, которое дополняется пакетами — готовыми сборниками скриптов для построения прикладных решений. Именно богатство выбора и функционал пакетов полюбились трейдерам для фундаментального анализа рынка (статистических вычислений, визуализации данных) и проверки торговых гипотез, включая машинное обучение.

Следуя этой тенденции, компания MQ встроила в 2019 году поддержку Python в MQL5. Такая более тесная интеграция "прямо из коробки" позволяет полностью перенести технический анализ и торговые алгоритмы в среду Python.

С технической точки зрения интеграция достигается за счет установки в Python пакета "MetaTrader5", который организует межпроцессное взаимодействие с терминалом (на момент написания книги — через механизм ipykernel/RPC).

Среди функций пакета имеются полные аналоги встроенных функций MQL5 для получения информации о терминале, торговом счете, символах в *Обзоре рынка*, котировках, тиках, стакане цен, ордерах, позициях и сделках. Кроме того, пакет позволяет переключать торговый счет, отправлять торговые приказы, проверять залоговые требования и оценивать потенциальные прибыли/убытки в реальном режиме времени.

Вместе с тем, интеграция с Python имеет и некоторые ограничения. В частности, в Python невозможно реализовать обработку событий, таких как *OnTick*, *OnBookEvent* и других. Из-за этого для проверки новых цен необходимо использовать бесконечный цикл, примерно также, как мы были вынуждены делать в скриптах MQL5. Столь же затруднен и анализ исполнения торговых приказов: в отсутствие *OnTradeTransaction* потребуется больше кода, чтобы узнать, была ли позиция закрыта полностью или частично. Для обхода этих ограничений можно организовать взаимодействие скрипта Python и MQL5, например, через сокеты. На сайте mql5.com можно найти статьи с примерами реализации такого "моста".

Таким образом, представляется, что наиболее органично применять Python в связке с MetaTrader 5 для задач машинного обучения, в которых происходит анализ котировок, тиков или истории торгового счета. К сожалению, получить показания индикаторов в Python нельзя.

7.9.1 Установка Python и пакета MetaTrader5

Для изучения материалов данной Главы на компьютере должен быть установлен Python. Если его еще нет, скачайте последнюю версию Python (например, на момент написания книги — 3.10) со страницы *https://www.python.org/downloads/windows*.

При установке Python рекомендуется отметить флаг "Add Python to PATH", чтобы можно было запускать скрипты Python из командной строки, находясь в любой папке.

После того как Python скачан и работоспособен, установите модуль MetaTrader5 из командной строки (здесь *pip* — стандартная программа менеджер пакетов Python):

```
pip install MetaTrader5
```

Впоследствии можно проверять обновление пакета такой командной:

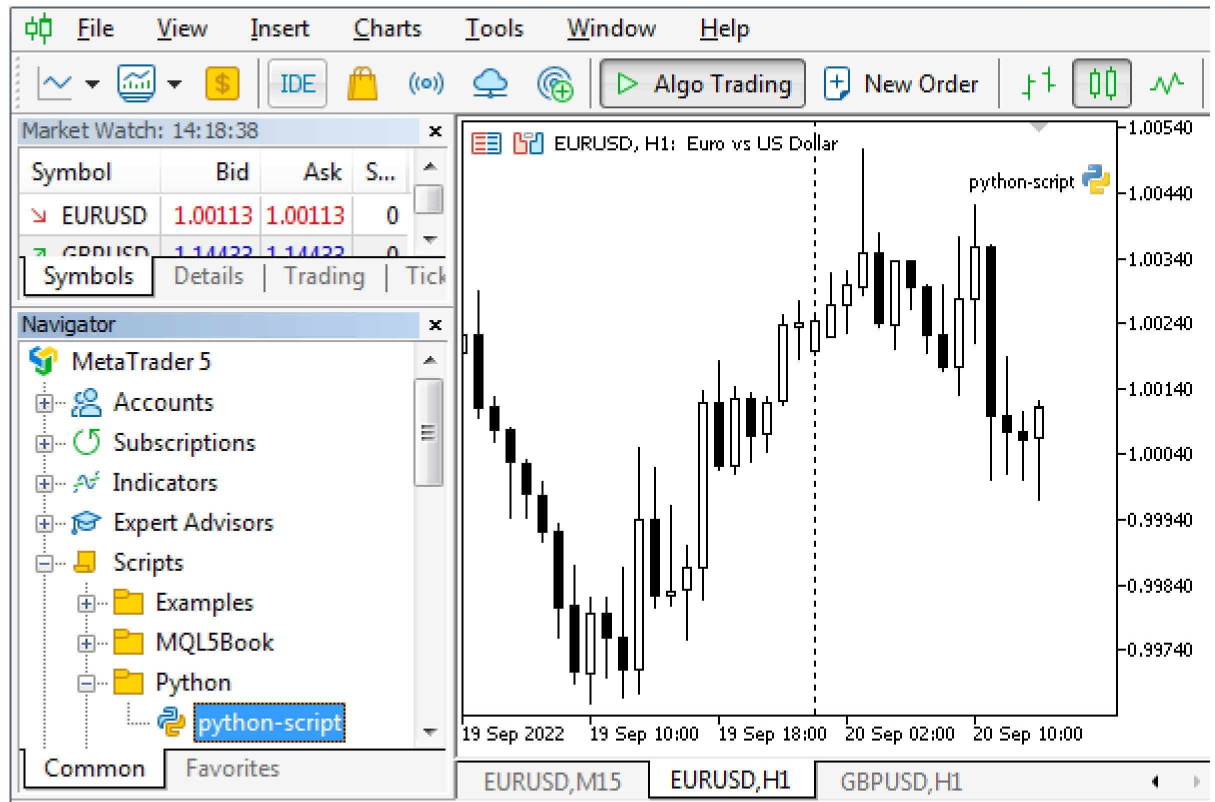
```
pip install --upgrade MetaTrader5
```

Для добавления других общеупотребительных пакетов используется аналогичный синтаксис. В частности, многие скрипты требуют пакеты анализа и визуализации данных — *pandas* и *matplotlib*, соответственно.

```
pip install matplotlib
pip install pandas
```

Создать новый скрипт на Python можно непосредственно из Мастера MQL5 в MetaEditor. Кроме имени скрипта пользователь может выбрать опции для импортирования нескольких пакетов, таких как *TensorFlow*, *NumPy* или *Datetime*.

Скрипты по умолчанию предлагается расположить в папке *MQL5/Scripts*. Вновь созданные и уже имеющиеся скрипты Python отображаются в навигаторе MetaTrader 5, помеченные особенной иконкой, и оттуда их можно запустить привычным способом. Python-скрипты могут выполняться на графике параллельно с другими MQL5-скриптами и советниками. Для остановки скрипта, если его исполнение зациклено, просто удалите его с графика.



Запуск скрипта Python в терминале

Скрипту Python, запущенному из терминала, через параметры командной строки передается название символа и таймфрейм графика. Например, мы можем запустить на графике EURUSD,H1 следующий скрипт, в котором аргументы доступны в виде массива *sys.argv*:

```
import sys

print('The command line arguments are:')
for i in sys.argv:
   print(i)
```

Он выведет в журнал экспертов:

```
The command line arguments are:
C:\Program Files\MetaTrader 5\MQL5\Scripts\MQL5Book\Python\python-args.py
EURUSD
60
```

Кроме того, скрипт Python можно запускать непосредственно из MetaEditor, если указать размещение установки Python в диалоге *Настроек* редактора, на закладке *Компиляторы* — тогда команда компиляции для файлов с расширением **.py* превращается в команду запуска.

Наконец, скрипты Python можно запускать и в своей родной среде, передавая их как параметры в вызовах *python.exe* из командной строки или из другой IDE (интегрированная среда разработки), адаптированной для Python, например, Jupyter Notebook.

Если в терминале разрешена алгоритмическая торговля, то по умолчанию разрешена и торговля из Python. Чтобы дополнительно защитить счета при использовании сторонних библиотек Python, в настройках платформы предусмотрена опция "Отключить автоматическую торговлю через внешний Python API". Таким образом, скриптам на Python можно избирательно

заблокировать торговлю, оставив её доступной для MQL-программ. Когда эта опция включена, вызовы торговых функций в скрипте Python будут возвращать ошибку 10027 (TRADE_RETCODE_CLIENT_DISABLES_AT) — алготрейдинг запрещен клиентским терминалом.

MQL5 vs Python

Python является интерпретируемым языком, в отличие от компилируемого MQL5. Для нас, как разработчиков это облегчает жизнь, потому что не нужна отдельная фаза компиляции, чтобы получить работающую программу. Однако скорость выполнения скриптов на Python заметно ниже, чем откомпилированных на MQL5.

Python — это язык с динамической типизацией: тип переменной определяется тем, какое значение мы в неё положили. С одной стороны это дает гибкость, но и требует осторожности, чтобы избежать непредвиденных ошибок. В MQL5 используется статическая типизации, то есть при описании переменных мы должны явно задать их тип, и компилятор следит за совместимостью типов.

Python сам "занимается" "уборкой мусора", то есть освобождает память, выделенную прикладной программой под объекты. В MQL5 мы должны следить за своевременным вызовом *delete* для динамических объектов.

В синтаксисе Python важную роль играют отступы в исходном коде. Если требуется записать составной оператор (например, цикл или условный) с блоком из нескольких вложенных операторов, то в Python для этой цели используются отступы пробелами или табуляцией (они должны быть равной величины внутри блока). Смешивать табуляции и пробелы запрещено. Неправильный отступ приведет к ошибке. В MQL5 мы формируем блоки составных операторов, заключая их в фигурные скобки { ... }, но форматирование при этом не играет роли, и вы можете применить любой понравившийся вам стиль, не нарушая работоспособности программы.

В функциях Python поддерживаются параметры двух типов: именованные и позиционные. Второй тип соответствует тому, к чему мы привыкли в MQL5: значение для каждого параметра должно передаваться строго на своем порядковом месте в списке аргументов (согласно прототипу функции). В отличие от этого, именованные параметры передаются в виде сочетания имени и значения (между ними ставится знак '=') и потому их можно указывать в произвольном порядке, например, *func(param2 = value2, param1 = value1)*.

7.9.2 Обзор функций пакета MetaTrader5 для Python

Функции API, доступные в Python, можно условно разделить на 2 группы: функции, имеющие полные аналоги в MQL5 API, и функции, доступные только в Python. Наличие второй группы отчасти обусловлено тем, что соединение Python и MetaTrader 5 должно быть организовано технически, прежде чем можно будет использовать прикладные функции. Это объясняет наличие и назначение пары функций *initialize* и *shutdown*: первая устанавливает подключение к терминалу, а вторая — завершает.

Важно, что в процессе инициализации может быть запущена требуемая копия терминала (если она еще не выполнялась) и выбран конкретный торговый счет. Кроме того, имеется возможность изменить торговый счет в контексте уже открытого подключения к терминалу: это делается функцией *login*.

После подключения к терминалу Python-скрипт может узнать краткую информацию о версии терминала с помощью функции *version*. Полная информация о терминале доступна через *terminal_info* — это полный аналог тройки *TerminalInfo*-функций, как бы объединенных в одном вызове.

В следующей таблице приведены прикладные функции Python и их аналоги в MQL5 API.

Python	**MQL5**
last_error	*GetLastError* (Внимание! в Python собственные коды ошибок)
account_info	*AccountInfoInteger*, *AccountInfoDouble*, *AccountInfoString*
terminal_info	*TerminalInfoInteger*, *TerminalInfoDouble*, *TerminalInfoDouble*
symbols_total	*SymbolsTotal* (все символы, включая пользовательские и отключенные)
symbols_get	*SymbolsTotal* + *SymbolInfo*-функции
symbol_info	*SymbolInfoInteger*, *SymbolInfoDouble*, *SymbolInfoString*
symbol_info_tick	*SymbolInfoTick*
symbol_select	*SymbolSelect*
market_book_add	*MarketBookAdd*
market_book_get	*MarketBookGet*
market_book_release	*MarketBookRelease*
copy_rates_from	*CopyRates* (по количеству баров, начиная с даты/времени)
copy_rates_from_pos	*CopyRates* (по количеству баров, начиная с номера бара)
copy_rates_range	*CopyRates* (в диапазоне дат/времени)
copy_ticks_from	*CopyTicks* (по количеству тиков, начиная с указанного времени)
copy_ticks_range	*CopyTicksRange* (в указанном временном диапазоне)
orders_total	*OrdersTotal*
orders_get	*OrdersTotal* + *OrderGet*-функции
order_calc_margin	*OrderCalcMargin*
order_calc_profit	*OrderCalcProfit*
order_check	*OrderCheck*
order_send	*OrderSend*
positions_total	*PositionsTotal*
positions_get	*PositionsTotal* + *PositionGet*-функции

Python	MQL5
history_orders_total	*HistoryOrdersTotal*
history_orders_get	*HistoryOrdersTotal* + *HistoryOrderGet*-функции
history_deals_total	*HistoryDealsTotal*
history_deals_get	*HistoryDealsTotal* + *HistoryDealGet*-функции

Функции из API Python имеют несколько особенностей.

Как уже было отмечено, функции могут иметь именованные параметры: при вызове функции такие параметры указываются вместе с именем и значением, в каждой паре имени и значения они объединены знаком равенства '='. Порядок указания именованных параметров неважен (в отличие от позиционных параметров, которые используются в MQL5 и должны следовать в строгом порядке, оговоренном прототипом функции).

Функции Python работают с типами данных, присущими Python. Сюда входят не только привычные числа и строки, но и несколько составных типов, отчасти аналогичных массивам и структурам MQL5.

Так многие функции возвращают специальные структуры данных Python: кортежи (*tuple*) и именованные кортежи (*namedtuple*).

Кортеж — это последовательность элементов произвольного типа. Её можно рассматривать как массив, но в отличие от массива, элементы кортежа могут быть разных типов. Также кортеж можно рассматривать как набор полей структуры.

Еще большее сходство со структурой можно найти у именованных кортежей, где каждый элемент получает идентификатор. Если в обычном кортеже для доступа к элементу можно использовать только индекс (в квадратных скобках, как и в MQL5, то есть [i]), то к именованному кортежу мы можем применить оператор разыменования (точку '.'), чтобы получить его "свойство" точно также как в структуре MQL5 (*tuple.field*).

Кроме того, кортежи и именованные кортежи не могут редактироваться в коде (то есть, являются константами).

Еще одним востребованным типом является словарь — ассоциативный массив, в котором хранятся пары ключ и значение, причем типы того и другого могут варьироваться. Доступ к значению словаря осуществляется с помощью оператора [], а между квадратных скобок указывается ключ (какого бы типа он ни был, например, строка), и в этом плане словари похожи на массивы. В словаре не может быть двух пар с одним и тем же ключом, то есть ключи всегда уникальны. В частности, именованный кортеж легко превратить в словарь с помощью метода *namedtuple._asdict()*.

7.9.3 Подключение скрипта Python к терминалу и счету

Функция *initialize* устанавливает соединение с терминалом MetaTrader 5 и имеет 2 формы: краткую (без параметров) и полную (с несколькими опциональными параметрами, первый из которых — *path* — позиционный, а все остальные — именованные).

bool initialize()
bool initialize(path, account = <ACCOUNT>, password = <"PASSWORD">,
  server = <"SERVER">, timeout = 60000, portable = False)

Параметр *path* задает путь к файлу терминала (*metatrader64.exe*) (обратите внимание: это неименованный параметр, в отличие от всех остальных, поэтому, если он указывается, должен идти первым в списке).

Если путь не указан, модуль попытается найти исполняемый файл самостоятельно (точный алгоритм разработчики не раскрывают). Для исключения неоднозначностей используйте вторую форму функции с параметрами.

В параметре *account* можно указать номер торгового счета. Если он не указан, то будет использован последний торговый счет в выбранном экземпляре терминала.

Пароль к торговому счету задается в параметре *password*, и тоже может быть опущен: в этом случае автоматически подставляется сохраненный в базе терминала пароль для указанного торгового счета.

По похожему принципу обрабатывается параметр *server* с именем торгового сервера (в том виде, как оно задано в терминале): если оно не указано, то автоматически подставляется сохраненный в базе терминала сервер для указанного торгового счета.

В параметре *timeout* указывается таймаут в миллисекундах, который дается на подключение (при превышении возникнет ошибка). По умолчанию используется значение 60000 (60 секунд).

Параметр *portable* содержит признак запуска терминала в "перемещаемом" режиме (по умолчанию равен *False*).

Функция возвращает *True* в случае успешного подключения к терминалу MetaTrader 5, а иначе — *False*.

Если потребуется, при выполнении вызова *initialize* может быть запущен терминал MetaTrader 5.

Например, подключение к конкретному торговому счету производится следующим образом.

```
import MetaTrader5 as mt5
if not mt5.initialize(login = 562175752, server = "MetaQuotes-Demo", password = "abc"
   print("initialize() failed, error code =", mt5.last_error())
   quit()
...
```

Еще одна функция *login* тоже подключается к торговому счету с указанными параметрами. Но при этом подразумевается, что соединение с терминалом уже установлено, то есть обычно функция используется для изменения счета.

bool login(account, password = <"PASSWORD">, server = <"SERVER">, timeout = 60000)

В параметре *account* указывается номер торгового счета. Это обязательный неименованный параметр, то есть он должен идти первым в списке.

Параметры *password*, *server* и *timeout* полностью аналогичны одноименным параметрам функции *initialize*.

Функция возвращает *True* в случае успешного подключения к торговому счету, а иначе — *False*.

shutdown()

Функция *shutdown* закрывает ранее установленное подключение к терминалу MetaTrader 5.

Пример использования вышеописанных функций см. в следующем разделе.

Когда соединение установлено, скрипт может узнать версию терминала.

tuple version()

Функция *version* возвращает краткую информацию о версии терминала MetaTrader 5 в виде кортежа из трех значений: номера версии, номера и даты сборки.

Тип поля	Описание
integer	Версия терминала MetaTrader 5 (текущая, 500)
integer	Номер сборки (например, 3456)
string	Дата сборки (например, '25 Feb 2022')

В случае ошибки функция вернет *None*, а код ошибки можно получить с помощью *last_error*.

Более полную информацию о терминале позволяет получить функция *terminal_info*.

7.9.4 Проверка ошибок: last_error

Функция *last_error* возвращает информацию о последней ошибке Python.

int last_error()

Целочисленные коды ошибок отличаются от тех, что выделены для ошибок MQL5 и возвращаются стандартной функцией *GetLastError*. В следующей таблице под сокращением IPC подразумевается термин "межпроцессного взаимодействия" (Inter-Process Communication).

Константа	Значение	Описание
RES_S_OK	1	Успех
RES_E_FAIL	-1	Общая ошибка
RES_E_INVALID_PARAMS	-2	Неверные аргументы/параметры
RES_E_NO_MEMORY	-3	Ошибка выделения памяти
RES_E_NOT_FOUND	-4	Запрошенная история не найдена
RES_E_INVALID_VERSION	-5	Версия не поддерживается
RES_E_AUTH_FAILED	-6	Ошибка авторизации
RES_E_UNSUPPORTED	-7	Метод не поддерживается
RES_E_AUTO_TRADING_DISABLED	-8	Алготрейдинг отключен
RES_E_INTERNAL_FAIL	-10000	Общая внутренняя ошибка IPC
RES_E_INTERNAL_FAIL_SEND	-10001	Внутренняя ошибка отправки данных IPC
RES_E_INTERNAL_FAIL_RECEIVE	-10002	Внутренняя ошибка отправки данных IPC
RES_E_INTERNAL_FAIL_INIT	-10003	Внутренняя ошибка инициализации IPC
RES_E_INTERNAL_FAIL_CONNECT	-10003	IPC отсутствует
RES_E_INTERNAL_FAIL_TIMEOUT	-10005	Таймаут IPC

В следующем скрипте (*MQL5/Scripts/MQL5Book/Python/init.py*) в случае ошибки при подключении к терминалу, выводим код ошибки и завершаем работу.

```
import MetaTrader5 as mt5
# покажем версию пакета MetaTrader5
print("MetaTrader5 package version: ", mt5.__version__)  #  5.0.37

# пробуем установить подключение или запустить терминал MetaTrader 5
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()
... # рабочая часть скрипта будет здесь
# завершаем подключение к терминалу
mt5.shutdown()
```

7.9.5 Получение информации о торговом счете

Функция *account_info* получает полную информацию о текущем торговом счете.

namedtuple account_info()

Функция возвращает информацию в виде структуры именованных кортежей (*namedtuple*). В случае ошибки результат равен значению *None*.

С помощью данной функции за один вызов можно получить всю информацию, которую в MQL5 предоставляют *AccountInfoInteger*, *AccountInfoDouble* и *AccountInfoString* со всеми вариантами поддерживаемых свойств. Названия полей в кортеже соответствуют названиям элементов перечислений без приставки "ACCOUNT_", приведенным к нижнему регистру.

К книге прилагается следующий скрипт *MQL5/Scripts/MQL5Book/Python/accountinfo.py*.

```
import MetaTrader5 as mt5

# установим подключение к терминалу MetaTrader 5
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

account_info = mt5.account_info()
if account_info != None:
   # выведем данные о торговом счете как есть
   print(account_info)
   # выведем данные о торговом счете в виде словаря
   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]))

# завершим подключение к терминалу MetaTrader 5
mt5.shutdown()
```

Результатом должен быть примерно такой вывод.

```
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 Получение информации о терминале

Функция *terminal_info* позволяет получить состояние и параметры подключенного терминала MetaTrader 5.

namedtuple terminal_info()

При успешном выполнении функция возвращает информацию в виде структуры именованных кортежей (*namedtuple*), а в случае ошибки - *None*.

С помощью данной функции за один вызов можно получить всю информацию, которую в MQL5 предоставляют *TerminalInfoInteger*, *TerminalInfoDouble* и *TerminalInfoDouble* со всеми вариантами поддерживаемых свойств. Названия полей в кортеже соответствуют названиям элементов перечислений без приставки "TERMINAL_", приведенным к нижнему регистру.

Например (см. *MQL5/Scripts/MQL5Book/Python/terminalinfo.py*):

```
import MetaTrader5 as mt5

# установим подключение к терминалу MetaTrader 5
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# выведем краткую информацию о версии MetaTrader 5
print(mt5.version())
# выведем полную информацию о настройках и состоянии терминала
terminal_info = mt5.terminal_info()
if terminal_info != None:
   # выведем данные о терминале как есть
   print(terminal_info)
   # выведем данные в виде словаря
   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]))

# завершим подключение к терминалу MetaTrader 5
mt5.shutdown()
```

Результат этого скрипта должен быть примерно следующим.

```
[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 Получение информации о финансовых инструментах

Группа функций пакета MetaTrader5 предоставляет информацию о финансовых инструментах.

Функция *symbol_info* возвращает информацию об одном финансовом инструменте в виде структуры именованного кортежа.

namedtuple symbol_info(symbol)

Имя интересующего вас финансового инструмента задается в параметре *symbol*.

За один вызов предоставляется вся информация, которую можно получить с помощью трех функций MQL5 со всеми свойствами: *SymbolInfoInteger*, *SymbolInfoDouble* и *SymbolInfoString*. Названия полей в именованном кортеже совпадают с названиями элементов перечислений, используемых в указанных функциях, но без префикса "SYMBOL_" и в нижнем регистре.

В случае ошибки функция возвращает *None*.

> Внимание! Для успешного выполнения функции запрашиваемый символ должен быть выбран в *Обзоре рынка*. Это можно сделать из Python с помощью вызова *symbol_select* (см. далее).

Пример (*MQL5/Scripts/MQL5Book/Python/eurjpy.py*):

```
import MetaTrader5 as mt5

# установим подключение к терминалу MetaTrader 5
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# убедимся, что EURJPY присутствует в Обзоре рынка, или прерываем алгоритм
selected = mt5.symbol_select("EURJPY", True)
if not selected:
   print("Failed to select EURJPY")
   mt5.shutdown()
   quit()

# выведем свойства символа EURJPY
symbol_info = mt5.symbol_info("EURJPY")
if symbol_info != None:
   # выведем данные как есть (как кортеж)
   print(symbol_info)
   # выведем пару конкретных свойств
   print("EURJPY: spread =", symbol_info.spread, ", digits =", symbol_info.digits)
   # выведем свойства символа в виде словаря
   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]))

# завершим подключение к терминалу MetaTrader 5
mt5.shutdown()
```

Результат:

```
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)

Функция *symbol_select* включает указанный символ в *Обзор рынка* или исключает из него. Символ задается первым параметром. Во втором параметре передается *True* или *False*, что означает, соответственно показ или скрытие символа.

Если второй необязательный неименованный параметр опущен, то по правилам приведения типов Python, *bool(None)* эквивалентно *False*.

Функция является аналогом *SymbolSelect*.

int symbols_total()

Функция *symbols_total* возвращает количество всех инструментов в терминале MetaTrader 5, с учетом пользовательских и не показанных в данный момент в окне *Обзор рынка*. Аналог функции *SymbolsTotal(false)*.

Следующая функция *symbols_get* возвращает массив кортежей с информацией о всех инструментах или избранных инструментах с названиями, удовлетворяющими заданному фильтру в необязательном именованном параметре *group*.

tuple[] symbols_get(group = "PATTERN")

Каждый элемент в кортеже-массиве является именованным кортежем с полным набором свойств по символу (подобный кортеж мы видели выше в контексте описания функции *symbol_info*).

Поскольку параметр только один, его название при вызове функции можно опускать.

В случае ошибки функция вернет специальное значение *None*.

Параметр *group* позволяет выбирать символы по имени, при необходимости используя знак подстановки '*' в начале и/или конце искомой строки. '*' означает 0 или несколько любых символов. Таким образом можно организовать поиск подстроки, которая встречается в имени с произвольным количеством других символов до или после указанного фрагмента. Например, "EUR*" означает символы, начинающиеся на "EUR" и имеющие любое продолжение имени (или просто "EUR"). Фильтр "*EUR*" вернет символы, в имени которых подстрока "EUR" встретится в любом месте.

Также параметр *group* может содержать несколько условий, разделенных запятыми. Каждое условие можно задавать как маску с использованием '*'. Для исключения инструментов можно использовать знак логического отрицания '!'. При этом все условия применяются последовательно, то есть сначала необходимо указать условия включения, а затем условия исключения. Например, group="*, !*EUR*" означает, что сначала нужно выбрать все символы и затем исключить те из них, что содержат в имени "EUR" (в любом месте).

Например, для вывода информации о кросс-курсах валют за исключением 4-х основных валют Forex, можно выполнить такой запрос:

```
crosses = mt5.symbols_get(group = "*,!*USD*,!*EUR*,!*JPY*,!*GBP*")
print('len(*,!*USD*,!*EUR*,!*JPY*,!*GBP*):', len(crosses)) # размер полученного масси
for s in crosses:
   print(s.name, ":", s)
```

Пример результата:

```
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_de
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_
```

Функция *symbol_info_tick* позволяет получить последний тик по указанному финансовому инструменту.

tuple symbol_info_tick(symbol)

В единственном обязательном параметре задается имя финансового инструмента.

Информация возвращается в виде кортежа с такими же полями, как в структуре *MqlTick*. Функция является аналогом *SymbolInfoTick*.

В случае ошибки возвращается *None*.

Для нормальной работы функции нужно, чтобы символ был включен в *Обзор рынка*. Продемонстрируем это в скрипте *MQL5/Scripts/MQL5Book/Python/gbpusdtick.py*.

```
import MetaTrader5 as mt5

# установим подключение к терминалу MetaTrader 5
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# попробуем включить символ GBPUSD в Обзоре рынка
selected=mt5.symbol_select("GBPUSD", True)
if not selected:
   print("Failed to select GBPUSD")
   mt5.shutdown()
   quit()

# выведем последний тик по символу GBPUSD в виде кортежа
lasttick = mt5.symbol_info_tick("GBPUSD")
print(lasttick)
# выведем значения полей тика в виде словаря
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]))

# завершим подключение к терминалу MetaTrader 5
mt5.shutdown()
```

Результат должен быть примерно таким:

```
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 Подписка на стакан цен

Python API включает 3 функции для работы со стаканом цен.

bool market_book_add(symbol)

Функция *market_book_add* производит подписку на получение событий об изменениях в стакане по указанному символу. Имя требуемого финансового инструмента указывается в единственном неименованном параметре.

Функция возвращает логический признак успешного выполнения.

Функция является аналогом *MarketBookAdd*. После завершения работы со стаканом, подписку следует отменить вызовом *market_book_release* (см. далее).

tuple[] market_book_get(symbol)

Функция *market_book_get* запрашивает текущее содержимое стакана цен для указанного символа. Результат возвращается в виде кортежа (массива) записей *BookInfo*. Каждая запись является аналогом структуры *MqlBookInfo*, а с точки зрения Python — это именованный кортеж с полями "type", "price", "volume", "volume_real". В случае ошибки возвращается значение *None*.

> Обратите внимание: по каким-то причинам в Python поле называется *volume_dbl*, хотя в MQL5 соответствующее поле называется *volume_real*.

Для работы с этой функцией требуется предварительно подписаться на получение событий в стакане цен с помощью функции *market_book_add*.

Функция является аналогом *MarketBookGet*. Обратите внимание, что скрипт Python не может получать события *OnBookEvent* напрямую и должен опрашивать содержимое стакана в цикле.

bool market_book_release(symbol)

Функция *market_book_release* отменяет подписку на получение событий об изменениях в стакане по указанному символу. В случае успешного выполнения функция возвращает *True*. Функция является аналогом *MarketBookRelease*.

Приведем простой пример (см. *MQL5/Scripts/MQL5Book/Python/eurusdbook.py*).

```
import MetaTrader5 as mt5
import time                  # подключаем пакет для паузы

# установим подключение к терминалу MetaTrader 5
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   mt5.shutdown()
   quit()

# подпишемся на получение обновлений стакана по символу EURUSD
if mt5.market_book_add('EURUSD'):
   # запустим 10 раз цикл для чтения данных из стакана
   for i in range(10):
      # получим содержимое стакана
      items = mt5.market_book_get('EURUSD')
      # выведем весь стакан одной строкой как есть
      print(items)
      # теперь выведем каждый ценовой уровень отдельно в виде словаря, для наглядност
      for it in items or []:
         print(it._asdict())
      # сделаем паузу в 5 секунд перед следующим запросом данных из стакана
      time.sleep(5)
   # отменим подписку на изменения стакана
   mt5.market_book_release('EURUSD')
else:
   print("mt5.market_book_add('EURUSD') failed, error code =", mt5.last_error())

# завершим подключение к терминалу MetaTrader 5
mt5.shutdown()
```

Пример результата:

```
(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 Чтение котировок

Python API позволяет получить массивы цен (баров) с помощью 3 функций, отличающихся способом указания диапазона запрашиваемых данных: по номерам баров или по времени. Все функции аналогичны различным формам *CopyRates*.

У всех функций два первых параметра служат для указания имени символа и таймфрейма. Для таймфреймов в mt5 определено перечисление TIMEFRAME, аналогичное перечислению ENUM_TIMEFRAMES в MQL5.

> Обратите внимание: по каким-то причинам в Python элементы этого перечисления имеют префикс TIMEFRAME_, в то время как элементы аналогичного перечисления в MQL5 имеют префиксы PERIOD_.

Идентификатор	Описание
TIMEFRAME_M1	1 минута
TIMEFRAME_M2	2 минуты
TIMEFRAME_M3	3 минуты
TIMEFRAME_M4	4 минуты
TIMEFRAME_M5	5 минут

Идентификатор	Описание
TIMEFRAME_M6	6 минут
TIMEFRAME_M10	10 минут
TIMEFRAME_M12	12 минут
TIMEFRAME_M12	15 минут
TIMEFRAME_M20	20 минут
TIMEFRAME_M30	30 минут
TIMEFRAME_H1	1 час
TIMEFRAME_H2	2 часа
TIMEFRAME_H3	3 часа
TIMEFRAME_H4	4 часа
TIMEFRAME_H6	6 часов
TIMEFRAME_H8	8 часов
TIMEFRAME_H12	12 часов
TIMEFRAME_D1	1 день
TIMEFRAME_W1	1 неделя
TIMEFRAME_MN1	1 месяц

Все 3 функции возвращают бары в виде массива пакета *numpy* с именованными столбцами *time*, *open*, *high*, *low*, *close*, *tick_volume*, *spread* и *real_volume*. Массив *numpy.ndarray* является более эффективным аналогом именованных кортежей. Для доступа к колонкам используйте нотацию с квадратными скобками, *array['column']*.

В случае ошибки возвращается *None*.

Все параметры функций являются обязательными, неименованными.

numpy.ndarray copy_rates_from(symbol, timeframe, date_from, count)

Функция *copy_rates_from* запрашивает бары, начиная с указанной даты (*date_from*) в количестве *count* баров. Дату можно задавать объектом *datetime* или в виде количества секунд, прошедших с 1970.01.01.

При создании объекта *datetime* Python использует локальный часовой пояс, в то время как терминал MetaTrader 5 хранит время тиков и открытия баров в UTC (GMT, без смещения). Поэтому, для выполнения функций, использующих время, необходимо создавать переменные *datetime* в UTC-времени. Для настройки таймзон можно использовать пакет *pytz*. Например (см. *MQL5/Scripts/MQL5Book/Python/eurusdrates.py*),

```
from datetime import datetime
import MetaTrader5 as mt5
import pytz                    # импортируем модуль pytz для работы с таймзоной
# установим подключение к терминалу MetaTrader 5
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   mt5.shutdown()
   quit()

# установим таймзону в UTC
timezone = pytz.timezone("Etc/UTC")

# создадим объект datetime в таймзоне UTC, чтобы не применялось смещение локальной та
utc_from = datetime(2022, 1, 10, tzinfo = timezone)

# получим 10 баров с EURUSD H1 начиная с 01.10.2022 в таймзоне UTC
rates = mt5.copy_rates_from("EURUSD", mt5.TIMEFRAME_H1, utc_from, 10)

# завершим подключение к терминалу MetaTrader 5
mt5.shutdown()

# выведем каждый элемент полученных данных (кортеж)
for rate in rates:
   print(rate)
```

Образец полученных данных:

```
(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)

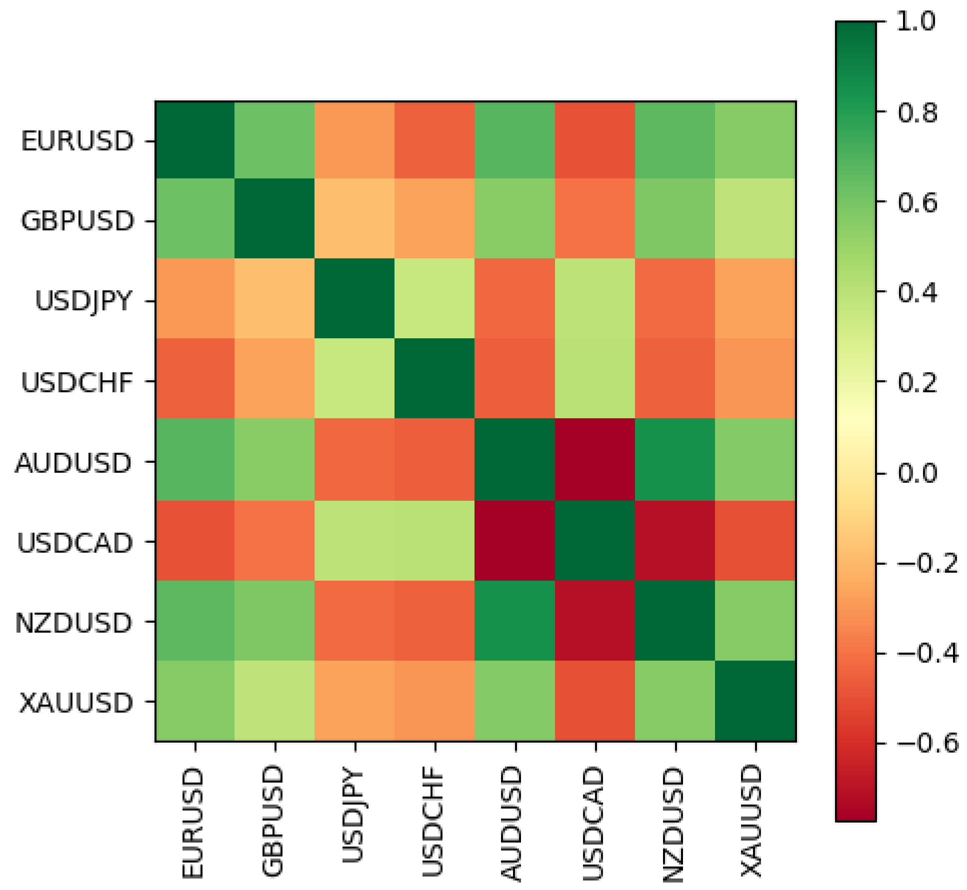
Функция *copy_rates_from_pos* запрашивает бары, начиная с указанного индекса *start*, в количестве *count*.

> Терминал MetaTrader 5 отдает бары только в пределах истории, доступной пользователю на графиках. Количество баров, которые доступны пользователю, задается в настройках параметром "Макс. баров в окне".

Следующий пример (*MQL5/Scripts/MQL5Book/Python/ratescorr.py*) демонстрирует графическое представление корреляционной матрицы нескольких валют на основе котировок.

```
import MetaTrader5 as mt5
import pandas as pd               # подключаем модуль pandas для вывода данных
import matplotlib.pyplot as plt   # подключаем модуль matplotlib для рисования

# установим подключение к терминалу MetaTrader 5
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   mt5.shutdown()
   quit()

# создадим путь в песочнице для картинки с результатом
image = mt5.terminal_info().data_path + r'\MQL5\Files\MQL5Book\ratescorr'

# список рабочих валют для расчета корреляции
sym = ['EURUSD','GBPUSD','USDJPY','USDCHF','AUDUSD','USDCAD','NZDUSD','XAUUSD']

# копируем цены закрытия баров в структуры DataFrame
d = pd.DataFrame()
for i in sym:        # для каждого символа по 1000 последних баров M1
   rates = mt5.copy_rates_from_pos(i, mt5.TIMEFRAME_M1, 0, 1000)
   d[i] = [y['close'] for y in rates]

# завершим подключение к терминалу MetaTrader 5
mt5.shutdown()

# вычислим изменение цен в процентах
rets = d.pct_change()

# вычислим корреляции
corr = rets.corr()

# рисуем корреляционную матрицу
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)
```

Файл с изображением *ratescorr.png* формируется в "песочнице" текущей рабочей копии MetaTrader 5. Интерактивный показ изображения в отдельном окне с помощью вызова *plt.show()* может не сработать, если ваша установка Python не включает дополнительную опцию (Optional Features) — "tcl/tk and IDLE" или если не добавить пакет *pip install tk*.



Корреляционная матрица валют Forex

numpy.ndarray copy_rates_range(symbol, timeframe, date_from, date_to)

Функция *copy_rates_range* позволяет получить бары в указанном диапазоне даты и времени, между *date_from* и *date_to*: оба значения задаются как количество секунд с начала 1970-го года, в часовом поясе UTC (поскольку Python использует в переменных *datetime* локальную таймзону, выполняйте конвертацию с помощью модуля *pytz*). В результат попадают бары с временем открытия *time* >= *date_from* и *time* <= *date_to*.

В следующем скрипте запросим бары в конкретном временном диапазоне.

```
from datetime import datetime
import MetaTrader5 as mt5
import pytz             # подключаем модуль pytz для работы с таймзоной
import pandas as pd     # подключаем модуль pandas для вывода данных в табличной форм

pd.set_option('display.max_columns', 500) # сколько столбцов показываем
pd.set_option('display.width', 1500)      # макс. ширина таблицы для показа

# установим подключение к терминалу MetaTrader 5
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# установим таймзону в UTC
timezone = pytz.timezone("Etc/UTC")
# создадим объекты datetime в таймзоне UTC, чтобы не применялось смещение локальной т
utc_from = datetime(2020, 1, 10, tzinfo=timezone)
utc_to = datetime(2020, 1, 10, minute = 30, tzinfo=timezone)

# получим бары с USDJPY M5 в интервале 2020.01.10 00:00 - 2020.01.10 00:30 в таймзоне
rates = mt5.copy_rates_range("USDJPY", mt5.TIMEFRAME_M5, utc_from, utc_to)

# завершим подключение к терминалу MetaTrader 5
mt5.shutdown()

# создадим из полученных данных DataFrame
rates_frame = pd.DataFrame(rates)
# сконвертируем время из количества секунд в формат datetime
rates_frame['time'] = pd.to_datetime(rates_frame['time'], unit = 's')

# выведем данные
print(rates_frame)
```

Пример результата:

```
                 time     open     high      low    close  tick_volume  spread  real_
0 2020-01-10 00:00:00  109.513  109.527  109.505  109.521           43       2
1 2020-01-10 00:05:00  109.521  109.549  109.518  109.543          215       8
2 2020-01-10 00:10:00  109.543  109.543  109.466  109.505           98      10
3 2020-01-10 00:15:00  109.504  109.534  109.502  109.517          155       8
4 2020-01-10 00:20:00  109.517  109.539  109.513  109.527           71       4
5 2020-01-10 00:25:00  109.526  109.537  109.484  109.520          106       9
6 2020-01-10 00:30:00  109.520  109.524  109.508  109.510          205       7
```

7.9.10 Чтение истории тиков

Python API включает 2 функции для чтения истории реальных тиков: *copy_ticks_from* — с указанием количества тиков, начиная с указанной даты, и *copy_ticks_range* — для всех тиков за указанный период.

Обе функции имеют 4 обязательных неименованных параметра, первый из которых предназначен для указания символа. Второй параметр задает начальное время запрашиваемых тиков. В третьем параметре передается либо требуемое количество тиков (в функции *copy_ticks_from*), либо конечное время тиков (в функции *copy_ticks_range*).

Последний параметр определяет, какого рода тики будут возвращены. В нем можно указать один из следующих флагов (COPY_TICKS):

Идентификатор	Описание
COPY_TICKS_ALL	Все тики
COPY_TICKS_INFO	Тики, содержащие изменения цен Bid и/или Ask
COPY_TICKS_TRADE	Тики, содержащие изменения цены Last и/или объема (Volume)

Обе функции возвращают тики в виде массива *numpy.ndarray* (из пакета *numpy*) с именованными столбцами *time*, *bid*, *ask*, *last* и *flags*. Значение поля *flags* является комбинацией битовых флагов из перечисления TICK_FLAG: каждый бит означает изменение соответствующего поля со свойством тика.

Идентификатор	Измененное свойство тика
TICK_FLAG_BID	Цена Bid
TICK_FLAG_ASK	Цена Ask
TICK_FLAG_LAST	Цена Last
TICK_FLAG_VOLUME	Объем (Volume)
TICK_FLAG_BUY	Цена последней покупки (Buy)
TICK_FLAG_SELL	Цена последней продажи (Sell)

numpy.ndarray copy_ticks_from(symbol, date_from, count, flags)

Функция *copy_ticks_from* запрашивает тики, начиная с указанного времени (*date_from*) в заданном количестве (*count*).

Функция является аналогом *CopyTicks*.

numpy.array copy_ticks_range(symbol, date_from, date_to, flags)

Функция *copy_ticks_range* позволяет получить тики за указанный диапазон времени.

Функция является аналогом *CopyTicksRange*.

В следующем примере (*MQL5/Scripts/MQL5Book/Python/copyticks.py*) сгенерируем интерактивную веб-страницу с графиком тиков (внимание: используется пакет *plotly* — напомним, что для его установки в Python нужно предварительно выполнить команду *pip install plotly*).

```
import MetaTrader5 as mt5
import pandas as pd
import pytz
from datetime import datetime

# подключаемся к терминалу
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# зададим имя файла для сохранения в "песочницу"
path = mt5.terminal_info().data_path + r'\MQL5\Files\MQL5Book\copyticks.html'

# копируем 1000 тиков EURUSD с конкретного момента в истории
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')

# завершим подключение к терминалу
mt5.shutdown()

# подключаем графический пакет и рисуем 2 ряда цен ask и bid в веб-странице
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)
```

Вот как может выглядеть результат.

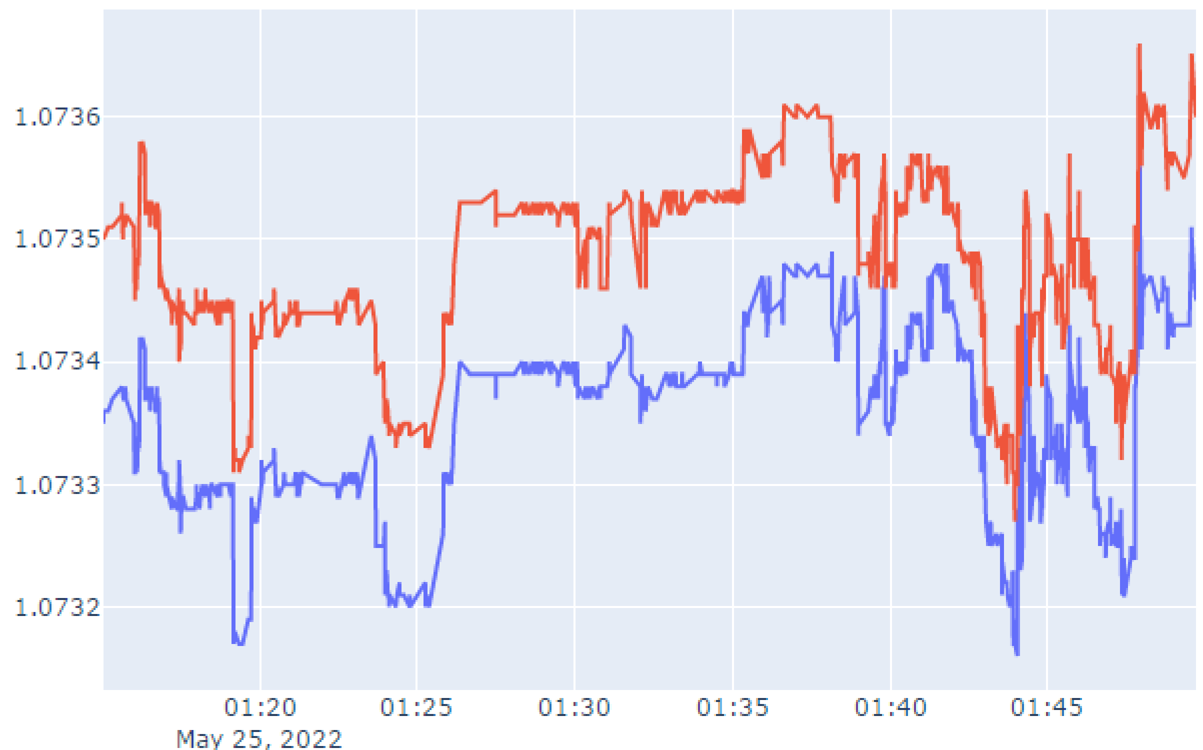


График тиков, полученных в скрипте Python

Веб-страница *copyticks.html* сгенерируется в подкаталоге *MQL5/Files/MQL5Book*.

7.9.11 Вычисление маржинальных требований и оценка прибыли

Разработчик на Python может непосредственно в скрипте вычислить залог и потенциальную прибыль или убыток предполагаемой торговой операции с помощью функций *order_calc_margin* и *order_calc_profit*. В случае успешного выполнения результат любой функций — вещественное число, а иначе — *None*.

float order_calc_margin(action, symbol, volume, price)

Функция *order_calc_margin* возвращает размер маржи (в валюте счета), необходимой для проведения указанной торговой операции *action* — один из двух элементов перечисления ENUM_ORDER_TYPE: ORDER_TYPE_BUY или ORDER_TYPE_SELL. В следующих параметрах задается имя финансового инструмента, объем торговой операции и цена открытия.

Функция является аналогом *OrderCalcMargin*.

float order_calc_profit(action, symbol, volume, price_open, price_close)

Функция *order_calc_profit* возвращает размер прибыли или убытка (в валюте счета) для указанного типа торговой операции, символа и объема, а также разницы в ценах входа и выхода из рынка.

Функция является аналогом *OrderCalcProfit*.

Рекомендуется делать проверку залога и предположительного результата торговой операции перед отправкой приказа.

7.9.12 Проверка и отправка торгового приказа

При необходимости вы можете осуществлять торговые операции непосредственно из скрипта Python. Пара функций — *order_check* и *order_send* — позволяет предварительно проверить и затем выполнить торговую операцию.

У обеих функций единственным параметром является структура запроса *TradeRequest* (на Python её можно проинициализировать как словарь, см. пример). Поля структуры полностью совпадают с *MqlTradeRequest*.

OrderCheckResult order_check(request)

Функция *order_check* проверяет правильность заполнения полей торгового запроса и достаточность средств для совершения требуемой торговой операции.

Результат функции возвращается в виде структуры *OrderCheckResult*. Она повторяет структуру *MqlTradeCheckResult*, но дополнительно содержит поле *request* с копией исходного запроса.

Функция *order_check* является аналогом *OrderCheck*.

Пример (*MQL5/Scripts/MQL5Book/python/ordercheck.py*):

```
import MetaTrader5 as mt5

# установим подключение к терминалу MetaTrader 5
...
# получим валюту счета для информации
account_currency=mt5.account_info().currency
print("Account currency:", account_currency)

# получим необходимые свойства символа сделки
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
# если символ недоступен в Обзоре рынка, добавим его
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()

# подготовим структуру запроса как словарь
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,
}

# выполним проверку и выведем результат как есть
result = mt5.order_check(request)
print(result)                       # [?этого в логе хелпа нет?]

# преобразуем результат в словарь и выведем поэлементно
result_dict = result._asdict()
for field in result_dict.keys():
   print("   {}={}".format(field, result_dict[field]))
   # если это структура торгового запроса, то выведем её тоже поэлементно
```

```
        if field == "request":
            traderequest_dict = result_dict[field]._asdict()
            for tradereq_filed in traderequest_dict:
                print("       traderequest: {}={}".format(tradereq_filed,
                traderequest_dict[tradereq_filed]))

# завершим подключение к терминалу
mt5.shutdown()
```

Результат:

```
Account currency: 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,
       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)

Функция *order_send* отправляет из терминала на торговый сервер запрос на совершение торговой операции.

Результат функции возвращается в виде структуры *OrderSendResult*. Она повторяет структуру *MqlTradeResult*, но дополнительно содержит поле *request* с копией исходного запроса.

Функция является аналогом *OrderSend*.

Пример (*MQL5/Scripts/MQL5Book/python/ordersend.py*):

```
import time
import MetaTrader5 as mt5

# установим подключение к терминалу MetaTrader 5
...
# назначим свойства рабочего символа
symbol = "USDJPY"
symbol_info = mt5.symbol_info(symbol)
if symbol_info is None:
   print(symbol, "not found, can not trade")
   mt5.shutdown()
   quit()

# если символ недоступен в Обзоре рынка, добавим его
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()

# подготовим структуру запроса для покупки
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,
}

# отправим торговый запрос на открытие позиции
result = mt5.order_send(request)
# проверим результат выполнения
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))
   # запросим результат в виде словаря и выведем поэлементно
   result_dict = result._asdict()
   for field in result_dict.keys():
```

```
            print("   {}={}".format(field, result_dict[field]))
            # если это структура торгового запроса, то выведем её тоже поэлементно
            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)
   # создадим запрос на закрытие
   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,
   }
   # отправим торговый запрос на закрытие позиции
   result = mt5.order_send(request)
   # проверим результат выполнения
   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))
      # запросим результат в виде словаря и выведем поэлементно
      result_dict = result._asdict()
      for field in result_dict.keys():
         print("   {}={}".format(field, result_dict[field]))
         # если это структура торгового запроса, то выведем её тоже поэлементно
         if field == "request":
            traderequest_dict = result_dict[field]._asdict()
            for tradereq_filed in traderequest_dict:
               print("       traderequest: {}={}".format(tradereq_filed,
```

```
                traderequest_dict[tradereq_filed]))

    # завершим подключение к терминалу
    mt5.shutdown()
```

Результат:

```
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,
       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 Получение количества и списка действующих ордеров

Для работы с активными ордерами в Python API предусмотрены следующие функции.

int orders_total()

Функция *orders_total* возвращает количество действующих ордеров.

Функция является аналогом *OrdersTotal*.

Подробную информацию о каждом ордере позволяет получить функция *orders_get*, имеющая несколько вариантов с возможностью фильтрации по символу или тикету. В любом варианте функция возвращает массив именованных кортежей *TradeOrder* (имена полей соответствуют

ENUM_ORDER_PROPERTY-перечислениям без префикса "ORDER_" и приведены к нижнему регистру). В случае ошибки результат равен *None*.

```
namedtuple[] orders_get()
namedtuple[] orders_get(symbol = <"SYMBOL">)
namedtuple[] orders_get(group = <"PATTERN">)
namedtuple[] orders_get(ticket = <TICKET>)
```

Функция *orders_get* без параметров возвращает ордера по всем символам.

Необязательный именованный параметр *symbol* дает возможность указать конкретное имя символа для отбора ордеров.

Необязательный именованный параметр *group* предназначен для указания шаблона поиска с использованием символа подстановки '*' (как заместителя произвольного количества любых символов, включая ноль символов в данном месте шаблона) и символа логического отрицания условия '!'. Принципы работы с шаблоном фильтрации были описаны в разделе Получение информации о финансовых инструментах.

При указании параметра *ticket* ищется один конкретный ордер.

Функция позволяет получить за один вызов все действующие ордера и является аналогом связки *OrdersTotal*, *OrderSelect* и *OrderGet*-функций.

В следующем примере (*MQL5/Scripts/MQL5Book/Python/ordersget.py*) запросим информацию об ордерах разными способами.

```
import MetaTrader5 as mt5
import pandas as pd
pd.set_option('display.max_columns', 500) # сколько столбцов показываем
pd.set_option('display.width', 1500)      # макс. ширина таблицы для показа

# установим подключение к терминалу MetaTrader 5
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# выведем информацию о действующих ордерах на символе GBPUSD
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))
   # выведем все действующие ордера
   for order in orders:
      print(order)
print()

# получим список ордеров на символах, чьи имена содержат "*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)))
   # выведем ордера в виде таблицы с помощью 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)

# завершим подключение к терминалу MetaTrader 5
mt5.shutdown()
```

Примерный результат:

```
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
0  554733548 2020-03-25 16:27:47   1585153667718     3 ...            0.2     1.25379
1  554733621 2020-03-25 16:27:51   1585153671419     2 ...            0.2     1.14370
2  554746664 2020-03-25 16:38:14   1585154294401     3 ...            0.2     0.93851
3  554746710 2020-03-25 16:38:17   1585154297022     2 ...            0.2     0.90527
```

7.9.14 Получение количества и списка открытых позиций

Функция *positions_total* возвращает количество открытых позиций.

```
int positions_total()
```

Функция является аналогом *PositionsTotal*.

Для получения подробной информации о каждой позиции предназначена функция *positions_get*, имеющая несколько вариантов. Все варианты возвращают массив именованных кортежей *TradePosition* с ключами, соответствующими свойствам позиций (см. элементы ENUM_POSITION_PROPERTY-перечислений, без префикса "POSITION_", в нижнем регистре). В случае ошибки результат равен *None*.

```
namedtuple[] positions_get()
namedtuple[] positions_get(symbol = <"SYMBOL">)
namedtuple[] positions_get(group = <"PATTERN">)
namedtuple[] positions_get(ticket = <TICKET>)
```

Функция без параметров возвращает все открытые позиции.

Функция с параметром *symbol* позволяет отобрать только позиции по конкретному символу.

Функция с параметром *group* обеспечивает фильтрацию по маске поиска с символами подстановки '*' (заменяется любые символы) и логического отрицания условия '!'. Подробности см. в разделе Получение информации о финансовых инструментах.

Вариант с параметров *ticket* выбирает позицию с конкретным тикетом (свойством POSITION_TICKET).

Функция *positions_get* позволяет получить за один вызов все позиции и их свойства, что делает её аналогом связки *PositionsTotal*, *PositionSelect* и *PositionGet*-функций.

В скрипте *MQL5/Scripts/MQL5Book/Python/positionsget.py* запросим позиции по конкретному символу и маске поиска.

```
import MetaTrader5 as mt5
import pandas as pd
pd.set_option('display.max_columns', 500) # сколько столбцов показываем
pd.set_option('display.width', 1500)      # макс. ширина таблицы для показа

# установим подключение к терминалу MetaTrader 5
if not mt5.initialize():
   print("initialize() failed, error code =", mt5.last_error())
   quit()

# получим открытые позиции на 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))
   # выведем все открытые позиции
   for position in positions:
      print(position)

# получим список позиций на символах, чьи имена содержат "*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)))
   # выведем позиции в виде таблицы с помощью 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)

# завершим подключение к терминалу MetaTrader 5
mt5.shutdown()
```

Вот каким может быть результат:

```
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, v
   sl=0.0, tp=0.0, price_current=0.9853, swap=-0.01, profit=6.24, symbol='USDCHF', co
positions_get(group="*USD*") = 2
       ticket                time  type  ...  identifier  volume  price_open  ... _cu
0  1468454363 2022-09-26 18:33:53     1  ...  1468454363    0.01     0.99145  ...  0.
1  1468475849 2022-09-26 18:44:00     0  ...  1468475849    0.01     1.06740  ...  1.
```

7.9.15 Чтение истории ордеров и сделок

Работа с ордерами и сделками в истории счета также возможна в скриптах Python. Для этих целей предназначены функции *history_orders_total*, *history_orders_get*, *history_deals_total*, *history_deals_get*.

int history_orders_total(date_from, date_to)

Функция *history_orders_total* возвращает количество ордеров в торговой истории в указанном интервале времени. Каждый из параметров задается объектом *datetime* или в виде количества секунд, прошедших с начала 1970-го года.

Функция является аналогом *HistoryOrdersTotal*.

Функция *history_orders_get* доступна в нескольких вариантах и поддерживает фильтрацию ордеров по подстроке в имени символа, тикету или идентификатору позиции. Все варианты возвращают массив именованных кортежей *TradeOrder* (имена полей соответствуют ENUM_ORDER_PROPERTY_-перечислениям без префикса "ORDER_" и приведены к нижнему регистру). Если подходящих ордеров нет, массив будет пустым. В случае ошибки функция вернет *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>)

Первый вариант отбирает ордера в указанном диапазоне времени (как и *history_orders_total*). В опциональном именованном параметре *group* можно указать шаблон поиска по подстроке названия символа (в нем можно использовать подстановочные символы замещения '*' и отрицания '!', см. раздел Получение информации о финансовых инструментах).

Второй вариант предназначен для поиска конкретного ордера по его тикету.

Последний вариант производит выборку ордеров по идентификатору позиции (свойство ORDER_POSITION_ID).

Любой вариант эквивалентен вызову нескольких функций MQL5: *HistoryOrdersTotal*, *HistoryOrderSelect* и *HistoryOrderGet*-функций.

Покажем на примере скрипта *historyordersget.py*, как получить количество и перечень исторических ордеров по разным условиям.

```
from datetime import datetime
import MetaTrader5 as mt5
import pandas as pd
pd.set_option('display.max_columns', 500) # сколько столбцов показываем
pd.set_option('display.width', 1500)      # макс. ширина таблицы для показа
...
# получим количество ордеров в истории за период (всего и по *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_errc
else :
   print("history_orders_get({}, {}, group=\"*GBP*\")={} of total {}".format(from_dat
   to_date, len(history_orders), total))

# выведем все отмененные исторические ордера по тикету позиции 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)))
   # выведем полученные ордера как есть
   for position_order in position_history_orders:
      print(position_order)
   # выведем эти ордера в виде таблицы с помощью 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)
...
```

Результат работы скрипта (приводится с сокращениями):

```
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, ma
   position_id=0, position_by_id=0, reason=3, volume_initial=0.01, volume_current=0.01, price_open=
   sl=0.0, tp=0.0, price_current=0.99311, price_stoplimit=0.0, symbol='EURUSD', comment='', externa
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, ma
   position_id=0, position_by_id=0, reason=3, volume_initial=0.01, volume_current=0.01, price_open=
   sl=0.0, tp=0.0, price_current=0.99284, price_stoplimit=0.0, symbol='EURUSD', comment='', externa
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, ma
   position_id=0, position_by_id=0, reason=3, volume_initial=0.01, volume_current=0.01, price_open=
   sl=0.0, tp=0.0, price_current=0.99286, price_stoplimit=0.0, symbol='EURUSD', comment='', externa
...

        ticket          time_setup           time_done  type  ... _initial  price_open  price_curre
0   1437318706 2022-08-24 13:34:25 2022-08-24 13:34:43     2           0.01     0.99301        0.993
1   1437331579 2022-08-24 13:42:25 2022-08-24 13:42:31     2           0.01     0.99281        0.992
2   1437331739 2022-08-24 13:42:33 2022-08-24 13:42:43     2           0.01     0.99285        0.992
...
```

Мы видим, что за сентябрь было всего 44 ордера, 15 из которых включали валюту GBP (количество нечетное из-за открытой позиции). Всего в истории 14 отмененных ордеров.

int history_deals_total(date_from, date_to)

Функция *history_deals_total* возвращает количество сделок в истории за указанный период.

Функция является аналогом *HistoryDealsTotal*.

Функция *history_deals_get* имеет несколько форм и предназначена для выборки сделок с возможностью фильтрации по тикету ордера или идентификатору позиции. Все формы функции возвращают массив именованных кортежей *TradeDeal*, в полях которых отражены свойства из ENUM_DEAL_PROPERTY_-перечислений (в названиях полей исключен префикс "DEAL_" и применен нижний регистр). В случае ошибки получим *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>)

Первая форма функции работает по аналогии с запросом исторических ордеров с помощью *history_orders_get*.

Вторая форма позволяет отобрать сделки, порожденные конкретным ордером, по его тикету (свойство DEAL_ORDER).

Наконец, третья форма запрашивает сделки, сформировавшие позицию с заданным идентификатором (свойство DEAL_POSITION_ID).

Функция позволяет получить за один вызов все сделки вместе с их свойствами, что является аналогом связки *HistoryDealsTotal*, *HistoryDealSelect* и *HistoryDealGet*-функций.

Приведем основную часть тестового скрипта *historydealsget.py*.

```
# зададим временной диапазон
from_date = datetime(2020, 1, 1)
to_date = datetime.now()

# получим  сделки по символам, имена которых не содержат ни "EUR" ни "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))
   # выведем все полученные сделки как есть
   for deal in deals:
      print("  ",deal)
   # выведем эти сделки в виде таблицы с помощью 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)
```

Пример результата:

```
history_deals_get(from_date, to_date, group="*,!*EUR*,!*GBP*") = 12
   TradeDeal(ticket=1109160642, order=0, time=1632188460, time_msc=1632188460852, typ
   TradeDeal(ticket=1250629232, order=1268074569, time=1645709385, time_msc=164570938
   TradeDeal(ticket=1250639814, order=1268085019, time=1645709950, time_msc=164570995
   TradeDeal(ticket=1250639928, order=1268085129, time=1645709955, time_msc=164570995
   TradeDeal(ticket=1250640111, order=1268085315, time=1645709965, time_msc=164570996
   TradeDeal(ticket=1250640309, order=1268085512, time=1645709973, time_msc=164570997
   TradeDeal(ticket=1250640400, order=1268085611, time=1645709978, time_msc=164570997
   TradeDeal(ticket=1250640616, order=1268085826, time=1645709988, time_msc=164570998
   TradeDeal(ticket=1250640810, order=1268086019, time=1645709996, time_msc=164570999
   TradeDeal(ticket=1445796125, order=1468026008, time=1664199450, time_msc=166419945
   TradeDeal(ticket=1445796155, order=1468026041, time=1664199452, time_msc=166419945
   TradeDeal(ticket=1446217804, order=1468454363, time=1664217233, time_msc=166421723

        ticket       order                time t… e… … position_id  volume      pr
0   1109160642           0 2021-09-21 01:41:00  2  0            0    0.00    0.00
1   1250629232  1268074569 2022-02-24 13:29:45  0  0   1268074569    0.01  1970.98
2   1250639814  1268085019 2022-02-24 13:39:10  1  1   1268074569    0.01  1970.09
3   1250639928  1268085129 2022-02-24 13:39:15  1  0   1268085129    0.01  1969.98
4   1250640111  1268085315 2022-02-24 13:39:25  0  1   1268085129    0.01  1970.17
5   1250640309  1268085512 2022-02-24 13:39:33  1  0   1268085512    0.10  1970.09
6   1250640400  1268085611 2022-02-24 13:39:38  0  1   1268085512    0.10  1970.22
7   1250640616  1268085826 2022-02-24 13:39:48  1  0   1268085826    1.10  1969.95
8   1250640810  1268086019 2022-02-24 13:39:56  0  1   1268085826    1.10  1969.88
9   1445796125  1468026008 2022-09-26 13:37:30  0  0   1468026008    0.10   144.13
10  1445796155  1468026041 2022-09-26 13:37:32  1  1   1468026008    0.10   144.12
11  1446217804  1468454363 2022-09-26 18:33:53  1  0   1468454363    0.01     0.99
```

7.10 Встроенная поддержка параллельных вычислений: OpenCL

OpenCL — это открытый стандарт параллельного программирования, который позволяет создавать приложения для одновременного выполнения на множестве ядер современных процессоров, различных по архитектуре, в частности, графических (GPU) или центральных (CPU).

Другими словами OpenCL позволяет задействовать для вычислений одной задачи все ядра центрального процессора или все вычислительные мощности видеокарты, что, в конечном счете, уменьшает время выполнения программы. Поэтому использование OpenCL является очень полезным для задач, связанных с трудоемкими вычислениями, однако важно отметить, что алгоритмы решения этих задач должны поддаваться разделению на параллельные потоки. К ним относятся, например, обучение нейронных сетей, преобразование Фурье, решение систем уравнений больших размерностей.

Например, применительно к трейдерской специфике, увеличение быстродействия может быть достигнуто у скрипта, индикатора или эксперта, который проводит сложный и длительный анализ исторических данных по нескольким символам и таймфреймам, и расчет для каждого из которых не зависит от других.

> Вместе с тем, у начинающих часто возникает вопрос, можно ли с помощью OpenCL ускорить процессы тестирования и оптимизации советников. Ответы на оба вопроса: нет. Тестирование воспроизводит реальный процесс последовательной торговли и потому каждый следующий бар или тик зависит от результатов предыдущих, что делает невозможным распараллеливание расчетов одного прохода. Что же касается оптимизации, то агенты тестера поддерживают только ядра центрального процессора. Это связано со сложностью полноценного анализа котировок или тиков, отслеживания позиций и подсчета баланса и эквити. Однако, если сложность вас не пугает, вы можете реализовать собственный "движок" оптимизации на ядрах графических карт, перенеся все вычисления, эмулирующие торговое окружение с требуемой достоверностью, в OpenCL.

OpenCL "расшифровывается" как Open Computing Language — открытый язык вычислений. Он похож на языки C и C++, а стало быть, и на MQL5. Однако для того, чтобы подготовить ("откомпилировать") программу на OpenCL, передать в неё входные данные, запустить параллельно на нескольких ядрах и получить результаты вычислений, применяется специальный программный интерфейс (набор функций). Это OpenCL API доступно и для MQL-программ, желающих организовать параллельное исполнение.

Для использования OpenCL совсем не обязательно иметь видеокарту на Вашем ПК — вполне достаточно и наличия центрального процессора, но в любом случае требуется наличие специальных драйверов от производителя (требуется версия OpenCL 1.1 и выше). Если на вашем компьютере установлены игры или другой софт (например, научный, редактор видео и пр.), работающий напрямую с видеокартами, то необходимая программная "прослойка", скорее всего, уже имеется. Это можно проверить, попытавшись запустить в терминале MQL-программу с обращением к OpenCL (хотя бы простой пример из поставки терминала, см. далее).

При отсутствии поддержки OpenCL вы увидите в журнале ошибку.

```
OpenCL OpenCL not found, please install OpenCL drivers
```

Если же на вашем компьютере есть подходящее устройство и для него была включена поддержка OpenCL, терминал выведет сообщение с названием и типом этого устройства (их может быть несколько). Например:

```
OpenCL Device #0: CPU GenuineIntel Intel(R) Core(TM) i7-2700K CPU @ 3.50GHz with Oper
OpenCL Device #1: GPU Intel(R) Corporation Intel(R) UHD Graphics 630 with OpenCL 2.1
```

Процедура установки драйверов описана для разных устройств в статье на сайте mql5.com. Разумеется, поддержка распространяется на наиболее популярные устройства от Intel, AMD, ATI, Nvidia.

Конечно, по количеству ядер и быстродействию распределенных вычислений центральные процессоры значительно уступают графическим "платам", но и хорошего многоядерного центрального процессора будет вполне достаточно для значительного увеличения производительности.

> Важно: Если на компьютере имеется видеокарта с поддержкой OpenCL, то ставить программную эмуляцию OpenCL на центральном процессоре не нужно!

Драйвера OpenCL-устройств автоматизируют распределение расчетов по ядрам. Например, если нужно выполнить миллион однотипных вычислений с различающимися векторами, а в распоряжении есть всего тысяча ядер, то драйвера будут автоматически запускать каждую следующую задачу по мере готовности предыдущих и освобождения ядер.

Подготовительные операции по настройке среды исполнения OpenCL в MQL-программе выполняются однократно с помощью функций вышеупомянутого OpenCL API.

1. Создание контекста для программы OpenCL (выбор устройства, например, видеокарты, ЦП или любого доступного): *CLContextCreate(CL_USE_ANY)*. Функция вернет дескриптор контекста (целое число, обозначим его условно *ContextHandle*).

2. Создание в полученном контексте OpenCL-программы: она "компилируется" на основе исходного кода на языке OpenCL с помощью вызова функции *CLProgramCreate*, в которую текст кода передается через параметр *Source*: *CLProgramCreate(ContextHandle, Source, BuildLog)*. Функция вернет дескриптор программы (целое число *ProgramHandle*). Здесь важно отметить, что внутри исходного кода этой программы должны присутствовать функции (хотя бы одна), помеченные специальным ключевым словом *__kernel* (или просто *kernel*): именно они содержат части алгоритма, подлежащие распараллеливанию (см. пример ниже). Разумеется, программист может для упрощения (декомпозиции исходного кода) разнести логические подзадачи функции-кернела на другие вспомогательные функции и вызывать их из кернела: при этом помечать вспомогательные функции словом *kernel* не надо.

3. Регистрация кернела для выполнения по имени одной из тех функций, что помечены в коде OpenCL-программы как "кернелобразующие": *CLKernelCreate(ProgramHandle, KernelName)*. Вызов этой функции вернет дескриптор кернела (целое число, допустим — *KernelHandle*). Вы можете подготовить в OpenCL-коде много разных функций и зарегистрировать их как разные кернелы.

4. При необходимости, создание буферов для массивов данных, передаваемых по ссылке в кернел и для возвращаемых значений/массивов: *CLBufferCreate(ContextHandle, Size * sizeof(double), CL_MEM_READ_WRITE)* и др. Буфера тоже идентифицируются и управляются с помощью дескрипторов.

Далее, однократно или, при необходимости, многократно (например, в обработчиках событий индикатора или эксперта), производятся непосредственно вычисления по следующей схеме:

I. Передача входных данных и/или привязка входных/выходных буферов с помощью *CLSetKernelArg(KernelHandle,...)* и/или *CLSetKernelArgMem(KernelHandle,..., BufferHandle)*. Первая функция обеспечивает установку скалярного значения, а вторая эквивалента передаче или

получению значения (или массива значений) по ссылке. На этом этапе происходит перемещение данных из MQL5 в исполнительное ядро OpenCL. Для записи данных в буфер используется *CLBufferWrite(BufferHandle,...)*. Параметры и буфера станут доступны OpenCL-программе во время выполнения кернела.

II. Выполнение параллельных вычислений вызовом конкретного кернела *CLExecute(KernelHandle,...)*. Функция-кернел сможет записать результаты своей работы в выходной буфер.

III. Получение результата с помощью *CLBufferRead(BufferHandle)*. На этом этапе происходит обратное перемещение данных из OpenCL в MQL5.

После завершения вычислений следует освободить все дескрипторы: *CLBufferFree(BufferHandle)*, *CLKernelFree(KernelHandle)*, *CLProgramFree(ProgramHandle)*, *CLContextFree(ContextHandle)*.

Данная последовательность условно обозначена на следующей схеме.

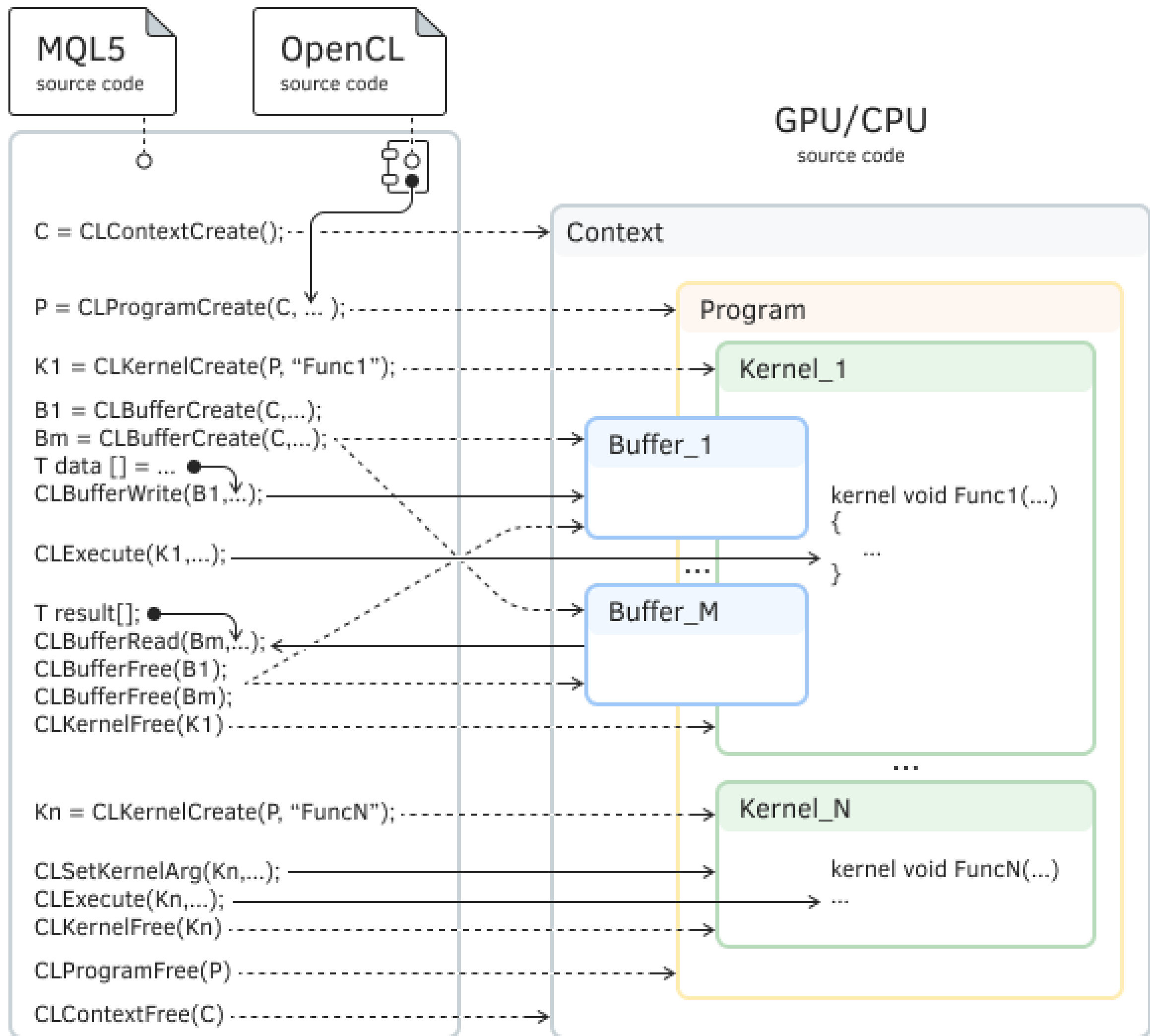


Схема взаимодействия MQL-программы и OpenCL-вложения

Исходный код OpenCL рекомендуется писать в отдельных текстовых файлах, которые затем можно подключать к MQL5-программе с помощью ресурсных переменных.

Стандартная библиотека заголовочных файлов, поставляемая с терминалом, содержит класс-обертку для работы с OpenCL: *MQL5/Include/OpenCL/OpenCL.mqh*.

А примеры использования OpenCL можно найти в каталоге *MQL5/Scripts/Examples/OpenCL/*. В частности, там имеется скрипт *MQL5/Scripts/Examples/OpenCL/Double/Wavelet.mq5*, производящий вейвлет-преобразование временного ряда (можно взять искусственную кривую по стохастической модели Вейерштрасса или приращения цен текущего финансового инструмента). В любом случае исходными данными для алгоритма выступает массив — двумерное изображение ряда.

При запуске этого скрипта, как и при запуске любой другой MQL-программы с OpenCL-кодом, терминал подберет наиболее быстродействующее устройство (если их несколько, и конкретное устройство не было выбрано в самой программе или не было уже определено ранее). Информация об этом выводится в закладку *Журнал* (журнал терминала, а не экспертов).

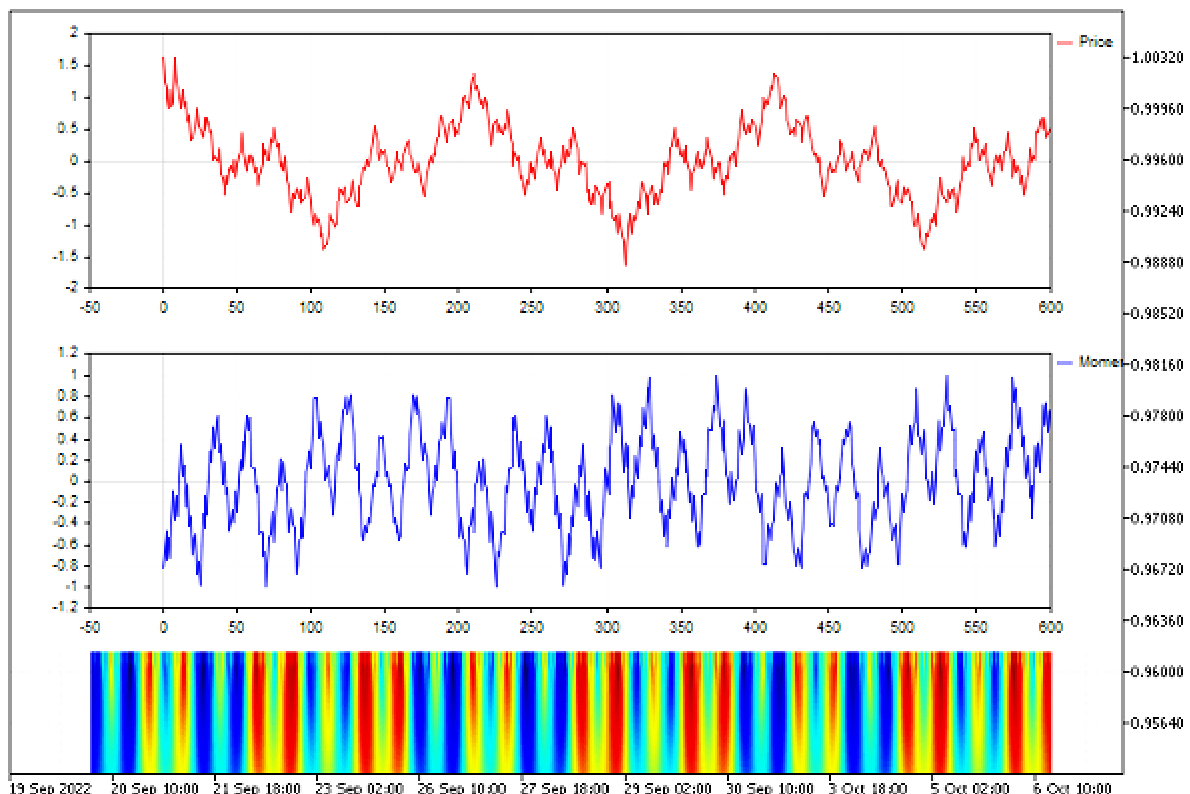
```
Scripts script Wavelet (EURUSD,H1) loaded successfully
OpenCL  device #0: GPU NVIDIA Corporation NVIDIA GeForce GTX 1650 with OpenCL 3.0 (16
OpenCL  device #1: GPU Intel(R) Corporation Intel(R) UHD Graphics 630 with OpenCL 3.0
OpenCL  device performance test started
OpenCL  device performance test 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
```

В результате выполнения скрипт выводит в закладку *Эксперты* записи с измерениями скорости расчетов обычным способом (последовательно, на ЦПУ) и параллельным (на ядрах OpenCL).

```
OpenCL: GPU device 'Intel(R) UHD Graphics 630' selected
time CPU=5235 ms, time GPU=125 ms, CPU/GPU ratio: 41.880000
```

Соотношение скоростей, в зависимости от специфики задачи, может достигать десятков.

На график скриптом выводятся: исходное изображение, его производная в виде приращений, а также результат вейвлет-преобразования.



Исходный смоделированный ряд, его приращения и вейвлет-преобразование

Обратите внимание, что графические объекты остаются на графике после завершения работы скрипта. Их нужно будет удалить вручную.

А вот как выглядит исходный OpenCL-код вейвлет-преобразования, вынесенный в отдельный файл *MQL5/Scripts/Examples/OpenCL/Double/Kernels/wavelet.cl*.

```
// требуется повышенная точность расчетов double
// (по умолчанию, без этой директивы получим float)
#pragma OPENCL EXTENSION cl_khr_fp64 : enable

// Вспомогательная функция 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;
}
```

Полную информацию о синтаксисе, встроенных функциях и принципах функционирования OpenCL можно найти на официальном сайте разработчика Khronos Group.

В частности, интересно отметить, что OpenCL поддерживает не только привычные скалярные числовые типы данных (начиная от *char* и заканчивая *double*), но и векторные *(u)charN*, *(u)shortN*,

(u)intN, (u)longN, floatN, doubleN, где N = {2|3|4|8|16} и обозначает длину вектора. В данном примере это не используется.

Помимо упомянутого ключевого слова *kernel* важную роль в организации параллельных вычислений играет функция *get_global_id*: она позволяет узнать в коде номер вычислительной подзадачи, выполняющейся в данный момент. Очевидно, что расчеты в разных подзадачах должны отличаться (иначе не было бы смысла задействовать много ядер). И в данном примере, поскольку задача подразумевает анализ двумерного изображения, её фрагменты удобнее идентифицировать с помощью двух ортогональных координат — именно их в вышеприведенном коде мы получаем с помощью двух вызовов *get_global_id(0)* и *get_global_id(1)*.

В принципе, размерность данных для задачи мы сами задаем при вызове в MQL5 функции *CLExecute* (см. далее).

В файле *Wavelet.mq5* исходный код OpenCL подключен с помощью директивы:

```
#resource "Kernels/wavelet.cl" as string cl_program
```

Размер изображения задан макросами:

```
#define SIZE_X 600
#define SIZE_Y 200
```

Для управления OpenCL используется стандартная библиотека с классом *COpenCL*. Его методы имеют похожие названия и внутри используют соответствующие встроенные функции OpenCL из MQL5 API. С ним предлагается ознакомиться самостоятельно.

```
#include <OpenCL/OpenCL.mqh>
```

В упрощенном виде (без проверок на ошибки и визуализации) MQL-код, обеспечивающий запуск преобразования, представлен ниже. Связанные с вейвлет-преобразованием действия сведены в класс *CWavelet*.

```
class CWavelet
{
protected:
   ...
   int         m_xsize;              // размеры изображения по осям
   int         m_ysize;
   double      m_wavelet_data_GPU[]; // результат попадет сюда
   COpenCL     m_OpenCL;             // объект-обертка
   ...
};
```

Основные "параллельные" вычисления организует его метод *CalculateWavelet_GPU*.

```
bool CWavelet::CalculateWavelet_GPU(double &data[], uint &time)
{
   int datacount = ArraySize(data); // размер изображения (количество точек)

   // компилируем cl-программу по её исходному коду
   m_OpenCL.Initialize(cl_program, true);

   // регистрируем единственную функцию-кернел из cl-файла
   m_OpenCL.SetKernelsCount(1);
   m_OpenCL.KernelCreate(0, "Wavelet_GPU");

   // регистрируем 2 буфера для ввода и вывода данных, записываем входной массив
   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];             // задача анализа двумерного изображения - отсюда разме
   uint offset[2] = {0, 0};  // начинаем с самого начала (а можно что-то пропустить)
   work[0] = m_xsize;
   work[1] = m_ysize;

   // задаем входные данные
   m_OpenCL.SetArgument(0, 1, datacount);
   m_OpenCL.SetArgument(0, 2, m_xsize);
   m_OpenCL.SetArgument(0, 3, m_ysize);

   time = GetTickCount();     // отсечка времени для замера скорости
   // запуск вычислений на GPU, двумерная задача
   m_OpenCL.Execute(0, 2, offset, work);

   // получение результатов в выходной буфер
   m_OpenCL.BufferRead(1, m_wavelet_data_GPU, 0, 0, m_xsize * m_ysize);

   time = GetTickCount() - time;

   m_OpenCL.Shutdown(); // освобождаем все ресурсы - вызов всех нужных функций CL***F
   return true;
}
```

В исходном коде примера присутствует закомментированная строка с вызовом *PreparePriceData* для подготовки входного массива на основе реальных цен: вы можете активировать её вместо предыдущей строки с вызовом *PrepareModelData* (которая генерирует искусственный ряд).

```
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);
   ... // визуализация ряда и приращений
   CWavelet wavelet;
   uint time_gpu = 0;
   wavelet.CalculateWavelet_GPU(momentum_data, time_gpu);
   ... // визуализация результата вейвлет-преобразования
}
```

Для работы с OpenCL выделен специальный набор кодов ошибок (с префиксом ERR_OPENCL_, начиная с кода 5100, ERR_OPENCL_NOT_SUPPORTED), доступный в справке. При возникновении проблем с исполнением OpenCL-программ, терминал выводит подробную диагностику в журнал с указанием кодов ошибок.

Заключение

Этот раздел завершает книгу. На протяжении 7 частей и множества глав мы знакомились с различными аспектами разработки MQL-программ, начиная с самых основ языка и заканчивая продвинутыми смежными технологиями, позволяющими постепенно переходить от создания отдельных узкоспециализированных инструментов трейдера к комплексным системам и продуктам.

Перед вами мир профессионального алготрейдинга, где полученные знания помогу воплотить ваши идеи в жизнь и добиться коммерческого успеха.

- Разрабатывайте собственные приложения и продавайте их через Маркет. Это крупнейший магазин программ для MetaTrader с готовой инфраструктурой для авторов. Вы получаете доступ к огромной аудитории, защиту и систему лицензирования для продуктов, а также готовую систему для приема платежей.
- Разрабатывайте приложения на заказ через Фриланс. Ваш ждет огромное количество заказов, удобная система работы и защита платежей.
- Делитесь опытом, публикуя свой код в Библиотеке. О ваших разработках узнают тысячи трейдеров из сообщества MQL5.community.

Ну и конечно, продолжайте учиться. На сайте www.mql5.com вы найдете огромное количество информации и готовых алгоритмов:

- Статьи по программированию, где профессиональные авторы детально разбирают прикладные задачи.
- Форум, где можно обменяться опытом и попросить совета у других разработчиков.
- Библиотека исходных кодов, которые помогут при изучении возможностей языка MQL5 и разработке собственных программ.

Наконец, хочется напомнить, что разработка программного обеспечение — это не только программирование, но и множество других не менее важных направлений: написание технического задания (даже если только для себя лично), проектирование, макетирование, дизайн интерфейса пользователя, документация, сопровождение. Все эти аспекты оказывают существенное влияние на эффективность вашей работы как программиста и качество конечного результата.

В частности, большинство прикладных задач можно "разложить по полочкам" на стандартные алгоритмы и принципы, которые уже давно применяются программистами, вне зависимости от языка. Сюда относятся и паттерны проектирования, и сборники структур данных, оптимизированных для конкретных задач, и средства автоматизации самой разработки. Все это можно и нужно применять на платформе MetaTrader 5 с помощью и в дополнение к MQL5. А книга — лишь первый шаг на пути профессионального роста.

Присоединяйтесь к сообществу разработчиков торговых роботов MQL5.community!